

Скользкие места C++

Как избежать проблем
при проектировании и компиляции
ваших программ

Стефан К. Дьюхэрст



Москва, 2017

C++ Gotchas

Avoiding Common Problems in Coding and Design

Stephen C. Dewhurst

✧ **Addison-Wesley**

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Стефан К. Дьюхэрст

Скользкие места C++

**Как избежать проблем
при проектировании и компиляции
ваших программ**

УДК 004.4
ББК 32.973.26-018.2
Д92

Стефан К. Дьюхэрст

Д92 Скользкие места С++. Как избежать проблем при проектировании и компиляции ваших программ. – М.: ДМК Пресс. – 264 с.: ил.

ISBN 9785-97060-475-5

Вы держите в руках руководство по тому, как не допускать и исправлять 99% типичных, разрушительных и просто любопытных ошибок при проектировании и реализации программ на языке С++. Эту книгу можно рассматривать также, как взгляд посвященного на нетривиальные особенности и приемы программирования на С++.

Обсуждаются как наиболее распространенные «ляпы», имеющиеся почти в любой программе на С++, так и сложные ошибки в использовании синтаксиса, преобразований типов, инициализации, управления памятью и ресурсами, полиморфизма, а также при проектировании классов и иерархий. Все ошибки и их последствия обсуждаются в контексте. Подробно описываются способы разрешения указанных проблем.

Автор знакомит читателей с идиомами и паттернами проектирования, с помощью которых можно решать типовые задачи. Читатель также узнает много нового о плохо понимаемых возможностях С++, которые применяются в продвинутых программах и проектах. На сайте <http://www.semantics.org> можно найти полный код примеров из книги.

В книге рассказывается, как миновать наиболее серьезные опасности, подстерегающие программиста на С++. Программисты найдут в ней практические рекомендации, которые позволят им стать настоящими экспертами.

Издание предназначено для всех программистов, желающих научиться писать правильные и корректно работающие программы на языке С++.

**УДК 004.4
ББК 32.973.26-018.2**

Original English language edition published by Pearson Education, Inc. Copyright ©

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-0-321-12518-7 (англ.)

ISBN 978-5-97060-475-5 (рус.)

Copyright © by Pearson Education, Inc.

© Перевод на русский язык, оформление,
издание, ДМК Пресс



Содержание

Предисловие	9
Благодарности	13
Глава 1. Основы	
Совет 1. Избыточное комментирование	15
Совет 2. Магические числа	17
Совет 3. Глобальные переменные	19
Совет 4. Отличайте перегрузку от инициализации аргументов по умолчанию	21
Совет 5. О неправильной интерпретации ссылок	22
Совет 6. О неправильной интерпретации const	25
Совет 7. Не забывайте о тонкостях базового языка	26
Совет 8. Отличайте доступность от видимости	29
Совет 9. О неграмотности	33
Лексика	33
Нулевые указатели	34
Акронимы	35
Совет 10. Не игнорируйте идиомы	35
Совет 11. Не мудрствуйте лукаво	38
Совет 12. Не ведите себя как дети	40
Глава 2. Синтаксис	
Совет 13. Не путайте массивы с инициализаторами	42
Совет 14. Неопределенный порядок вычислений	43
Порядок вычисления аргументов функции	43
Порядок вычисления подвыражений	44
Порядок вычисления размещающего new	45
Операторы, которые фиксируют порядок вычислений	46
Некорректная перегрузка операторов	47
Совет 15. Помните о предшествовании	47
Приоритеты и ассоциативность	47
Проблемы, связанные с приоритетом операторов	48
Проблемы, связанные с ассоциативностью	49
Совет 16. Подводные камни в предложении for	50
Совет 17. Принцип «максимального куска»	53

Совет 18. О порядке следования спецификаторов в объявлениях	54
Совет 19. Функция или объект?	55
Совет 20. Перестановка квалификаторов типа	55
Совет 21. Автоинициализация	56
Совет 22. Статические и внешние типы	58
Совет 23. Аномалия при поиске операторной функции	58
Совет 24. Тонкости оператора ->	60

Глава 3. Препроцессор

Совет 25. Определение литералов с помощью #define	62
Совет 26. Определение псевдофункций с помощью #define	64
Совет 27. Не увлекайтесь использованием директивы #if	66
Использование директивы #if для отладки	66
Использование #if для переносимости	68
А как насчет классов?	69
Практика – критерий истины	70
Совет 28. Побочные эффекты в утверждениях	70

Глава 4. Преобразования

Совет 29. Преобразование посредством void *	73
Совет 30. Срезка	76
Совет 31. Преобразование в указатель на константу	78
Совет 32. Преобразование в указатель на указатель на константу	79
Совет 33. Преобразование указателя на указатель на базовый класс	82
Совет 34. Проблемы с указателем на многомерный массив	82
Совет 35. Бесконтрольное понижающее приведение	84
Совет 36. Неправильное использование операторов преобразования ..	84
Совет 37. Непреднамеренное преобразование с помощью конструктора	88
Совет 38. Приведение типов в случае множественного наследования ...	91
Совет 39. Приведение неполных типов	92
Совет 40. Приведения в старом стиле	93
Совет 41. Статические приведения	94
Совет 42. Инициализация формальных аргументов временными объектами	97
Совет 43. Время жизни временных объектов	100
Совет 44. Ссылки и временные объекты	101
Совет 45. Неоднозначность при использовании dynamic_cast	104
Совет 46. Контравариантность	108

Глава 5. Инициализация

Совет 47. Не путайте инициализацию и присваивание	111
---	-----

Совет 48. Правильно выбирайте область видимости переменной	114
Совет 49. Внимательно относитесь к операциям копирования	116
Совет 50. Побитовое копирование объектов классов	119
Совет 51. Не путайте инициализацию и присваивание в конструкторах	121
Совет 52. Несогласованный порядок членов в списке инициализации ...	123
Совет 53. Инициализация виртуальных базовых классов	124
Совет 54. Инициализация базового класса в конструкторе копирования	128
Совет 55. Порядок инициализации статических данных во время выполнения	131
Совет 56. Прямая инициализация и инициализация копированием ...	133
Совет 57. Прямая инициализация аргументов	136
Совет 58. Что такое оптимизация возвращаемого значения?	137
Совет 59. Инициализация статических членов в конструкторе	141

Глава 6. Управление памятью и ресурсами

Совет 60. Различайте выделение и освобождение памяти для скаляров и для массивов	143
Совет 61. Контроль ошибок при выделении памяти	146
Совет 62. Подмена глобальных new и delete	148
Совет 63. Об области видимости и активации функций-членов new и delete	150
Совет 64. Строковые литералы в выражении throw	151
Совет 65. Обработывайте исключения правильно	154
Совет 66. Внимательно относитесь к адресам локальных объектов ...	157
Исчезающие фреймы стека	157
Затирание статических переменных	158
Идиоматические трудности	159
Проблемы локальной области видимости	159
Исправление ошибки путем добавления static	160
Совет 67. Помните, что захват ресурса есть инициализация	161
Совет 68. Правильно используйте auto_ptr	164

Глава 7. Полиморфизм

Совет 69. Кодирование типов	168
Совет 70. Невиртуальный деструктор базового класса	172
Неопределенное поведение	172
Виртуальные статические функции-члены	173
Всех обманом	174
Исключения из правил	175
Совет 71. Соккрытие неvirtуальных функций	176
Совет 72. Не делайте шаблонные методы слишком гибкими	179
Совет 73. Перегрузка виртуальных функций	180

Совет 74. Виртуальные функции с аргументами по умолчанию	181
Совет 75. Вызовы виртуальных функций из конструкторов и деструкторов	183
Совет 76. Виртуальное присваивание	185
Совет 77. Различайте перегрузку, переопределение и сокрытие	187
Совет 78. О реализации виртуальных функций и механизма переопределения	192
Совет 79. Вопросы доминирования	197

Глава 8. Проектирование классов

Совет 80. Интерфейсы get/set	201
Совет 81. Константные и ссылочные данные-члены	204
Совет 82. В чем смысл константных функций-членов?	206
Синтаксис	206
Простая семантика и механизм работы	207
Семантика константной функции-члена	208
Совет 83. Различайте агрегирование и использование	210
Совет 84. Не злоупотребляйте перегрузкой операторов	214
Совет 85. Приоритеты и перегрузка	216
Совет 86. Операторы, являющиеся членами и друзьями класса	217
Совет 87. Проблемы инкремента и декремента	218
Совет 88. Неправильная интерпретация шаблонных операций копирования	221

Глава 9. Проектирование иерархий

Совет 89. Массивы объектов класса	224
Совет 90. Не всегда один контейнер можно подставить вместо другого	226
Совет 91. Что такое защищенный доступ?	229
Совет 92. Применение открытого наследования для повторного использования кода	232
Совет 93. Конкретные открытые базовые классы	235
Совет 94. Не пренебрегайте вырожденными иерархиями	236
Совет 95. Не злоупотребляйте наследованием	237
Совет 96. Управление на основе типов	240
Совет 97. Космические иерархии	242
Совет 98. Задание «интимных» вопросов объекту	244
Совет 99. Опрос возможностей	248

Список литературы	252
--------------------------------	-----

Предметный указатель	253
-----------------------------------	-----



Предисловие

Эта книга – результат почти двадцатилетней работы, полной мелких разочарований, серьезных ошибок, бессонных ночей и выходных, добровольно проведенных за клавиатурой компьютера. Я включил в нее 99 глав, в которых описываются «скользкие места» (gotcha) в языке C++, которые иногда являются источниками распространенных ошибок и путаницы, а иногда просто вызывают интерес. С большинством из них я сталкивался лично (как это ни печально).

У слова «gotcha» довольно туманная история и множество определений. В этой книге мы будем понимать под ним типичную проблему, возникающую при проектировании и программировании на языке C++, которую можно предотвратить. В книге описаны самые разные проблемы такого рода: мелкие синтаксические тонкости, серьезные огрехи при проектировании и поведение, которое противно всем «нормам общежития».

Почти десять лет, как я начал включать замечания об отдельных скользких местах в материалы курса по C++, который я читаю. Мне казалось, что, обращая внимание студентов на типичные ошибки и просчеты, и одновременно показывая, как следует решать задачу правильно, я буду поспособствовать тому, что новые поколения программистов на C++ не станут повторять грехов своих предшественников. В общем и целом, эта идея оказалась удачной, и меня попросили подготовить собрание взаимосвязанных скользких мест для презентации на конференциях. Презентации завоевали популярность (не я один такой?), в результате чего я получил предложение написать книгу на эту тему.

Когда заходит речь о том, как не поскользнуться при работе с C++ или исправить последствия ошибки, нельзя не затронуть такие смежные вопросы, как наиболее распространенные паттерны проектирования, идиомы и технические детали языка.

Эта книга не о паттернах проектирования, но мы часто будем ссылаться на них как на средство обойти скользкое место. Названия паттернов по традиции принято писать с большой буквы, например: Template Method (Шаблонный Метод) или Bridge (Мост). При упоминании паттерна мы вкратце опишем его суть, если это не слишком сложно, но за подробным обсуждением отсылаем к работам, специально посвященным паттернам. Более полное описание конкретных паттернов, равно как и глубокое обсуждение этой темы в общем, можно найти в книге Эриха Гаммы и др. «Design patterns»*. Описания паттернов Acyclic Visitor (Ациклический ациклический Посетительпосетитель), Monostate (Моносостояниемоносостояние) и Null Object (Пустой пустой Объектобъект) можно найти в книге

* Имеется русский перевод: Гамма и др. «Паттерны проектирования». (Прим. перев.)

Robert Martin «Agile Software Development» («Разработка программ с удовольствием»).

С точки зрения скользких мест, у паттернов проектирования есть два важных свойства. Во-первых, каждый паттерн – это описание апробированной, неизменно приносящей успех техники проектирования, которую можно адаптировать под конкретные условия, возникающие при решении новых задач. Во-вторых, и это даже более важно, само упоминание о том, что в приложении используется тот или иной паттерн, документирует не только примененную технику, но также причины и результаты ее применения.

Например, если мы видим, что при проектировании программы был использован паттерн Bridge, то сразу понимаем, что реализация абстрактного типа данных разбита на интерфейсный класс и класс реализации. Кроме того, мы знаем, что сделано это было для того, чтобы разорвать связь между интерфейсом и реализацией, в результате чего изменение реализации никак не затронет пользователей интерфейса. Знаем мы и то, какие накладные расходы во время выполнения влечет за собой такое разделение, как организован исходный код, реализующий абстрактный тип данных, и целый ряд других деталей. Название паттерна – это недвусмысленная ссылка на кладезь информации и опыта, стоящий за соответствующей техникой. Обдуманное и правильное применение паттернов и связанной с ними терминологии при проектировании и документировании программ помогает понять код и не споткнуться на скользком месте.

C++ – это сложный язык программирования, а чем сложнее язык, тем важнее употребление идиом. В контексте языка программирования под идиомой понимается широко распространенная и имеющая всем понятный смысл комбинация низкоуровневых языковых средств, приводящая к появлению высокоуровневой конструкции. Такова же роль паттернов в проектировании программ. Поэтому мы и можем говорить об операциях копирования, функциональных объектах, интеллектуальных указателях и возбуждении исключений в C++, не опускаясь до уровня деталей реализации.

Важно подчеркнуть, что идиома – это не просто известная комбинация языковых средств, но и определенные ожидания относительно того, как эта комбинация будет себя вести. Каков смысл операции копирования? Чего ожидать в случае возбуждения исключения? Многие советы в этой книге касаются распознавания идиом и их использования в проектировании и кодировании. Можно сказать и так: многие из описанных скользких мест – не что иное, как игнорирование какой-то идиомы C++, а для решения проблемы часто всего-то и нужно, что следовать подходящей идиоме (см. «Совет 10»).

Немало глав в этой книге посвящено описанию некоторых нюансов языка, которые часто понимают неправильно, что и приводит к проблемам. Хотя некоторые примеры могут показаться надуманными, но незнание соответствующего материала не позволит вам стать настоящим экспертом по C++. Эти «закоулки» сами по себе могли бы стать предметом весьма любопытных и полезных исследований. Существуют они в C++ не без причины, а опытные программисты нередко прибегают к ним при разработке нетривиальных приложений.

Между «скользкими местами» и паттернами проектирования можно провести и еще одну аналогию: важность изложения предмета на сравнительно простых примерах. Простые паттерны очень важны. В некотором смысле они даже важнее технически более трудных паттернов, поскольку выше вероятность их широкого применения.

Точно также, описанные в этой книге скользкие места сильно разнятся по сложности: от простого увещания действовать как ответственные профессионалы (см. «Совет 12») до предупреждения избегать неверной интерпретации правила доминирования при виртуальном наследовании (см. «Совет 79»). Но по аналогии с паттернами, в повседневной практике призыву подходить к делу ответственно скорее найдется место, нежели правилу доминирования.

Через всю книгу красной нитью проходят две темы. Первая – это исключительная важность соглашений. Особенно актуально это для такого сложного языка, как C++. Следование принятым соглашениям позволяет эффективно и точно доносить свои мысли до других людей. Вторая тема – это осознание того факта, что написанный нами код кому-то предстоит сопровождать. Сопровождение может быть прямым (и тогда наша программа должна быть понятна компетентному специалисту), или косвенным и (в этом случае нужно быть уверенным, что код останется правильным, даже если его поведение будет модифицировано в результате изменений, которые будут внесены в отдаленном будущем).

Настоящая книга построена в виде собрания коротких эссе, в каждом из которых описывается одна или несколько взаимосвязанных проблем и даются советы, как избежать их или устранить последствия. Я не уверен, что можно написать внутренне целостную книгу на эту тему в силу «анархической» природы материала. Тем не менее, материал разбит на главы в соответствии с общей природой проблем или областями, в которых они обычно встречаются.

Кроме того, при обсуждении одного скользкого места неизбежно приходится затрагивать и другие. Когда это имеет смысл, то есть практически всегда, я даю прямые отсылки. Связность изложения внутри одной темы иногда также оказывается под вопросом. Часто, прежде чем переходить к описанию проблемы, нужно представить контекст, в котором она проявляется. А это, в свою очередь, влечет за собой обсуждение какой-то техники, идиомы, паттерна или нюанса языка и может далеко увести от темы раздела. Я старался свести такое «растекание мыслию по древу» к минимуму, но, думается, было бы нечестно отказаться от него полностью. Для эффективного программирования на C++ нужно осознанно применять знания из таких разных областей, что практически невозможно вообразить, как исследовать этиологию этого процесса, не прибегая к подобным эклектичным рассуждениям.

Разумеется, вовсе не обязательно – и даже не рекомендуется – читать эту книгу подряд, от «Совета 1» к «Совету 99». Прием лекарства в таких дозах может навеки отвратить вас от программирования на C++. Гораздо лучше начать с того места, на котором вам уже случалось поскользнуться, или с того, которое кажется вам интересным, а потом следовать по ссылкам. А можно вообще читать в случайном порядке.

В тексте применяется ряд типографских эффектов, призванных облегчить усвоение материала. Во-первых, неправильный или не рекомендуемый код напечатан на сером фоне, а правильный – на белом. Во-вторых, приведенный в тексте код для краткости и ясности слегка отредактирован. Поэтому примеры не будут компилироваться без дополнительного кода. Исходный код для нетривиальных примеров можно скачать с сайта автора по адресу www.semantics.org. Для таких случаев в тексте приводится сокращенный путь, например: `gotcha00/somocode.cpp`.

И напоследок одно предостережение: чего ни в коем случае не следует делать со «скользкими местами», так это повышать их в статусе до идиом или паттернов. Один из признаков правильного использования паттерна или идиомы состоит в том, что идея ее применения в данном контексте возникает подсознательно, спонтанно.

Распознавание же скользкого места можно уподобить условному рефлексу на опасность: обжегшись на молоке, дуешь на воду. Но, как и в случае спичек и огнестрельного оружия, совершенно необязательно обжигаться или получать рану в голову самому, чтобы научиться распознавать опасности опасность и сторониться ее. Обычно бывает достаточно предупреждения. Читайте эту книгу средством прямо смотреть в глаза опасностям, подстерегающим вас в темных закоулках C++.

Стефан К. Дьюхэрст
Карвер, Массачусетс
июль 2002



Благодарности

Редакторы часто устаиваются лишь краткого упоминания в благодарностях, например, такого: «... я также благодарен своему редактору, который, наверное, чем-то занимался, пока я корпел над рукописью». Но без моего редактора, Дэбби Лафферти (Debbie Lafferty), эта книга вообще не увидела бы света. Когда я явился к ней с малоинтересным предложением написать заурядный учебник по программированию, она предложила вместо этого расширить главу, посвященную скользким местам. Я отказывался. Она настаивала. Последнее слово осталось за ней. К счастью, Дэбби была достаточно тактична, во всяком случае, извечного редакторского: «Ну, мы же вам говорили», я от нее не слышал. Ну и, кроме того, она-то уж точно кое-чем занималась, пока я корпел над рукописью.

Я также выражаю благодарность рецензентам, которым делились своим временем и опытом, чтобы эта книга получилась лучше. Рецензировать сырую рукопись – занятие небystрое, часто утомительное, иногда вызывающее раздражение. Это акт профессиональной любезности, не приносящий почти никаких материальных выгод (см. «Совет 12»). Я весьма ценю глубокие, а иногда колкие комментарии своих рецензентов. Советами по техническим вопросам и нормам общественной морали, исправлениями, фрагментами кода и ехидными замечаниями со мной поделились Стив Клэмидж (Steve Clamage), Томас Гшвинд (Thomas Gschwind), Брайан Керниган (Brian Kernighan), Патрик МакКиллен (Patrick McKillen), Джеффри Олдэм (Jeffrey Oldham), Дэн Сакс (Dan Saks), Мэттью Уилсон (Matthew Wilson) и Леор Золман (Leor Zolman).

Леор начал рецензировать книгу задолго до того, как появилась рукопись, поскольку посылал мне ядовитые комментарии на публикации в Web ранних вариантов некоторых из описанных в этой книге скользких мест. Сара Хьюинс (Sarah Hewins) – мой лучший друг и самый строгий критик – заслужила оба эти титула, рецензируя разные версии рукописи. Дэвид Р. Дьюхэрст (David R. Dewhurst) часто откладывал весь проект на неопределенное время. Грег Комей (Greg Comeau) любезно позволил мне пользоваться своим замечательным и полностью поддерживаемым стандарт C++ компилятором для проверки кода.

Как и любая нетривиальная книга по C++, эта является плодом трудов многих людей. На протяжении ряда лет мои студенты, клиенты и коллеги обогащали мой, не сказать, чтобы счастливый, опыт преодоления скользких мест C++, а многие помогали находить решения проблем. Шаблон `Select` в «Совете 11» и политика `OpNewCreator` из «Совета 70» были впервые были опубликованы к книге Andrei Alexandrescu «Modern C++ Design»*.

* Имеется русский перевод: Андрей Александреску «Современное проектирование на C++», издательский дом «Вильямс», 2002. (Прим. перев.)

С проблемой возврата ссылки на константный аргумент, описанной в «Совете 44», я впервые столкнулся в книге Cline и др. «C++ FAQ» («Часто задаваемые вопросы по C++»), после чего она немедленно стала появляться в коде, написанном моими клиентами. Там же описывается упомянутый в «Совете 73» способ, позволяющий уйти от применения перегруженных виртуальных функций.

Шаблон `Cptr` из «Совета 83» – это модифицированная версия шаблона `CountedPtr`, описанного в книге Nicolai Josuttis «The C++ Standard Library» (Стандартная библиотека C++).

Скотт Мейерс немало написал о нежелательности перегрузки операторов `&&`, `||`, `и`, `,` в своей книге «More Effective C++»*. В книге «Effective C++»[†] он подробно обсуждает необходимость возврата результата бинарного оператора по значению, о чем идет речь в(см. «Совете 58», а в книге «Effective STL» (Эффективное использование STL) описывает неправильное применение шаблона `auto_ptr` (см. «Совет 68»). Упомянутая в «Совете 87» техника возврата константного значения из операторов постфиксного инкремента и декремента также описана в книге «More Effective C++».

От Дэна Сакса я впервые услышал убедительные аргументы в пользу файла с опережающими объявлениями («Совет 8»). Он же первым описал «сержантский оператор» («Совет 17») и убедил меня не проверять выход за границы диапазона значений при инкременте и декремента перечислений («Совет 87»).

* Имеется русский перевод: Скотт Мейерс «Наиболее эффективное использование C++», издательство ДМК, 2000. (Прим. перев.)

** Имеется русский перевод: Скотт Мейерс «Эффективное использование C++», издательство ДМК, 2000. (Прим. перев.)



Глава 1. Основы

Тот факт, что некоторая проблема описана в разделе «Основы», не означает, что она не может быть серьезной или часто встречающейся. На самом деле, широкое распространение ошибок, описанных в этой главе, — гораздо больший повод для обеспокоенности, нежели чем технически более серьезные проблемы, которые обсуждаются в последующих главах. Сама простота и фундаментальность рассматриваемых ниже ошибок подразумевает, что в той или иной степени они встречаются почти в любом коде на C++.

Совет 1. Избыточное комментирование

Многие комментарии не нужны. Из-за них труднее читать и сопровождать код, а часто они только сбивают с толку сопровождающего программу программиста. Рассмотрим следующий пример:

```
a = b; // присвоить переменной a значение b
```

Этот комментарий ничего не сообщает о смысле предложения сверх того, что уже написано в самом коде, поэтому он бесполезен. Даже хуже, чем бесполезен. Он мешает. Во-первых, он отвлекает читателя от кода, увеличивая объем текста, который тот должен воспринять. Во-вторых, сопровождающему добавляется работы, поскольку комментарии приходится изменять, если модифицируется текст программы. В-третьих, при сопровождении этим часто пренебрегают.

```
c = b; // присвоить переменной a значение b
```

Добросовестный программист, сопровождающий программу, не может просто предположить, что комментарий не соответствует действительности. Он обязан проследить за ходом исполнения программы, чтобы понять, : то ли это ошибка, то ли любезность (с является ссылкой на a), то ли некоторая тонкость (присваивание значения переменной c в дальнейшем каким-то образом приведет к присваиванию того же значения переменной a). С самого начала не следовало сопроводить эту строку комментарием:

```
a = b;
```

Этот код и так совершенно ясен, без всяких комментариев, которые могут стать неверными в ходе сопровождения. Это Данное замечание вполне согласуется с выстраданным наблюдением, что самый эффективный код – это тот, который не написан. То же относится и к комментариям: лучшим является комментарий, который не нужно писать, поскольку и без него код самодокументирован.

Примеры излишних комментариев также часто встречаются в определениях классов либо в результате неверно понятого стандарта кодирования, либо в программах начинающих программистов:

```
class C {
    // Открытый интерфейс
    public:
        C(); // конструктор по умолчанию
        ~C(); // деструктор
    // . . .
};
```

Такое впечатление, что читаешь какую-то шпаргалку. Если программисту, который сопровождает вашу программу, надо напоминать, что означает метка `public:`, ему вряд ли стоит ему вообще поручать сопровождение. Ни один из этих комментариев ничего не дает опытному программисту на C++, а только загромождает код и увеличивает шансы на внесение ошибок в ходе сопровождения.

```
class C {
    // Public Interface
    protected:
        C( int ); // конструктор по умолчанию
    public:
        virtual ~C(); // деструктор
    // . . .
};
```

У программистов часто есть серьезный стимул не писать «лишних» строк в исходном тексте. Как это ни смешно звучит, но если некоторую конструкцию (функцию, открытый интерфейс класс и так далее) можно рационально и, следуя принятым соглашениям, уместить на одной «странице», то есть примерно в 30–40 строк, то ее легче понять. Если она переходит на вторую страницу, то понять ее вдвое сложнее. А если необходима и третья, то сложность восприятия увеличивается примерно в четыре раза.

Особенно отвратительна привычка помещать историю изменений в виде комментариев в начале или в конце файлов с исходными текстами:

```
/* 6/17/02 SCD исправил дурацкую ошибку */
```

Это полезная информация или сопровождающий просто хвастается? Комментарий станет абсолютно ненужным через неделю-другую после вставки, но торчать в тексте он будет годами, отвлекая внимание многих сопровождающих. Куда лучше оставлять такие комментарии в системе управления версиями; исходный текст программы на C++ – не место для списка вещей, предназначенных для стирки.

Один из лучших способов избежать комментариев и сделать код понятным и удобным для сопровождения: следовать простому и четко сформулированному соглашению об именовании и выбирать имена так, чтобы они отражали назначенные сущности (функции, класса, переменной и так далее). Особенно важны имена аргументов в объявлениях. Взгляните на следующую функцию, которая принимает три аргумента одного и того же типа:

```
/*
    Источник выполняет действие над целью.
    Arg1 - код действия, arg2 - источник, arg3 - цель.
*/
void perform( int, int, int );
```


Вроде бы ничего страшного, но представьте, что аргументов не три, а семь или восемь. Может быть, стоит поступить так:

```
void perform( int actionCode, int source, int destination );
```

Уже лучше, хотя еще необходим однострочный комментарий, описывающий, что делает функция (но не как она это делает). Имена формальных аргументов в объявлениях хороши тем, что, в отличие от комментариев, они сопровождаются вместе с остальным кодом, хотя прямого отношения к семантике программы и не имеют. Не могу представить себе программиста, который изменил бы смысл второго и третьего аргумента функции `perform`, не изменив их имена, но так и вижу легионы программистов, которые проведут такую модификацию, забыв про комментарий.

Кэти Старк (Kathy Stark) лучше всех выразила эту мысль в своей книге «Programming in C++» (Программирование на C++): «Если в программе используются осмысленные и мнемонические имена, то необходимость в дополнительных комментариях возникает лишь изредка. Если же используемые имена ничего не означают, то маловероятно, что комментарий сделает код яснее.»

Еще один способ свести число комментариев к минимуму – пользоваться стандартными или хорошо известными компонентами:

```
printf( "Hello, World!" ); // вывести "Hello, World" на экран
```

Этот комментарий бесполезен, к тому же и верен-то не всегда. Дело не в том, что стандартные компоненты обязательно самодокументированы, а в том, что они уже хорошо документированы и всем известны.

```
swap( a, a+1 );
sort( a, a+max );
copy( a, a+max, ostream_iterator<T>(cout, "\n") );
```

Поскольку `swap`, `sort` и `copy` – стандартные компоненты, дополнительный комментарий к ним только засоряет текст и вносит неточность в описание стандартных операций.

Я отнюдь не хочу сказать, что любой комментарий по природе своей вреден. Напротив, часто они комментарии необходимы, но их нужно сопровождать, а это обычно сложнее, чем сопровождать сам исходный текст, который они призваны документировать. Комментарий не должен повторять очевидное или нести информацию, которую лучше хранить в каком-нибудь другом месте. Задача не в том, чтобы любой ценой избавиться от комментариев, а в том, что свести их объем к минимуму, необходимому для того, чтобы код было проще понять и сопровождать.

Совет 2. Магические числа

Здесь под «магическими числами» я понимаю числовые литералы, употребляемые в контексте, где следовало бы воспользоваться именованными константами:

```
class Portfolio {
// . . .
Contract *contracts_[10];
char id_[10];
};
```

Основная проблема, связанная с магическими числами, в том, что у них нет никакой внятной семантики; они могут означать все, что угодно. Число 10 – это всего лишь 10, а не максимальное число контрактов или длина идентификатора. Поэтому при чтении или сопровождении кода приходится выяснять, что означает каждый литерал. А это работа ненужная и часто приводящая к не совсем правильным результатам работа.

Например, в примере выше может оказаться, что портфель заказов изначально был спроектирован неудачно, поскольку он может содержать не более десяти заказов. Это маловато, поэтому в какой-то момент мы решаем увеличить емкость до 32. (Если бы мы задумались о безопасности и корректности, то воспользовались бы стандартным классом `vector`). Но теперь нам предстоит просмотреть все исходные файлы, в которых используется класс `Portfolio`, найти все вхождения литерала 10 и понять, какие из них относятся к числу заказов.

На самом деле, все может обстоять еще хуже. В больших проектах, существующих на протяжении длительного времени, иногда информация о том, что максимальное число заказов равно 10, просачивается наружу и используется в коде, который вообще не включает заголовочного файла `Portfolio`:

```
for( int i = 0; i < 10; ++i )
// ...
```

Означает ли число 10 в этом фрагменте максимальное число заказов? Или длину идентификатора? Или что-то совсем из другой оперы?

Случайное сочетание числовых литералов в одном контексте может давать самые отвратительные примеры кодирования:

```
if( Portfolio *p = getPortfolio() )
for( int i = 0; i < 10; ++i )
p->contracts_[i] = 0, p->id_[i] = '\0';
```

Теперь сопровождающему надо как-то разорвать связь между инициализацией различных компонентов `Portfolio`, которые и не должны были находиться в одном месте, если бы не случайное совпадение значений двух разных по своей сути величин. Нет никакого оправдания для создания такого рода проблем, когда избежать их можно было бы совсем простым способом:

```
class Portfolio {
// . . .
enum { maxContracts = 10, idlen = 10 };
Contract *contracts_[maxContracts];
char id_[idlen];
};
```

Перечисления не занимают памяти в исполняемом коде, их применение не вызывает никаких накладных расходов, зато мы получаем осмысленные имена обозначаемых ими понятий в нужной области видимости.

Не столь очевидный недостаток магических чисел – неопределенность их типа, в результате чего для хранения числа может потребоваться разный объем памяти. Например, тип литерала 40000 зависит от платформы. Оно может быть представлено как типом `int`, так и типом `long`. Если мы не хотим создавать себе проблемы (например, из-за неоднозначности перегрузки) при переносе програм-

мы на новую платформу, лучше точно сказать, что мы имеем в виду, а не полагаться на компилятор:

```
const long patienceLimit = 40000;
```

И еще одна неприятность состоит в том, что литералы не имеют адреса. Проявляется она нечасто, но иногда бывает полезно указать на константу или связать с ней ссылку:

```
const long *p1 = &40000; // ошибка!  
const long *p2 = &patienceLimit; // Правильно.  
const long &r1 = 40000; // допустимо, однако см. совет 44  
const long &r2 = patienceLimit; // Правильно.
```

Достоинств у магических чисел нет, а недостатков масса. Пользуйтесь пересчетами или инициализированными константами.

Совет 3. Глобальные переменные

Редко для объявления глобальной переменной находят оправдания. Глобальные переменные затрудняют повторное использование и сопровождение кода. Первое связано с тем, что любой код, ссылающийся на глобальную переменную, тем самым зависит от нее и, значит, может использоваться только вместе с этой переменной. Сопровождение же усложняется из-за того, что трудно понять, где именно используется данная глобальная переменная, поскольку доступ к ней имеет любая часть программы.

Глобальные переменные увеличивают число зависимостей между компонентами, поскольку часто служат примитивным механизмом передачи сообщений. Даже если глобальные переменные работают, устранить их из большой программы оказывается практически невозможно. Это **если** они работают. Так как глобальные переменные никак не защищены, всякий, кто недавно приступил к сопровождению программы, может случайно вывести из строя те ее части, которые зависят от глобальных переменных.

Пытаясь обосновать использование глобальных переменных, часто говорят, что они, мол, удобны. Это иллюзорный или эгоистический аргумент, поскольку сопровождение программы обычно продолжается дольше, чем первоначальная разработка. Предположим, что некоторой некой системе требуется доступ к некоторому глобальному «окружению», которое (так сформулировано в требованиях) всегда существует в единственном числе. К несчастью, мы решили остановиться на глобальной переменной:

```
extern Environment * const theEnv;
```

Требования тоже со временем изменяются. Незадолго до поставки продукта заказчику обнаружилось, что одновременно может существовать два окружения. Или три. А, может быть, их число задается во время запуска программы. Или вообще может динамически изменяться. Как водится, исправление нужно внести в последнюю минуту. В большом проекте, где применяются строго регламентированные процедуры контроля версий, для изменения каждого файла может потребоваться немало времени, даже если модификация минимальна и очевидна. На

это могут уйти недели или месяцы. А откажись мы от использования глобальной переменной, все заняло бы пять минут:

```
Environment *theEnv();
```

Достаточно инкапсулировать доступ в функцию, и мы сможем реализовать обобщение за счет перегрузки или инициализации аргументов по умолчанию, не внося существенных изменений в исходный текст:

```
Environment *theEnv( EnvCode whichEnv = OFFICIAL );
```

Еще одна, не столь очевидная проблема, касающаяся глобальных переменных, – это необходимость статической инициализации во время выполнения. Если начальное значение статической переменной нельзя вычислить на этапе компиляции, то приходится это делать на этапе выполнения, и тогда последствия могут быть катастрофическими (см. «Совет 55»).

```
extern Environment * const theEnv = new OfficialEnv;
```

Если доступ к глобальной информации контролирует функция или класс, то задание начального значения можно отложить до момента, когда это будет безопасно:

➤➤ gotcha03/environment.h

```
class Environment {
public:
    static Environment &instance();
    virtual void opl() = 0;
    // . . .
protected:
    Environment();
    virtual ~Environment();
private:
    static Environment *instance_;
    // . . .
};
```

➤➤ gotcha03/environment.cpp

```
// . . .
Environment *Environment::instance_ = 0;

Environment &Environment::instance() {
    if( !instance_ )
        instance_ = new OfficialEnv;
    return *instance_;
}
extern Environment * const theEnv = new OfficialEnv;
```

В данном случае мы применили простую реализацию паттерна Singleton (Одиночка) для выполнения отложенной «инициализации» (строго говоря, это присваивание) статического указателя на окружение и потому можем быть уверены, что никогда не возникнет более одного объекта `Environment`. Обратите внимание, что в классе `Environment` нет открытого конструктора, поэтому для получения статического указателя пользователь вынужден воспользоваться функцией-членом `instance`. Поэтому создание объекта `Environment` откладывается до момента первого обращения.

```
Environment::instance().opl();
```

Важнее, однако, тот факт, что контролируемый доступ позволяет гибко адаптировать паттерн Singleton к изменяющимся требованиям, не затрагивая других частей исходного текста. Позже, если мы сделаем программу многопоточной или разрешим существование нескольких экземпляров окружения, достаточно будет модифицировать реализацию Singleton точно так же, как мы выше модифицировали функцию-обертку.

Избегайте использования глобальных переменных. Для достижения тех же результатов имеются более безопасные и гибкие механизмы.

Совет 4. Отличайте перегрузку от инициализации аргументов по умолчанию

Перегрузка функций имеет мало общего с инициализацией аргументов по умолчанию. Но эти два языковых средства часто путают, поскольку они могут породить внешне схожие интерфейсы. Тем не менее, внутренняя семантика таких интерфейсов совершенно различна:

➤> gotcha04/c12.h

```
class C1 {
public:
    void f1( int arg = 0 );
    // . . .
};
```

➤> gotcha04/c12.cpp

```
// . . .
C1 a;
a.f1(0);
a.f1();
```

Проектировщик класса C1 решил применить в объявлении функции f1 аргумент с начальным значением по умолчанию. В результате пользователь C1 может при вызове f1 указать аргумент явно или опустить его, подразумевая значение 0. В обоих показанных выше вариантах последовательность вызова будет одной и той же:

➤> gotcha04/c12.h

```
class C2 {
public:
    void f2();
    void f2( int );
    // . . .
};
```

➤> gotcha04/c12.cpp

```
// . . .
C2 a;
a.f2(0);
a.f2();
```

Класс C2 реализован совершенно иначе. У пользователя есть выбор между двумя различными функциями с одним и тем же именем `f2`. Какая из них будет вызвана, зависит от числа переданных аргументов. В предыдущем примере вызывалась одна и та же функция. Здесь же вызываются разные функции, то есть семантика принципиально иная.

Различие между этими двумя интерфейсами станет еще нагляднее, если мы попытаемся взять адреса функций-членов `C1::f1` и `C2::f2`:

➤➤ `gotcha04/c12.cpp`

```
void (C1::*pmf)() = &C1::f1; //ошибка!
void (C2::*pmf)() = &C2::f2;
```

При данной реализации класса C2 указатель `pmf` будет ссылаться на функцию-член `f2` без аргументов. Поскольку переменная `pmf` указывает на функцию-член без аргументов, то компилятор инициализирует его указателем на первую версию `f2`. В случае же класса C1 мы получим ошибку компиляции, поскольку существует лишь одна функция-член с именем `f1`, и она принимает аргумент типа `int`.

Обычно перегрузка применяется для того, чтобы подчеркнуть, что несколько функций имеют схожую семантику, но разные реализации. Инициализация же по умолчанию, как правило, служит лишь для того, чтобы упростить вызов функции. Перегрузка и аргументы по умолчанию – это разные языковые средства, у них различные цели и разное поведение. Отличайте одно от другого. (См. также «советы Совет 73» и «Совет 74»).

Совет 5. О неправильной интерпретации ссылок

Со ссылками связаны две распространенные ошибки. Во-первых, их часто путают с указателями. Во-вторых, не пользуются ими тогда, когда это имеет прямой смысл. Во многих случаях использование указателей в C++ – это пережиток C, и употребление ссылки было бы уместнее.

Ссылка – это не указатель. Ссылка – эт, ао другое имя того, чем она инициализируется. По существу, к ссылке можно применить только одну операцию – инициализацию. Затем она становится просто еще одним способом обратиться к сущности, которой инициализирована. (См. однако «Совет 44»). У ссылки нет адреса, иногда под нее даже память не отводится.

```
int a = 12;
int &ra = a;
int *ip = &ra; // ip ссылается на a
a = 42; // ra == 42
```

По этой причине нельзя объявлять ссылку на ссылку, указатель на ссылку или массив ссылок. (Хотя комитет по стандартизации C++ рассматривает возможность разрешить ссылки на ссылки, по крайней мере, в некоторых контекстах.)

```
int &&ri = ra; // ошибка!
int &*pri; // ошибка!
int &ar[3]; // ошибка!
```

К ссылкам неприменимы квалификаторы `const` и `volatile`, так как псевдонимы не могут быть ни `const`, ни `volatile`, хотя ссылка может ссылаться на сущность, объявленную как `const` или `volatile`. Попытка объявить ссылку `const` или `volatile` приводит к ошибке компиляции:

```
int &const cri = a; // компилятор выдаст ошибку . . .
const int &rci = a; // Правильно
```

Странно, но C++ не запрещает применять квалификаторы `const` и `volatile` к имени типа, являющегося ссылкой. Ошибка в этом случае не выдается, но квалификатор игнорируется.

```
typedef int *PI;
typedef int &RI;
const PI p = 0; // константный указатель
const RI r = a; // просто ссылка!
```

Не существует нулевых ссылок, как и ссылок на `void`:

```
C *p = 0; // нулевой указатель
C &rC = *p; // неопределенное поведение
extern void &rv; // ошибка!
```

Ссылка – это всего лишь псевдоним, а псевдоним должен на что-то ссылаться.

Заметим однако, что ссылка не обязательно должна относиться к имени простой переменной. Иногда бывает удобно связать ссылку с `lvalue` (см. «Совет 6»), являющемуся результатом вычисления более сложного выражения:

```
int &el = array[n-6][m-2];
el = el*n-3;
string &name = p->info[n].name;
if( name == "Joe" )
    process( name );
```

Возврат ссылки из функции позволяет выполнить присваивание результату вызова. Канонический пример – функция взятия индекса в абстрактном массиве:

➤➤ [gotcha05/array.h](#)

```
template <typename T, int n>
class Array {
public:
    T &operator [] (int i)
        { return a_[i]; }
    const T &operator [] (int i) const
        { return a_[i]; }
    // . . .
private:
    T a_[n];
};
```

Возврат ссылки делает возможным естественным синтаксис присваивания элементу массива:

```
Array<int,12> ia;
ia[3] = ia[0];
```

Ссылки можно использовать и для возврата дополнительных значений из функции:

```
Name *lookup( const string &id, Failure &reason );
```

```
// . . .
string ident;
// . . .
Failure reasonForFailure;
if( Name *n = lookup( ident, reasonForFailure ) ) {
    // Поиск завершился успешно . . .
}
else {
    // Ошибка поиска. Выяснить причину . . .
}
}
```

Результат приведения объекта к ссылочному типу принципиально отличается от приведения к тому же типу, но без ссылки:

```
char *cp = reinterpret_cast<char *>(a);
reinterpret_cast<char *>(a) = cp;
```

В первом случае целое преобразуется в указатель. (Мы предпочли оператор `reinterpret_cast` приведению в старом стиле – `(char *)a`. См. «Совет 40».) В результате мы получили копию того же числа, интерпретируемую как указатель.

Смысл второго предложения совершенно иной. В результате приведения к ссылочному типу сам целочисленный объект интерпретируется как указатель. Он становится `lvalue`, ему можно присваивать значение. (Получим ли мы при этом дамп памяти – другой вопрос. Применение `reinterpret_cast` в общем случае считается «непереносимой конструкцией».) Попытка добиться аналогичного результата путем приведения к не-ссылочному типу закончится неудачей, так как мы получаем `rvalue`, а не `lvalue`:

```
reinterpret_cast<char *>(a) = 0; // ошибка!
```

Ссылка на массив сохраняет информацию о его размерности. При взятии указателя эта информация теряется:

```
int ary[12];
int *pary = ary; // указывает на первый элемент
int (&rary)[12] = ary; // ссылается на весь массив
int ary2[3][4];
int (*pary2)[4] = ary2; // указывает на первый элемент
int (&rary2)[3][4] = ary2; // ссылается на весь массив
```

Этим свойством иногда можно воспользоваться при передаче массива в качестве аргумента функции. (См. «Совет 34».)

Можно также связать ссылку с функцией:

```
int f( double );
int (* const pf)(double) = f; // константный указатель на функцию
int (&rf)(double) = f; // ссылка на функцию
```

На практике различие между константным указателем на функцию и ссылкой на функцию невелико, разве что указатель можно явно разыменовать. Поскольку ссылка является псевдонимом, к ней такая операция неприменима, но можно явно преобразовать ее в указатель на функцию, а затем уже разыменовать:

```
a = pf( 12.3 ); // использовать указатель
a = (*pf)(12.3); // использовать указатель
a = rf( 12.3 ); // использовать ссылку
a = f( 12.3 ); // использовать функцию
```



```
a = (*rf)(12.3); // преобразовать ссылку в указатель и разыменовать  
a = (*f)(12.3); // преобразовать функцию в указатель и разыменовать
```

Отличайте ссылки от указателей.

Совет 6. О неправильной интерпретации const

Идея константности в C++ проста, но может не соответствовать вашему предвзятому мнению о том, что такое константа.

Для начала обратите внимание на различие в объявлении переменной как константы и ее инициализации литералом:

```
int i = 12;  
const int ci = 12;
```

Целочисленный литерал 12 – это не константа. Это литерал. У него нет адреса, и его значение никогда не изменится. Целочисленная переменная `i` – это объект. У нее есть адрес, а ее значение изменяемо. Константная целочисленная переменная `ci` – тоже объект. У нее есть адрес, но (в данном случае) ее значение останется неизменным.

Мы говорим, что `i` и `ci` можно использовать в качестве lvalue, тогда как литерал 12 может быть только rvalue. Эта терминология берет начало от псевдовыражения `L = R`, из которого видно, что lvalue может стоять в левой части операции присваивания, а rvalue – только в правой его части. Однако, к языку C++, равно как и к стандартному C, это определение не вполне применимо, поскольку `ci` – это lvalue, но присваивать ему ничего нельзя, то есть это неизменяемое lvalue. Можете считать, что lvalue – это имена ячеек, которые могут содержать значения, тогда как rvalue – просто значения, с которыми не связан никакой адрес.

```
int *ip1 = &12; // ошибка!  
12 = 13; // ошибка!  
const int *ip2 = &ci; // правильно  
ci = 13; // ошибка!
```

Лучше всего считать, что слово `const` в объявлении `ip2` выше означает ограничение на то, какие операции можно производить над `ci` через указатель `ip2`, а не на то, что вообще можно делать с `ci`. Рассмотрим объявление указателя на `const`:

```
const int *ip3 = &i;  
i = 10; // правильно  
*ip3 = 10; // ошибка!
```

Здесь мы имеем указатель на целочисленную константу, которая ссылается на целое, не являющееся константой. Применение `const` в этом случае просто налагает ограничение на возможные способы использования `ip3`. Никто не гарантирует, что переменная `i` не может изменяться, говорится лишь, что ее нельзя изменить через указатель `ip3`. Еще более удивительной является комбинация квалификаторов `const` и `volatile`:

```
extern const volatile time_t clock;
```

Наличие квалификатора `const` означает, что нам не разрешено модифицировать значение переменной `clock`, а квалификатор `volatile` говорит, что значение `clock`, тем не менее, может (и будет) изменяться.

Совет 7. Не забывайте о тонкостях базового языка

Большинство программистов на C++ убеждены, что уж они-то прекрасно знают ту часть языка, которую можно назвать «базовой», то есть унаследованную от C. Однако даже самые опытные из них иногда пребывают в неведении относительно некоторых «темных» деталей операторов и предложений базового C/C++.

Вряд ли кто-нибудь назовет логические операторы «темной деталью», но начинающие программисты на C++ пользуются ими недостаточно уверенно. Разве у вас не вызывает у вас раздражения такой код:

```
bool r = false;
if( a < b )
    r = true;
```

Вместо такого:

```
bool r = a<b;
```

Надо ли считать до восьми, увидев следующие строки:

➤➤ gotcha07/bool.cpp

```
int ctr = 0;
for( int i = 0; i < 8; ++i )
    if( options & 1<<(8+i) )
        if( ctr++ ) {
            cerr << "Выбрано слишком много опций";
            break;
        }
```

Или лучше написать так:

➤➤ gotcha07/bool.cpp

```
typedef unsigned short Bits;
inline Bits repeated( Bits b, Bits m )
    { return b & m & (b & m)-1; }
// . . .
if( repeated( options, 0xFFF00 ) )
    cerr << "Выбрано слишком много опций";
```

Разве что-то случилось с булевой логикой?

Аналогично, многие программисты не знают о том, что результат вычисления условного оператора – это lvalue (см. «Совет 6»), если оба потенциальных результата являются lvalue. Из-за этого пробела в знаниях им приходится писать такой код:

```
// вариант 1
if( a < b )
    a = val();
else if( b < c )
    b = val();
else
    c = val();

// вариант 2
a<b ? (a = val()) : b<c ? (b = val()) : (c = val());
```

Альтернатива с применением `lvalue`, очевидно, короче:

```
// вариант #3
(a<b?a:b<c?b:c) = val();
```

Хотя такое эзотерическое знание может показаться не столь полезным, как уверенное владение булевой логикой, но во многих контекстах C++ допускаются только выражения (например, в списках инициализации членов в конструкторе, в выражениях `throw` и так далее).

Обратите еще внимание на то, что величина `val` в вариантах 1 и 2 встречается несколько раз, а в варианте 3 – всего один. Если `val` – функция, то большого значения это не имеет. Однако, если `val` – макрос препроцессора, то многократные расширения могут произвести нежелательные побочные эффекты (см. «Совет 26»). В таких контекстах применение эффективного условного оператора вместо предложения `if` может оказаться существенным. Вообще говоря, я не рекомендую применять такую конструкцию повсеместно, но знать о ее существовании все же полезно. Она может пригодиться опытному программисту на C++ в тех редких случаях, когда без нее не обойтись, или она по какой-то причине она предпочтительнее прочих. В язык C++ она включена не без причины.

Как это ни странно, но многие плохо понимают семантику встроенного оператора взятия индекса. Все мы знаем, что этот оператор можно применять к именам массивов и к указателям:

```
int ary[12];
int *p = &ary[5];
p[2] = 7;
```

Встроенный оператор взятия индекса – это просто сокращенная запись для некоторых арифметических операций с указателями и разыменования. Выражение `p[2]` в точности эквивалентно `*(p+2)`. Многие программисты на C++ с опытом работы на C знают также, что разрешено использовать отрицательные значения индексов, так что выражение `p[-2]` корректно и эквивалентно `*(p-2)` или, если хотите, `*(p+-2)`. Однако, похоже, не всем известно, что сложение коммутативно, поэтому большинство программистов на C++ удивляются, увидев, что можно индексировать целое указателем:

```
(-2)[p] = 6;
```

Но это же простое преобразование: `p[-2]` эквивалентно `*(p+-2)`, а это, в свою очередь, эквивалентно `*(-2+p)`, что есть в точности `(-2)[p]` (скобки нужны потому, что приоритет оператора `[]` выше, чем у унарного минуса).

Какое применение может найти это тривиальное наблюдение? Для начала отметим, что коммутативность оператора взятия индекса имеет место только для его встроенного использования применительно к указателям. Иными словами, видя выражение типа `6[p]`, мы можем сразу сказать, что это встроенный оператор, а не перегруженная версия `operator []` (хотя `p` не обязательно указатель или массив). Кроме того, это отличная тема для разговоров на вечеринке. Впрочем, прежде чем применять подобный синтаксис в промышленном коде, познакомьтесь с «Советом 11».

Для большинства программистов на C++ предложение `switch` относится к числу базовых. Многие даже не осознают, насколько оно базовое. Абстрактный синтаксис предложения `switch` очень прост:

```
switch( expression ) statement
```

А вот следствия из такого простого синтаксиса бывают поразительными.

Обычно за выражением `expression` в `switch` следует блок. Внутри него имеется несколько меток `case`, которые, по существу, представляют собой не что иное, как вычисляемый `goto` внутри блока. Первая тонкость, с которой сталкиваются начинающие программисты на С и C++, — это «проваливание». Это значит, что, в отличие от многих других современных языков программирования, после того как исполнение внутри `switch` дошло до нужной метки, работа считается выполненной. Куда направится программа дальше, целиком зависит от программиста:

```
switch( e ) {
default:
theDefault:
    cout << "default" << endl;
    // проваливаемся . . .
case 'a':
case 0:
    cout << "group 1" << endl;
    break;
case max-15:
case Select<(MAX>12),A,B>::Result::value:
    cout << "group 2" << endl;
    goto theDefault;
}
```

Когда «проваливание» применяется осознанно (а не по ошибке, как бывает чаще,) принято оставлять комментарий, сообщающий тем, кто будет сопровождать программу, что автор именно это имел в виду. В противном случае сопровождающий не преминет вставить ненужный `break`.

Отметим, что метки `case` должны быть целочисленными константными выражениями. Иными словами, компилятор должен иметь возможность вычислить их значения на этапе компиляции. Но, как показывает приведенный выше несколько надуманный пример, в том, как определить константное выражение, вам предоставлена большая свобода. Выражение `case` может принадлежать интегральному типу или быть объектом, преобразуемым в интегральный тип. Например, оно могло бы быть именем объекта класса, в котором объявлен оператор преобразования в интегральный тип.

Обратите внимание, что абстрактный синтаксис предложения `switch` допускает даже менее структурированные конструкции, чем показано выше. В частности, метки `case` могут находиться в любом месте `switch` и не обязательно на одном и том же уровне:

```
switch( expr )
default:
    if( cond1 ) {
        case 1: stmt1;
        case 2: stmt2;
    }
}
```

```
else {  
    if( cond2 )  
        case 3:stmt2;  
    else  
        case 0: ;  
}
```

Такая конструкция может показаться бессмысленной (и так оно и есть), но и подобные экзотические аспекты базового языка иногда бывают полезны. Например, описанное свойство предложения switch используется для реализации эффективного обхода сложной внутренней структуры данных в одном компиляторе C++:

➤➤ [gotcha07/iter.cpp](#)

```
bool Postorder::next() {  
    switch( pc )  
    case START:  
        while( true )  
            if( !lchild() ) {  
                pc = LEAF;  
                return true;  
            }  
        case LEAF:  
            while( true )  
                if( sibling() )  
                    break;  
            else  
                if( parent() ) {  
                    pc = INNER;  
                    return true;  
                }  
        case INNER: ;  
            }  
            else {  
                pc = DONE;  
            }  
        case DONE: return false;  
    }  
}
```

В этом коде мы воспользовались непривычными особенностями скромного предложения switch в целях реализации семантики сопрограммы для операции next, применяемой при обходе дерева.

Я получал резко негативные, иногда даже оскорбительные отзывы на использование всех приведенных выше конструкций. Согласен, что не стоит взваливать такой код на хрупкие плечи начинающего программиста сопровождения, но, будучи хорошо инкапсулированы и должным образом документированы, подобные изыски могут найти себе место в оптимизированном или узкоспециализированном коде. Знакомство с экзотическими возможностями базового языка тоже бывает полезно.

Совет 8. Отличайте доступность от видимости

В языке C++ не реализовано сокрытие данных; реализованы только различные уровни доступа. Закрытые и защищенные члены класса не являются невиди-

мыми, они лишь недоступны. Как и в случае со многими другими видимыми, но недоступными объектами (например, менеджерами), это может стать источником проблем.

Самая очевидная проблема состоит в том, что приходится перекомпилировать весь код, в котором используется некоторый класс, даже если изменились только «невидимые» аспекты реализации. Рассмотрим простой класс, в который был добавлен новый член данных:

```
class C {
public:
    C( int val ) : a_( val ),
                 b_( a_ ) // добавлено
    {
        int get_a() const { return a_; }
        int get_b() const { return b_; } // добавлено
private:
    int b_; // добавлено
    int a_;
};
```

Здесь изменилось несколько аспектов класса, часть из них видима, а часть — нет.

К видимым относится изменение размера класса, поскольку был добавлен новый член данных. Это отразится на всем коде, в котором встречаются объекты этого класса, производится разыменование или арифметические операции над указателями на объекты класса или еще каким-то образом упоминаются размер класса либо имена его членов. Отметим также, что добавление нового члена данных изменило смещение члена `a_` от начала класса, следовательно, все существующие ссылки на член `a_` и указатели на него стали недействительными. Кроме того, некорректным стало поведение списка инициализации членов в конструкторе, поскольку `b_` инициализируется неопределенным значением (см. «Совет 52»).

Основные невидимые изменения касаются семантики неявного конструктора копирования и оператора присваивания, которые генерируются для класса `C` компилятором. По умолчанию, они определены как встраиваемые функции и, следовательно, вставляются в любой код, где инициализируются или копируются объекты `C` (см. «Совет 49»).

В результате описанной модификации класса `C` (не будем обращать внимание на вышеупомянутую ошибку) приходится перекомпилировать почти весь код, где `C` используется. В больших проектах это может занять немало времени. Если класс `C` определен в заголовочном файле, то перекомпиляции подлежит весь код, который включает этот файл (или файлы, включающие его). В какой-то мере исправить ситуацию можно, воспользовавшись «опережающими» (то есть неполными) объявлениями класса `C` в контекстах, где полная информация о нем не нужна:

```
class C;
```

Наличие такого неполного объявления все же позволяет объявлять указатели и ссылки на `C` при условии, что мы не выполняем никаких операций, для которых

нужно знать размер `C` или его членов. В частности, это относится к случаю, когда объект `C` является подобъектом базового класса (см. «Совет 39»).

Этот подход может оказаться эффективным, но чтобы избежать сложностей при сопровождении, неполное объявление класса должно браться из того же источника, что и определение класса. Иначе говоря, автор сложного компонента, который предполагается использовать подобным образом, должен предоставить заголовочный файл с опережающими объявлениями.

Например, если полное определение класса `C` находится в файле `c.h`, то можно предоставить еще и файл `cfwd.h`, в котором находятся только неполные объявления классов. В тех случаях, когда полное определение `C` не нужно, можно будет включить `cfwd.h` вместо `c.h`. Предоставлять файл с опережающими объявлениями необходимо потому, что в будущем определение `C` может измениться и стать несовместимым с прежним опережающим объявлением. Например, то, что раньше было классом `C`, может превратиться в `typedef`:

```
template <typename T>
class Cbase {
    // . . .
};
typedef Cbase<int> C;
```

Ясно, что автор заголовочного файла `c.h` стремился экранировать пользователей класса `C` от изменений в нем, но теперь любой код, в котором встречается неполное объявление `C`, перестанет компилироваться:

```
#include "c.h"
// . . .
class C; // ошибка! C - это имя typedef'a, а не класса
```

Наличие файла `cfwd.h` позволило бы избежать таких проблем. Этот подход применен в реализации части `iostream` стандартной библиотеки, где заголовочный файл `iosfwd` соответствует файлу `iostream`.

Чаше необходимость перекомпилировать код, в котором используется класс `C`, затрудняет наложение «заплат» (обычно исправлений ошибок) на установленные программы. Возможно, самый эффективный способ отделить интерфейс класса от его реализации и, следовательно, добиться истинного сокрытия данных заключается в применении паттерна **Bridge** (Мост).

Паттерн `Bridge` подразумевает разделение класса на две части: интерфейс и реализацию:

```
>> gotcha08/cbridge.h
```

```
class C {
public:
    C( int val );
    ~C();
    int get_a() const;
    int get_b() const;
private:
    Cimpl *impl_;
};
```

```
>> gotcha08/cbridge.cpp
```

```
class Cimpl {
```

```

public:
    Cimpl( int val ) : a_( val ), b_( a_ ) {}
    ~Cimpl() {}
    int get_a() const { return a_; }
    int get_b() const { return b_; }
private:
    int a_;
    int b_;
};

C::C( int val )
    : impl_( new Cimpl( val ) ) {}
C::~~C()
    { delete impl_; }
int C::get_a() const
    { return impl_->get_a(); }
int C::get_b() const
    { return impl_->get_b(); }

```

Интерфейсная часть содержит исходный интерфейс класса C, тогда как его реализация перенесена в отдельный файл, скрытый от пользователей. В новой версии класса C есть только указатель на реализацию, а все остальное, в том числе и функции-члены, клиентскому коду не видны. Любые изменения в реализации C, не затрагивающие некоторые не затрагивают интерфейс класса, теперь не выйдут за пределы одного-единственного файла реализации.

С использованием паттерна **Bridge**, очевидно, связаны некоторые накладные расходы во время выполнения, поскольку теперь для представления C требуются два объекта, а не один, и, кроме того, все функции-члены вызываются косвенно и не могут встраиваться. Однако, возможность многократно сократить время компиляции и обновлять клиентский код без перекомпиляции часто перевешивает эти издержки. Такая методика успешно применяется уже много лет и получила немало забавных названий, в частности, «идиома `rimpl`» и «улыбка Чеширского кота».

Недоступные члены также могут влиять на семантику производных классов и базовых классов, если к последним производится доступ через интерфейс производного класса. Например, рассмотрим следующий базовый и производный класс:

```

class B {
public:
    void g();
private:
    virtual void f(); // добавлена
};
class D : public B {
public:
    void f();
private:
    double g; // добавлен
};

```


Добавление закрытой виртуальной функции в базовый класс В сделало виртуальной функцию $f()$ в производном классе, которая раньше виртуальной не была. Добавление закрытого члена данных в класс D скрыло функцию с тем же именем, унаследованную от В. Наследование часто называют повторным использованием в виде «прозрачного ящика», поскольку изменения существенно затрагивают и базовый, и производный классы.

Сгладить остроту этих проблем помогает, в частности, следование простому соглашению об именовании, в соответствии с которым имена следует разделять по функциональности. Как правило, лучше всего применять различные соглашения для имен типов, закрытых членов данных и всех остальных имен. В этой книге мы будем писать имена типов с заглавной буквы, в конец имен данных-членов (все они закрыты!) добавлять пробел, а остальные имена (за немногими исключениями) начинать с маленькой-со строчной буквы. Если бы мы следовали этому соглашению в примере выше, то функция-член $g()$ базового класса не оказалась бы скрытой в классе D. Всеми силами противьтесь искушению ввести сложное соглашение об именовании, поскольку придерживаться его, скорее всего, не будете.

И еще. Никогда не пытайтесь кодировать тип переменной в ее имени. Например, выбор для целочисленного индекса имени `iIndex` сильно затрудняет понимание и сопровождение кода. Во-первых, имя должно описывать абстрактную семантику программной сущности, а не то, как она реализована (абстракция данных может относиться даже к предопределенным типам). Во-вторых, тип переменной нередко изменяется, и тогда возникает несоответствие с именем. Имя переменной становится в этом случае источником дезинформации о ее типе.

Другие подходы обсуждаются в советах 70, 73, 74 и 77.

Совет 9. О неграмотности

Когда несколько лет назад широкий мир вторгся в пределы уютного замкнутого мирка C++, он принес с собой ряд достойных порицания оборотов и приемов кодирования. В этом разделе я попробую научить вас правильному идиоматичному употреблению лексики C++, применяемой при описании его поведения.

Лексика

В таблице 1.1 перечислены наиболее распространенные ошибки словоупотребления и показано, как надо говорить правильно.

Таблица 1.1. Типичные ошибки словоупотребления и правильные варианты

Неправильно	Правильно
Чисто виртуальный базовый класс	Абстрактный класс
Метод	Функция-член
Виртуальный метод	???
Разрушен (Destructed)	Уничтожен (Destroyed)
Оператор приведения	Оператор преобразования

Нет такого понятия как «чисто виртуальный» базовый класс. Есть чисто виртуальные функции, а класс, который содержит такую функцию и не переопределяет ее, называется абстрактным.

В C++ нет методов. Это в языках (Java и Smalltalk) методы. Говоря об объектно-ориентированном проектировании и желая выказать особую претенциозность, вы можете употребить термины «сообщение» и «метод», но при переходе к обсуждению реализации вашего проекта на C++ все же пользуйтесь терминами «вызов функции» и «функция-член».

Некоторые, в основном вполне квалифицированные специалисты по C++, говорят «destructed» (разрушен) в противоположность «constructed» (сконструирован). Это просто безграмотный английский! Правильно говорить «destroyed» (уничтожен).

В C++ действительно есть операторы приведения (cast operator), или преобразования типов (type conversion operator). Их четыре: `static_cast`, `dynamic_cast`, `const_cast` и `reinterpret_cast`. Но термин «оператор приведения» (cast operator) часто неправильно употребляют применительно к оператору преобразования, являющемуся функцией-членом класса, который описывает, как объект класса можно преобразовать в другой тип:

```
class C {
    operator int *()const; // оператор преобразования
    // . . .
};
```

Разумеется, оператор преобразования разрешается вызывать и явно (с помощью оператора приведения), если только вы знаете, что есть что.

См. также обсуждение различий между константным указателем и указателем на `const` в «Совете 31».

Нулевые указатели

Было время, когда программа на C++ могла «рухнуть», если для представления нулевого указателя использовался символ препроцессора `NULL`.

```
void doIt( char * );
void doIt( void * );
C *cp = NULL;
```

Проблема в том, что `NULL` определяется по-разному на разных платформах:

```
#define NULL ((char *)0)
#define NULL ((void *)0)
#define NULL 0
```

Из-за этого переносимость программы на C++ оказывалась под угрозой:

```
doIt( NULL ); // зависит от платформы или неоднозначно
C *cp = NULL; // ошибка?
```

На самом деле, непосредственно представить нулевой указатель в C++ нельзя, но гарантируется, что числовой литерал `0` можно преобразовать в нулевой указатель любого типа. Именно так программисты на C+ обычно обеспечивают переносимость и корректность своего кода. В стандарте сказано, что определения

типа (void *)0 недопустимы, так что технических проблем с использованием NULL в любом случае не возникнет (правда, сам факт, что это символ препроцессора, заставляет глядеть на него относиться к нему с подозрением). Тем не менее, настоящие программисты на C++ по-прежнему представляют нулевой указатель литералом 0. Применяя любой другой способ, вы рискуете показаться безнадежно старомодным.

Акронимы

У программистов на C++ есть какая-то болезненная приверженность к акронимам, хотя и не в такой тяжелой форме, как у менеджеров. В следующий раз, когда ваш коллега заявит, что «RVO неприменима к POD, поэтому лучше определить копирующий стог», обратитесь к таблице 1.2.

Таблица 1.2. Значение некоторых распространенных акронимов

Акроним	Значение
POD (Plain old data)	Добрые старые данные, структура в смысле C
POF (Plain old function)	Добрая старая функция, функция в смысле C
RVO (Return value optimization)	Оптимизация возвращаемого значения
NRV (Named RVO)	Оптимизация именованного возвращаемого значения
ctor (Constructor)	Конструктор
dtor (Destructor)	Деструктор
ODR (One definition rule)	Правило одного определения

Совет 10. Не игнорируйте идиомы

Давно замечено, что лучшие писатели иногда не обращают внимания на правила риторики. Но в таких случаях читатель обнаруживает в предложении какие-то достоинства, с лихвой компенсирующие нарушение правил. Если бы автор не был уверен в том, что делает, то, вероятно, постарался бы следовать писаным правилам. (Strunk and White «The Elements of Style»¹).

Эта цитата из классического руководства по англоязычной прозе часто приводится и в книгах, посвященных стилю программирования. Я полностью согласен и с самой цитатой, и со скрывающимся за ней подтекстом – ограничивать свои проявления. И все же я нахожу ее не вполне удовлетворительной, поскольку, будучи вырвана из контекста, она ничего не говорит о том, почему в общем случае лучше придерживаться правил риторики и в чем эти правила заключаются. Мне всегда больше нравилось принадлежащее Уайту сравнение со звериной тропой, чем олимпийская непогрешимость Странка:

¹ Автором этой цитаты на самом деле является Вильям Странк, поскольку она появилась в первом издании книги, еще до того как она была извлечена из забвения Уайтом в 1959 году.

Живой язык можно сравнить со звериной тропой: ее протоптали сами звери, и они идут по ней или сходят с нее, следуя своим желаниям или потребностям. От ежедневного употребления тропа меняет направление. Зверь не обязан любой ценой идти по узкой тропе, которую сам же и проложил с учетом особенностей местности, но часто поступает именно так, потому что это удобно, а сойдя с нее, он не будет знать, где находится и куда направляется (Е. В. White, из статьи в журнале «The New Yorker»).

Языки программирования не так сложны, как естественные, и нашей цели – написания понятного кода – достичь не так сложно, как писать ясную, отточенную прозу. Но все же язык программирования, подобный C++, сложен настолько, что для эффективного программирования на нем приходится прибегать к целому ряду стандартных оборотов и идиом. Язык C++ не содержит непререкаемых предписаний, то есть допускает заметную гибкость, но идиоматическое использование его средств – это способ эффективно и понятно донести идею проекта до аудитории. Незнание идиом или сознательное пренебрежение ими ведет к путанице и неправильному применению.

Многие рекомендации из этой книги предполагают свободное владение и употребление идиом в процессе проектирования и кодирования на C++. Часть упоминаемых «скользких мест» – это просто результат отхода от той или иной идиомы. А чтобы решить проблему, часто не требуется ничего большего, чем следование подходящей идиоме. И тому есть причина: набор идиом кодирования и проектирования на C++ создан и постоянно совершенствуется в результате усилий всего сообщества программистов на этом языке. Подходы, оказавшиеся нерепродуцируемыми или утратившие актуальность, перестают применяться и отбрасываются. Выживают лишь те идиомы, которые эволюционируют вместе с окружением. Знание – и употребление на практике – идиом проектирования и кодирования – один из лучших способов создания ясных, эффективных и удобных для сопровождения программ и проектов на C++.

Любой компетентный профессиональный программист должен всегда заботиться о том, чтобы и код, и проект существовали в контексте какой-то идиомы. Зная о существовании идиом, мы уже можем решать, остаться ли на узкой тропе или осознанно сойти с нее, чтобы достичь стоящей перед нами цели. Но придерживаться идиом выгодно, а игнорировать их рискованно.

Я бы не хотел, чтобы у вас создалось неправильное впечатление, будто идиомы C++ – это своего рода смирительная рубашка, управляющая всеми аспектами процесса проектирования. Совсем нет. При правильном использовании идиомы могут облегчить как само проектирование, так и документирование проекта, никак не ограничивая при этом свободу творчества проектировщика. Но бывает, что даже самая разумная и широко распространенная идиома не укладывается в контекст проекта, и тогда проектировщик вынужден уйти с протоптанных дорожек.

Одна из самых известных и полезных идиом C++ связана с операцией копирования. Каждый абстрактный тип данных в C++ должен принять какое-то решение по поводу оператора присваивания и конструктора копирования. Программист

должен либо разрешить компилятору сгенерировать их, либо написать самостоятельно, либо запретить их использование вовсе (см. «Совет 49»).

Если программист сам кодирует эти операции, то мы точно знаем, что они должны были быть написаны. Но «стандартный» способ их написания с годами менялся. И в этом одно из преимуществ идиом по сравнению с незыблемыми правилами: идиома эволюционирует, чтобы соответствовать контексту, в котором применяется.

```
class X {
public:
    X( const X & );
    X &operator =( const X & );
    // . . .
};
```

Хотя язык C++ допускает большую свободу в определении операций копирования, но почти всегда имеет смысл объявлять их, как показано выше: обе операции принимают ссылку на константу, а оператор присваивания не виртуальный и возвращает ссылку на не-const. Ясно, что ни одна из этих операций не должна изменять свой аргумент. Это просто бессмысленно.

```
X a;
X b( a ); // a не изменится
a = b; // b не изменится
```

Но бывают и исключения. К стандартному шаблону `auto_ptr` предъявляются необычные требования. Это дескриптор ресурса, который должен освободить выделенную из кучи память, когда она больше не нужна.

```
void f() {
    auto_ptr<Blob> blob( new Blob );
    // . . .
    // автоматическое удаление выделенной для Blob памяти
}
```

Прекрасно, но что если допустить к этому коду неопытного студента?

```
void g( auto_ptr<Blob> arg ) {
    // . . .
    // автоматическое удаление выделенной для Blob памяти
}

void f() {
    auto_ptr<Blob> blob( new Blob );
    g( blob );
    // повторное удаление выделенной для Blob памяти!!!
}
```

Можно было бы запретить для `auto_ptr` операции копирования, но это сильно ограничило бы его применимость и сделало бы невозможными целый ряд полезных идиом, связанных с `auto_ptr`. Другой вариант – включить в `auto_ptr` счетчик ссылок, но тогда возросли бы накладные расходы. В стандарте при реализации `auto_ptr` было принято решение сознательно отойти от идиомы операции копирования:

```
template <class T>
class auto_ptr {
```

```

public:
    auto_ptr( auto_ptr & );
    auto_ptr &operator =( auto_ptr & );
    // . . .
private:
    T *object_;
};

```

(В стандартном `auto_ptr` реализован также ряд шаблонных функций-членов, соответствующих нестандартным операциям копирования, но к ним применимы аналогичные рассуждения. См. также «Совет 88».) Здесь правая часть каждой операция неконстантна! Когда `auto_ptr` инициализируется или присваивается другому `auto_ptr`, источник инициализации или присваивания отказывается от владения выделенным из кучи объектом, на который указывал, для чего сбрасывает в нуль внутренний указатель на объект.

Как часто бывает при отходе от идиоматического употребления, первоначально вокруг правильного способа использования `auto_ptr` возникла путаница. Однако такой отход от известной идиомы позволил разработать целый ряд новых полезных идиом, касающихся владения объектами, а применение объектов `auto_ptr` как «источников» и «стоков» данных стало плодотворной идеей в проектировании. Иными словами, осознанный отказ от известной и успешно применяемой идиомы привел к появлению семейства новых идиом.

Совет 11. Не мудрствуйте лукаво

Языки C++ и C, похоже, привлекают излишне большое число желающих присоединиться. (Приходилось вам когда-нибудь слышать о конкурсе «Озадачивающий Eiffel»?) Такие программисты, вероятно, думают, что кратчайшим путем между двумя точками является большая окружность в евклидовом пространстве на сфере.

О чем я говорю? В кругах, близких к C++ (евклидовых или нет) хорошо известно, что форматирование кода нужно только для удобства чтения человеком; с точки зрения семантики программы важна лишь последовательность лексем. Последнее, впрочем, весьма важно; так, следующие две строки означают совершенно разные вещи (см. однако «Совет 87»).

```

a+++++b; // ошибка!
a+++ ++b; // правильно.

```

как и такие две строки (см. «Совет 17»):

```

ptr->*m; // правильно.
ptr-> *m; // ошибка!

```

А потому многие программисты на C++ делают вывод, что форматирование не существенно для семантики программы, лишь бы поток символов правильно разбивался на лексемы. Как бы ни объявлять переменную, в одной или в нескольких строках, результат будет одинаковым. (Некоторые отладчики и другие инструментальные средства оперируют номерами строк, а не более точным индикатором места в программе. Это вынуждает программистов применять неудобное

или неестественное разбиение на строки, чтобы получить внятные сообщения об ошибках, поставить точку прерывания и так далее. Но это не проблема C++ как такового, а вопрос для проектировщиков сред разработки.)

```
long curLine = __LINE__ ; // номер текущей строки
long        curLine
            =          __LINE__
            ; // то же самое объявление
```

Но эти программисты ошибаются. Взгляните на простой механизм выбора типа на этапе компиляции, применяемый в технологии метапрограммирования шаблонов:

➤> gotchall/select.h

```
template <bool cond, typename A, typename B>
struct Select {
    typedef A Result;
};
template <typename A, typename B>
struct Select<false, A, B> {
    typedef B Result;
};
```

Во время конкретизации шаблона `Select` на этапе компиляции вычисляется условие, а затем в зависимости от булевого результата конкретизируется та или иная версия шаблона. Это, по сути дела, предложение `if` этапа компиляции, которое говорит: «Если условие истинно, то вложенным типом `Result` будет `A`, иначе `B`».

➤> gotchall/lineno.cpp

```
Select< sizeof(int)==sizeof(long), int, long >::Result temp = 0;
```

В этом предложении переменная `temp` будет иметь тип `int`, если типы `int` и `long` занимают одно и то же число байтов в памяти. В противном случае `temp` объявляется как `long`.

Обратимся вновь к объявлению `curLine`. Зачем тратьте лишнее место для размещения `long`, если это необязательно? Давайте-ка прибегнем к неоправданно сложному трюку:

➤> gotchall/lineno.cpp

```
const char CM = CHAR_MAX;
const Select< __LINE__ <=CM, char, long>::Result curLine = __LINE__ ;
```

Работает (и даже правильно), но строка стала слишком длинной, и программист, который стал сопровождать программу после вас, ее немного переформатировал:

➤> gotchall/lineno.cpp

```
const Select< __LINE__ <=CM, char, long>::Result
curLine = __LINE__ ;
```

И тем самым внес ошибку. Можете ее найти?

Что если это объявление встретится в строке с номером `CHAR_MAX` (а значение этой константы может быть совсем небольшим, обычно 127)? Тогда тип

`curLine` окажется `char`, и инициализирована она будет максимальным значением, допустимым для этого типа. Стоит нам поместить инициализатор на следующую строку, как мы попытаемся инициализировать значение типа `char` величиной на единицу большей максимального значения для этого типа. В результате номер строки окажется отрицательным числом (вероятно, 128). Умно, ничего не скажешь.

Излишнее «умничанье» – это типичная проблема программистов на C++. Помните, что почти всегда лучше придерживаться соглашений, выражать свои мысли ясно, пусть даже это приведет к чуть менее эффективной программе, чем проявлять ненужное хитроумие, которое приводит к запутанным и не пригодным для сопровождения программам.

Совет 12. Не ведите себя как дети

Мы, программисты, всегда готовы раздавать советы, но с трудом соглашаемся следовать им сами. Мы посылаем проклятия в адрес глобальных переменных, плохих имен, магических чисел и тому подобного, но зачастую вставляем их в собственные программы. Этот феномен многие годы не давал мне покоя, пока в одном журнале я не прочел статью, описывающую аналогичное поведение у подростков. По-видимому, молодым людям свойственно критиковать рискованное поведение других, но в силу какой-то «странной фантазии» считать, что, если они будут сами вести себя так, то ничего страшного не случится. Программисты, как класс, тоже страдают от задержки эмоционального развития.

Мне приходилось работать над проектами, в которых некоторые программисты не только отказывались следовать стандартам кодирования, но угрожали уволиться, если их будут заставлять делать отступ из четырех, а не из двух пробелов. Встречал я и ситуации, когда одна группа отказывалась приходить на собрания, если там присутствовали представители другой группы. Я видел, как программисты намеренно не документируют и всячески затуманивают код, чтобы больше никто не мог его сопровождать. Я наблюдал, как талантливые в общем-то программисты отказывались принимать советы слишком старых/слишком молодых/слишком «правильных»/увлекающихся пирсингом коллег, и из-за этого разражались случались катастрофы.

Каким бы ни был уровень эмоционального развития профессионального программиста, у него, как у любого взрослого человека или, по крайней мере, профессионала, есть обязанности. (Познакомьтесь с позицией Ассоциации вычислительных машин по этому поводу, изложенной в «Кодексе этического и профессионального поведения ACM» (ACM Code of Ethics and Professional Conduct), а также с «Кодексом этического и профессионального поведения разработчика программного обеспечения» (Software Engineering Code of Ethics and Professional Practice)).

Во-первых, выбранная нами профессия обязывает выполнять работу качественно в соответствии с наивысшими стандартами.

Во-вторых, у нас есть обязанности перед обществом и планетой, на которой мы обитаем. Наша профессия – это в равной мере и наука, и ее практическое при-

менение. Если наш труд не помогает сделать мир, в котором мы живем, лучше, то это пустая трата нашего таланта, времени и, в конечном итоге, жизни.

В-третьих, наш долг перед сообществом – делиться своим опытом, если того требует политика правительства. Наше общество становится все более технологическим, а самые важные решения обычно принимают люди, разбирающиеся в юриспруденции или политике, но технически безграмотные. Например, в одном штате когда-то действовал закон, по которому число π приравнялось к 3. Это смешной пример (правда, колесный транспорт немного трясло, пока закон не отменили), чего не скажешь о многих других решениях, принятых недостаточно информированными людьми. Наш долг – предоставить участникам политических дебатов рациональное техническое и количественное обоснование.

В-четвертых, наш долг перед коллегами – работать совместно. Это означает, что необходимо следовать местным стандартам кодирования и проектирования (если они недостаточно хороши, надо попытаться их изменить, но не игнорировать), писать код, удобный для сопровождения, прислушиваться к чужому мнению и делиться собственным опытом.

Это не следует расценивать, как призыв быть «рубахой-парнем» или безоговорочно принимать корпоративную униформу или видение мира. Некоторые мои знакомые, профессионалы, работать с которыми было очень приятно, странно одевались, имели нетрадиционные политические взгляды и необычные привычки. Но все они с уважением относились ко мне лично и к моим идеям (не стеснясь ругать меня, когда я того заслуживал, и указывать на мои ошибки), и, приступая к работе со мной, честно стремились достичь тех целей, которые мы перед собой поставили.

В-пятых, наш долг перед другими – делиться своими знаниями и опытом.

В-шестых, у каждого из нас есть долг перед самим собой. Ваша работа и ваши мысли должны вас удовлетворять, вы не должны раскаиваться в выборе профессии. Если вы занимаетесь своим делом с удовольствием, если оно является неотъемлемой частью вашей жизни, то перечисленные выше обязанности покажутся не обузой, а радостью.

Глава 2. Синтаксис

Язык C++ обладает сложной лексической и синтаксической структурой. Частично сложность унаследована от C, а частично необходима для поддержки определенных языковых средств.

В этой главе мы рассмотрим ряд скользких мест, имеющих отношение к синтаксису. Некоторые из них правильнее было бы назвать опечатками, которые, тем не менее, компилируются и приводят к неожиданным результатам во время исполнения. Другие иллюстрируют проблему слабой связи между синтаксической структурой фрагмента кода и его поведением при выполнении. Третьи обусловлены гибкостью синтаксиса, из-за чего два программиста могут прийти к разным выводам относительно семантики одного и того же кода.

Совет 13. Не путайте массивы с инициализаторами

Мы ведь можем распределить массив из 12 целых чисел в куче, не так ли? Конечно:

```
int *ip = new int(12);
```

Пока все хорошо. А теперь воспользуемся этим массивом. Когда он перестанет быть нужным, почистим за собой:

```
for( int i = 0; i < 12; ++i )  
    ip[i] = i;  
delete [] ip;
```

Обратите внимание на пустые квадратные скобки. Их наличие говорит компилятору, что `ip` указывает на массив, а не на одиночное целое число. А так ли это?

На самом деле, `ip` указывает именно на одиночное целое, инициализированное значением 12. Мы сделали типичную опечатку, спутав круглые и квадратные скобки. И доступ внутри цикла (ко всем элементам, кроме имеющего индекс 0), и удаление некорректны. Но компилятор вряд ли сможет обнаружить эту ошибку. Поскольку указатель может указывать как на единственный объект, так и на массив объектов, то синтаксически как обращение к элементам по индексу внутри цикла, так и удаление массива правильны. И разочарование постигнет нас только во время исполнения.

А, возможно, и тогда все обойдется. Обращаться к памяти за границей объекта нельзя (хотя язык разрешает указывать на адрес, следующий непосредственно за последним элементом объекта). Удалять скаляр как массив тоже неправильно. Но из того, что вы делаете нечто незаконное, еще не следует, что вас обязательно поймают (вспомните про Уолл Стрит). На некоторых платформах этот код может

выполниться нормально, а на других приведет к аварийному завершению программы. Или программа будет вести себя нестабильно в зависимости от того, как пользуется кучей конкретный поток или процесс. Корректно память выделяется так:

```
int *ip = new int[12];
```

Но еще лучше не выделять память вовсе, а воспользоваться стандартной библиотекой:

```
std::vector<int> iv( 12 );
for( int i = 0; i < iv.size(); ++i )
    iv[i] = i;
// явного удаления нет ...
```

Стандартный шаблон `vector` почти так же эффективен, как встроенный массив, но он безопаснее, с ним проще работать, и он самодокументирован. В общем случае отдавайте предпочтение классу `vector`, а не низкоуровневым массивам. Кстати, такая же синтаксическая проблема может возникнуть и при простом объявлении, только ее обычно легче обнаружить:

```
int a[12]; // массив из 12 целых
int b(12); // целое, инициализированное значением 12
```

Совет 14. Неопределенный порядок вычислений

Происхождение C++ от C нигде не проявляется с такой очевидностью, как при рассмотрении порядка вычислений. Это ловушка, в которую легко попадают непосвященные. В этом разделе мы покажем несколько проявлений одной и той же проблемы: и C, и C++ предоставляют компилятору большую свободу при определении того, как вычислять выражение. Эта гибкость позволяет оптимизировать код, но программист должен быть внимателен и избегать необоснованных предположений о порядке вычислений.

Порядок вычисления аргументов функции

```
int i = 12;
int &ri = i;
int f( int, int );
// . . .
int result1 = f( i, i *= 2 ); // не переносимо
```

Порядок вычисления аргументов функции не определен. Поэтому функции `f` могут быть переданы аргументы 12 и 24 или 24 и 24. Осмотрительный программист не станет модифицировать аргумент, который появляется более одного раза в списке аргументов, но и это не спасает:

```
int result2 = f( i, ri *= 2 ); // не переносимо
int result3 = f( p(), q() ); // как повезет
```

В первом случае `ri` – это псевдоним `i`, поэтому о значении `result2` можно сказать не больше, чем о значении `result1`. Во втором случае мы предположили,

что порядок, в котором вызываются функции `p` и `q`, не имеет значения. Но даже если это сейчас и так, то в будущем может измениться, а такое ограничение на реализацию `p` и `q` нигде не документировано.

Лучше избегать побочных эффектов при вычислении аргументов функции:

```
result1 = f( i, i*2 );
result2 = f( i, ri*2 );
int a = p();
result3 = f( a, q() );
```

Порядок вычисления подвыражений

Порядок вычисления подвыражений также не фиксирован:

```
a = p() + q();
```

Функция `p` может вызываться раньше `q`, а может и позже. Правила предествования и ассоциативности операторов не влияют на порядок вычислений:

```
a = p() + q() * r();
```

Функции `p`, `q` и `r` могут вычисляться в любом из шести возможных порядков. Тот факт, что у оператора умножения более высокий приоритет, гарантирует лишь, что результаты вызова `q` и `r` будут перемножены до того, как выполнится сложение с результатом вызова `q`. Аналогично, левая ассоциативность оператора «плюс» не гарантирует определенного порядка вызова функций `p`, `q` и `r` в примере ниже; можно лишь утверждать, что результаты вызовов будут складываться слева направо:

```
a = p() + q() + r();
```

Скобки тоже не помогут:

```
a = (p() + q()) * r();
```

Сначала действительно будет выполнено сложение результатов вызова `p` и `q`, но будет ли функция `r` вызвана до или после этого, мы не знаем. Единственный надежный способ зафиксировать порядок вычисления подвыражений, явно завести временные переменные:

```
a = p();
int b = q();
a = (a + b) * r();
```

Как часто возникает такая проблема? Достаточно часто, чтобы испортить один-другой выходной каждый год. На рис. 2.1 показан фрагмент абстрактного синтаксического дерева, используемого в реализации арифметического калькулятора.

Следующая реализация не переносима.

»» gotcha14/e.cpp

```
int Plus::eval() const
{ return l_>eval() + r_>eval(); }
int Assign::eval() const
{ return id->set( e_>eval() ); }
```

Проблема в реализации функции `Plus::eval`: порядок вычисления левого и правого поддерева не определен. Имеет ли это значение для сложения? Ведь оно

должно быть коммутативным, не так ли? Но рассмотрим вычисление следующего выражения:

```
(a = 12) + a
```

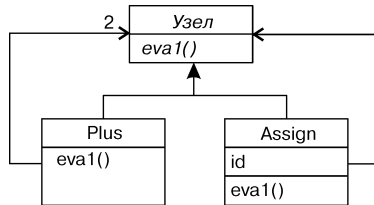


Рис. 2.1. Иерархия узлов абстрактного синтаксического дерева для простого калькулятора (фрагмент). У узла Plus есть левое и правое поддереву; у оператора присваивания только одно поддереву, представляющее его правую часть

В зависимости от порядка вычисления левого и правого поддеревьев в функции `Plus::eval` значением этого выражения может быть как 24, так и предыдущее значение `a` плюс 12. Если от нашего калькулятора требуется, чтобы присваивание выполнялось раньше сложения, то в реализации `Plus::eval` необходимо задействовать временную переменную:

➤ `gotcha14/e.cpp`

```
int Plus::eval() const {
    int lft = l_>eval();
    return lft + r_>eval();
}
```

Порядок вычисления размещающего `new`

Надо признать, что эта проблема возникает не часто. Синтаксис размещающего оператора `new` допускает передачу аргументов не только инициализатору (обычно конструктору) размещаемого объекта, но и функции `operator new`, которая производит выделение памяти.

```
Thing *pThing =
    new (getHeap(), getConstraint()) Thing( initval() );
```

Первый список аргументов передается функции `operator new`, которая может принимать аргументы, а второй – конструктору класса `Thing`. Общее замечание относительно порядка вычисления аргументов функции применимо к каждому из этих двух списков. Мы не знаем, что будет вычислено раньше: аргументы для `operator new` или для конструктора `Thing`, хотя точно знаем, что функция `operator new` будет вызвана раньше конструктора (поскольку нам нужно сначала получить память для объекта, который мы собираемся инициализировать).

Операторы, которые фиксируют порядок вычислений

На поведение некоторых операторов можно положиться в большей степени, если они употребляются самостоятельно. Так, оператор «запятая» фиксирует порядок вычисления своих подвыражений:

```
result = expr1, expr2;
```

В этом выражении сначала вычисляется `expr1`, затем `expr2`, а затем результат присваивается переменной `result`. Этим можно воспользоваться для написания необычно выглядящего кода:

```
return f(), g(), h();
```

Правда, автору этого кода надо бы пройти курс социализации. Применяйте более традиционный стиль кодирования, если хотите облегчить жизнь тем, кто будет сопровождать вашу программу:

```
f();
g();
return h();
```

Едиственное общеупотребительное применение оператор «запятая» находит в части оператора `for`, где происходит увеличение переменной цикла, если таких переменных более одной:

```
for( int i = 0, j = MAX; i <= j; ++i, -j ) // ...
```

Отметим, что первая запятая в объявлении `i` и `j` – это не оператор «запятая», а часть объявления двух переменных типа `int`.

«Закорачивающие» логические операторы `&&` и `||` более полезны, они позволяют записывать сложные выражения кратко и идиоматично:

```
if( f() && g() ) // ...
if( p() || q() || r() ) // ...
```

Первое выражение означает: «Вызвать `f`. Если результат ложный, то и все условие ложно. Если же результат истинный, вызвать `g`. Значением всего условия будет результат вычисления `g`.» Второе условие читается так: «Вызвать `p`, `q` и `r` в этом порядке. Если все три вызова возвращают ложь, то все условие ложно; в противном случае, условие истинно.» Учитывая, насколько компактнее эти операторы позволяют сделать код, неудивительно, что программисты на `C` и `C++` применяют их с такой готовностью.

Тернарный условный оператор («?:») также фиксирует порядок вычисления своих аргументов:

```
expr1 ? expr2 : expr3
```

Сначала вычисляется первое выражение – условие; в зависимости от результата вычисляется второе или третье выражение. Результатом всего оператора является результат вычисления последнего выражения.

```
a = f()+g() ? p() : q();
```

В данном случае у нас имеются некоторые гарантии относительно порядка вычислений. Мы знаем, что `f` и `g` будут вызваны раньше `p` и `q` (хотя и не знаем, в каком точно порядке) и что из двух функций `p` и `q` будет вызвана ровно одна.

Для удобства чтения было бы неплохо добавить скобки, хотя, строго говоря, они излишни.

```
a = (f()+g()) ? p() : q();
```

В противном случае сопровождающий по незнанию или в спешке может предположить (неверно), что сложение выполняется позже условного оператора:

```
a = f()+g() ? p() : q();
```

Некорректная перегрузка операторов

Однако, как бы ни были полезны встроенные версии этих операторов, перегружать их не стоит. В C++ перегрузка операторов – это «синтаксическая приправа», не более чем приятный для взгляда синтаксис вызова функции. Например, можно было бы перегрузить оператор `&&`, так чтобы он принимал два аргумента типа `Thing`:

```
bool operator &&( const Thing &, const Thing & );
```

Если использовать этот оператор в инфиксной нотации, то сопровождающий может предположить, что он допускает «закорачивание», как и встроенный оператор. Однако это не так:

```
Thing &tf1();  
Thing &tf2();  
// ...  
if( tf1() && tf2() ) // ...
```

Семантически этот код эквивалентен вызову функции:

```
if( operator &&( tf1(), tf2() ) ) // ...
```

А выше мы видели, что в этом случае вызываются обе функции `tf1` и `tf2`, причем порядок вызова не определен. То же самое относится к операторам `operator ||` и `operator ,`. К счастью, перегружать оператор `?:` запрещается.

Совет 15. Помните о предшествовании

В этом разделе мы не будем говорить, кто должен сидеть рядом с Послом на званом обеде: Графиня или Баронесса (у этой задачи все равно нет решения). Нет, мы собираемся обсудить, как наличие нескольких уровней приоритетности операторов в выражениях на языке C++ может стать причиной докучных проблем.

Приоритеты и ассоциативность

Обычно от наличия разных уровней приоритетности операторов язык только выигрывает, поскольку это позволяет проще записывать сложные выражения, не отвлекаясь на скобки. (Заметим, однако, что все равно полезно расставлять скобки в длинных и не очевидных с первого взгляда выражениях. Но в простых, не вызывающих сомнения случаях ненужные скобки лучше опускать.)

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

В этом выражении оператор `*` имеет наивысший приоритет, поэтому связанное им выражение вычисляется первым. У оператора присваивания самый низкий приоритет, поэтому эта операция выполняется последней.

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

Здесь мы знаем, что операции сложения выполняются раньше присваиваний, поскольку приоритет оператора `+` выше, чем у оператора `=`, но какое сложение и какое присваивание будет выполняться первым? Для ответа на этот вопрос надо принять во внимание ассоциативность операторов. В C++ есть левоассоциативные и правоассоциативные операторы. Левоассоциативный оператор, каковым является `+`, теснее связан с аргументом в левой части. Поэтому сначала складываются `a` и `b`, а потом к результату прибавляется `c`.

Оператор присваивания правоассоциативен, так что сначала результат вычисления `a+b+c` присваивается `a`, после чего значение `a` присваивается `b`. В некоторых языках встречаются неассоциативные операторы; если, к примеру, оператор `@` неассоциативен, то выражение `a@b@c` недопустимо. Но в нашем «родном» C++ неассоциативных операторов нет.

Проблемы, связанные с приоритетом операторов

Библиотека `iostream` спроектирована так, чтобы свести употребление скобок к минимуму:

```
cout << "a+b = " << a+b << endl;
```

Приоритет оператора `+` выше, чем у оператора сдвига влево, поэтому компилятор разбирает это выражение так, как нам нужно: сначала вычисляется `a+b`, а потом результат отправляется в `cout`.

```
cout << a ? f() : g();
```

В этом примере использование единственного в C++ тернарного оператора становится источником неприятностей, но не потому, что `?:` тернарный; дело в том, что его приоритет ниже, чем у оператора `<<`. Таким образом, это выражение означает, что нужно «сдвинуть» `a` в `cout`, а результат вычисления этого выражения использовать в качестве условия в операторе `?:`. Трагизм ситуации заключается в том, что это совершенно корректный код! (Потоковый объект, в частности, `cout` обладает оператором `operator void *`, который неявно преобразует свой операнд к типу `void *`, а он может быть преобразован в `false` или `true` в зависимости от того, является указатель нулевым или нет.) Поэтому здесь приходится воспользоваться скобками:

```
cout << (a ? f() : g());
```

Если вы хотите, чтобы вас считали совершенно нормальным человеком, можете сделать еще один шаг:

```
if ( a )
    cout << f();
else
    cout << g();
```

Здесь, конечно, нет той изысканности, что в предыдущем примере, зато код легко читать и сопровождать.

Не многие программисты на C++ попадают в ловушки, связанные с приоритетом оператора указания на объекты, поскольку хорошо известно, что у операторов `>` и `.` очень высокий приоритет. Следовательно, выражение `a = ++ptr->mem` означает: «инкрементировать член `mem` объекта, на который указывает `ptr`». Если бы мы хотели сначала инкрементировать указатель, то надо было бы написать так: `a = (++ptr)->mem` или так: `++ptr; a = ptr->mem;` или уж на худой конец так: `a = (++ptr, ptr->mem).`

Указатели на члены классов — это совсем другая история. Их следует размысливать в контексте объекта класса (см. «Совет 46»). Для этого существуют два специальных оператора: `->*` для перехода от указателя на член к указателю на объект класса и `*` — для перехода от указателя на член к самому объекту класса.

Указатели на функции-члены часто становятся причиной головной боли, но серьезных синтаксических проблем не вызывают:

```
class C {
    // ...
    void f( int );
    int mem;
};
void (C::*pfmem)(int) = &C::f;
int C::*pdmem = &C::mem;
C *cp = new C;
// ...
cp->*pfmem(12); // ошибка!
```

Ошибку компиляции мы получаем потому, что оператор вызова функции имеет более высокий приоритет, чем оператор `->*`, но мы не можем вызвать функцию-член по указателю, не размыслив ее предварительно. Скобки здесь необходимы:

```
(cp->*pfmem)(12);
```

С указателями на данные-члены дело обстоит сложнее. Рассмотрим следующее выражение:

```
a = ++cp->*pdmem
```

Переменная `cp` — это тот же указатель на объект класса, что и выше, а `pdmem` — не имя члена, а указатель на член. В данном случае, поскольку у оператора `->*` приоритет выше, чем у `++`, то `cp` сначала инкрементируется, а потом уже размысливается указатель на член. Если только `cp` не указывает внутрь массива объектов, то такое размыменование, скорее всего, кончится бедой.

Мало найдется программистов на C++, которые хорошо понимают, что такое указатели на члены класса. Чтобы упростить в дальнейшем сопровождение вашей программы, старайтесь быть проще, когда работаете с ними:

```
++cp;
a = cp->*pdmem;
```

Проблемы, связанные с ассоциативностью

В C++ большинство операторов левоассоциативны, а неассоциативных операторов нет вовсе. Но это не мешает в остальном вполне образованным программистам пытаться использовать операторы примерно так:

```
int a = 3, b = 2, c = 1;
// ...
if( a > b > c ) // корректно, но, скорее всего, ошибочно
```

Этот код абсолютно корректен, но, скорее всего, ошибочен. Значение выражения $3 > 2 > 1$, конечно же, `false`. Оператор «больше», как и большинство операторов в C++, левоассоциативен, поэтому сначала вычисляется подвыражение $3 > 2$, которое равно `true`. Теперь у нас осталось выражение `true > 1`. Значение `true` преобразуется в целое число, после чего вычисляется выражение $1 > 1$, которое равно `false`.

В данном случае программист, вероятно, хотел записать условие `a > b && b > c`. Если в силу каких-то неочевидных причин программист действительно хотел выразить то, что написано в тексте, то лучше было бы сделать это так: `a > b ? 1 > c : 0 > c` или, быть может, так: `(c - (a > b)) < 0`. То и другое выглядят достаточно странно, чтобы привлечь более пристальное внимание сопровождающего. И это тот случай, когда комментарий не повредил бы. (См. «Совет 1».)

Совет 16. Подводные камни в предложении for

В языке C++ есть несколько мест, где разрешено объявлять переменную с ограниченной областью видимости, отличающейся от блока. Например, можно объявить переменную в условии предложения `if`. Она будет видна в предложениях, управляемых этим условием, причем в обеих ветвях.

```
if( char *theName = lookup( name ) ) {
    // сделать что-то с name ...
}
// здесь theName покидает область видимости
```

Раньше такую переменную пришлось бы объявлять вне предложения `if`, по этому она оставалась бы видимой и дальше, после того как мы закончили с ней работать. Чем не источник неприятностей?

```
char *theName = lookup( name );
if( theName ) {
    // сделать что-то с name ...
}
// theName здесь все еще доступна ...
```

Вообще говоря, всегда лучше ограничить область видимости переменной той частью программы, где она используется. В ходе сопровождения по причинам, которые я не в состоянии понять, «висящие» переменные часто повторно используются для каких-то невообразимых целей. Поэтому они оказывают на документирование и сопровождение программы, мягко говоря, «негативный» эффект. (См. также «Совет 48».)

```
theName = new char[ ISBN_LEN ]; // нужен буфер для ISBN
```

То же самое относится к предложению `for`: переменную цикла можно объявить в первой части заголовка:

```
for( int i = 0; i < bufSize; ++i ) {
```

```
    if( !buffer[i] )
        break;
}
if( i == bufSize ) // раньше было корректно, теперь нет; i вне области
                  // видимости
// ...
```

Много лет такая запись в С++ была допустима, но потом область действия переменной цикла изменилась. Раньше она простиралась от точки объявления (непосредственно перед инициализатором, см. «Совет 21») до конца блока, объемлющего предложение `for`. После изменения семантики область видимости стала ограничена сами предложением `for`. Большая часть программистов полагает, что такое изменение разумно по многим причинам: оно более согласованно с другими частями языка, упрощает оптимизацию циклов и так далее. Но вместе с тем приходится исправлять старые программы, которые зависели от этой семантики.

А иногда этот процесс бывает болезненным. Оцените возможность скрытого изменения смысла следующего кода:

```
int i = 0;
void f() {
    for( int i = 0; i <bufSize; i++ ) {
        if( !buffer[i] )
            break;
    }
    if( i == bufSize ) // i из области видимости файла!
        // . . .
}
```

К счастью, подобные ошибки редки, и хороший компилятор предупредит вас о возникновении такой ситуации. Отнеситесь к предупреждению серьезно (и не отключайте режим выдачи предупреждений) и старайтесь избегать сокрытия имен из внешних областей видимости именами, объявленными во внутренних областях. И перестаньте пользоваться глобальными переменными. (См. «Совет 3».)

Как ни странно, самое зловерное воздействие изменение области действия в предложении `for` оказало на способ записи таких предложений программистами:

```
int i;
for( i = 0; i < bufSize; ++i ) {
    if( isprint( buffer[i] ) )
        message( buffer[i] );
    // . . .
    if( some_condition )
        continue;
    // . . .
}
```

Это код на С, а не на С++. Да, у него есть достоинство: его смысл одинаков и для старой, и для новой семантики; но давайте посмотрим, что мы теряем. Во-первых, переменная цикла остается видимой после выхода из предложения `for`. Во-вторых, переменная `i` не инициализирована. Ни то, ни другое не смертельно, когда код только разрабатывается. Но в ходе сопровождения менее опытный

программист может попытаться использовать неинициализированную переменную `i` как перед входом в предложение `for`, так и после выхода из него, то есть в момент, когда, по мысли автора, `i` уже не должно существовать.

Другая проблема в том, что из-за этого изменения некоторые программисты вообще перестают пользоваться предложением `for`:

```
int i = 0;
while( i < bufSize ) {
    if( isprint( buffer[i] ) )
        message( buffer[i] );
    // . . .
    if( some_condition )
        continue; // беда!
    // . . .
    ++i;
}
```

Дело в том, что предложения `while` и `for` не эквивалентны. Например, если в теле цикла есть предложение `continue`, то в семантике программы происходит трудноуловимое изменение. В данном случае она войдет в бесконечный цикл, а это обычно ясно свидетельствует об ошибке. Но не всегда можно рассчитывать на такую удачу.

Если вам повезло работать исключительно на платформах, где поддерживается новая семантика предложения `for`, то лучше всего будет адаптировать свои программы, как только выйдет обновленная версия компилятора.

К несчастью, чаще бывает так, что код должен компилироваться на разных платформах, где семантика предложения `for` несовместима. В таком случае кажется логичным написать все предложения `for` так, чтобы они имели один и тот же смысл при любом способе трансляции.

```
int i;
for( i = 0; i < bufSize; ++i ) {
    if( isprint( buffer[i] ) )
        message( buffer[i] );
    // . . .
}
```

Но я все же рекомендую переписать все предложения `for` в соответствии с новой семантикой. Чтобы обойти проблемы с областью видимости переменной цикла, можно погрузить предложение `for` в блок:

```
{for( int i = 0; i < bufSize; ++i ) {
    if( isprint( buffer[i] ) )
        message( buffer[i] );
    // . . .
}}
```

Выглядит достаточно уродливо, чтобы на это обратили внимание и удалили лишний блок, как только появится такая возможность. Кроме того, эта конструкция явно свидетельствует о намерении автора написать предложение `for` с учетом новой семантики, а не оставлять эту модификацию тому, кто будет сопровождать программу.

Совет 17. Принцип «максимального куска»

Что вы делаете, столкнувшись с подобным выражением?

```
++++p->*mp
```

А не доводилось ли вам сталкиваться с «сержантским оператором»?¹

```
template <typename T>
class R {
    // ...
    friend ostream &operator <<< // сержантский оператор?
        T >( ostream &, const R & );
};
```

Не задавались ли вы вопросом, корректно ли следующее выражение?

```
a+++++b
```

Добро пожаловать в мир «больших кусков». На одной из ранних стадий трансляции программы на С++ работает так называемый «лексический анализатор», задача которого, разбить входной поток на отдельные лексические единицы или лексемы. Встретив последовательность символов типа `->*`, лексический анализатор может выделить три лексемы (`-`, `>` и `*`), две лексемы (`->` и `*`) или одну лексему (`->*`), и все это будет разумно. Чтобы избежать неоднозначности, анализатор всегда выделяет самую длинную из возможных лексем: «максимальный кусок».

Выражение `a+++++b` недопустимо, как и выражение `a+++++b`; нельзя применять операцию постинкремента к `gvalue`, каковым является выражение `a++`. Если вы хотели применить постинкремент к `a`, а затем прибавить результат к тому, что получается после прединкремента `b`, то надо было вставить хотя бы один пробел: `a+++++b`. Если вы питаете хоть малейшее уважение к тем, кто будет читать ваш код, то добавите и еще один, хотя, строго говоря, он не обязателен: `a+++++b`. И никто не станет критиковать вас, если вы включите еще и скобки: `(a++)+(++b)`.

Принцип «максимального куска» решает намного больше проблем, чем порождает, но есть два случая, когда он мешает. Первый – это конкретизация шаблонов аргументами, которые сами являются шаблонами. Например, стандартная библиотека позволяет объявить список (`list`) из векторов (`vector`) строк (`string`):

```
list<vector<string>> lovos; // ошибка!
```

К несчастью, две соседних закрывающих угловых скобки интерпретируются в этом случае как оператор сдвига, и мы получаем синтаксическую ошибку. Необходим пробел:

```
list< vector<string> > lovos;
```

Другая ситуация возникает, когда используется значение по умолчанию для аргумента, являющегося указателем:

```
void process( const char * = 0 ); // ошибка!
```

¹ Три лычки (`<<<<`) нашиваются на погоны сержантов армии США (Прим. перев.)

Здесь компилятор думает, что мы пытаемся использовать оператор присваивания в объявлении формального аргумента. Синтаксическая ошибка. Эта ошибка проходит по категории «по заслугам и награда»; ничего не случилось бы, если бы автор дал формальному аргументу какое-то имя. Оно не только стало бы самой лучшей документацией, но и предотвратило бы ошибку из-за применения принципа «максимального куска»:

```
void process( const char *processId = 0 );
```

Совет 18. О порядке следования спецификаторов в объявлениях

С точки зрения языка порядок следования спецификаторов в объявлениях не имеет значения:

```
int const extern size = 1024; // допустимо, но странно
```

Но без основательных причин не стоит нарушать соглашение, лучше следовать сложившемуся де факто стандарту упорядочения спецификаторов: спецификатор компоновки, квалификатор типа, тип.

```
extern const int size = 1024; // нормально
```

Каков тип переменной `ptr` ниже?

```
int const *ptr = &size;
```

Правильно. Это указатель на константное целое, но вы не поверите, сколько программистов считают, что это объявление константного указателя на целое:

```
int * const ptr2 = &size; // ошибка!
```

Конечно же, это два совсем разных типа, поскольку первый может ссылаться на константное целое, а второй – нет. В разговорной речи многие программисты называют указатели на константные данные «const-указателями». Это неудачная идея, так как правильный смысл (указатель на константные данные) дойдет, как это ни забавно, только до невежд и собьет с толку любого компетентного программиста на C++, который поверит вам на слово (константный указатель на не-константные данные).

Следует признать, что в стандартной библиотеке есть понятие `const_iterator`, обозначающее – и этому нет прощения – итератор, который ссылается на константные элементы; сам итератор константным не является. (Но из того, что у комитета по стандартизации выдался тяжелый день, не следует, что вы должны повторять его ошибку.) Проводите различие между «указателем на const» и «константным указателем». (См. «Совет 31»).

Поскольку технически порядок спецификаторов в объявлениях не важен, то указатель на const можно объявить двумя способами:

```
const int *pci1;
int const *pci2;
```

Некоторые эксперты по C++ рекомендуют вторую форму, поскольку, по их мнению, в сложных объявлениях указателей она легче читается:

```
int const * const *ppl;
```

Размещение квалификатора `const` последним в списке спецификаторов позволяет читать модификаторы указателя в обратном порядке, то есть справа налево: `pp1` – это указатель на константный указатель на `const int`. Традиционное расположение такого простого прочтения не допускает.

```
const int * const *pp2; // то же, что pp1
```

Однако эта запись не намного сложнее предыдущей, а программисты, которые будут читать и сопровождать код, содержащий такие вычурные объявления, вероятно, смогут в них разобраться. Важнее то, что указатели на указатели и другие подобные им объявления встречаются редко, особенно в тех интерфейсах, на которые могут натолкнуться менее опытные программисты. Как правило, они скрыты глубоко в недрах реализации. Гораздо чаще встречаются указатели на константы, поэтому имеет смысл следовать соглашению, чтобы избежать недопонимания:

```
const int *pcil; // правильно: указатель на const
```

Совет 19. Функция или объект?

Когда объект инициализируется конструктором по умолчанию, не следует указывать пустой список инициализации, поскольку компилятор интерпретирует его как объявление функции:

```
String s("Semantics, not Syntax!"); // явный инициализатор
String t; // инициализация по умолчанию
String x(); // объявление функции
```

Эта неоднозначность внутренне присуща языку C++. По существу, при разработке стандарта «подбросили монету» и решили: пусть `x` будет объявлением функции. Отметим, что в выражениях `new` такая неоднозначность не возникает:

```
String *sp1 = new String(); // никакой неоднозначности ...
String *sp2 = new String; // то же, что и выше
```

Вторая форма предпочтительнее, поскольку она более распространена и ортогональна по отношению к объявлению объектов.

Совет 20. Перестановка квалификаторов типа

Не существует такого понятия, как константный или изменяющийся массив, поэтому квалификаторы типы (`const` или `volatile`), указанные для массива, будут автоматически переставлены на правильную позицию в объявлении типа:

```
typedef int A[12];
extern const A ca; // массив из 12 константных целых
typedef int *AP[12][12];
volatile AP vm; // двумерный массив volatile-указателей на int
volatile int *vm2[12][12]; // двумерный массив указателей на volatile int
```

Это разумно, так как массив – это не что иное, как своего рода литеральный указатель на свои элементы. С ним не ассоциировано никакой памяти, которая может быть константной или изменяющейся, поэтому квалификаторы могут применяться только к его элементам. Однако имейте в виду, что компиляторы часто

некорректно реализуют эту операцию в сложных случаях. Например, компилятор решил (ошибочно), что тип переменной `vm` выше совпадает с типом `vm2`.

В отношении объявлений функций все обстоит несколько хитрее. В прошлом наиболее распространенные реализации C++ допускали аналогичную перестановку квалификаторов и для них:

```
typedef int FUN( char * );
typedef const FUN PF; // раньше: функция, возвращающая const int
// теперь: недопустимо
```

Ныне стандарт говорит, что квалификатор типа можно применять в объявлении функции к `typedef`'у «верхнего уровня», и что `typedef` можно использовать только для объявления нестатической функции-члена:

```
typedef int MF() const;
MF nonmemfunc; // ошибка!
class C {
    MF memfunc; // правильно.
};
```

Наверное, лучше избегать такого использования. Современные компиляторы не всегда реализуют его правильно, а читателей-людей оно только запутывает.

Совет 21. Автоинициализация

Каково значение внутренней переменной `var` в следующем коде?

```
int var = 12;
{
    double var = var;
    // ...
```

Не определено. В C++ имя попадает в область действия перед началом раздела инициализатора, поэтому любая ссылка на это имя внутри инициализатора оказывается ссылкой на необъявленную переменную! Мало найдется программистов, которые захотят написать такое странное объявление, но наткнуться на подобную проблему можно в результате копирования кода из другого места:

```
int copy = 12; // какая-то глубоко упрятанная переменная
// ...
int y = (3*x+2*copy+5)/z; // вырезать ...
// ...
void f() {
    // нужна копия начального значения y ...
    int copy = (3*x+2*copy+5)/z; // и вставить сюда!
    // ...
```

Препроцессор может привести к такой же ошибке, как небрежное копирование (см. «Совет 26»).

```
int copy = 12;
#define Expr ((3*x+2*copy+5)/z)
// ...
void g() {
    int copy = Expr; // дежа вю, все повторилось ...
    // ...
```


Другое проявление той же проблемы возникает, когда выбранное соглашение об именовании не различает имена типов и не-типов:

```
struct buf {
    char a, b, c, d;
};
// ...
=== Page 54 ===
void aFunc() {
    char *buf = new char[ sizeof( buf ) ];
    // ...
}
```

Локальное имя `buf` относится (вероятно) к 4-байтовому буферу, достаточно большому, чтобы вместить значение типа `char *`. Эта ошибка могла бы оставаться незамеченной очень долго, особенно, если размер структуры `struct buf` случайно окажется таким же, как размер указателя. Если соглашение об именовании таково, что имена типов нельзя спутать с именами не-типов, то такая проблема никогда не возникнет (см. «Совет 12»).

```
struct Buf {
    char a, b, c, d;
};
// ...
void aFunc() {
    char *buf = new char[ sizeof( Buf ) ]; // правильно
    // ...
}
```

Теперь мы знаем, как избежать канонических проявлений этого «скользкого места»:

```
int var = 12;
{
    double var = var;
    // ...
}
```

А что вы скажете насчет такой вариации на ту же тему?

```
const int val = 12;
{
    enum { val = val };
    // ...
}
```

Каково значение перечисляемой константы `val`? Тоже не определено? Подумайте как следует. Ее значение равно 12, и причина в том, что точка объявления перечисляемой константы `val`, в отличие от объявления переменной, расположена после инициализатора (или, говоря более формально, после определения перечисляемой константы). Значение `val` после знака `=` в `enum` относится к константе в объемлющей области видимости. Продолжая эту тему, мы можем рассмотреть еще более запутанную ситуацию:

```
const int val = val;
{
    enum { val = val };
    // ...
}
```

К счастью, такое определение перечисляемой константы недопустимо. Инициализатор здесь не является целым константным выражением, поскольку ком-

пилятор не может определить значение `val` из объемлющей области видимости на этапе компиляции.

Совет 22. Статические и внешние типы

Нет таких вещей в C++. Однако опытные программисты часто сбивают с толку начинающих объявлениями, подобными показанному ниже. На первый взгляд кажется, что к типу применен спецификатор компоновки (см. «Совет 11»):

```
static class Repository {
    // ...
} repository; // статический
Repository backUp; // не статический
```

Хотя указывать для типа вид компоновки не запрещено, этот спецификатор всегда относится к объекту или функции, но не к самому типу. Лучше выражать свои намерения яснее:

```
class Repository {
    // ...
};
static Repository repository;
Repository backUp;
```

Отметим, что вместо спецификатора компоновки `static` лучше пользоваться безымянным пространством имен:

```
namespace {
    Repository repository;
    Repository backUp;
}
```

Теперь имена `repository` и `backUp` имеют внешнюю компоновку и, стало быть, могут применяться для большего числа целей, чем статические имена (например, при конкретизации шаблонов). Однако, как и статические имена, они недоступны вне текущей единицы трансляции.

Совет 23. Аномалия при поиске операторной функции

Перегруженные операторы – это, в действительности, просто обычные функции, являющиеся или не являющиеся членами класса, при вызове которых можно использовать инфиксный синтаксис. Это не более чем «синтаксическая приправа»:

```
class String {
public:
    String &operator =( const String & );
    friend String operator +( const String &, const String & );
    String operator -();
    operator const char *() const;
    // ...
};
String a, b, c;
```

```
// ...
a = b;
a.operator =( b ); // то же самое
a + b;
operator +( a, b ); // то же самое
a = -b;
a.operator =( b.operator -() ); // то же самое
const char *cp = a;
cp = a.operator const char *(); // то же самое
```

Очевидно, что инфиксная нотация намного понятнее. Обычно мы пользуемся этой нотацией при вызове перегруженных операторов; в конце концов, именно для этого мы операторы и перегружаем.

Но бывают и исключения, когда обычный синтаксис вызова функции выглядит понятнее инфиксного. Стандартный пример – обращение к оператору присваивания, определенному в базовом классе, из реализации оператора присваивания в производном классе:

```
class A {
protected:
    A &operator =( const A & );
    // . . .
};
class B : public A {
public:
    B &operator =( const B & );
    // . . .
};

B &B::operator =( const B &b ) {
    if( &b != this ) {
        A::operator =( b ); // понятнее чем
                           // (*static_cast<A*const>(this))=b
        // присвоить значения локальным членам
    }
    return *this;
}
```

Функциональная форма вызова также применяется вместо инфиксной, когда последняя выглядит настолько странно (хотя и абсолютно корректна), что читатель будет вынужден потратить пару минут на то, чтобы понять ее смысл:

```
value_type *Iter::operator ->() const
{ return &operator *(); } // вместо &*( *this)
```

Существуют, кроме того, неоднозначные ситуации, в которых явное предпочтение нельзя отдать ни тому, ни другому синтаксису:

```
bool operator !=( const Iter &that ) const
{ return !(*this == that); } // или !operator ==(that)
```

Заметим, однако, что порядок поиска имени для инфиксной и функциональной формы различен. И это может приводить к неожиданным результатам:

```
class X {
public:
    X &operator %( const X & ) const;
    void f();
};
```

```

    // . . .
};
X &operator %( const X &, int );
void X::f() {
    X &anX = *this;
    anX % 12; // правильно, не член
    operator %( anX, 12 ); // ошибка!
}

```

Когда используется функциональная форма, для поиска имени функции применяется стандартная процедура. Если речь идет о функции-члене `X::f`, компилятор сначала ищет функцию с именем `operator %` в классе `X`. Если имя найдено, он не будет продолжать поиск других функций с тем же именем в объемлющих областях видимости.

К несчастью, мы передаем бинарному оператору три аргумента. Поскольку у функции-члена `operator %` есть неявный аргумент `this`, то наличие еще и двух явных аргументов говорит компилятору о том, что мы пытаемся превратить бинарный оператор `%` в тернарный. Правильно было бы либо явно сказать, что это не функция-член (`::operator %(anX, 12)`), либо передать ожидаемое число аргументов функции-члену (`operator %(anX)`).

При использовании инфиксной нотации компилятор будет искать функцию-член `operator %` и одноименную функцию, не являющуюся членом, в области видимости левого операнда. В случае выражения `anX % 12` компилятор найдет две функции-кандидата и выберет функцию, не являющуюся членом, как и должно быть.

Совет 24. Тонкости оператора `->`

Встроенный оператор `->` бинарный: левым операндом является указатель, а правым — имя члена класса. Перегруженный же оператор `->` — унарная функция-член!

➤ gotcha24/ptr.h

```

class Ptr {
public:
    Ptr( T *init );
    T *operator ->();
    // . . .
private:
    T *tp_;
};

```

Обращение к перегруженной функции `->` должно вернуть нечто, что можно передать оператору `->` для доступа к члену.

➤ gotcha24/ptr.cpp

```

Ptr p( new T );
p->f(); // p.operator ->()->f()!

```

Можно взглянуть на эту ситуацию так: лексема `->` не «поглощается» перегруженной версией `operator ->`, а остается во входном потоке и рано или поздно будет востребована встроенным оператором `->`. Как правило, при перегрузке опе-

ратора -> добавляется та или иная семантика, делающая из него «интеллектуальный указатель»:

➤➤ gotcha24/ptr.cpp

```
T *Ptr::operator ->() {
    if( today() == TUESDAY )
        abort();
    else
        return tp_;
}
```

Мы уже упоминали, что перегруженный оператор -> должен возвращать «нечто», что позволит получить доступ к члену. Это «нечто» не обязательно быть встроенным указателем, а может оказаться объектом класса, который и сам переопределяет оператор ->:

➤➤ gotcha24/ptr.h

```
class AugPtr {
public:
    AugPtr( T *init ) : p_( init ) {}
    Ptr &operator ->();
    // . . .
private:
    Ptr p_;
};
```

➤➤ gotcha24/ptr.cpp

```
Ptr &AugPtr::operator ->() {
    if( today() == FRIDAY )
        cout << '\a' << flush;
    return p_;
}
```

Это дает возможность распределить обязанности между несколькими интеллектуальными указателями:

➤➤ gotcha24/ptr.cpp

```
AugPtr ap( new T );
ap->f(); // ap.operator ->().operator ->()->f()!
```

Отметим, что последовательность активаций оператора -> всегда определяется по статическому типу объекта, содержащего оператора ->, и цепочка обращений к функции-члену оператора -> обрывается, когда получен встроенный указатель. Например, применение -> к AugPtr всегда даст такую последовательность вызовов: сначала AugPtr::operator ->, затем Ptr::operator ->, затем встроенный ->, применяемый к указателю типа T *. (См. более реалистичный пример использования оператора -> в «Совете 83».)



Глава 3. Препроцессор

Обработка текста программы на C++ препроцессором – это, наверное, самая опасная фаза трансляции. Препроцессор видит только лексемы («слова», из которых состоит исходный текст программы) и не обращает внимания на синтаксические и семантические особенности языка. Можно сказать, что препроцессор не осознает собственной мощи и, как многие сильные, но глупые существа, способен причинить немало вреда.

Основной вывод настоящей главы: применяйте препроцессор там, где нужно много «силы», но мало знаний о C++, и не подпускайте его ни к чему, требующему «тонкого обращения».

Совет 25. Определение литералов с помощью #define

Программисты на C++ не пользуются директивой #define для определения литералов, поскольку в C++ это может стать причиной ошибок и непереносимости. Рассмотрим привычное для C применение #define:

```
#define MAX 1<<16
```

Основной недостаток символов препроцессора в том, что препроцессор расширяет их до того, как их имел возможность увидеть собственно компилятор C++. Препроцессор ничего не знает о правилах видимости, действующих в C++.

```
void f( int );  
void f( long );  
// ...  
f( MAX ); // какая f?
```

К тому моменту, как компилятор приступает к разрешению перегрузки, символ препроцессора MAX оказывается всего лишь целым значением 1<<16. В зависимости от платформы число 1<<16 может иметь тип int или long. Следовательно, на разных платформах могут вызываться разные функции f.

Директива #define не обращает внимания на области видимости. Очень многие средства C++ инкапсулированы в пространствах имен. У такого подхода много достоинств, включая то, что разные средства не вступают в конфликт друг с другом. К сожалению, #define «плюет» на границы пространств имен:

```
namespace Influential {  
#   define MAX 1<<16  
// ...  
}  
namespace Facility {  
const int max = 512;
```

```
// ...
}
// ...
int a[MAX]; // беда!
```

Программист забыл импортировать имя `max`, к тому же неправильно написал вместо него `MAX`. Но препроцессор просто заменил `MAX` на `1<<16`, поэтому код все равно компилируется. «Интересно, почему программа потребляет так много памяти...».

Решить все эти проблемы можно, воспользовавшись инициализированной константой:

```
const int max = 1<<9;
```

Теперь `max` имеет один и тот же тип на любой платформе, и имя `max` подчиняется обычным правилам областей видимости. Отметим, что использование `max` почти так же эффективно, как и применение `#define`, поскольку компилятору разрешено не выделять для этой константы память, а просто подставить ее начальное значение всюду, где она используется в качестве `rvalue`. Однако, поскольку имя `max` – это все же `lvalue` (только неизменяемое, см. «Совет 6»), то у него есть адрес, и мы можем указать на него. Для литерала это было бы невозможно:

```
const int *pmax = &Facility::max;
const int *pMAX = &MAX; // ошибка!
```

С директивой `#define` связана еще одна проблема: подстановки, выполняемые препроцессором, имеют лексический, а не синтаксический характер. В примере выше определение `MAX` с помощью `#define` не вызвало проблем, но посмотрите, как легко они могут появиться:

```
int b[MAX*2];
```

Поскольку мы не заключили выражение в правой части `#define` в скобки, то сейчас мы пытаемся объявить поистине огромный массив целых чисел:

```
int b[ 1<<16*2 ];
```

Да, эта ошибка – всего лишь результат неправильного употребления `#define`, но она вообще не могла бы возникнуть, если бы мы воспользовались инициализированной константой.

Та же проблема существует и для области видимости класса. В примере ниже мы хотим, чтобы значение было доступно внутри класса и больше нигде. Традиционно в C++ для таких целей применяют перечисления:

```
class Name {
// ...
void capitalize();
enum { nameLen = 32 };
char name_[nameLen];
};
```

Перечисляемая константа `nameLen` не занимает памяти и доступна только в области видимости класса, которая, разумеется, включает и все функции-члены:

```
void Name::capitalize() {
for( int i = 0; i < nameLen; ++i )
```

```

if( name_[i] )
    name_[i] = toupper( name_[i] );
else
    break;
}

```

Разрешается также, хотя не все компиляторы еще поддерживают эту возможность, объявлять и инициализировать константные статические данные-члены интегральных типов целочисленным константным выражением внутри тела класса (см. «Совет 59»).

```

class Name {
    // . . .
    static const int nameLen_ = 32;
};
// . . .
const int Name::nameLen_;    // здесь нет инициализатора!

```

Однако может случиться, что компилятор не сумеет оптимизировать код так, чтобы не выделять память под такой статический член данных, поэтому лучше для определения простых целочисленных констант лучше пользоваться старыми добрыми перечислениями.

Совет 26. Определение псевдофункций с помощью #define

В языке С директива #define часто применяется для определения псевдофункций, когда снижение накладных расходов на вызов функции считается важнее безопасности:

```
#define repeated(b, m) (b & m & (b & m)-1)
```

Разумеется, и такое использование препроцессора ведет к обычным для него проблемам. В частности, приведенное выше определение некорректно:

```

typedef unsigned short Bits;
enum { bit01 = 1<<0, bit02 = 1<<1, bit03 = 1<<2, // ...
Bits a = 0;
const Bits mask = bit02 | bit03 | bit06;
// ...
if( repeated( a+bit02, mask ) ) // беда!
// ...

```

Здесь мы допустили типичную ошибку: забыли расставить скобки. Правильное определение не оставляет места никаким случайностям:

```
#define repeated(b, m) ((b) & (m) & ((b) & (m))-1)
```

За исключением побочных эффектов. Стоит слегка изменить обращение к этой псевдофункции, и результат окажется и неправильным, и неоднозначным:

```

if( repeated( a+=bit02, mask ) ) // большая беда!
// ...

```

При вычислении первого аргумента возникает побочный эффект. Если бы repeated была настоящей функцией, то побочный эффект проявлялся бы только один раз, еще перед вызовом функции. Но при имеющемся определении repeated

он проявляется дважды, причем порядок не определен (см. «Совет 14»). Псевдофункции особенно опасны тем, что в тексте программы неотличимы от настоящих функций, хотя имеют совершенно другую семантику. Из-за такого сходства даже опытные программисты на C++ иногда пользуются псевдофункциями неправильно, так как думают, что вызывают обычную функцию.

В C++ почти всегда псевдофункции следует предпочесть встраиваемую функцию, поскольку она-то обладает обычной для функций семантикой, такой же, как у невстраиваемых функций:

```
inline Bits repeated( Bits b, Bits m )
{ return b & m & (b & m)-1; }
```

Макросы, используемые в качестве псевдофункций, подвержены тем же проблемам с областью видимости, как и макросы-константы (см. «Совет 25»).

➤➤ gotcha26/execbump.cpp

```
int kount = 0;
#define execBump( func ) (func(), ++kount)
// . . .
void aFunc() {
    extern void g();
    int kount;
    while( kount++ < 10 )
        execBump( g ); // инкрементировать локальную kount!
}
```

Пользователь псевдофункции `execBump` не знал (будем надеяться), что в ней используется переменная `kount` и непреднамеренно модифицировал значение локальной переменной `kount` вместо глобальной. Лучше бы в этом случае написать настоящую функцию:

➤➤ gotcha26/execbump.cpp

```
int kount = 0;
inline void execBump( void (*func)() )
{ func(); ++kount; }
```

Во встраиваемой функции идентификатор `kount` во время компиляции связывается с глобальной переменной. Это имя не будет ссылаться на какую-либо другую переменную `kount` в момент вызова функции. (Но это не оправдывает самого факта использования глобальной переменной; см. «Совет 3».)

Еще лучше было бы воспользоваться функциональным объектом, чтобы повысить степень инкапсуляции счетчика:

➤➤ gotcha26/execbump.cpp

```
class ExecBump { // паттерн Monostate. см Совет 69.
public:
    void operator ()( void (*func)() )
        { func(); ++count_; }
    int get_count() const
        { return count_; }
private:
    static int count_;
};
// ...
```

```
int ExecBump::count_ = 0;
// ...
void aFunc() {
    extern void g();
    ExecBump exec;
    int count = 0;
    while( count++ < 10 )
        exec( g );
}
```

Использование псевдофункций бывает оправдано сравнительно редко, разве что если речь идет о символах препроцессора `__LINE__`, `__FILE__`, `__DATE__` и `__TIME__`:

`>> gotcha28/myassert.h`

```
#define myAssert( e ) ((!e)?void(std::cerr << "Ошибка: " \
    << #e << " строка " << __LINE__ << std::endl): void())
```

См. также «Совет 28».

Совет 27. Не увлекайтесь использованием директивы `#if`

Использование директивы `#if` для отладки

Как мы обычно вставляем в программу отладочный код? Всем известно, что с помощью препроцессора:

```
void buggy() {
#ifdef NDEBUG
    // отладочный код ...
#endif
    // рабочий код ...
#ifdef NDEBUG
    // снова отладочный код ...
#endif
}
```

И все ошибаются. Почти у каждого давно работающего программиста найдется кошмарная история о том, как отладочная версия отлично работала, но стоило определить символ `NDEBUG`, как промышленная программа таинственно работать перестала.

А ничего таинственного здесь нет. Ведь мы по сути дела обсуждаем две совсем разные программы, пусть они и сгенерированы из одних и тех же исходных файлов. Пришлось один и тот же исходный текст даже откомпилировать дважды, чтобы убедиться хотя бы в его синтаксической правильности. Правильный подход – отказаться от идеи отладочной версии и писать единственную программу:

```
void buggy() {
    if( debug ) {
        // отладочный код ...
    }
}
```

```
// рабочий код ...
if( debug ) {
    // снова отладочный код ...
}
}
```

А как же быть с отладочным кодом? Он останется в исполняемом файле промышленной версии? И на него будет расходоваться память? А на обработку лишних ветвей условного предложения будет тратиться время? Нет, если отладочного кода в исполняемом файле не будет. Компиляторы прекрасно справляются с задачей выявления и удаления неиспользуемого кода. Намного лучше, чем это делаем мы с помощью директив `#ifdef`. Нужно лишь ясно выразить свое намерение:

```
const bool debug = false;
```

Выражение `debug` в стандарте названо «целочисленным константным выражением». Каждый компилятор C++ должен уметь вычислять подобные константные выражения на этапе компиляции; так транслируются границы массивов, метки `case` и длины битовых полей. Любой сколько-нибудь качественный компилятор сможет убрать недостижимый код вида

```
if( false ) {
    // недостижимый код ...
}
```

Да-да, даже тот компилятор, на который вы жаловались своему руководству последние пять лет, тоже справится с этим. И, хотя в конечном итоге недостижимый код будет удален, компилятор предварительно выполнит его полный анализ и статический семантический контроль. Следуя определению константного выражения, данному в стандарте, компилятор сумеет удалить недостижимый код, охраняемый даже более сложными выражениями, например:

```
if( debug && debuglvl > 5 && debugopts&debugmask ) {
    // потенциально недостижимый код ...
}
```

Компилятор сумеет устранить ненужный код и в более сложных случаях. Например, можно попытаться задействовать мою любимую встраиваемую функцию в условном выражении:

```
typedef unsigned short Bits;
inline Bits repeated( Bits b, Bits m )
{ return b & m & (b & m)-1; }
// ...
if( debug && repeated( debugopts, debugmask ) ) {
    // потенциально недостижимый код ...
    error( "Разрешена только одна опция" );
}
```

Однако, при наличии вызова функции (все равно, встраиваемой или нет) выражение перестает быть константным, поэтому не гарантируется, что компилятор станет вычислять его на этапе компиляции, поэтому недостижимый код может и остаться в программе. Если вы настаиваете на удалении кода, то такое решение не

переносимо. Некоторые программисты, слишком долгое время работавшие на C, могут предложить такой выход из положения:

```
#define repeated(b, m) ((b) & (m) & ((b) & (m)) - 1)
```

Не делайте этого (см. «Совет 26»).

Отметим, что иногда оставлять в приложении условно компилируемый код даже полезно, например, чтобы задать значения констант на этапе компиляции:

```
const bool debug =
#ifdef NDEBUG
    false
#else
    true
#endif
;
```

Впрочем, и такая малая толика условно компилируемого кода не обязательна. В общем случае, лучше выбирать между отладочной и промышленной версией в файле сборки проекта (makefile или аналогичный механизм).

Использование `#if` для переносимости

«Однако, – начинаете вы с видом знатока – мой код не должен зависеть от платформы. Поэтому я использую `#if`, чтобы учесть специфику разных платформ». И в доказательство своей правоты вы демонстрируете примерно такой код:

```
void operation() {
    // переносимый код
#ifdef PLATFORM_A
    // что-то сделать ...
    a(); b(); c();
#endif
#ifdef PLATFORM_B
    // сделать то же самое ...
    d(); e();
#endif
}
```

Этот код не является платформенно-независимым. Он зависит от многих платформ. Любое изменение на любой из платформ потребует не только перекомпиляции исходных текстов, но и внесения изменений в них для всех платформ. Вы добились максимальной зависимости от платформы: достижение, достойное восхищения, правда, несколько непрактичное.

Но это еще ерунда по сравнению с настоящей проблемой, затаившейся в реализации функции `operation`. Функции – это абстракции. Функция `operation` – это абстракция некоторой операции, которая по-разному реализована на разных платформах. Работая на языке высокого уровня, мы часто можем использовать один и тот же код для реализации одной абстракции на разных платформах. Например, выражение `a = b + c`, где `a`, `b` и `c` – значения типа `int`, для разных процессоров транслируется в разные команды, но его смысл таков, что (вообще говоря) исходный текст на любой платформе будет один и тот же. Так бывает не всегда,

особенно если наша операция определена в терминах, зависящих от операционной системы или функций из конкретной библиотеки.

Из реализации функции `operation` видно, что «то же самое» должно происходить на обеих поддерживаемых платформах, и так оно, наверное, в начале и было. Но в ходе сопровождения ошибки обычно обнаруживаются и исправляются на какой-то одной платформе. Вы и ахнуть не успеете, как смысл функции `operation` на разных платформах перестанет совпадать, и вам придется сопровождать два совершенно разных приложения. Отметим, что такое различие в поведении оказалось требованием, потому что пользователи уже привыкли к платформенно-зависимой семантике `operation` и стали полагаться на нее. Правильнее было бы с самого начала реализовать функцию `operation` так, чтобы она обращалась к платформенно-зависимому коду через не зависящий от платформы интерфейс:

```
void operation() {
    // переносимый код ...
    doSomething(); // переносимый интерфейс ...
}
```

Когда абстракция выделяется явно, вероятность того, что в ходе сопровождения ее семантика на различных платформах сохранится, значительно повышается. Объявление функции `doSomething` должно находиться в платформенно-зависимой части исходного текста. Разные реализации `doSomething` определяются в разных платформенно-зависимых исходных файлах (если `doSomething` — встраиваемая функция, она должна быть определена в платформенно-зависимом заголовочном файле). Выбор платформы производится в файле сборки проекта `makefile`. Не надо никаких `#if`. К тому же, добавление или удаление какой-то платформы не потребует внесения изменений в исходный текст.

А как насчет классов?

Как и функция, класс является абстракцией. У всякой абстракции есть реализация, которая может выбираться либо на этапе компиляции, либо на этапе выполнения. Как и в случае функции, использование директивы `#if` для выбора реализации класса сопряжено с опасностями:

```
class Doer {
#   if ONSERVER
    ServerData x;
#   else
    ClientData x;
#   endif
    void doit();
    // . . .
};

void Doer::doit() {
#   if ONSERVER
    // что-то делается для сервера ...
#   else
    // что-то делается для клиента ...
#   endif
}
```

Строго говоря, этот код не является незаконным, если только символ `ONSERVER`, встречающийся в определении класса `Doer`, не будет определяться и отменяться в разных единицах трансляции. Но иногда очень хотелось бы запретить такой код. Часто бывает, что разные версии `Doer` определяются в разных единицах трансляции, а затем компонуется вместе без ошибок. Ошибки, возникающие при этом во время выполнения, обычно бывает очень трудно найти.

К счастью, такой способ внесения ошибок в программу теперь не так широко распространен, как в прежние времена. Очевидный способ выразить подобную вариативность, — воспользоваться полиморфизмом:

```
class Doer { // платформенно-независимый
public:
    virtual ~Doer();
    virtual void doit() = 0;
};
class ServerDoer : public Doer { // платформенно-зависимый
    void doit();
    ServerData x;
};
class ClientDoer : public Doer { // платформенно-зависимый
    void doit();
    ClientData x;
};
```

Практика — критерий истины

Мы рассмотрели некоторые довольно простые примеры попыток с помощью одного исходного текста представить разные программы. Складывается впечатление, что с помощью идиом и паттернов совсем несложно будет переделать исходный текст так, чтобы сопроводать его стало удобнее.

К сожалению, реальность часто оказывается куда сложнее. Как правило, исходный текст параметризуется не одним символом (скажем, `NDEBUG`), а несколькими, причем каждый из них может принимать несколько значений, и эти символы используются в сочетании друг с другом. Выше мы показали, что каждая комбинация символов и их значений порождает новое приложение со своим абстрактным поведением. С практической точки зрения, даже если и возможно отделить друг от друга приложения, определяемые этими символами, переделка неизбежно приведет к изменению поведения хотя бы на одной платформе.

Но подобная переделка рано или поздно становится необходимостью, когда невозможно точно определить абстрактную семантику программы, а чтобы установить, является ли код синтаксически корректным, приходится выполнять сотни компиляций с разными значениями символов. Гораздо лучше вообще не прибегать к использованию `#if` для создания разных версий программы.

Совет 28. Побочные эффекты в утверждениях

Мне не нравятся многие способы использования директивы `#define`, но я готов смириться со стандартным макросом `assert`, который определен в заго-

Совет 28. Побочные эффекты в утверждениях 71

ловке `<cassert>`. Я даже призываю пользоваться им, при условии, конечно, что это делается правильно. А вот с правильным использованием часто возникают сложности.

Хотя есть много вариаций, но обычно макрос `assert` определяется как-то так:

➤ `gotcha28/myassert.h`

```
#ifndef NDEBUG
#define assert(e) ((e) \
    ? (void)0 \
    : __assert_failed(#e, __FILE__, __LINE__ ) )
#else
#define assert(e) ((void)0)
#endif
```

Если символ `NDEBUG` определен, значит, речь идет не об отладочной версии, и `assert` является пустышкой. В противном случае `assert` расширяется (в данной реализации) в условное выражение, в котором проверяется некоторое условие. Если это условие ложно, выдается диагностическое сообщение и вызывается функция `abort`.

В общем случае использовать `assert` лучше, чем вставлять комментарии, документирующие предусловия, постусловия и инварианты. Если отладка включена, то `assert` выполняет проверку условий во время выполнения, поэтому его не так легко игнорировать, как комментарий (см. «Совет 1»). В отличие от комментариев, утверждения `assert`, ставшие некорректными, обычно исправляются, так как на вызов `abort` трудно не обратить внимания:

➤ `gotcha28/myassert.cpp`

```
template <class Cont>
void doit( Cont &c, int index ) {
    assert( index >= 0 && index < c.size() ); // #1
    assert( process( c[index] ) ); // #2
    // ...
}
```

Но в примере выше мы применили `assert` неправильно. В строке с меткой `#2` ошибка очевидна, так как внутри `assert` вызывается функция, которая может иметь побочный эффект. В таком случае поведение программы будет зависеть от того, определен символ `NDEBUG` или нет. При таком использовании отладочная версия программы может работать правильно, а промышленная – нет. Стоит включить отладку, и ошибка исчезает. Выключаете - и ...

Строка с меткой `#1` не так очевидна. Функция-член `size` класса `Cont`, по всей видимости, константная, поэтому у нее не должно быть побочных эффектов. Так? Не так. Ничто, кроме привычного смысла имени `size` (размер), не обещает семантики константности. Но даже если функция `size` константна, нет гарантии, что ее вызов не приведет к побочным эффектам. Даже если логическое состояние `c` при вызове не изменится, может измениться физическое состояние (см. «Совет 82»). И, наконец, не забывайте, что утверждения служат для «вылавливания» ошибок. Пусть даже обращение к `size` не окажет влияния на последующее поведение кода, но ведь в ее реализации могут быть ошибки. Хотелось бы, чтобы использова-

ние `assert` вскрывало ошибки, а не прятало их. При правильном употреблении можно избежать даже потенциального побочного эффекта при проверке условия:

```
template <class Cont>
void doit( Cont &c, int index ) {
    const int size = c.size();
    assert( index >= 0 && index < size ); // правильно
    // ...
}
```

Конечно, утверждения – это не панацея, но они занимают свою нишу, расположенную где-то между комментариями и исключениями, и помогают документировать программы и обнаруживать некорректное поведение. Основной недостаток утверждений заключается в том, что `assert` – это псевдофункция и, как таковая, страдает от всех болезней, присущих псевдофункциям (см. «Совет 26»). Но вместе с тем она стандартна, поэтому ее негативные стороны хорошо известны. При правильном употреблении утверждения могут оказаться очень полезными.



Глава 4. Преобразования

Сложность системы типов в языке C++ вполне соответствует его выразительной мощи. Но и без того не простая концепция осложняется наличием определяемых пользователем преобразований, которые могут неявно применяться на этапе компиляции. Возможность расширять язык C++ путем добавления новых абстрактных типов данных возлагает на проектировщика ответственность за создание безопасной, эффективной и внутренне непротиворечивой системы типов. Поскольку типы в C++ контролируются в основном статически, то эффективный дизайн должен замаскировать внутреннюю сложность.

К сожалению, плохое кодирование может испортить даже самый удачный проект. В этой главе мы рассмотрим некоторые типичные ошибки, которые сводят на нет статическую безопасность типов. Мы также познакомимся с некоторыми аспектами C++, которые часто понимаются неверно, что тоже может приводить к компрометации статической системы безопасности типов.

Совет 29. Преобразование посредством `void *`

Даже программисты на C знают, что `void *` – это близкий родственник приведения, от которого надо по возможности держаться подальше. Как и в случае приведения, преобразование типизированного указателя в `void *` приводит к утрате всей полезной информации о типе. Обычно при использовании `void *` исходный тип указателя можно «запомнить» и затем восстановить. Если восстановить тип корректно, то все будет прекрасно работать (с той оговоркой, что необходимость запоминать тип для последующего приведения означает дополнительную работу для проектировщика):

```
void *vp = new int(12);  
// ...  
int *ip = static_cast<int *>(vp);    // будет работать
```

К сожалению, даже такое простое применение `void *` открывает двери для проблем, связанных с переносимостью. Напомним, что для относительно безопасного и переносимого приведения типа (когда без него не обойтись) лучше применять оператор `static_cast`. Например, `static_cast` пригоден для преобразования указателя на базовый класс в указатель на открыто наследующий ему производный класс. Для небезопасных, платформенно-зависимых преобразований мы вынуждены применять оператор `static_cast`. Так, этот оператор можно использовать для преобразования целого в указатель или между двумя указателями несвязанных типов:

```
char *cp = static_cast<char *>(ip); // ошибка!  
char *cp = reinterpret_cast<char *>(ip); // работает.
```

Наличие в программе `static_cast` недвусмысленно говорит и вам, и читателю вашей программы, что вы не только выполняете приведение типов, но и делаете это потенциально не переносимым способом. Использование для той же цели `void *` в качестве промежуточного типа скрывает эту важную информацию:

```
char *cp = static_cast<char *>(vp); // поместить адрес int в char *!
```

Дело обстоит еще хуже. Рассмотрим пользовательский интерфейс, который позволяет сохранить, а затем извлечь адрес объекта «Widget»:

```
typedef void *Widget;
void setWidget( Widget );
Widget getWidget();
```

Пользователи такого интерфейса понимают, что должны запомнить тип `Widget`, для которого вызывается функция `setWidget`, чтобы потом эту информацию можно было восстановить:

```
// В каком-то заголовочном файле ...
class Button {
    // ...
};
class MyButton : public Button {
    // ...
};
// где-то в другом месте ...
MyButton *mb = new MyButton;
setWidget( mb );

// совсем в другом месте ...
Button *b = static_cast<Button *>(getWidget()); // может и сработать!
```

Обычно такой код будет работать, пусть даже мы теряем какую-то информацию о типе при извлечении `Widget`. Сохраненный `Widget` ссылался на `MyButton`, а после восстановления превратился в `Button`. Причина, по которой этот код часто оказывается работоспособным, связана со способом размещения объекта в памяти.

Обычно та часть объекта производного класса, которая относится к базовому классу, имеет смещение 0 относительно начала объекта в памяти, то есть подобъект базового класса трактуется как первый член данных производного класса. А данные собственно производного класса размещаются ниже, как показано на рис. 4.1. Поэтому адрес объекта производного класса такой же, как у объекта его базового класса. (Заметим, однако, что стандарт гарантирует правильность результата, только если адрес, хранящийся в переменной типа `void *`, преобразуется точно в тот же тип, который был у объекта до преобразования его в `void *`. О том, как такой код может работать неправильно даже в случае одиночного наследования, см. «Совет 70».)

Однако этот код нестабилен, поскольку в ходе сопровождения в него может быть внесена ошибка. В частности, так произойдет в случае вполне корректного применения множественного наследования:

```
// В каком-то заголовочном файле ...
class Subject {
```



Рис. 4.1. Типичное размещение объекта производного класса в памяти при одиночном наследовании

```
// ...
};
class ObservedButton : public Subject, public Button {
// ...
};
// где-то в другом месте ...
ObservedButton *ob = new ObservedButton;
setWidget( ob );
// ...
Button *badButton = static_cast<Button *>(getWidget()); // беда!
```

Проблема в том, как объект производного класса размещается в памяти в случае множественного наследования. У объекта `ObservedButton` есть части, принадлежащие двум базовым классам, и лишь адрес одной из них совпадает с адресом объекта в целом. Как правило, объект первого базового класса (в данном случае `Subject`) размещается со смещением 0 от начала объекта производного класса, за ним следует память, занятая объектами последующих базовых классов (в данном случае `Button`), и в самом конце – дополнительные данные-члены, определенные в производном классе (см. рис. 4.2.). В случае множественного наследования у одного объекта оказывается несколько адресов.

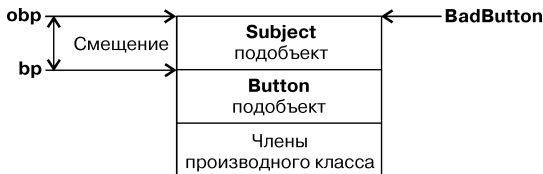


Рис. 4.2. Типичное размещение объекта производного класса в памяти при множественном наследовании. Объект `ObservedButton` содержит подобъекты своих базовых классов в `Subject` и `Button`. Из-за потери информации `badButton` ссылается на адрес, не принадлежащий объекту `Button`

Обычно это не составляет проблемы, поскольку компилятору известны все смещения, и он может выполнить нужную корректировку:

```
Button *bp = new ObservedButton;
ObservedButton *obp = static_cast<ObservedButton *>(bp);
```

В примере выше `bp` правильно указывает на часть `Button` объекта `ObservedButton`, а не на начало самого объекта. Когда мы приводим указатель на `Button`

к указателю на `ObservedButton`, компилятор добавляет к адресу смещение, так что теперь указатель направлен на начало объекта `ObservedButton`. Это несложно, потому что компилятор может воспользоваться своим знанием о смещениях подобъектов каждого базового класса, если только ему известны типы базового и производного классов.

А вот тут-то и возникает проблема. Воспользовавшись функцией `setWidget`, мы отбросили всю полезную информацию о типе. Теперь при приведении результата, возвращенного `getWidget`, к типу `Button` компилятор не в состоянии выполнить корректировку адреса. И, следовательно, указатель на `Button` фактически указывает на `Subject`!

У указателей на `void` есть свои применения, как и у приведений типов, но злоупотреблять ими не стоит. Никогда не следует включать `void *` в интерфейс, одна часть которого требует от пользователя указать информацию, потерянную при работе с другой частью.

Совет 30. Срезка

Срезка происходит тогда, когда объект производного класса копируется в объект базового класса. В результате данные и поведение, специфичные для производного класса, «срезаются», что обычно приводит к ошибке или непредсказуемому поведению.

```
class Employee {
public:
    virtual ~Employee();
    virtual void pay() const;
    // . . .
protected:
    void setType( int type )
        { myType_ = type; }
private:
    int myType_; // плохая мысль, см. совет 69
};
class Salaried : public Employee {
    // ...
};
Employee employee;
Salaried salaried;
employee = salaried; // срезка!
```

Присваивание объекта `salaried` объекту `employee` совершенно законно, так как `Salaried` «является разновидностью» `Employee`, но результат, скорее всего, будет не таким, как вы ожидаете. После присваивания поведение `employee`, включая как виртуальные, так и не виртуальные его функции, будет таким, как определено в классе `Employee`. Никакие данные-члены, специфичные для класса `Salaried`, не копируются.

Хуже всего то, что состояние объекта `employee` – это копия части объекта `salaried`, унаследованной от класса `Employee`. Что же в этом плохого? Дело в том, что объект производного класса `Salaried` может хранить в унаследован-

ной от Employee базовой части значения, специфичные для Salaried, которые для объекта класса Employee не имеют смысла (см. «Совет 91»).

В качестве иллюстрации предположим, что классы, производные от Employee, хранят в своих Employee-подобъектах какой-то код, идентифицирующий тип. (Сразу отмечу, что это неудачный способ проектирования, я привожу его лишь в качестве иллюстрации. См. «Совет 69».) После срезки объект employee будет вести себя как Employee, заявляя при этом, что он Salaried.

На практике противоречия между состоянием и поведением срезанного объекта бывают гораздо тоньше, а потому и разрушительнее.

Чаще всего срезка возникает, когда объект производного класса передается по значению для инициализации формального параметра, являющегося объектом базового класса.

```
void fire( Employee victim );
// ...
fire( salaried ); // срезка!
```

Избежать этой проблемы можно, передавая объект не по значению, а по ссылке (или по указателю). В таком случае никакой срезки не будет, поскольку объект производного класса нигде не копируется, а формальный аргумент становится лишь псевдонимом фактического (см. «Совет 5»).

```
void rightSize( Employee &asset );
// ...
rightSize( salaried ); // срезки нет
```

Со срезкой могут быть связаны и другие проблемы, но это бывает гораздо реже. Например, можно скопировать подобъект базового класса из одного объекта производного класса в другой объект производного класса, правда, уже иного:

```
Employee *getNextEmployee(); // получить объект класса, производного от
                             // Employee
// ...
Employee *ep = getNextEmployee();
*ep = salaried; // срезка!
```

Возникновение проблем из-за срезки, как правило, является свидетельством глубоких изъянов при проектировании иерархии классов. Лучший и самый простой способ никогда не сталкиваться со срезкой — избегать конкретных базовых классов (см. «Совет 93»).

```
class Employee {
public:
    virtual ~Employee();
    virtual void pay() const = 0;
    // . . .
};
void fire( Employee ); // ошибка, к счастью
void rightSize( Employee & ); // правильно
Employee *getNextEmployee(); // правильно
Employee *ep = getNextEmployee(); // правильно
*ep = salaried; // ошибка, к счастью
Employee e2( salaried ); // ошибка, к счастью
```

Невозможно создать объект абстрактного базового класса, поэтому большая часть ситуаций, приводящих к срезке, будет перехвачена еще на этапе компиляции.

Отметим, что изредка срезка применяется намеренно, чтобы модифицировать поведение или тип объекта производного класса. Обычно в таких случаях данные не срезаются, а срезка просто «иначе интерпретирует» данные базового класса, наделяя их поведением производного. Такая техника полезна, хотя применяется редко, и никогда не должна раскрываться как часть доступного пользователям интерфейса.

Совет 31. Преобразование в указатель на константу

Для начала разберемся с терминологией. «Константный указатель», или «const-указатель», – это указатель, являющийся константным. Это не значит, что то, на что он указывает, является константой. Да, в стандартной библиотеке C++ есть понятие `const_iterator`, обозначающее неконстантный итератор, указывающий внутрь последовательности константных элементов, но это симптом «проектирования комитетом» или другой похожей болезни.

```
const char *pci; // указатель на константу
char * const cpi = 0; // константный указатель
char const *pci2; // указатель на константу, см. совет 18
const char * const cpci = 0; // константный указатель на константу
char *ip; // указатель
```

Стандарт допускает преобразования, «повышающие степень константности». Например, можно скопировать указатель на не-константу в указатель на константу. Среди прочего, это позволяет передавать указатель на неконстантный символ стандартным функциям `strcmp` или `strlen`, несмотря на то, что согласно объявлению, они принимают указатель на константу. Интуитивно мы понимаем, что, разрешая указателю на константу ссылаться на неконстантные данные, мы не нарушаем ограничения, налагаемого объявлением. Мы также понимаем, что обратное неверное, поскольку в этом случае мы получили бы больше прав, чем допускает объявление данных:

```
size_t strlen( const char * );
// ...
int i = strlen( cpi ); // правильно ...
pci = ip; // правильно ...
ip = pci; // ошибка!
```

Заметим, что язык занимает консервативную позицию: может быть, модифицировать данные, на которые ссылается указатель на константу, и допустимо в том смысле, что не приведет к немедленному дампу памяти. Так бывает, если данные на самом деле не константны или, хотя и константны, но платформа не размещает их в памяти, доступной только для чтения. Однако квалификатор `const` – это скорее выражение намерений проектировщика, нежели физическое свойство. Можно считать, что язык подкрепляет желание проектировщика.

Совет 32. Преобразование в указатель на указатель на константу

Простота идеи, лежащей в основе преобразования в указатель на константу, не распространяется на случай преобразования в указатель на указатель на константу. Рассмотрим попытку преобразовать указатель на указатель на `char` в указатель на указатель на `const char` (то есть `char**` в `const char**`):

```
char **ppc;
const char **ppcc = ppc; // ошибка!
```

Выглядит безобидно, но, как и многие, на первый взгляд, безобидные преобразования, открывает путь к обходу системы типов:

```
const T t = init;
T *pt;
const T **ppt = &pt; // ошибка, к счастью
*ppt = &t; // поместить const T * в T *!
*pt = value; // попрашиваемся с t!
```

Эта животрепещущая тема обсуждается в разделе 4.4. стандарта, под заголовком «Преобразования квалификаторов». (Технически, ключевые слова `const` и `volatile` называются «квалификаторы типа», но в стандарте C++ часто применяется термин «cv-квалификаторы». Я предпочитаю первое название.) Там мы находим следующие простые правила, определяющие, когда преобразование возможно:

Преобразование может добавить cv-квалификаторы на любом уровне многоуровневых указателей, кроме первого, при соблюдении следующих правил:

Два указательных типа T_1 и T_2 подобны, если существует тип T и целое число $n > 0$ такое, что:

T_1 есть $cv_1, 0$ указателей на $cv_1, 1$ указатель на ... $cv_1, n-1$ указатель на cv_1, n T

и

T_2 есть $cv_2, 0$ указателей на $cv_2, 1$ указатель на ... $cv_2, n-1$ указатель на cv_2, n T где каждый cv_i, j — это `const`, `volatile`, `const volatile` или ничего.

Другими словами, два указателя подобны, если у них одинаковый базовый тип и одно и то же число «звездочек». Так, типы `char * const **` и `const char ***const` подобны, `aint * const *` и `int ***` — нет.

n -кортеж, состоящий из cv-квалификаторов, следующих за первым в указательном типе, например, $cv_1, 1, cv_1, 2, \dots, cv_1, n$ в указательном типе T_1 , называется сигнатурой *cv-квалификации* указательного типа. Выражение типа T_1 можно преобразовать в тип T_2 тогда и только тогда, когда выполнены следующие условия:

- Указательные типы подобны.
- Для каждого $j > 0$, если `const` входит в cv_1, j , то `const` входит в cv_2, j , и то же верно для `volatile`.
- Если cv_1, j и cv_2, j различны, то `const` входит в каждый cv_2, k для $0 < k < j$.

Вооружившись этими правилами и терпением, мы можем установить законность следующих преобразований указателей:

```
int * * * const cnnn = 0;
    // n==3, signature == none, none, none
int * * const * ncn = 0;
    // n==3, signature == const, none, none
int * const * * nncn = 0;
    // signature == none, const, none
int * const * const * nccn = 0;
    // signature == const, const, none
const int * * * nnnn = 0;
    // signature == none, none, const

// примеры применения правил
ncnn = cnnn; // правильно
nncn = cnnn; // ошибка!
nccn = cnnn; // правильно
ncnn = cnnn; // правильно
nnnn = cnnn; // ошибка!
```

Как ни странно звучат эти правила, необходимость в их применении возникает довольно часто. Рассмотрим типичную ситуацию:

```
extern char *namesOfPeople[];
for( const char **currentName = namesOfPeople; // ошибка!
     *currentName; currentName++ ) // ...
```

В моей практике типичная реакция на эту ошибку заключалась в отправке отчета поставщику компилятора, применении `const_cast` для снятия константности и получении затем дампа памяти. Но, как обычно бывает, прав компилятор, а не разработчик.

Рассмотрим более специфическую версию приведенного выше примера:

```
typedef int T;
const T t = 12345;
T *pt;
const T **ppt = (const T **)&t; // опасное приведение!
*ppt = &t; // поместить const T * в T *!
*pt = 54321; // попросаемся с t!
```

Трагическая сторона этого кода состоит в том, что ошибка может оставаться незамеченной годами, пока не проявится во время рутинного сопровождения. Например, мы можем использовать значение `t`:

```
cout << t; // вероятно, будет выведено 12345
```

Поскольку компилятору разрешено подставлять инициализатор константы вместо самой константы, это предложение, вероятно, выведет значение 12345 даже после того, как мы изменили его на 54321. Но позже чуть-чуть иное использование `t` выведет ошибку на чистую воду:

```
const T *pct = &t;
// ...
cout << t; // будет выведено 12345
cout << *pct; // будет выведено 54321!
```

Зачастую лучше избегать сложностей, связанных с указателями на указатели, и пользоваться ссылками или стандартной библиотекой. Например, в C принято

передавать адрес указателя (то есть указатель на указатель), чтобы модифицировать значение самого указателя:

➤➤ gotcha32/gettoken.cpp

```
// get_token возвращает указатель на следующую цепочку
// символов, ограниченных символами, входящими в ws.
// Указатель, переданный в качестве аргумента, после возврата будет
// указывать на символ, следующий за выделенной лексемой.
char *get_token( char **s, char *ws = " \\t\\n" ) {
    char *p;
    do
        for( p = ws; *p && **s != *p; p++ );
    while( *p ? *(**s)++ : 0 );
    char *ret = *s;
    do
        for( p = ws; *p && **s != *p; p++ );
    while( *p ? 0 : **s ? (**s)++ : 0 );
    if( **s ) {
        **s = '\\0';
        ++**s;
    }
    return ret;
}

extern char *getInputBuffer();
char *tokens = getInputBuffer();
// ...
while( *tokens )
    cout << get_token( &tokens ) << endl;
```

В С++ предпочтительно передавать аргумент-указатель как ссылку на неконстанту. Это делает реализацию функции несколько чище и, что важнее, она перестает быть такой запутанной:

➤➤ gotcha32/gettoken.cpp

```
char *get_token( char *&s, char *ws = " \\t\\n" ) {
    char *p;
    do
        for( p = ws; *p && *s != *p; p++ );
    while( *p ? *s++ : 0 );
    char *ret = s;
    do
        for( p = ws; *p && *s != *p; p++ );
    while( *p ? 0 : *s ? s++ : 0 );
    if( *s ) *s++ = '\\0';
    return ret;
}
// ...
while( *tokens )
    cout << get_token( tokens ) << endl;
```

Исходный пример можно реализовать более безопасно с помощью компонентов из стандартной библиотеки:

```
extern vector<string> namesOfPeople;
```

Совет 33. Преобразование указателя на указатель на базовый класс

С похожей ситуацией мы сталкиваемся, когда имеем дело с указателем на указатель на производный класс:

```
D1 d1;
D1 *d1p = &d1; // правильно
B **ppb1 = &d1p; // ошибка, к счастью
D2 *d2p;
B **ppb2 = &d2p; // ошибка, к счастью
*ppb2 = *ppb1; // теперь d2p указывает на D1!
```

Выглядит знакомо? Как свойство константности не сохраняется, если ввести еще один уровень косвенности, так не сохраняется и свойство «является». Хотя указатель на производный класс «является» указателем на открытый базовый класс, но указатель на указатель на производный класс уже не является указателем на указатель на открытый базовый класс. Как и в случае с примером `const`, ситуация, в которой проявляется ошибка, на первый взгляд, выглядит надуманной. Однако легко привести пример, в котором эта ошибка возникает из-за плохо спроектированного интерфейса, усугубленного неправильным его использованием:

```
void doBs( B *bs[], B *pb ) {
    for( int i = 0; bs[i]; ++i )
        if( somecondition( bs[i], pb ) )
            bs[i] = pb; // oops!
}
// ...
extern D1 *array[];
D2 *aD2 = getMeAD2();
doBs( ( B ** )array, aD2 ); // и снова остается пожелать приведениям смерти
```

И здесь разработчик счел, что компилятор ошибся, и решил обойти систему типов за счет приведения. Но в данном случае значительная доля вины лежит на проектировщице интерфейса функции. Безопаснее было бы воспользоваться контейнером, который не обманешь приведением, как массив.

Совет 34. Проблемы с указателем на многомерный массив

В языках C и C++ есть лишь рудиментарная поддержка массивов. По сути дела, имя массива – это всего лишь указательный литерал, который ссылается на первый элемент массива:

```
int a[5];
int * const pa = a;
int * const *ppa = &pa;
const int alen = sizeof(a)/sizeof(a[0]); // alen == 5
```

Единственное практическое различие между именем массива и константным указателем заключается в том, что в результате применения оператора `sizeof`

к имени массива мы получаем размер массива, а не указателя. Ну и еще имя массива не занимает памяти и, следовательно, не имеет адреса. Выразимся точнее: у массива есть адрес, и этот адрес обозначается именем массива, но само имя массива адреса не имеет:

```
int *ip = a; // a - указатель на первый элемент массива
int (*ap)[5] = &a; // &a - адрес массива, а не имени a
int (*ap2)[sizeof(a)/sizeof(a[0])] = &a; // то же самое
int **pip = &ip; // &ip - адрес указателя, а не массива
```

Также обстоит дело и с многомерными массивами, точнее, с массивами массивов. Но помните, что типом первого элемента в многомерном массиве является массив, а не базовый тип:

```
int aa[2][3];
const int aalen = sizeof(aa)/sizeof(aa[0]); // aalen = 2
```

Таким образом, `aa` – это, по существу, литеральный указатель на первый элемент массива, состоящего из трех целых чисел, а вовсе не указатель на целое. Это может приводить к странному, хотя технически корректным, результатам:

```
void processElems( int *, size_t );
void processElems( void *, size_t );
// ...
processElems( a, aalen );
processElems( aa, aalen ); // беда!
```

Первое обращение к перегруженной функции `processElems` соответствует версии, принимающей аргумент типа `int *`; имя массива `a` – это просто замаскированный `int *`. Второе обращение соответствует версии `processElems`, которая принимает `void *`, и вряд ли программист это имел в виду. Тип имени многомерного массива – указатель на его первый элемент, то есть массив конкретного размера, а не указатель на базовый тип массива. Не существует неявного преобразования из `int (*) [3]` (это указатель на массив из трех целых) в `int *`, но есть такое преобразование в `void *`.

```
int (* const paa)[3] = aa;
int (* const *ppaa)[3] = &paa;
void processElems( int (*) [3], size_t );
// ...
processElems( aa, aalen ); // правильно
```

С многомерными массивами вообще трудно работать. Лучше воспользоваться контейнерами из стандартной библиотеки или специальными контейнерами, которые реализуют абстрактный многомерный массив. Если в конкретной задаче не требуются именно встроенные многомерные массивы, то обычно лучше инкапсулировать их. Безответственно предлагать наивному пользователю такой интерфейс:

```
int *(* (*aryCallback)(int (*) [n])) [n];
```

Это (ну, разумеется) не что иное, как указатель на функцию, которая принимает указатель на массив из `n` указателей на `int` и возвращает указатель того же типа. Ладно, это всего лишь доведенная до абсурда демонстрация. (См. «Совет 11».) С помощью `typedef` все можно существенно упростить:

```
typedef int *(*PA)[n];
PA (*aryCallback)(PA); // так гуманнее
```

Совет 35. Бесконтрольное понижающее приведение

Приведение указателя на базовый класс к указателю на производный класс («понижающее приведение») может в результате дать некорректный адрес, как показано на рис. 4.3. При выполнении арифметических операций над приводимым указателем компилятор предполагает, что адрес базового класса принадлежит базовой части производного класса:

```
class A { public: virtual ~A(); };
class B { public: virtual ~B(); };
class D : public A, public B {};
class E : public B {};
B *bp = getMeAB(); // получить объект класса, производного от B
D *dp = static_cast<D*>(bp); // безопасно???
```

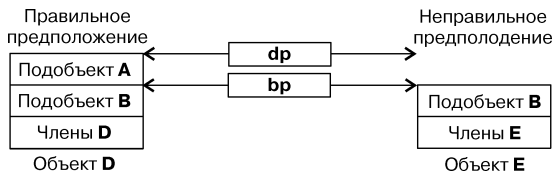


Рис. 4.3. Результат некорректного статического приведения: приведение указателя на подобъект B объекта E к указателю на D

Лучше всего проектировать иерархию классов так, чтобы понижающие приведения были не нужны; систематическое использование понижающих приведений – признак неудачного дизайна. Если понижающее приведение все-таки необходимо, то стоит обратиться к оператору `dynamic_cast`, который во время выполнения производит проверку корректности приведения:

```
if ( D *dp = dynamic_cast<D *>(bp) ) {
    // приведение успешно
}
else {
    // ошибка приведения
}
```

Совет 36. Неправильное использование операторов преобразования

Чрезмерное использование операторов преобразования усложняет код. Поскольку компилятор применяет их неявно, то наличие слишком большого числа операторов преобразования в классе может стать причиной неоднозначности:

```
class Cell {
public:
    // ...
    operator int() const;
    operator double() const;
    operator const char *() const;
    typedef char **PPC;
    operator PPC() const;
    // и т.д...
};
```

Класс Cell отвечает столь многим требованиям, что пользователи часто получают более одного ответа, и тогда возникает неоднозначность во время компиляции. Хуже того, когда неоднозначность не возникает и программа компилируется без ошибки, трудно сказать точно, какое именно неявное преобразование применил компилятор. В общем случае, когда требуется слишком много преобразований, лучше отказаться от операторов преобразования вовсе и ограничиться более прямолинейными явными функциями преобразования:

```
class Cell {
public:
    // ...
    int toInt() const;
    double toDouble() const;
    const char *toPtrConstChar() const;
    char **toPtrPtrChar() const;
    // и т.д...
};
```

Как правило, в классе должно быть не более одного оператора преобразования. Если их два, надо приглядеться к дизайну внимательнее. Когда их три и больше, надо менять дизайн.

Даже единственный оператор преобразования в сочетании с конструктором может стать причиной неоднозначности:

```
class B;

class A {
public:
    A( const B & );
    // ...
};

class B {
public:
    operator A() const;
    // ...
};
```

Есть два способа неявно преобразовать B в A: конструктор класса A и оператор преобразования из класса B. Результат – неоднозначность:

```
extern A a;
extern B b;
a = b; // ошибка! неоднозначность
a = b.operator A(); // правильно, но выглядит странно
a = A(b); // ошибка! неоднозначность
```

Обратите внимание, что не существует прямого способа вызвать конструктор или получить его адрес. Поэтому выражение `A(b)` – это не вызов конструктора, хотя подобные выражения часто являются результатом вызова конструкторов. Это требование преобразовать `b` к типу `A` любым возможным способом, и оно по-прежнему неоднозначно. (К сожалению, большинство компиляторов не выдают в этом случае ошибку и пользуются для выполнения преобразования конструктором класса `A`.)

Обычно лучше отказаться от операторов преобразования, объявить конструктор с одним аргументом `explicit` и избегать неявных преобразований за исключением тех случаев, когда они действительно необходимы. Если имеются и не-`explicit` конструкторы, и операторы преобразования, эвристическое правило таково: пользуйтесь конструкторами для преобразования из определенных пользователем типов, а операторами преобразования – только для преобразования во встроенные типы.

Назначение операторов преобразования: более тесно интегрировать абстрактный тип данных в существующую систему типов путем предоставления неявных преобразований, построенных по образцу преобразований, которые поддерживают встроенные типы. Было бы ошибкой применять оператор преобразования для реализации «дополнительных» преобразований:

```
class Complex {
    // ...
    operator double() const;
};
Complex velocity = x + y;
double speed = velocity;

class Container {
    // ...
    virtual operator Iterator *() const = 0;
};
Container &c = getNewContainer();
Iterator *i = c;
```

Здесь проектировщик класса `Complex` хотел дать средства для вычисления длины вектора, определяемого комплексным числом. Однако пользователь этого интерфейса может предположить, что преобразование к типу `double` возвращает вещественную часть комплексного числа, или его мнимую часть, или угол вектора, или придумать еще какую-нибудь разумную интерпретацию. Смысл преобразования неясен.

Проектировщик абстрактного интерфейса `Container` хотел реализовать паттерн `Factory Method` (фабричный метод), который возвращает указатель на итератор для конкретного контейнера, производного от класса `Container`. Однако мы не преобразовываем `Container` в `Iterator`, следовательно, реализация фабричного метода в виде преобразования только вводит в заблуждение и потому не годится. Кроме того, в будущем могут возникнуть проблемы, если Фабричному Методу потребуется передать аргумент. Поскольку оператор преобразования не может принимать аргументов, то его придется заменить не-оператор-

ной функцией. Это заставит всех пользователей класса `Container` искать в своих программах и переделывать неявные обращения к оператору преобразования.

Гораздо лучше оставить операторам преобразования то, для чего они предназначены, то есть преобразования. Во всех рассмотренных случаях предпочтительнее воспользоваться не-операторными функциями:

```
class Complex {
    // ...
    double magnitude() const;
};
Complex velocity = x + y;
double speed = velocity.magnitude();

class Container {
    // ...
    virtual Iterator *genIterator() const = 0;
};
Container &c = getNewContainer();
Iterator *i = c.genIterator();
```

Я утверждаю, что эта рекомендация относится даже к случаю простого преобразования к типу `bool` (а иногда и `void *`), цель которого, – сообщить, находится ли объект в пригодном для использования состоянии:

```
class X {
public:
    virtual operator bool() const = 0;
    // ...
};
// . . .
extern X &a;
if( a ) {
    // а пригоден для использования ...
```

И здесь «дополнительная функциональность» оператора преобразования только приводит к неточности. В будущем мы можем захотеть различать недействительные, непригодные для использования и заперченные объекты. Лучше выражать свои намерения более определенно:

```
class X {
public:
    virtual bool isValid() const = 0;
    virtual bool isUsable() const = 0;
    // ...
};
// ...
if( a.isValid() ) {
    // ...
```

В части `iostream` стандартной библиотеки операторы преобразования используются, чтобы быстро проверить состояние потока:

```
if( cout ) // cout в хорошем состоянии?
    // ...
```

Это обеспечивает функция `operator void *`, и приведенное выше предложение должно транслироваться примерно так:

```
if( static_cast<bool>(cout.operator void *()) ) // ...
```

Если поток `iostream` в плохом состоянии, то оператор преобразования возвращает нулевой указатель, в противном случае, ненулевой указатель. Поскольку преобразование из указателя в тип `bool` относится к числу определенных, то им можно воспользоваться для проверки состояния потока. К несчастью, его также можно использовать для задания значения указателя на `void`:

```
void *coutp = cout; // странно и почти бесполезно
cout << cout << cin << cerr; // печатает какие-то void *
```

Однако, наличие преобразования из `iostream` в `void *`, хотя и выглядит странно, но не вызывает столько проблем, сколько могло бы вызвать преобразование к типу `bool`:

```
cout >> 12; // к счастью, не компилируется
```

Здесь мы допустили типичную ошибку: применили к потоку вывода оператор сдвига вправо вместо оператора сдвига влево. Если бы можно было преобразовать поток вывода непосредственно в `bool`, то это предложение компилировалось бы. Объект `cout` был бы преобразован в значение типа `bool` которое, в свою очередь, было бы преобразовано в `int`, и результат был бы сдвинут на 12 битов вправо. Очевидно, что преобразование `cout` в `void *` предпочтительнее, но еще лучше было бы отказаться от операторов преобразования вовсе и предоставить ясную и недвусмысленную функцию-член:

```
if( !cout.fail() )
    // ...
```

Совет 37. Непреднамеренное преобразование с помощью конструктора

Конструктор с одним аргументом описывает как инициализацию, так и преобразование. Подобно оператору преобразования, компилятор может неявно принимать конструктор для преобразования типов. Иногда это бывает удобно:

```
class String {
public:
    String( const char * );
    operator const char *() const;
    // . . .
};
String name1( "Fred" ); // непосредственная инициализация
name1 = "Joe"; // неявное преобразование
const char *cname = name1; // неявное преобразование
String name2 = cname; // неявное преобразование,
// инициализация копированием
String name3 = String( cname ); // явное преобразование,
// инициализация копированием
```

(см. «Совет 56».) Однако неявные преобразования с помощью конструктора часто приводят к непонятному коду и появлению тонких ошибок. Рассмотрим шаблон контейнера, реализующего стек фиксированного размера:

```
template <class T>
class BoundedStack {
```



```

public:
    BoundedStack( int maxSize );
    ~BoundedStack();
    bool operator ==( const BoundedStack & ) const;
    void push( const T & );
    void pop();
    const T &top() const;
    // ...
};

```

В нашем типе `BoundedStack` есть обычные для стека операции `push`, `pop` и так далее, а также возможность сравнивать два стека на равенство. При создании экземпляра `BoundedStack<T>` необходимо указать максимальный размер

```

BoundedStack<double> s( 128 );
s.push( 37.0 );
s.push( 232.78 );
// ...

```

К несчастью, конструктор с одним аргументом может быть применен для преобразования в тех случаях, когда мы предпочли бы получить ошибку компиляции:

```

if( s == 37 ) { // беда!
    // ...
}

```

В данном случае мы, скорее всего, хотели записать условие типа `s.top() == 37`. Но компилятор ничего не скажет, поскольку может преобразовать целое значение `37` в объект типа `BoundedStack<double>` и передать его в качестве аргумента функции `BoundedStack<double>::operator ==`. По существу, компилятор генерирует такой код:

```

BoundedStack<double> stackTemp( 37 );
bool resultTemp( s.operator ==( stackTemp ) );
stackTemp.~BoundedStack<double>();
if( resultTemp ) {
    // ...
}

```

Этот код, хотя и допустим, но некорректен и влечет высокие накладные расходы. Безопаснее было бы объявить конструктор класса `BoundedStack` с ключевым словом `explicit`. Оно сообщает компилятору, что данный конструктор нельзя применять для неявных преобразований, хотя явное использование в таком качестве и не запрещено:

```

template <class T>
class BoundedStack {
public:
    explicit BoundedStack( int maxSize );
    // ...
};
// ...
if( s == 37 ) { // ошибка, к счастью
    // ...
}
if( s.top() == 37 ) { // правильно, преобразования нет
    // ...
}
if( s == static_cast< BoundedStack<double> >(37) ) { // правильно ...
    // ...
}

```

Докучливых неявных преобразований следует опасаться гораздо больше, чем изредка возникающей необходимости применить явное преобразование, поэтому большинство конструкторов с одним аргументом рекомендуется объявлять `explicit`.

Отметим, что наличие ключевого слова `explicit` в объявлении конструктора влияет также на то, какой синтаксис инициализации допустим при объявлении объекта класса. Слегка изменим объявление класса `String` выше и посмотрим, какие виды инициализации после этого станут законными:

```
class String {
public:
    explicit String( const char * );
    operator const char *() const;
    // ...
};
String name1( "Fred" ); // правильно.
name1 = "Joe"; // ошибка!
const char *cname = name1; // неявное преобразование, правильно.
String name2 = cname; // ошибка!
String name3 = String( cname ); // явное преобразование, правильно.
```

Неявная генерация временного объекта, имеющая место при инициализации `name2` копированием, теперь ошибочна, равно как и аргумент функции `String::operator =`. Инициализация `name3` по-прежнему допустима, поскольку это преобразование явное (хотя все же было бы лучше для инициализации применить `static_cast`; см. «Совет 40»). Как обычно, непосредственная инициализация предпочтительнее инициализации копированием (см. «Совет 56»).

Прежде чем оставить тему `explicit`, рассмотрим поучительную, хотя и вышедшую из моды, технику имитации семантики `explicit` в отсутствие этого ключевого слова:

```
class StackInit {
public:
    StackInit( size_t s ) : size_( s ) {}
    int getSize() const { return size_; }
private:
    int size_;
};
template <class T>
class BoundedStack {
public:
    BoundedStack( const StackInit &init );
    // ...
};
```

Поскольку конструктор класса `BoundedStack` не объявлен `explicit`, компилятор попытается неявно преобразовать любой объект `StackInit` в `BoundedStack`. Однако компилятор не будет пытаться преобразовать в `StackInit` целое число с последующим неявным преобразованием `StackInit` в `BoundedStack`. Стандарт четко говорит, что разрешено только одно неявное определенное пользователем преобразование:

```
BoundedStack<double> s( 128 ); // правильно.
```

```
BoundedStack<double> t = 128; // правильно.
if( s == 37 ) { // ошибка!
    // ...
}
```

Этот прием дает почти такое же поведение, как и ключевое слово `explicit`. Объявления `s` и `t` допустимы, так как для преобразования `128` в `StackInit` нужно только одно пользовательское преобразование. Но компилятор не будет пытаться преобразовать значение `37` в `BoundedStack<double>`, поскольку для этого требуется два преобразования: из `int` в `StackInit` и из `StackInit` в `BoundedStack<double>`.

Совет 38. Приведение типов в случае множественного наследования

В случае множественного наследования у объекта может быть несколько адресов. Подобъект каждого базового класса может иметь собственный уникальный адрес, и каждый такой адрес считается корректным адресом всего объекта. (В плохо спроектированных иерархиях с одиночным наследованием объект тоже может иметь два действительных адреса. См. «Совет 70».)

```
class A { /* . . . */ };
class B { /* . . . */ };
class C : public A, public B { /* . . . */ };
// ...
C *cp = new C;
A *ap = cp; // правильно
B *bp = cp; // правильно
```

В приведенном выше примере память для подобъекта `B` объекта `C`, скорее всего, будет располагаться с фиксированным смещением – «дельтой» – от начала объекта `C`. Преобразование указателя `cp` на производный класс в указатель `B *` поэтому сводится к корректировке `cp` на дельту. Это преобразование безопасно относительно типов и автоматически выполняется компилятором.

Возможность существования нескольких адресов у объекта заставляет `C++` точно определять семантику сравнения указателей:

```
if( bp == cp ) // ...
```

В этом предложении мы не спрашиваем, состоят ли два указателя из одной и той же комбинации битов. Нет, смысл его другой: «Указывают ли эти указатели на один и тот же объект?» Реализация такой проверки может быть довольно сложной, но все равно остается эффективной, безопасной и автоматической. Вероятно, компилятор реализует сравнение указателей как-то так:

```
if( bp ? (char *)bp-delta==(char *)cp : cp==0 )
```

Для преобразования, учитывающего дельта-арифметику над адресами объектов классов, годятся приведения как нового, так и старого стиля. Однако, в отличие от рассмотренных выше преобразований, нет никакой гарантии, что в результате приведения получится действительный адрес. (Оператор `dynamic_cast` дает такую гарантию, но с ним связаны другие проблемы. См. советы 97, 98 и 99.)

```
B *gimmeAB();
```

```
bp = gimmeAB();
cp = static_cast<C *>(bp); cp = (C *) bp;
typedef C *CP;
cp = CP( bp );
```

Все три приведения выполняют операции дельта-арифметики над `bp`, но результат будет правильным лишь, если объект `B`, на который указывает `bp`, есть часть объемлющего объекта `C`. Если это не так, то получится неверный адрес, как в случае такого вот «креативного» кода в стиле `C`:

```
cp = (C *) ((char *)bp-delta)
```

Оператор `reinterpret_cast` делает именно то, что подразумевает его название: интерпретирует комбинацию битов, переданную в качестве аргумента, как нечто иное, не модифицируя сами биты. Иначе говоря, он «отключает» дельта-арифметику. (Если быть совершенно точным, то стандарт говорит, что такое приведение зависит от реализации, но принято считать, что речь идет об «отключении прохода по иерархии». Тем не менее, стандарт не гарантирует именно такого поведения, поэтому `reinterpret_cast` может изменить битовую структуру указателя.)

```
cp = reinterpret_cast<C *>(bp); // да я хочу получить дамп памяти ...
```

Все описанные выше приведения требуют, чтобы объект, на который направлен указатель типа `C *`, принял на себя больше ответственности, чем обещает его интерфейс. Проект плох изначально, поскольку мы слишком мало знаем о возможностях объекта и используем статическое приведение типа, чтобы заставить его сыграть роль, к которой он, возможно, не готовился. Лучше всего избегать статических приведений объектов классов. Позже я приведу аргументы в пользу того, что и динамических приведений следует избегать. Вот тогда картина будет полной.

Совет 39. Приведение неполных типов

У неполных типов нет определения, тем не менее на них можно объявлять указатели и ссылки, а также функции, которые принимают аргументы и возвращают результат в виде таких типов. Эта общепринятая и полезная практика:

```
class Y;
class Z;
Y *convert( Z * );
```

Проблема возникает тогда, когда программист заходит слишком далеко; невведение – благо только до определенного предела:

```
Y *convert( Z *zp )
{ return reinterpret_cast<Y *>(zp); }
```

Здесь применение `reinterpret_cast` необходимо, поскольку у компилятора нет никакой информации о взаимосвязи между типами `Y` и `Z`. Поэтому лучшее, что он может предложить, – «интерпретировать» комбинацию битов, составляющую указатель на `Z`, как указатель на `Y`. Иногда это может даже сработать:

```
class Y { /* . . . */ };
class Z : public Y { /* . . . */ };
```

Вполне может статься, что подобъект *Y* базового класса, содержащийся в объекте *Z*, имеет тот же адрес, что и весь объект в целом. Но в будущем такое положение может измениться, и в ходе сопровождения приведение перестанет быть корректным. (См. «Совет 38» и «Совет 70».)

```
class X { /* . . . */ };
class Z : public X, public Y { /* . . . */ };
```

Использование `reinterpret_cast`, скорее всего, отключает дельта-арифметику, поэтому мы получаем неверный адрес *Y*.

На самом деле, `reinterpret_cast` – это не единственная возможность, поскольку в нашем распоряжении есть еще и приведения в старом стиле. На первый взгляд, это может показаться даже более удачным решением, поскольку при таких приведениях дельта-арифметика выполняется, если у компилятора достаточно информации. Однако в действительности эта гибкость лишь усугубляет проблему, поскольку выглядящие одинаковыми преобразования могут давать совершенно разные результаты в зависимости от того, какая информация доступна компилятору:

```
Y *convert( Z *zp )
{ return (Y *)zp; }
// ...
class Z : public X, public Y { // ...
// ...
Z *zp = new Z;
Y *yp1 = convert( zp );
Y *yp2 = (Y *)zp;
cout << zp << ' ' << yp1 << ' ' << yp2 << endl;
```

Значение *yp1* будет соответствовать *zp* или *yp2* в зависимости от того, находится ли определение `convert` до или после определения класса *Z*.

Ситуация может стать намного сложнее, если `convert` – шаблонная функция, конкретизируемая в разных объектных файлах. В таком случае результат приведения может зависеть от причуд компоновщика (см. «Совет 11»).

В данном случае предпочтительнее использовать `reinterpret_cast`, а не приведения в старом стиле, поскольку, если уж результат будет неправильным, то при любых обстоятельствах. Но я бы воздержался и от того, и от другого.

Совет 40. Приведения в старом стиле

Не пользуйтесь приведениями в старом стиле. Они слишком мощны и слишком просты в применении. Рассмотрим такой заголовочный файл:

```
// emp.h
// ...
const Person *getNextEmployee();
// ...
```

Он используется в приложении повсеместно, в том числе и в таком фрагменте:

```
#include "emp.h"
// ...
Person *victim = (Person *)getNextEmployee();
```

```
dealWith( victim );
// ...
```

Отметим, что любое отбрасывание константности потенциально опасно и не переносимо. Предположим, однако, что автор этого кода обладает большим даром провидения, чем все мы, и решил, что именно в этом случае приведение корректно и переносимо. Но код все равно неправилен по двум причинам. Во-первых, запрошенное преобразование гораздо сильнее, чем необходимо. Во-вторых, автор впал в ребяческий грех, понадеявшись на «вторичную семантику»: в коде предполагается, что наблюдаемое, но официально не подтвержденное поведение абстракции, выражаемой функцией `getNextEmployee`, будет поддержано и в будущем.

По сути дела, использование `getNextEmployee` в данном случае предполагает, что первоначальная реализация функции никогда не изменится. Разумеется, это не так. Вскоре программист, сопровождающий файл `emp.h`, поймет, что работники (`Employee`) – это не просто люди (`Person`), и исправит ошибку:

```
// emp.h
// ...
const Employee *getNextEmployee();
// ...
```

К сожалению, приведение по-прежнему остается допустимым, хотя смысл его теперь изменился: оно не просто отбрасывает константность объекта, но и меняет набор операций, применимых к этому объекту. Используя приведение, мы говорим компилятору, что знаем о типах больше, чем он. Изначально, может, так оно и было, но после изменения заголовочного файла, скорее всего, никто не удосужился просмотреть все места, где оно используется, и распоряжение, отданное компилятору, перестало быть разумным. Если бы мы воспользовались подходящим приведением в новом стиле, то компилятор сумел бы обнаружить изменение и выдал бы сообщение об ошибке:

```
#include "emp.h"
// ...
Person *victim = const_cast<Person *>(getNextEmployee());
dealWith( victim );
```

Отметим, что использование `const_cast`, хотя и, несомненно, лучше приведения в старом стиле, все же опасно. Мы по-прежнему полагаемся на предположение о том, что официально не заявленная – и, быть может, случайная – связь между функциями `getNextEmployee` и `dealWith`, которая и делает возможным применение `const_cast`, будет существовать и дальше.

Совет 41. Статические приведения

Под «статическими» мы понимаем, кто бы мог подумать, «не динамические» приведения типов. Под такое определение попадает не только оператор `static_cast`, но и `reinterpret_cast`, `const_cast`, а также приведения в старом стиле.

Основная проблема статических приведений заключается именно в их статичности. Используя такую конструкцию, мы просим компилятор согласиться

с нашим мнением о возможностях объекта, а не с тем, что он видит. Хотя часто статические приведения дают код, который в момент разработки был правилен, но он не адаптируется автоматически при последующих изменениях структуры типа объекта. Поскольку эти изменения обычно находятся далеко от места приведения, то человек, сопровождающий программу, не всегда модифицирует соответствующий код. В то же время дополнительный эффект приведения типов состоит в отключении диагностики, которую компилятор в противном случае не преминул бы выдать.

Приведения – необязательно зло, но прибегать к ним нужно, соблюдая умеренность, так, чтобы изменение кода, удаленного от места приведения, не делало последнее некорректным. С практической точки зрения, из этих требований вытекает, что в общем случае следует избегать приведения абстрактных типов данных и – в особенности – абстрактных типов в иерархии наследования.

Рассмотрим следующую простую иерархию:

```
class B {
public:
    virtual ~B();
    virtual void op1() = 0;
};
class D1 : public B {
public:
    void op1();
    void op2();
    virtual int thisop();
};
```

С ней ассоциирована функция, которая служит фабрикой для создания объектов, производных от класса B. Первоначально в иерархии мог быть всего один производный класс, поэтому реализация тривиальна:

```
B *getAB() { return new D1; }
```

Но затем самому автору или сопровождающему мог понадобиться доступ к определенной в классе D1 функциональности объекта, возвращенного функцией `getAB`. Правильный подход в этом случае, перепроектировать программу так, чтобы тип объекта был известен статически. Если это невозможно или нецелесообразно, то можно прибегнуть к оператору `dynamic_cast` (после самокритичного анализа). Использование статического приведения почти никогда не приводит к добру, что мы и наблюдаем ниже:

```
B *bp = getAB();
D1 *d1p = static_cast<D1 *>(bp);
d1p->op1();
d1p->op2();
int a = d1p->thisop();
```

Этот код работает только потому, что возвращаемый объект действительно имеет тип D1. Но такое везение долго не продлится, и в иерархию рано или поздно будет добавлен новый класс вместе с обновленной фабрикой:

```
class D2 : public B {
public:
```

```

void op1();
void op2();
virtual char thatop();
};
// ...
B *getAB() {
    if( rand() & 1 )
        return new D1;
    else
        return new D2;
}

```

Обратите внимание, что эти изменения вполне могли затронуть совсем другую часть программы, в которую не часто заглядывает человек, сопровождающий код, который содержит статическое приведение. Остается только надеяться, что из-за модификации функции `getAB` придется хотя бы перекомпилировать этот код, но даже это не гарантируется. Да и если код все же будет перекомпилирован, статическое приведение типов все равно подавляет диагностику компилятора. Почти ничего нельзя сказать о том, как поведет себя этот код, если `getAB` вернет объект типа `D2`, но вполне возможно, что он даже будет кое-как работать. Ниже в комментариях описано обычно наблюдаемое поведение:

```

B *bp = getAB(); // возвращает D2
D1 *d1p = static_cast<D1 *>(bp); // делаем вид, что D2 - это D1
d1p->op1(); // #1: вызывается D2::op1!
d1p->op2(); // #2: вызывается D1::op2!!
int a = d1p->thisop(); // #3: вызывается D2::thatop!!!

```

Несмотря на отсутствие гарантий такого поведения, строка с меткой `#1`, вероятно, будет работать «правильно». Но, конечно, лучше бы вызывать функцию `op1` через интерфейс базового класса, так как в этом случае правильное поведение гарантируется.

Строка с меткой `#2` более проблематична. Это вызов не виртуальной функции-члена класса `D1`. Увы, мы запрашиваем ее у объекта класса `D2`, а это приводит к неопределенному поведению во время выполнения. Может даже и сработать.

Но самыми большими неприятностями грозит строка с меткой `#3`. Статически мы вызываем виртуальную функцию-член `thisop` класса `D1`, которая возвращает `int`. Динамически же вызывается функция-член `thatop` класса `D2`, возвращающая `char`. Если этот код и не вызовет аварийного завершения программы, то мы попытаемся скопировать `char` в `int`.

Применение статического приведения типов часто означает, по меткому наблюдению Скотта Мейерса, что «нарушены договоренности между вами и компилятором». По сути дела, статическое приведение не только требует от компилятора каких-то действий «потому что я так сказал» (подобный тон при разговоре с человеком гарантирует конец всякого полезного общения), но и является признаком неуважения к открытому интерфейсу, предлагаемому абстрактным типом данных, который вы приводите к другому. Конечно, отыскание продуманного решения с учетом объявленных возможностей объекта требует больше времени и умения, чем грубое приведение типа, зато результатом будет более стабильный, переносимый и удобный для использования код или интерфейс.

Совет 42. Инициализация формальных аргументов временными объектами

Рассмотрим класс `String`, в котором объявлены операторы проверки на равенство и неравенство:

```
class String {
public:
    String( const char * = "" );
    ~String();
    friend bool operator ==( const String &, const String & );
    friend bool operator !=( const String &, const String & );
    // . . .
private:
    char *s_;
};
inline bool
operator ==( const String &a, const String &b )
    { return strcmp( a.s_, b.s_ ) == 0; }

inline bool
operator !=(const String &a, const String &b )
    { return !(a == b); }
```

Отметим, что в данном конкретном случае используется не-`explicit` конструктор с одним аргументом и операторы не являются членами класса. Тем самым мы приглашаем пользователей оценить достоинства неявных преобразований для упрощения своего кода:

```
String s( "Hello, World!" );
String t( "Yo!" );
if( s == t ) {
    // ...
}
else if( s == "Howdy!" ) { // неявное преобразование
    // ...
}
```

Проверка первого условия `s == t` эффективна. Два формальных аргумента-ссылки функции `operator ==` инициализируются соответственно `s` и `t`, после чего для выполнения сравнения вызывается функция `strcmp`. Если компилятор решит встроить `operator ==` (а он, вероятно, так и сделает, если не включен режим отладки), то во время выполнения все сведется к простому вызову `strcmp`.

Проверка второго условия `s == «Howdy!»` не так эффективна, хотя и правильна. Чтобы инициализировать второй аргумент `operator ==`, компилятор должен создать временный объект `String` и инициализировать его строковым литералом «`Howdy!`». После возврата из функции временный объект должен быть уничтожен. По существу, этот код эквивалентен следующему:

```
String temp( "Howdy!" );
bool result = operator ==( s, temp );
temp.~String();
if( result ) {
    // ...
}
```

В данном случае удобство неявного преобразования может компенсировать дополнительные накладные расходы, поскольку реализация как класса `String`, так и пользовательского кода оказывается короткой и прозрачной.

Однако, по крайней мере, в двух случаях неявное преобразование неприемлемо. Первый — это, конечно, когда такие преобразования применяются очень интенсивно и вызывают заметное снижение быстродействия или увеличивают потребление памяти. Второй случай возникает, когда наличие преобразования из `const char *` в `String` приводит к неоднозначности и сложности в других местах класса `String`, и его проектировщик решает сделать конструктор `String` `explicit`.

Эту проблему легко решить перегрузкой операторов проверки на равенство объектов класса `String`:

```
class String {
public:
    explicit String( const char * = "" );
    ~String();
    friend bool operator ==( const String &, const String & );
    friend bool operator !=( const String &, const String & );
    friend bool operator ==( const String &, const char * );
    friend bool operator !=( const String &, const char * );
    friend bool operator ==( const char *, const String & );
    friend bool operator !=( const char *, const String & );
    // ...
};
```

Теперь для любой допустимой комбинации аргументов существует точное соответствие, и компилятору не нужно генерировать временные объекты `String`. К сожалению, сам класс `String` при этом становится тяжелее и труднее для понимания, поэтому такой подход к оптимизации рекомендуется лишь, если профилирование доказало его целесообразность.

Начинающие программисты на C++ часто допускают типичную ошибку: передают объекты по значению в ситуации, где лучше бы передавать их по ссылке. Рассмотрим функцию, которая принимает аргумент типа `String`:

```
String munge( String s ) {
    // сделать что-то с s ...
    return s;
}
// ...
String t( "Munge Me" );
t = munge( t );
```

Трудно найти хорошие слова в адрес этого кода, но для программистов, только приступающих к изучению C++, он не редкость. При вызове `munge` необходимо вызвать конструктор копирования для инициализации формального аргумента `s`, затем конструктор копирования для создания возвращаемого значения и, наконец, деструктор для уничтожения локальной переменной `s`. Поскольку мы присваиваем результат обработки `t` функцией `munge` самой этой переменной `t`, то хочется надеяться, что компилятор поймет это и сделает оператор присваивания «пустышкой». Напрасные надежды. Компилятор обязан поместить возвращенное

`munge` значение во временную переменную (которая позже должна быть уничтожена), поэтому присваивание не будет оптимизировано. Таким образом, мы имеем шесть вызовов функций.

Гораздо лучше переписать функцию `munge` так, чтобы она принимала ссылку на переменную типа `String`:

```
void munge( String &s ) {
    // сделать что-то с s ...
}
// ...
munge( t );
```

Всего один вызов функции. У этих двух версий несколько различающаяся семантика: теперь любое изменение `s` внутри `munge` отражается на самом фактическом аргументе, а не на возвращаемом значении. (Это различие может стать существенным, если внутри `munge` произойдет исключение или `munge` вызовет другую функцию, которая ссылается на `t`.) Но в целом сложность уменьшилась, код стал компактнее и быстрее.

Передача по ссылке особенно важна при реализации шаблонов, так как невозможно заранее предсказать, во что обойдется передача аргумента в случае той или иной конкретизации:

```
template <typename T>
bool operator >( const T &a, const T &b )
{ return b < a; }
```

Накладные расходы при передаче аргумента по ссылке фиксированы, невысоки и не зависят от типа аргумента. Не исключено, что аргументы некоторых типов, например, встроенных или небольших простых классов эффективнее передавать по значению. Если это так важно, шаблон можно перегрузить (если это шаблон функции) или специализировать (если это шаблон класса).

Есть случаи, когда принято передавать аргументы по значению. Так, в стандартной библиотеке шаблонов C++ по значению передаются «функциональные объекты». (Функциональным объектом называется объект класса, в котором перегружен оператор вызова функции. Это такой же объект, как и любой другой, но при работе с ним допустим синтаксис вызова функции.)

Например, можно объявить функциональный объект, который будет выступать в роли «предиката», то есть функции, выносящей вердикт «да или нет» о своем аргументе:

```
struct IsEven : public std::unary_function<int,bool> {
    bool operator ()( int a )
        { return !(a & 1); }
};
```

У объекта `IsEven` нет ни данных-членов, ни виртуальных функций, ни конструктора, ни деструктора. Передача такого объекта по значению обходится недорого (а часто и вовсе бесплатно). На самом деле, при работе с STL считается хорошим тоном передавать функциональные объекты как анонимные временные объекты:

```
extern int a[n];
int *thatsOdd = partition( a, a+n, IsEven() );
```

Выражение `IsEven()` создает анонимный временный объект типа `IsEven`, который передается по значению алгоритму `partition` (см. «Совет 43»). Конечно, такое соглашение подразумевает, что функциональные объекты, используемые совместно с STL, невелики и могут быть эффективно переданы по значению.

Совет 43. Время жизни временных объектов

При некоторых обстоятельствах компилятор вынужден создавать временные объекты. Стандарт говорит, что время жизни такого объекта простирается от точки создания до конца самого внешнего объемлющего выражения (в стандарте это называется «полным выражением»). Типичная проблема заключается в том, что программа зависит от существования уже уничтоженных временных объектов:

```
class String {
public:
    // ...
    ~String()
    { delete [] s_; }
    friend String operator +( const String &, const String & );
    operator const char *() const
    { return s_; }
private:
    char *s_;
};
// ...
String s1, s2;
printf( "%s", (const char *) (s1+s2) ); // #1
const char* p = s1+s2; // #2
printf( "%s", p ); // #3
```

При реализации бинарного оператора `+` в классе `String` часто требуется, чтобы возвращаемое значение сохранялось во временной переменной. Так обстоит дело в обоих приведенных выше примерах. В предложении с меткой #1 результат вычисления `s1+s2` помещается во временный объект, который затем преобразуется в `const char *` и передается функции `printf`. После возврата из `printf` временный объект типа `String` уничтожается. Код работает, потому что временный объект существует на протяжении всего времени, пока используется.

В предложении #2 результат вычисления `s1+s2` помещается во временный объект, который, как и раньше, преобразуется в `const char *`. Но разница в том, что он уничтожается сразу после инициализации указателя `p`. Когда `p` передается `printf`, он указывает на буфер, принадлежащий уже разрушенному объекту `String`. Неопределенное поведение.

Самое неприятное последствие этой ошибки в том, что программа может продолжать работать (по крайней мере, во время тестирования). Например, когда временный объект `String` удаляет свой буфер для хранения символов, оператор удаления массива может просто пометить память как неиспользуемую, не изменяя ее содержимого. Если эта память не задействована для другой цели между строками #2 и #3, то будет казаться, что программа работает нормально. Но стоит использовать этот код в многопоточном приложении, как он начнет время от времени «рушиться».

Лучше воспользоваться сложным выражением или явно объявить временный объект с увеличенным временем жизни:

```
String temp = s1+s2;  
const char *p = temp;  
printf( "%s", p );
```

Заметим, однако, что ограниченное время жизни временных объектов часто можно обратить себе на пользу. При использовании стандартной библиотеки очень распространена практика адаптации компонентов с помощью функциональных объектов:

```
class StrLenLess  
: public binary_function<const char *, const char *, bool> {  
public:  
    bool operator() ( const char *a, const char *b ) const  
        { return strlen(a) < strlen(b); }  
};  
// ...  
sort( start, end, StrLenLess() );
```

Выражение `StrLenLess()` заставляет компилятор сгенерировать анонимный временный объект, который существует до возврата из алгоритма `sort`. Альтернатива – воспользоваться явно поименованной переменной – длиннее и засоряет текущую область видимости бесполезным именем (см. «Совет 48»).

```
StrLenLess comp;  
sort( start, end, comp );  
// comp все еще в области видимости ...
```

Еще одна неприятность, связанная со временем жизни временных объектов, может возникнуть в унаследованном коде, написанном для компилятора, созданного до принятия стандарта. Раньше не было четких правил определения времени жизни временных объектов. В результате некоторые компиляторы уничтожали такие объекты в конце того блока, где они появились на свет, другие – в конце предложения и так далее. Переделывая унаследованный код, внимательно относиться к внешне незаметным изменениям семантики из-за изменившегося времени жизни временных объектов.

Совет 44. Ссылки и временные объекты

Ссылка – это псевдоним своего инициализатора (см. «Совет 7»). После инициализации ссылку можно употреблять везде, где употребляется сам инициализатор, без изменения семантики. Ну почти...

```
int a = 12;  
int &r = a;  
++a; // то же, что ++r  
int *ip = &r; // то же, что &a  
int &r2 = 12; // ошибка! 12 – это литерал
```

Ссылка должна быть инициализирована lvalue; грубо говоря, это означает, что инициализатор должен иметь не только значение, но и адрес (см. «Совет 6»). В случае ссылок на константы ситуация несколько осложняется. Инициализатор

ссылки на константу по-прежнему должен быть lvalue, но компилятор в этом случае имеет право создать lvalue из инициализатора, который lvalue не является.

```
const int &r3 = 12; // правильно.
```

Ссылка `r3` – это псевдоним для анонимного временного значения типа `int`, которое было неявно создано и инициализировано компилятором. Обычно время жизни созданных компилятором временных объектов ограничено самым внешним объемлющим выражением. Однако в данном случае стандарт гарантирует, что временный объект будет существовать столько же, сколько и ссылка, которую он инициализирует. Отметим, что временный объект никак не связан с породившим его инициализатором, поэтому следующий неприглядный и опасный код, к счастью, не отразится на значении литерала 12:

```
const_cast<int &>(r3) = 11; // присвоить значение временному объекту
                          // или умереть ...
```

Компилятор также изготовит временный объект для инициализатора-lvalue, тип которого отличается от типа инициализируемой им ссылки:

```
const string &name = "Fred"; // правильно.
short s = 123;
const int &r4 = s; // правильно.
```

Здесь мы испытываем некоторое семантическое затруднение, так как идея ссылки как псевдонима инициализатора оказывается под угрозой. Легко забыть, что инициализатор ссылки – это на самом деле анонимный временный объект, а не то, что мы видим в исходном тексте. Например, любое изменение переменной `s` типа `short` никак не отразится на ссылке `r4`:

```
s = 321; // r4 still == 123
const int *ip = &r4; // это не &s
```

Действительно ли это проблема? Может стать таковой с вашей помощью. Взгляните на следующую попытку достичь переносимости за счет применения `typedef`. Быть может, существует глобальный для всего проекта заголовочный файл, в котором подменяются платформенно-зависимые имена для типов целых разных размеров:

```
// Заголовочный файл big/sizes.h
typedef short Int16;
typedef int Int32;
// ...
```

```
// Заголовочный файл small/sizes.h
typedef int Int16;
typedef long Int32;
// ...
```

(Обратите внимание, что мы не пользуемся директивами `#if`, чтобы вписать все `typedef`'ы для всех платформ в один файл. Этот непродуманный поступок способен отравить вам выходные, подпортить репутацию и сломать жизнь. См. «Совет 27».) В этом нет ничего плохого, если все разработчики будут пользоваться именно такими именами. Увы, они поступают так далеко не всегда:

```
#include <sizes.h>
```

```
// . . .
Int32 val = 123;
const int &theVal = val;
val = 321;
cout << theVal;
```

Если мы ведем разработку на «большой» платформе, то `theVal` – это псевдоним `val`, и мы выводим в `cout` значение 321. Если позже мы решим воспользоваться предполагаемой независимостью от платформы и перекомпилируем код для «маленькой» платформы, то `theVal` уже будет ссылаться на временный объект, и в поток будет выведено 123. Изменение семантики произошло «молча» и обычно не так заметно, как изменившаяся выходная информация.

Еще одна потенциальная проблема заключается в том, что инициализация ссылки на константу может привести к затруднениям со временем жизни временных объектов. Мы уже видели, что компилятор обеспечит таким временным объектам столь же долгую жизнь, как инициализированным ими ссылкам, и вроде бы никакой опасности не предвидится. Но взгляните на следующую простую функцию:

```
const string &
select( bool cond, const string &a, const string &b ) {
    if( cond )
        return a;
    else
        return b;
}
// . . .
string first, second;
bool useFirst = false;
// . . .
const string &name = select( useFirst, first, second ); // правильно
```

На первый взгляд, абсолютно безобидная функция. Она ведь просто возвращает один из своих аргументов. Но проблема именно в предложении `return`. Вот другая функция, в которой проблема проявляется более наглядно:

```
const string &crashAndBurn() {
    string temp( "Fred" );
    return temp;
}
// . . .
const string &fred = crashAndBurn();
cout << fred; // беда!
```

Здесь мы явно возвращаем ссылку на локальную переменную. После возврата локальная переменная уже не существует, так что у пользователя этой функции остается ссылка на уничтоженный объект. К счастью, большинство компиляторов предупредят о такой ситуации. Но в примере ниже на предупреждение не рассчитывайте, компилятор просто не в состоянии его выдать:

```
const string &name = select( useFirst, "Joe", "Besser" );
cout << name; // беда!
```

Проблема в том, что второй и третий аргументы функции `select` – это ссылки на константы, поэтому они будут инициализированы временными строковыми

объектами. Хотя эти временные строки и не локальны по отношению к функции `select`, но существовать будут лишь до конца самого внешнего объемлющего выражения, а оно кончается после возврата из `select`, но до того, как возвращенное значение используется. Возможный выход из положения – погрузить вызов функции в большее выражение:

```
cout << select( useFirst, "Joe", "Besser" ); // работает, но очень хрупко
```

Такой код будет работать, если его писал эксперт, но сломается в руках новичка.

Безопаснее избегать возврата формального аргумента, который является ссылкой на константу. Для функции `select` в нашем распоряжении по меньшей мере два разумных варианта. Стандартный тип `string` не является полиморфным (то есть не имеет виртуальных функций), а потому можно предполагать, что ссылочные аргументы связаны с объектами типа `string`, а не производного от него. Таким образом, можно возвращать строку по значению, не опасаясь срезки, но при этом неизбежны накладные расходы на вызов конструктора копирования `string` для инициализации возвращаемого значения:

```
string
select( bool cond, const string &a, const string &b ) {
    if( cond )
        return a;
    else
        return b;
}
```

Другой вариант – объявить формальные аргументы ссылками на не-константы, тогда компилятор выдаст ошибку, если для их инициализации потребуется создавать временные объекты. В результате предыдущий пример просто перестанет компилироваться:

```
string &
select( bool cond, string &a, string &b ) {
    if( cond )
        return a;
    else
        return b;
}
```

Ни то, ни другое решение не кажется особо привлекательным, но все-таки это лучше, чем оставить в программе возможность для проявления ошибки.

Совет 45. Неоднозначность при использовании `dynamic_cast`

Ясное дело, вам стыдно. Вы даже не хотите обсуждать эту тему с коллегами. Возможно, это неблагоприятно отразится на личных взаимоотношениях с ними. Но, если вас приперли к стенке, если вы оказались один на один с плохо спроектированным модулем, столкнулись с невыполнимыми требованиями, выставленными начальством, и необходимостью закончить все к завтрашнему дню, то, может быть, имеет смысл обратиться к `dynamic_cast`.

Предположим, что вам нужно определить, соответствует ли конкретный объект, представляющий экран, экрану для ввода или какому-нибудь другому. Проблема в том, что вы вклинились в какой-то код общего характера, относящийся к экранам любого вида. Первое ваше побуждение – расширить интерфейс экранных типов, чтобы они предоставляли необходимую информацию.

```
class Screen {
public:
    //...
    virtual bool isEntryScreen() const
    { return false; }
};
class EntryScreen : public Screen {
public:
    bool isEntryScreen() const
    { return true; }
};
// . . .
Screen *getCurrent();
// . . .
if( getCurrent()->isEntryScreen() )
    // . . .
```

Но этот подход позволил бы задавать слишком «интимные» вопросы объекту класса Screen. Базовый класс Screen явно предлагает задать такой вопрос: «А не являешься ли ты объектом EntryScreen?» Ну, раз уж шлюзы открыты, то ничто не помешает тем, кто будет сопровождать программу, задать и еще более личные вопросы (см. «Совет 98»):

```
class Screen {
public:
    //...
    virtual bool isEntryScreen() const
    { return false; }
    virtual bool isPricingScreen() const
    { return false; }
    virtual bool isSwapScreen() const
    { return false; }
    // и так до бесконечности ...
};
```

Конечно, наличие такого интерфейса почти наверняка гарантирует, что его будут использовать:

```
// ...
if( getCurrent()->isEntryScreen() )
    // ...
else if( getCurrent()->isPricingScreen() )
    // ...
else if( getCurrent()->isSwapScreen() )
    // ...
```

А это уже почти предложение switch, только более медленное и неудобное для сопровождения. Меньшее зло – сцепить зубы и все-таки один раз воспользоваться оператором dynamic_cast. Хочется надеяться, что приведение будет дос-

таточно глубоко упрятано, чтобы не стать примером для подражания, и когда-нибудь во время основательной переработки кода будет удалено:

```
if( EntryScreen *es = dynamic_cast<EntryScreen *>(sp) ) {
    // сделать что-то с экраном для ввода ...
}
```

Если приведение завершилось успешно, то `es` будет указывать на `EntryScreen`, и это может означать, что объект действительно принадлежит классу `EntryScreen` или является подобъектом более специализированного объекта. Но что означает неудачное приведение?

Оператор `dynamic_cast` может возвращать нуль по одной из четырех причин. Во-первых, приведение может быть некорректным. Если `sp` не указывает на объект класса `EntryScreen` или производного от него, то приведение не пройдет. Во-вторых, если `sp` равно нулю, то нулевым будет и результат приведения. В-третьих, приведение к недоступному базовому классу или из такого класса тоже закончится неудачей. И, наконец, причиной провала может стать неоднозначность.

Неоднозначности, возникающие в ходе преобразования типов, редко встречаются в хорошо спроектированных иерархиях. Но если иерархия построена неудачно или к ней неправильно обращаются, то можно попасть в беду.

На рис. 4.4 показана простая иерархия с множественным наследованием. Предположим, что тип `A` полиморфен (имеет виртуальную функцию) и что используется только открытое наследование. В таком случае у объекта `D` будет два подобъекта типа `A`, то есть по меньшей мере один из них является неvirtуальным базовым классом:

```
D *dp = new D;
A *ap = dp; // ошибка! неоднозначность
ap = dynamic_cast<A *>(dp); // ошибка! неоднозначность
```

Инициализация `ap` неоднозначна, так как может относиться к любому из двух разумных адресов `A`. Но поскольку мы имеем адрес одного из двух подобъектов `A`, ссылка на все остальные подтипы в иерархии уже будет однозначной:

```
B *bp = dynamic_cast<B *>(ap); // работает
C *cp = dynamic_cast<C *>(ap); // работает
```

На какой бы подобъект `A` ни указывал `ap`, преобразование его в указатель на подобъекты `B` или `C` либо на весь объект `D` однозначна, поскольку полный объект содержит лишь по одному экземпляру каждого такого подобъекта.

Интересно отметить, что если бы оба подобъекта `A` были виртуальными базовыми классами, то неоднозначность не возникла бы, поскольку объект `D` содержал бы единственный подобъект `A`:

```
D *dp = new D;
A *ap = dp; // правильно, неоднозначности нет
ap = dynamic_cast<A *>(dp); // правильно, неоднозначности нет
```

Но ее можно снова ввести, чуть усложнив иерархию, как показано на рис. 4.5. В этой модифицированной иерархии нет прежней неоднозначности, поскольку объект `D` все еще содержит единственный подобъект `A`:

```
A *ap = new D; // неоднозначности нет
```

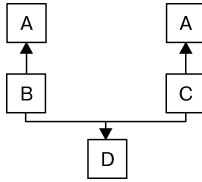


Рис. 4.4. Иерархия с множественным наследованием без виртуальных базовых классов. Объект класса D содержит два подобъекта класса A

Зато теперь неоднозначность проявляется по-другому:

```
E *ep = dynamic_cast<E *>(ap); // неоднозначность!
```

Указатель `ap` можно преобразовать в один из двух подобъектов E. Эту неоднозначность можно обойти, если выразить свои намерения яснее:

```
E *ep = dynamic_cast<B *>(ap); // работает
```

Объект D содержит единственный подобъект B, поэтому преобразование `A * в B *` однозначно, а для последующего преобразования `B * в его открытый базовый класс` не нужно приведения. Заметим, однако, что такое решение требует включения в код детальной информации о структуре иерархии ниже классов A и E. Лучше упростить иерархию, чтобы избежать появления динамической неоднозначности.

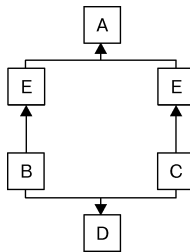


Рис. 4.5. Иерархия с множественным виртуальным и неvirtуальным наследованием нескольких подобъектов одного типа. Полный объект D содержит единственный подобъект A с двумя подобъектами E

Раз уж мы рассматриваем оператор `dynamic_cast`, то должны остановиться на некоторых тонкостях его семантики. Во-первых, этот оператор не обязательно динамичен, то есть может и не выполнять проверки во время выполнения. Когда выполняется приведение указателя (или ссылки) на производный класс к одному из его открытых базовых классов, проверка не нужна, поскольку компилятор может установить допустимость приведения статически. Разумеется, в таком случае применять `dynamic_cast` вообще не имеет смысла, так как преобразование из производного класса в открытый базовый относится к числу предопределенных. (Хотя правила языка в этом отношении могут показаться излишними, часто это

упрощает программирование шаблонов, когда типы, которыми мы манипулируем, заранее неизвестны.)

Кроме того, допустимо приводить указатель или ссылку на полиморфный тип к `void *`. В этом случае результатом будет начало «самого производного» (наиболее глубокого расположенного в иерархии наследования) или полного объекта, на который направлен указатель. Конечно, мы так и не знаем, на что указываем, но хотя бы понимаем, куда...

Совет 46. Контравариантность

Правила преобразования указателей на члены класса логичны, но зачастую противоречат интуиции. Знакомство с реализацией указателей на члены поможет прояснить ситуацию.

Указатель направлен на некоторую область памяти; он содержит адрес, который нужно разыменовать, чтобы получить доступ к содержимому памяти. (В коде ниже встречаются два порочных приема: открытый член данных и сокрытие не-виртуальной функции базового класса. Я поступил так только для иллюстрации, не нужно расценивать это как завуалированное приглашение поступать подобным образом. См. «Совет 8» и «Совет 71».)

```
class Employee {
public:
    double level_;
    virtual void fire() = 0;
    bool validate() const;
};
class Hourly : public Employee {
public:
    double rate_;
    void fire();
    bool validate() const;
};
// ...
Hourly *hp = new Hourly, h;
// ...
*hp = h;
```

Отметим, что адрес члена данных конкретного объекта – это еще не указатель на член. Это лишь простой указатель, ссылающийся на конкретный член конкретного объекта:

```
double *ratep = &hp->rate_;
```

Указатель на член – это вообще не указатель. Указатель на член не содержит никакого адреса и не ссылается ни на какой конкретный объект или ячейку памяти. А ссылается он на конкретный член неопределенного объекта. Поэтому, чтобы разыменовать указатель на член, необходим объект:

```
double Hourly::*hvalue = &Hourly::rate_;
hp->*hvalue = 1.85;
h.*hvalue = hp->*hvalue;
```

Операторы `.*` и `->*` – это бинарные операторы, которые разыменовывают указатель на член при наличии объекта класса или указателя на объект соответ-

венно (см., «Совет 15» и «Совет 17»). Указатель на член `hvalue` был инициализирован так, чтобы он указывал на член `rate_` класса `Hourly`, а затем разыменован для доступа к члену `rate_` конкретного объекта `h` класса `Hourly` или объекта, на который указывает `hp`.

Указатель на член данных обычно реализуется в виде смещения. То есть при взятии адреса члена данных, как в случае `&Hourly::rate_`, мы получаем число байтов от начала объекта класса, которому принадлежит данный член. Как правило, это значение увеличивается на 1, оставляя значение 0 для представления нулевого указателя на член данных. Разыменование указателя на член данных обычно заключается в прибавлении хранящего в указателе смещения (уменьшенного на 1) к адресу объекта класса. Затем получающийся указатель разыменовывается для получения доступа к нужному члену объекта класса. Например, выражение

```
h.*hvalue = 1.85
```

могло бы быть оттранслировано так:

```
*(double *) ((char *) &h + (hvalue-1)) = 1.85
```

Рассмотрим еще один указатель на член данных:

```
double Employee::*evaluate = &Employee::level_;
Employee *ep = hp;
```

Поскольку `Hourly` «является разновидностью» `Employee`, то можно разыменовывать указатель `evaluate` на член данных с помощью указателя на объект и того, и другого класса. Это хорошо известное предопределенное преобразование из производного класса в открытый базовый:

```
ep->*evaluate = hp->*evaluate;
```

Объект `Hourly` можно подставлять вместо `Employee`. Но попытка выполнить аналогичное преобразование с указателями на члены не проходит:

```
evaluate = hvalue; // ошибка!
```

Не существует преобразования из указателя на член производного класса в указатель на член его открытого базового класса. Зато обратное преобразование допустимо:

```
hvalue = evaluate; // правильно
```

Это явление называется «контравариантностью»; предопределенные преобразования для указателей на члены классов в точности противоположны их аналогам для обычных указателей на классы. (Не путайте контравариантность с ковариантными типами возвращаемых значений; см. «Совет 77».) Немного поразмыслив, вы поймете, какая логика стоит за этими противоречащими интуиции правилами. Поскольку `Hourly` «является разновидностью» `Employee`, то он содержит `Employee` в качестве подобъекта. Следовательно, любое смещение от начала `Employee` является также смещением и от начала `Hourly`. Напротив, некоторые смещения от начала `Hourly` уже недействительны для `Employee`. Отсюда вытекает, что указатель на член открытого базового класса можно безопасно преобразовать в указатель на член производного класса, но не наоборот:

```
T SomeClass::*mptr;
... ptr->*mptr ...
```

В этом фрагменте указатель `ptr` можно преобразовать в указатель на объект типа `SomeClass` или любого открыто наследующего ему. Указатель на член `mPtr` может содержать адрес члена `SomeClass` или адрес любого доступного базового класса `SomeClass`.

Контравариантность применима также к указателям на функции-члены. Это в такой же мере противоречит интуиции и вместе с тем не менее осмысленно:

```
void (Employee::*action1)() = &Employee::fire;
(hourly->*action1)(); // Hourly::fire
bool (Employee::*action2)() const = &Employee::validate;
(hourly->*action2)(); // Employee::validate
```

Реализации указателей на функции у разных производителей различаются, но обычно представляют собой небольшую структуру. В ней содержится информация, необходимая для того, чтобы отличать виртуальные функции от не виртуальных, а также другие платформенно-зависимые данные, относящиеся к деталям реализации структуры объектов при наследовании. Первый вызов из примера выше (через указатель `action1`) сводится к косвенному виртуальному вызову функции-члена `Hourly::fire`, поскольку `&Employee::fire` – это указатель на виртуальную функцию-член. Вызов через `action2` приводит к вызову функции `&Employee::validate`, так как `&Employee::validate` – указатель на не виртуальную функцию.

```
action2 = &Hourly::validate; // ошибка!
bool (Hourly::*action3)() = &Employee::validate; // правильно
```

И снова контравариантность. Запрещено присваивать адрес функции `validate` производного класса указателю на член базового класса, но можно инициализировать указатель на член производного класса адресом функции-члена базового класса. Как и в случае указателей на члены данных, причина кроется в безопасности доступа к членам. Реализация `Hourly::validate` может попытаться обратиться к членам данных (и функциям), которые отсутствуют в классе `Employee`. С другой стороны, все члены, доступные функции `Employee::validate`, доступны и в классе `Hourly`.



Глава 5. Инициализация

Семантика инициализации в C++ сложна, изобилует тонкостями и очень важна.

Причины сложности лежат очень глубоко. Программирование на C++ складывается в основном из использования классов для реализации абстрактных типов данных. По сути дела, мы расширяем базовый язык за счет включения в него новых типов. Мы связаны двумя обязательствами. С одной стороны, дизайн языка программирования должен обеспечивать создание удобных для использования, интегрируемых типов. С другой стороны, компилятор должен эффективно транслировать нашу реализацию абстрактных типов данных. Детали инициализации и копирования объектов классов очень важны для эффективного использования абстракций данных в промышленном коде.

Не менее, чем эффективность, важна, конечно, и корректность. Незнание сложной семантики инициализации в C++ может привести к неправильному использованию языка.

В этой главе мы рассмотрим, как реализуется инициализация и как убедить компилятор оптимизировать определенные пользователем операции инициализации и копирования. Кроме того, мы познакомимся с некоторыми типичными ошибками, возникающими из-за неправильного понимания семантики инициализации.

Совет 47. Не путайте инициализацию и присваивание

С технической точки зрения, присваивание и инициализация имеют мало общего. Это различные операции, и применяются они в разных ситуациях. Инициализация – это процесс превращения неформатированной памяти в объект. Когда речь идет об объекте класса, при этом могут быть задействованы внутренние механизмы для предоставления виртуальных функций и виртуальных базовых классов, информации о типе во время выполнения и другой зависящей от типа информации (см. «Совет 53» и «Совет 78»). Присваивание – это процесс замены существующего состояния готового объекта новым состоянием. Присваивание не отражается на внутренних механизмах, определяющих зависящее от типа поведение объекта. Присваивание никогда не применяется к неформатированной памяти.

Но если семантика конструирования путем копирования важна при одних обстоятельствах, то есть шанс, что при других обстоятельствах важна семантика присваивания копированием, и наоборот. Если забыть, что присваивание и инициализацию надо рассматривать совместно, возможны ошибки:

```
class SloppyCopy {  
public:
```

```

    SloppyCopy &operator =( const SloppyCopy & );
    // Примечание: компилятор генерирует SloppyCopy(const SloppyCopy &)
    // по умолчанию ...
private:
    T *ptr;
};

void f( SloppyCopy ); // передача по значению

SloppyCopy sc;
f( sc ); // псевдоним того, на что указывает указатель;
        // возможно, ошибка

```

Передача аргументов выполняется путем инициализации, а не присваивания; формальный аргумент функции `f` инициализируется фактическим аргументом `sc`. Инициализацию производит конструктор копирования класса `SloppyCopy`. Если в этом классе конструктор копирования не объявлен явно, то его сгенерирует компилятор. В данном случае созданная компилятором версия будет работать некорректно (см. «Совет 49» и «Совет 53»).

Существует идиома, в основе которой лежит предположение о том, что хотя конструирование и присваивание копированием – это разные операции, но они должны иметь сходную семантику:

```

extern T a, b;
b = a;
T c( a );

```

В этом фрагменте пользователь типа `T` ожидает, что значения `b` и `c` будут согласованы. Другими словами, для последующих операций должно быть безразлично, получил ли объект типа `T` свое значение в результате присваивания или инициализации. Это предположение о согласованности настолько глубоко укоренилось в сообществе программистов на C++, что даже стандартная библиотека опирается на него:

➤> gotcha47/rawstorage.h

```

template <class Out, class T>
class raw_storage_iterator
    : public iterator<output_iterator_tag,void,void,void,void> {
public:
    raw_storage_iterator& operator =( const T& element );
    // ...
protected:
    Out cur_;
};

template <class Out, class T>
raw_storage_iterator<Out, T> &
raw_storage_iterator<Out,T>::operator =( const T &val ) {
    T *elem = &cur_; // получить указатель на элемент
    new ( elem ) T(val); // оператор размещения и конструктор копирования
    return *this;
}

```

Класс `raw_storage_iterator` применяется для разметки неинициализированной памяти. Обычно оператор присваивания требует, чтобы оба его аргумента были инициализированными объектами, иначе может возникнуть пробле-

ма, когда во время присваивания будет произведена попытка «очистить» левый аргумент перед установкой нового значения. Например, если объект, которому присваивается новое значение, содержит указатель на буфер, выделенный из кучи, то оператор присваивания обычно сначала освобождает эту память. Если же объект не инициализирован, то удаление указателя может привести к неопределенному поведению:

>> gotcha47/rawstorage.cpp

```
struct X {
    T *t_;
    X &operator =( const X &rhs ) {
        if( this != &rhs )
            { delete t_; t_ = new T(*rhs.t_); }
        return *this;
    }
    // . . .
};
// . . .
X x;
X *buf = (X *)malloc( sizeof(X) ); // неформатированная память ...
X &rx = *buf; // грязный трюк ...
rx = x; // вероятно, ошибка!
```

Алгоритм `copy` из стандартной библиотеки копирует входную последовательность в выходную, вызывая для копирования каждого элемента оператор присваивания:

```
template <class In, class Out>
Out std::copy( In b, In e, Out r ) {
    while( b != e )
        *r++ = *b++; // присвоить исходный элемент целевому
    return r;
}
```

Применение `copy` к неинициализированному массиву объектов класса `X`, скорее всего, закончится катастрофой:

>> gotcha47/rawstorage.cpp

```
X a[N];
X *ary = (X *)malloc( N*sizeof(X) );
copy( a, a+N, ary ); // присваивание неформатированной памяти!
```

Присваивание несколько напоминает (но не идентично!) уничтожение с последующим вызовом конструктора копирования. Класс `raw_storage_iterator` реализует присваивание неформатированной памяти, интерпретируя присваивание как конструирование путем копирования, но без «уничтожения», которое может вызвать проблемы. Это будет работать только в предположении, что присваивание и конструирование копированием дают примерно одинаковые результаты.

>> gotcha47/rawstorage.cpp

```
raw_storage_iterator<X *, X> ri( ary );
copy( a, a+N, ri ); // работает!
```

Отсюда не следует, что проектировщик класса `X` должен быть хорошо знаком со всеми (возможно, непонятными и запутанными) деталями стандартной библио-

теки, чтобы реализовать свой класс правильно. Но он обязан знать общее идиоматическое предположение о согласованности присваивания и инициализации копированием. Абстрактный тип данных, который не поддерживает такую согласованность, не может эффективно использоваться совместно со стандартной библиотекой и будет не так полезен, как согласованный в этом смысле класс.

Часто программисты впадают в заблуждение, думая, что в следующем предложении как-то задействовано присваивание:

```
T d = a; // это не присваивание
```

Знак = в данном случае – не оператор присваивания, а указание на то, что `d` инициализируется значением `a`. И это правильно, иначе мы стали бы что-то присваивать неформатированной памяти (см. «Совет 56»).

Совет 48. Правильно выбирайте область видимости переменной

Одна из самых типичных ошибок при программировании на C и C++ – это использование неинициализированных переменных. Этой проблемы вообще не должно существовать! Отделение объявления переменной от ее инициализации редко дает какие-то преимущества:

```
int a;  
a = 12;  
string s;  
s = "Joe";
```

Это же просто глупо. Целое число будет иметь неопределенное значение до выполнения присваивания в следующей строке. Строка корректно инициализируется своим конструктором по умолчанию, но это значение сразу же затирается последующим оператором присваивания (см. также «Совет 51»). В обоих случаях инициализацию надо было совместить с объявлением:

```
int a = 12;  
string s( "Joe" );
```

Реальная опасность состоит в том, что в ходе сопровождения между объявлением неинициализированной переменной и первым присваиванием ей может быть вставлен какой-то код. Типичный сценарий не так тривиален, как в примере выше:

```
bool f( const char *s ) {  
    size_t length;  
    if( !s ) return false;  
    length = strlen( s );  
    char *buffer = (char *)malloc( length+1 );  
    // ...  
}
```

Здесь переменная `length` не только не инициализирована, она еще и должна была бы быть константой. Автор этого кода забыл, что в C++, в отличие от C, объявление является предложением; точнее это «предложение объявления», так что объявление может находиться в любом месте, где допустимо предложение:

```
bool f( const char *s ) {
    if( !s ) return false;
    const size_t length = strlen( s );
    char *buffer = (char *)malloc( length+1 );
    // ...
}
```

Рассмотрим еще одну распространенную ошибку, которая обычно вносится на этапе сопровождения. Код, подобный показанному ниже, встречается сплошь и рядом:

```
void process( const char *id ) {
    Name *function = lookupFunction( id );
    if( function ) {
        // ...
    }
}
```

В объявлении переменной `function` пока что нет ничего плохого, но при сопровождении может возникнуть проблема. Мы уже говорили выше, что сопровождающие часто используют имеющиеся локальные переменные для совершенно других целей. Почему? Наверное, просто потому, что они уже есть:

```
void process( const char *id ) {
    Name *function = lookupFunction( id );
    if( function ) {
        // сделать что-то с функцией ...
    }
    else if( function = lookupArgument( id ) ) {
        // обработать аргумент ...
    }
}
```

Пока еще не ошибка, хотя думается мне, что код обработки аргумента окажется довольно сложным для неподготовленного читателя («В этой части кода всюду, где написано 'function', имеется в виду 'argument'.») Но что случится, если автор первоначальной версии решит немного подправить обработку функции?

```
void process( const char *id ) {
    Name *function = lookupFunction( id );
    if( function ) {
        // сделать что-то с функцией ...
    }
    else if( function = lookupArgument( id ) ) {
        // обработать аргумент ...
    }
    // ...
    if( function ) {
        // сделать что-то еще с функцией ...
    }
}
```

Теперь мы, возможно, попытаемся обработать аргумент как функцию.

Обычно лучше всего ограничивать область видимости имени той частью кода, где автор намеревался его использовать. Имена, которые находятся в области видимости, но больше не используются, подобны ничем не занятым подросткам; они просто болтаются без дела, напрашиваясь на неприятности. В первоначальной

версии следовало бы ограничить область видимости переменной `function` участком программы, где она действительно нужна:

```
void process( const char *id ) {
    if( Name *function = lookupFunction( id ) ) {
        // ...
    }
}
```

Ограничение области видимости имени переменной подавляет искушение использовать его повторно, в результате окончательная реализация функции после изменения станет более рациональной:

```
void process( const char *id ) {
    if( Name *function = lookupFunction( id ) ) {
        // ...
        postprocess( function );
    }
    else if( Name *argument = lookupArgument( id ) ) {
        // ...
    }
}
```

Язык C++ признает важность инициализации и ограничения области видимости имен. Он предоставляет программисту ряд средств, позволяющих гарантировать, что каждое имя инициализировано и находится в области видимости, точно соответствующей предполагаемому использованию.

Совет 49. Внимательно относитесь к операциям копирования

C++ очень серьезно относится к операциям копирования. Они исключительно важны в программировании на этом языке и в особенности это касается классов. Операции копирования настолько важны, что если вы не определите их для класса, то компилятор сделает это за вас. А иногда компилятор даже может проигнорировать ваши определения и подставить свои. В одних случаях сгенерированные компилятором операции корректны, в других – нет. Поэтому важно четко понимать, что компилятор ожидает от операции копирования.

Отметим, что конструктор копирования и копирующий оператор присваивания (наряду с другими конструкторами и деструктором) не наследуются от базовых классов. Следовательно, каждый класс должен самостоятельно реализовать для себя копирование.

По умолчанию конструктор копирования реализует почленную инициализацию. Это не имеет ничего общего с принятым в C побитовым копированием структур. Рассмотрим пример реализации простого класса:

```
template <int maxlen>
class NBString {
public:
    explicit NBString( const char *name );
    // . . .
private:
```

```

    std::string name_;
    size_t len_;
    char s_[maxlen];
};

```

В предположении, что операции копирования не определены, компилятор сгенерирует их автоматически. Они будут открытыми встраиваемыми членами.

```

NBString<32> a( "String 1" );
// ...
NBString<32> b( a );

```

Неявный конструктор копирования выполняет почленную инициализацию, вызывая конструктор копирования класса `string` для инициализации члена `b.name_` значением `a.name_`, члена `b.len_` – значением `a.len_`, а элементов массива `b.s_` – значениями соответствующих элементов массива `a.s_`. (На самом деле, из-за какой-то необъяснимой прихоти, стандарт говорит, что «скалярным» типам, к которым относятся все встроенные типы, перечисления и указатели, в неявном конструкторе копирования значения присваиваются, а не инициализируются. Мне не понять, какими мотивами руководствовались члены комитета по стандартизации, формулируя именно такое определение, но для скалярных типов результат все равно будет один и тот же, будь то инициализация или присваивание.)

```
b = a;
```

Аналогично, неявный оператор присваивания выполняет почленное присваивание, вызывая оператор присваивания класса `string` для присваивания значения `a.name_` члену `b.name_`, значения `a.len_` члену `b.len_` и значения элементов массива `a.s_` соответствующим элементам массива `b.c_`.

Такие определения неявных операций копирования дают правильную и традиционную семантику копирования (см. также «Совет 81»). Но рассмотрим немного иную реализацию класса, представляющего именованную строку ограниченной длины:

```

class NBString {
public:
    explicit NBString( const char *name, int maxlen = 32 )
        : name_(name), len_(0), maxlen_(maxlen),
          s_(new char[maxlen] )
        { s_[0] = '\0'; }
    ~NBString()
        { delete [] s_; }
    // ...
private:
    std::string name_;
    size_t len_;
    size_t maxlen_;
    char *s_;
};

```

Теперь конструктор задает максимальный размер строки, а память для хранения символов больше не находится внутри объекта `NBString`. И операции копирования, сгенерированные компилятором, перестают быть правильными:

```
NBString c( "String 2" );
NBString d( c );
NBString e( "String 3" );
e = c;
```

В результате применения неявного конструктора копирования члены `s_` объектов `c` и `d` указывают на один и тот же выделенный из кучи буфер. При уничтожении объектов `c` и `d` их деструкторы попытаются освободить одну и ту же область памяти, то есть имеет место двойное удаление. С другой стороны, когда `c` присваивается `s`, память, на которую указывал `e.s_`, окажется недоступной после того, как `e.s_` станет указывать туда же, куда указывает `c.s_`. При уничтожении же объектов возникнет та же проблема, что и выше. (См. также «Совет 53», где вкратце обсуждается, что происходит с оператором присваивания, сгенерированным компилятором, при наличии виртуальных базовых классов.)

Чтобы написать корректную реализацию, необходимо отказаться от услуг компилятора и взяться за дело самостоятельно:

```
class NBString {
public:
    // . . .
    NBString( const NBString & );
    NBString &operator =( const NBString & );
private:
    std::string name_;
    size_t len_;
    size_t maxlen_;
    char *s_;
};
// . . .
NBString::NBString( const NBString &that )
    : name_(that.name_), len_(that.len_), maxlen_(that.maxlen_),
      s_(strcpy(new char[that.maxlen_], that.s_))
{}
NBString &NBString::operator =( const NBString &rhs ) {
    if( this != &rhs ) {
        name_ = rhs.name_;
        char *temp = new char[rhs.maxlen_];
        len_ = rhs.len_;
        maxlen_ = rhs.maxlen_;
        delete [] s_;
        s_ = strcpy( temp, rhs.s_ );
    }
    return *this;
}
```

Любой проектировщик классов должен внимательно относиться к операциям копирования. Их надо либо предоставлять явно (и не забывать изменять, когда модифицируется реализация класса), либо поручить генерацию компилятору (ревизуя это решение при каждом изменении реализации), либо вообще запретить с помощью следующей идиомы:

```
class NBString {
public:
    // ...
```

```
private:
    NBString( const NBString &);
    NBString &operator =( const NBString & );
    // ...
};
```

Объявление без предоставления определения операций копирования закрытыми запрещает копирование объектов класса. Компилятор не будет пытаться генерировать неявные версии, и большая часть программы не будет иметь доступа к закрытым членам. Любая попытка скопировать объект, предпринятая членами класса или его друзьями, будет перехвачена на этапе компоновки.

Практически невозможно обмануть компилятор в вопросах реализации операций копирования. Ниже показаны изобретательные, но тщетные попытки такого рода:

```
class Derived;
class Base {
public:
    Derived &operator =( const Derived & );
    virtual Base &operator =( const Base & );
};
class Derived : public Base {
public:
    using Base::operator =; // скрыт
    template <class T>
    Derived &operator =( const T & ); // не оператор присваивания
    Derived &operator =( const Base & ); // не оператор присваивания
};
```

Мы уже знаем, что операции копирования не наследуются, но `using`-объявление, которое импортирует пригодный для использования оператор присваивания из не виртуального базового класса, не мешает компилятору сгенерировать собственную версию, которая скрывает то, что импортирована явно. (Отметим попутно, что в базовом классе явно упомянут производный класс, а это признак плохого дизайна. См. «Совет 69».)

Не поможет и реализация оператора присваивания в виде шаблонной функции; члены-шаблоны никогда не могут выступать в роли операций копирования (см. «Совет 88»). Виртуальный оператор присваивания из базового класса переопределяется в производном классе, но переопределенный оператор присваивания для копирования не используется (см. «Совет 76»). Язык C++ очень настойчив в этом отношении: либо пишите операции копирования сами, либо поручите компилятору, третьего не дано.

Совет 50. Побитовое копирование объектов классов

Нет ничего плохого в том, чтобы позволить компилятору автоматически сгенерировать операции копирования, хотя делать это стоит только для простых классов или, если быть точным, для классов с простой структурой. Более того, для простых классов даже лучше поручить эту работу компилятору из соображений

эффективности. Рассмотрим класс, который по существу является лишь элементарным набором данных:

```
struct Record {
    char name[maxname];
    long id;
    size_t seq;
};
```

Имеет прямой смысл дать компилятору возможность реализовать для него операции копирования. Такие классы называются POD-классами (сокр. от Plain Old Data – «добрые старые данные»; см. «Совет 9») и представляют собой обычные структуры в смысле языка C. Стандарт четко определяет, что неявные операции копирования в таких случаях должны следовать семантике, принятой для C-структур, то есть быть побитовыми.

В частности, если на данной платформе есть машинная команда «быстро скопировать n байтов», то компилятор может воспользоваться ей для побитового копирования. Такая оптимизация применима даже к не-POD классам. Так, операции копирования для нешаблонной версии класса `NBString` (см. «Совет 49») можно было бы реализовать, вызвав соответствующую операцию из класса `string` для члена `name_`, а остаток объекта скопировать побитово.

Случается, что кодировщик класса решает взять на себя побитовое копирование. Обычно это ошибочное решение, так как компилятор знает о внутреннем устройстве класса и специфике платформы гораздо больше, чем программист. Ручное побитовое копирование обычно оказывается и более медленным, и в большей мере подверженным ошибкам, чем версия, сгенерированная компилятором:

```
class Record {
public:
    Record( const Record &that )
        { *this = that; }
    Record &operator =( const Record &that )
        { memcpy( this, &that, sizeof(Record) ); return *this; }
    // . . .
private:
    char name[maxname];
    long id;
    size_t seq;
};
```

Наша POD-структура `Record` превратилась в настоящий класс, поэтому мы решили предоставить для нее явные операции копирования. Это излишне, так как компилятор создал бы очень эффективные и правильные версии. Но настоящая проблема возникает по мере эволюции класса:

```
class Record {
public:
    virtual ~Record();
    Record( const Record &that )
        { *this = that; }
    Record &operator =( const Record &that )
        { memcpy( this, &that, sizeof(Record) ); return *this; }
    // . . .
```



```
private:
    char name[maxname];
    long id;
    size_t seq;
};
```

Теперь картина уже не такая радужная. Побитовое копирование больше не отвечает структуре класса. Добавление виртуальной функции заставляет компилятор включить механизм ее реализации, обычно указатель на таблицу виртуальных функций (см. «Совет 78»).

Неявные операции копирования, сгенерированные компилятором, учитывают наличие этого механизма: конструктор копирования правильно установит указатель, а оператор присваивания не станет его модифицировать. Но наша реализация с применением `memcpy` затрет указатель на таблицу виртуальных функций сразу после того, как он будет установлен конструктором копирования. То же самое сделает и оператор присваивания. Подобные ошибки могут возникать и при многих других модификациях класса: наследовании виртуальному базовому классу, добавлении члена данных, для которого определены нетривиальные операции копирования, использование указателей на неинкапсулированную память и так далее.

В общем случае неразумно самостоятельно реализовывать побитовое копирование объекта любого класса, если только нет неоспоримых свидетельств в пользу того, что это даст ощутимое увеличение производительности. Если вы все-таки примете такое решение, не забывайте пересматривать его при каждом изменении реализации класса.

Разумеется, применять побитовое копирование класса вне его реализации совсем уж не рекомендуется. Реализация операций копирования с помощью `memcpy` рискована. Вольные игры с битами самоубийственны.

```
extern Record *exemplaryRecord;
char buffer[sizeof(Record)];
memcpy( buffer, exemplaryRecord, sizeof(buffer) );
```

Автор этого кода, наверное, постеснялся (по крайней мере, должен бы) и запретил его подальше от файла с реализацией класса `Record`. Любое изменение `Record`, несовместимое с побитовым копированием, не будет обнаружено, пока не проявится во время выполнения. Если от написания подобного куда не уйти, то нужно делать это так, чтобы вызывались собственные операции копирования класса (см. «Совет 62»):

```
(void) new (buffer) Record( *exemplaryRecord );
```

Совет 51. Не путайте инициализацию и присваивание в конструкторах

В конструкторе инициализируются все члены класса, нуждающиеся в инициализации. Не присваиваются, а именно инициализируются. В инициализации нуждаются константы, ссылки, объекты классов, имеющих конструкторы, и по-

добъекты базовых классов. (Однако см. «Совет 81» касательно константных и ссылочных данных-членов.)

```
class Thing {
public:
    Thing( int );
};
class Melange : public Thing {
public:
    Melange( int );
private:
    const int aConst;
    const int &aRef;
    Thing aClassObj;
};
// ...
Melange::Melange( int )
{ } // ошибки!
```

Компилятор обнаружит в конструкторе класса `Melange` четыре ошибки: отсутствие инициализации базового класса и трех членов данных. Любая ошибка, обнаруженная во время компиляции, не слишком серьезна, поскольку мы можем исправить ее до того, как она отравит чью-нибудь жизнь:

```
Melange::Melange( int arg )
: Thing( arg ), aConst( arg ), aRef( aConst ), aClassObj( arg )
{ }
```

Настоящая проблема возникает тогда, когда программист забывает выполнить инициализацию, но при этом все равно получается допустимый с точки зрения компилятора код:

```
class Person {
public:
    Person( const string &name );
    // ...
private:
    string name_;
};
// ...
Person::Person( const string &name )
{ name_ = name; }
```

Этот совершенно корректный код увеличивает размер и почти удваивает время работы конструктора класса `Person`. У типа `string` есть конструктор, поэтому его необходимо инициализировать. Но у него есть и конструктор по умолчанию, который вызывается, в случае отсутствия явного инициализатора. Следовательно, конструктор `Person` вызывает конструктор по умолчанию класса `string`, после чего результат его работы немедленно переписывается оператором присваивания. Гораздо лучше сразу инициализировать член `string` и забыть о нем:

```
Person::Person( const string &name )
: name_( name )
{ }
```

В общем случае отдавайте предпочтение спискам инициализации членов, а не присваиванию в теле конструктора.

Конечно, не нужно доводить эту рекомендацию до абсурда. Во всем нужна умеренность. Рассмотрим конструктор нестандартного класса `String`:

```
class String {
public:
    String( const char *init = "" );
    // ...
private:
    char *s_;
};
// ...
String::String( const char *init )
: s_(strcpy(new char[strlen(init?init:"")+1],init?init:"") )
{}

```

Здесь мы зашли слишком далеко, разумнее было бы выполнить присваивание в теле конструктора:

```
String::String( const char *init ) {
    if( !init ) init = "";
    s_ = strcpy( new char[strlen(init)+1], init );
}

```

В списке инициализации членов нельзя инициализировать два вида членов: статические данные-члены и массивы. Разговор о статических данных-членах отложим до «Совета 59». Отдельным элементам массива значения должны быть присвоены в теле конструктора. Альтернативой массиву, зачастую более предпочтительной, служит стандартный контейнер, например, `vector`.

Совет 52. Несогласованный порядок членов в списке инициализации

Порядок инициализации компонентов объекта класс фиксирован в стандарте языка C++ (см. также «Совет 49»).

- Подобъекты виртуальных базовых классов, вне зависимости от того, как далеко они расположены вверх по иерархии наследования.
- Невиртуальные непосредственные базовые классы в порядке перечисления в списке базовых классов.
- Данные-члены класса в порядке объявления.

Отсюда следует, что любой конструктор класса должен выполнять инициализацию именно в этом порядке. Точнее говоря, порядок перечисления элементов списка инициализации членов не имеет значения для компилятора:

```
class C {
public:
    C( const char *name );
private:
    const int len_;
    string n_;
};
// . . .
C::C( const char *name )
: n_( name ), len_( n_.length() ) // ошибка!!!
{}

```

Член `len_` объявлен первым, поэтому он будет инициализирован раньше `n_` несмотря на то, что в списке инициализации членов находится после `n_`. В данном случае мы пытаемся вызвать функцию-член объекта, который еще не был инициализирован, что дает неопределенный результат.

Считается хорошим тоном располагать элементы списка инициализации членов в том порядке, в котором объявлены базовые классы и данные-члены, иначе говоря, в том порядке, в котором они будут инициализироваться в действительности. По мере возможности следует избегать зависимостей от порядка следования членов в списке инициализации:

```
C::C( const char *name )
    : len_( strlen(name) ), n_( name )
    {}
```

См. «Совет 67» о том, почему порядок инициализации не связан с порядком следования членов в списке инициализации.

Совет 53. Инициализация виртуальных базовых классов

Подобъект виртуального базового класса размещается в памяти иначе, чем подобъект неvirtуального базового класса. Члены неvirtуального базового класса размещаются так, будто это просто данные-члены производного класса (рис. 5.1). Поэтому внутри объекта может быть несколько подобъектов одного и того же неvirtуального базового класса.

```
class A { members };
class B : public A { members };
class C : public A { members };
class D : public B, public C { members };
```

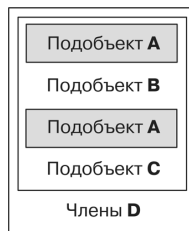


Рис. 5.1. Типичное размещение в памяти объекта при условии множественного неvirtуального наследования. В объекте D есть два подобъекта A

Объект виртуального базового класса входит в объект производного класса только один раз, даже если в графе наследования встречается неоднократно (как на рис. 5.2):

```
class A { members };
class B : public virtual A { members };
```

```
class C : public virtual A { members };
class D : public B, public C { members };
```

Для простоты мы показали уже вышедший из обихода список реализации виртуальных базовых классов с помощью указателей. В том месте, где в полном объекте должен был бы появиться подобъект виртуального базового класса А, на самом деле стоит указатель на общую память для всех подобъектов А. Чаще ссылка на разделяемый подобъект виртуального базового класса реализуется в виде смещения или информации, которая хранится в таблице виртуальных функций. Так или иначе, но последующее обсуждение применимо к любой реализации.

Как правило, память для разделяемого подобъекта виртуального базового класса выделяется в конце полного объекта. В примере выше полный объект принадлежит классу D, а память для подобъекта класса А находится после всех членов D. Объект, для которого «самым производным» классом является D, размещался бы в памяти иначе.

Немного подумав, вы поймете, что только «самый производный» (расположенный дальше других от начала иерархии) класс точно знает, где должна находиться память для подобъекта виртуального базового класса. Объект типа В может быть полным или подобъектом другого объекта. Поэтому именно на самый производный класс возлагается обязанность инициализировать все подобъекты виртуальных базовых классов в графе наследования. Ему и предоставляется механизм для доступа к этим подобъектам.

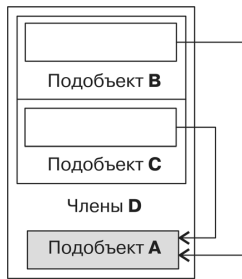


Рис. 5.2. Типичное размещение в памяти объекта при условии множественного виртуального наследования. В объекте D есть только один подобъект А

В случае объекта, для которого самым производным является класс В, как на рис. 5.3, конструкторы В должны были инициализировать подобъект А и установить указатель на него:

```
B::B( int arg )
: A( arg ) {}
```

В случае объекта, для которого самым производным является класс D, как на рис. 5.2, конструкторы D инициализируют подобъект А и указатели на В и С, а равно непосредственные базовые классы D.

```
D::D( int arg )
: A( arg ), B( arg ), C( arg+1 ) {}
```

После того как подобъект A инициализирован конструктором D, он не будет еще раз инициализироваться конструктором B или C. (Компилятор может добиться этого, заставив конструктор D передавать некий флаг или указатель на A конструктору B и C, говоря тем самым: «Да, кстати, не надо инициализировать A.» Никакой мистики.) Взгляните еще на один конструктор D:

```
D::D()
: B( 11 ), C( 12 ) {}
```

Это типичный источник непонимания и ошибок при использовании виртуальных базовых классов. Конструктор D по-прежнему инициализирует виртуальный подобъект A, но делает это неявно, вызывая конструктор A по умолчанию. Когда конструктор D вызывает конструктор подобъекта B, он не инициализирует A заново, и потому явного обращения к конструктору A с аргументами не происходит.

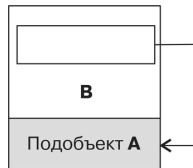


Рис. 5.3. Типичное размещение в памяти объекта при условии одиночного виртуального наследования. В объекте B имеется только один подобъект A, но все равно сослаться на него приходится косвенно

Чтобы не усложнять программы, применяйте виртуальные базовые классы только тогда, когда без них не обойтись. (Верно и обратное: если проект настоятельно требует использования виртуальных базовых классов, не отказывайтесь от них.) Обычно проще всего проектировать виртуальные базовые классы как «интерфейсные». В интерфейсном классе нет никаких данных, а все его функции, как правило, чисто виртуальны (за исключением, быть может, деструктора). В нем нет конструктора вовсе или есть только простой конструктор по умолчанию:

```
class A {
public:
    virtual ~A();
    virtual void op1() = 0;
    virtual int op2( int src, int dest ) = 0;
    // ...
};
inline A::~~A()
{ }
```

Следование этому совету поможет избежать ошибок не только в конструкторах, но и в операторах присваивания. В частности, стандарт говорит, что генери-

руемый компилятором оператор присваивания может выполнять присваивание подобъекту виртуального базового класса многократно, но может этого и не делать. Если все виртуальные базовые классы являются интерфейсными, то присваивание – это пустая операция (вспомните, что такие механизмы реализации класса, как указатель на таблицу виртуальных функций, задействуются только во время инициализации, но не во время присваивания), так что ее многократное выполнение ни к каким неприятностям не приводит.

Общее решение задачи реализации оператора присваивания в иерархии, содержащей виртуальные базовые классы, заключается в том, чтобы в каком-то смысле имитировать семантику конструирования объектов, включающих подобъекты виртуальных базовых классов.

Рассмотрим первую реализацию приведенного выше класса D (рис. 5.1.), которая содержит два (невиртуальных) подобъекта A. Здесь, как и в случае конструктора D, предоставляемый программистом оператор присваивания можно реализовать целиком в терминах непосредственных базовых классов:

➤ gotcha53/virtassign.cpp

```
D &D::operator =( const D &rhs ) {
    if( this != &rhs ) {
        B::operator =( *this ); // присвоить значение подобъекту B
        C::operator =( *this ); // присвоить значение подобъекту C
        // присвоить значения членам, специфичным для D ...
    }
    return *this;
}
```

В этой реализации сделано разумное допущение о том, что базовые классы B и C сами позаботятся о присваивании значений своим (невиртуальным) подобъектам. Но, как и при конструировании, этот простой поэтапный подход к присваиванию для виртуального наследования не работает. И в этом случае присваивать значения подобъектам виртуальных базовых классов должен самый производный класс; он же должен как-то предотвратить повторное присваивание промежуточным базовым классам:

➤ gotcha53/virtassign.cpp

```
D &D::operator =( const D &rhs ) {
    if( this != &rhs ) {
        A::operator =( *this ); // присвоить значение виртуальному A
        B::nonvirtAssign( *this ); // присвоить значение B, не трогая
        // часть A
        C::nonvirtAssign( *this ); // присвоить значение C, не трогая
        // часть A
        // присвоить значения членам, специфичным для D ...
    }
    return *this;
}
```

Здесь мы ввели в классы B и C специальные функции-члены, похожие на оператор присваивания. Работают они точно так же, как последний, но не выполняют присваивание подобъектам виртуальных базовых классов. Это эффективно, но, очевидно, сложно; кроме того, от D требуется «интимное» знакомство со структу-

рой иерархии выше его непосредственных базовых классов. Стоит этой структуре измениться, и класс D придется перерабатывать. Выше уже отмечалось, что на роль виртуальных базовых классов лучше всего выбирать интерфейсные.

Из того, как размещаются в памяти подобъекты виртуальных базовых классов, вытекает одно следствие: запрещается выполнять статическое понижающее приведение от виртуального базового класса к любому производному от него.

```
A *ap = gimmeanA();
D *dp = static_cast<D *>(ap); ошибка!
dp = (D *)ap; // ошибка!
```

Можно выполнить оператор `reinterpret_cast` для приведения от виртуального базового класса к производному. Как показано на рис. 5.4, это, скорее всего, даст некорректный и непригодный для использования адрес. Единственный надежный способ выполнить понижающее приведение указателя или ссылки на виртуальный базовый класс, воспользоваться оператором `dynamic_cast` (см. «Совет 45»):

```
if( D *dp = dynamic_cast<D *>(ap) ) {
    // сделать что-то с dp ...
}
```

Однако систематическое употребление `dynamic_cast` — признак неудачного проекта. (См. «Совет 98» и «Совет 99»).

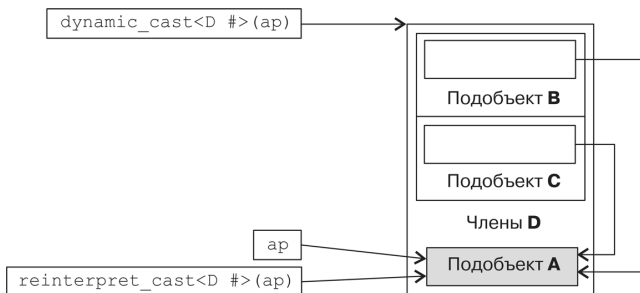


Рис. 5.4. Вероятный эффект статического и динамического приведения при множественном виртуальном наследовании. При такой реализации объект D имеет три действительных адреса, и для корректного приведения нужно знать смещения различных подобъектов от начала полного объекта

Совет 54. Инициализация базового класса в конструкторе копирования

Вот парочка простых компонентов:

```
class M {
public:
    M();
```



```

M( const M & );
~M();
M &operator =( const M & );
// . . .
};
class B {
public:
    virtual ~B();
protected:
    B();
    B( const B & );
    B &operator =( const B & );
    // . . .
};

```

Воспользуемся ими для создания нового класса и попытаемся заставить компилятор проделать как можно большую часть работы:

```

class D : public B {
    M m_;
};

```

Поскольку класс D не наследует конструкторы, деструктор и оператор присваивания от своего базового класса, то компилятор сгенерирует эти операции самостоятельно, позаимствовав соответствующие реализации от компонентов (см. «Совет 49»). Например, реализация конструктора по умолчанию D станет открытой встраиваемой функцией. Этот конструктор сначала вызовет конструктор по умолчанию базового класса B, а затем конструктор по умолчанию члена M. Деструктор, как обычно, выполнит действия в обратном порядке: сначала уничтожит член, а потом вызовет деструктор базового класса.

Операции копирования интереснее. Сгенерированная компилятором версия выполнит почленную инициализацию, как если бы мы написали:

```

D::D( const D &init )
: B( init ), m_( init.m_ )
{}

```

Сгенерированный компилятором оператор присваивания выполняет почленное присваивание, примерно так:

```

D &D::operator =( const D &that ) {
    B::operator =( that );
    m_ = that.m_;
    return *this;
}

```

Предположим, что мы добавили в класс новый член данных, в котором эти операции не определены. Например, указатель на размещенный в куче объект X:

```

class D : public B {
public:
    D();
    ~D();
    D( const D & );
    D &operator =( const D & );
private:
    M m_;
};

```

```
X *xp_; // новый член данных
};
```

Теперь все эти операции придется написать явно. Конструктор по умолчанию и деструктор не вызывает сложностей, и мы можем поручить компилятору выполнить большую часть работы:

```
D::D()
    : xp_( new X )
    {}
D::~D()
    { delete xp_; }
```

Компилятор неявно вызовет конструкторы по умолчанию и деструкторы для базового класса и члена `m_`. Заманчиво было бы обойтись столь же малой кровью при реализации конструктора копирования и копирующего оператора присваивания, но не получится:

```
D::D( const D &init )
    : xp_( new X(*init.xp_) )
    {}
D &D::operator =( const D &rhs ) {
    delete xp_;
    xp_ = new X(*rhs.xp_);
    return *this;
}
```

Обе эти реализации откомпилируются без ошибок, но во время выполнения будут делать не то, что нужно. Наш конструктор копирования инициализирует член `xp_` копией того, на что указывает инициализатор `xp_`, но базовый класс и член `m_` инициализируются соответственно с помощью конструкторов `B` и `M` по умолчанию, а не их конструкторами копирования. В случае присваивания значения подобъекта базового класса и члена `m_` вообще не изменятся.

Раз уж вы не позволили компилятору написать за вас эти функции-члены, то придется предоставить полную реализацию:

```
D::D( const D &init )
    : B( init ), m_( init.m_ ), xp_( new X(*init.xp_) )
    {}
D &D::operator =( const D &rhs ) {
    if( this != &rhs ) {
        B::operator =( rhs );
        m_ = rhs.m_;
        delete xp_;
        xp_ = new X(*rhs.xp_);
    }
    return *this;
}
```

То же относится к конструктору по умолчанию и деструктору, но в данном случае неявный вызов конструкторов по умолчанию для базового класса и `m_` обеспечивает правильное поведение. Я предпочитаю подход, при котором приходится меньше нажимать на клавиши, но вы вольны выбрать и явное решение:

```
D::D()
: B(), m_(), xp_( new X )
{}

```

Совет 55. Порядок инициализации статических данных во время выполнения

Все статические данные в программе на C++ инициализируются перед обращением к ним. По большей части, инициализация производится во время загрузки программы, еще до начала исполнения. Если явный инициализатор отсутствует, данные инициализируются «нулями»:

```
static int question; // 0
extern int answer = 42;
const char *terminalType; // null
bool isVT100; // false
const char **ptt = &terminalType;

```

Эти операции инициализации происходят «одновременно», и их порядок не играет роли.

Допустима также статическая инициализация во время выполнения. Но в этом случае не дается никаких гарантий относительно порядка инициализации объектов, находящихся в разных единицах трансляции. (Единица трансляции – это, по существу, файл, полученный после обработки препроцессором.) Это часто приводит к ошибкам, поскольку порядок инициализации может измениться даже, если исходный текст остался неизменным:

```
// в файле term.cpp
const char *terminalType = getenv( "TERM" );

// в файле vt100.cpp
extern const char *terminalType;
bool isVT100 = strcmp( terminalType, "vt100" )==0; // ошибка?

```

Существует неявная зависимость от порядка инициализации между `terminalType` и `isVT100`, но язык C++ не дает, да и не может дать гарантий относительно того, какая переменная будет инициализирована первой. Эта проблема обычно возникает при переносе работающей программы на другую платформу, где случайно порядок статической инициализации переменных из разных единиц трансляции оказался иным. Может она проявиться и в отсутствие каких-либо модификаций кода, если внесены изменения в порядок сборки или компонент, который раньше компоновался статически, теперь стал компоноваться динамически.

Имейте в виду, что инициализация статических объектов классов также считается статической инициализацией во время выполнения:

```
class TermInfo {
public:
    TermInfo()
        : type_( ::terminalType )
    {}

```

```
private:
    std::string type_;
};
// ...
TermInfo myTerm; // статическая инициализация во время выполнения!
```

Лучший способ избежать проблем из-за статической инициализации во время выполнения: свести к минимуму число внешних переменных, в том числе и статически инициализируемых данных-членов классов (см. «Совет 3»).

Если это не выход, то надо стремиться, чтобы зависимость от порядка инициализации существовала только между переменными в одной единице трансляции. В этом случае никакой неопределенности нет: статические переменные инициализируются в том порядке, в котором встречаются в тексте. Например, если определения `terminalType` и `isVT100` записаны именно в таком порядке и внутри одного файла, то проблем с переносимостью не возникнет. Но и при таком подходе может возникнуть проблема, если какая-либо внешняя функция, в том числе и член класса, воспользуется статической переменной, поскольку не исключено, что эта функция вызвана (прямо или косвенно) в ходе статической инициализации переменной, находящейся в другой единице трансляции:

```
extern const char *termType()
{ return terminalType; }
```

Еще одно решение – применить вместо инициализации отложенное (*lazy*) вычисление. Обычно это сводится к той или иной вариации на тему паттерна Singleton (Одиночка) (см. «Совет 3»).

И в качестве последнего средства мы можем запрограммировать инициализацию явно, применив какой-либо из стандартных приемов. Один такой прием называется «счетчиком Шварца», поскольку Джерри Шварц (Jerry Schwarz) придумал и применил его в библиотеке `iostream`:

➤ `gotcha55/term.h`

```
extern const char *terminalType;
// прочие вещи, нуждающиеся в инициализации ...
class InitMgr { // счетчик Шварца
public:
    InitMgr()
        { if( !count_++ ) init(); }
    ~InitMgr()
        { if( !--count_ ) cleanup(); }
    void init();
    void cleanup();
private:
    static long count_; // один на весь процесс
};
namespace { InitMgr initMgr; } // один на каждый включаемый файл
```

➤ `gotcha55/term.cpp`

```
extern const char *terminalType = 0;
long InitMgr::count_ = 0;
void InitMgr::init() {
    if( !(terminalType = getenv( "TERM" )) )
```

```

        terminalType = "VT100";
    // другие операции инициализации ...
}
void InitMgr::cleanup() {
    // очистка ...
}

```

Счетчик Шварца подсчитывает, сколько раз был включен заголовочный файл, в котором он находится. Существует один на весь процесс экземпляр статического члена `count_` класса `InitMgr`. Однако при каждом включении файла `term.h` создается новый объект типа `InitMgr`, и каждый из них требует статической инициализации. Конструктор `InitMgr`, глядя на член `count_`, определяет, первая ли это инициализация объекта `InitMgr` в процессе. Если да, то инициализация выполняется.

Обратно, если процесс завершается нормально, то статические объекты, имеющие деструкторы, будут уничтожены. При уничтожении каждого объекта `InitMgr` его деструктор уменьшает счетчик `count_` на 1. Когда `count_` станет равен нулю, выполняется необходимая очистка.

Хотя эта техника довольно надежна, особо бестолковое кодирование может обмануть даже счетчик Шварца. Так что в общем случае старайтесь минимизировать употребление статических переменных и избегать статической инициализации во время выполнения.

Совет 56. Прямая инициализация и инициализация копированием

Мне доводилось сталкиваться с довольно небрежными примерами инициализации. Рассмотрим простой класс `Y`:

```

class Y {
public:
    Y( int );
    ~Y();
};

```

Нередко следующие три способа инициализации такого объекта применяют так, как будто они эквивалентны. Как будто это не имеет значения. Как будто.

```

Y a( 1066 );
Y b = Y(1066);
Y c = 1066;

```

Вообще-то, все три способа, скорее всего, приведут к генерации одного и того же кода, тем не менее, они не эквивалентны. Инициализация `a` называется прямой, ее результат в точности тот, что мы ожидаем; происходит непосредственное обращение к конструктору `Y : Y (int)`.

Инициализации `b` и `c` несколько сложнее. То и другое — инициализация копированием. В случае `b` мы просим создать временный анонимный объект типа `Y`, инициализированный значением `1066`. Затем этот объект передается в качестве параметра конструктору копирования класса `Y` для инициализации `b`. В конце

вызывается деструктор временного анонимного объекта. По сути дела мы попросили компилятор сгенерировать код, подобный следующему:

```
Y temp( 1066 ); // инициализировать временный объект
Y b( temp ); // конструктор копирования
temp.~Y(); // вызов деструктора
```

Семантика инициализации `e` такая же, но анонимный временный объект создается неявно.

Изменим реализацию `Y`, добавив собственный конструктор копирования, и посмотрим, что получится:

```
class Y {
public:
    Y( int );
    Y( const Y & )
        { abort(); }
    ~Y();
};
```

Ясно, что объекты `Y` не хотят, чтобы их копировали. Но после перекомпиляции программы все три инициализации выполняются нормально, процесс не завершается. Что же происходит?

Дело в том, что стандарт явно разрешает компилятору преобразовывать программу с целью исключить порождение временных переменных и обращения к конструктору копирования, генерируя тот же самый код, что и при прямой инициализации. Отметим, что это не просто «оптимизация», поскольку изменяется поведение программы (в данном случае процесс не завершается). Большинство компиляторов C++ выполняют такое преобразование, хотя стандарт этого и не требует. В условиях такой неопределенности лучше точно говорить, что вы имеете в виду, и пользоваться прямой инициализацией в объявлениях объектов классов:

```
Y a(1066), b(1066), c(1066);
```

По какой-то странной прихоти у вас может возникнуть желание запретить компилятору выполнять описанное преобразование, возможно, потому, что порождение временных объектов и вызов конструктора копирования дают какой-то побочный эффект. А, может, вы просто поставили себе целью написать большое и медленное приложение. К сожалению, гарантировать нужную семантику нелегко, так как любой совместимый со стандартом компилятор вправе применить такое преобразование. О способе избежать его переносимым способом (без использования платформенно-зависимых директив `#pragma` или флагов компилятора) даже думать страшно, так что просто взгляните на следующий код:

```
struct {
    char b_[sizeof(Y)];
} aY; // выровненный буфер размера не меньше Y
new (&aY) Y(1066); // создать временный объект
Y d( reinterpret_cast<Y &>(aY) ); // конструктор копирования
reinterpret_cast<Y &>(aY).~Y(); // уничтожить временный объект
```

Здесь почти продублирована семантика инициализации без преобразования. (Память для `aY`, скорее всего, не будет позднее использована повторно во фрейме

стека так, как это было бы в случае сгенерированного компилятором временного объекта. См. «Совет 66».) Но есть и более простые способы писать объемные и медленные программы.

Говоря о рассматриваемом преобразовании, важно понимать, что компилятор применяет его после того, как проверит семантику исходной программы. Если инициализация до преобразования была некорректной, то будет выдано сообщение об ошибке, даже если в результате преобразования можно было бы получить правильный код. Рассмотрим такой класс X:

```
class X {
public:
    X( int );
    ~X();
    // . . .
private:
    X( const X & );
};

X a( 1066 ); // правильно
X b = 1066; // ошибка!
X c = X(1066); // ошибка!
```

Для инициализации `b` и `c` без преобразования необходим доступ к конструктору копирования `X`, но автор `X` решил запретить копирование объектов `X`, сделав конструктор копирования закрытым. И, хотя преобразование устранило бы необходимость в обращении к конструктору копирования, код все равно считается некорректным.

Как прямая инициализация, так и инициализация копированием применимы не только к типам класса, но в этом случае результат предсказуем и переносим:

```
int i(12); // прямая инициализация
int j = 12; // копирование, результат тот же
```

При инициализации таких типов можете выбирать тот способ, который кажется вам наиболее понятным. Но имейте в виду, что внутри шаблона, когда тип переменной до конкретизации неизвестен, обычно все-таки лучше пользоваться прямой инициализацией. Рассмотрим упрощенную версию алгоритма «длина последовательности», параметризованного не только типом итератора (`In`), но и типом числового счетчика (`N`):

>> `gotcha56/seqlength.cpp`

```
template <typename N, typename In>
void seqLength( N &len, In b, In e ) {
    N n( 0 ); // именно так, а НЕ "N n = 0;"
    while( b != e ) {
        ++n;
        ++b;
    }
    len = n;
}
```

При такой реализации использование прямой инициализации позволяет подставить (согласен, это необычно) определенный пользователем числовой тип, ко-

торый не допускает копирования. Если бы в реализации `seqLength` мы применили инициализацию объекта `N` копированием, это было бы невозможно.

С точки зрения простоты и переносимости лучше применять прямую инициализацию в объявлениях объектов классов или объектов, которые могут оказаться объектами типа класса.

Совет 57. Прямая инициализация аргументов

Все мы знаем, что формальные аргументы инициализируются фактическими, но как именно: прямо или копированием? Это легко проверить экспериментально:

```
class Y {
public:
    Y( int );
    ~Y();
private:
    Y( const Y & );
    // ...
};
void f( Y yFormalArg ) {
    // ...
}
// ...
f( 1337 );
```

Если бы передача аргументов была реализована как прямая инициализация, то вызов `f` был бы правильным. Если же она реализована в виде инициализации копированием, то компилятор выдал бы сообщение о попытке неявного доступа к закрытому конструктору копирования в классе `Y`. Большинство компиляторов разрешают такой вызов, поэтому вроде бы напрашивается вывод, что передача аргументов реализована как прямая инициализация. Но компиляторы не правы или, по крайней мере, устарели в этом отношении. Стандарт говорит, что передача аргументов должна быть реализована путем инициализации копированием, так что приведенный выше вызов `f` некорректен. Инициализация `yFormalArg` полностью аналогична следующему объявлению:

```
Y yFormalArg = 1337; // ошибка!
```

Если вы хотите, чтобы ваш код был стандартным, переносимым и остался правильным после того, как все компиляторы будут вести себя в соответствии с этим аспектом стандарта, то таких вызовов `f` следует избегать.

Возможны также некоторые проблемы с производительностью. Если бы функция, обращающаяся к `f`, имела доступ к закрытому конструктору копирования `Y`, то этот вызов был бы правильным, но означал бы что-то в таком роде:

```
Y temp( 1337 );
yFormalArg( temp );
// тело f . . .
yFormalArg.~Y();
temp.~Y();
```

Другими словами, инициализация формального аргумента состояла бы из следующих шагов: создание временного объекта, конструирование формального

аргумента путем копирования, уничтожение формального аргумента после возврата из функции и уничтожение временного объекта. Четыре вызова функций, не считая собственно обращения к `f`. К счастью, большинство компиляторов выполняют преобразование программы с целью избавиться от создания временного объекта и обращения к конструктору копирования, так что генерируемый код получается таким же, как при прямой инициализации:

```
yFormalArg( 1337 );  
// тело f ...  
yFormalArg.~Y();
```

Однако даже такое решение оптимально не во всех случаях. Что если аргумент `yFormalArg` инициализируется объектом `Y`?

```
Y aY( 1453 );  
f( aY );
```

В этом случае мы имеем конструирование `yFormalArg` путем копирования `aY` и уничтожение `yFormalArg` после возврата из `f`. Гораздо лучше будет по возможности вообще избежать передачи объектов класса по значению, используя вместо этого передачу по ссылке на константу:

```
void fprime( const Y &yFormalArg );  
// ...  
fprime( 1337 ); // работает! конструктор копирования не вызывается  
fprime( aY ); // работает, эффективно.
```

В первом случае компилятор создает временный объект `Y`, инициализированный значением `1337`, и использует его для инициализации ссылочного формального аргумента. Временный объект уничтожается сразу после возврата из `fprime`. (См. «Совет 44», в котором обсуждается серьезная опасность, сопряженная с возвратом такого аргумента). С точки зрения эффективности это эквивалентно показанному выше преобразованному решению, но зато является корректным кодом на C++. Со вторым обращением к `fprime` вообще не связаны никакие накладные расходы на создание временных объектов, а, кроме того, отпадает необходимость в вызове деструктора после возврата из функции.

Совет 58. Что такое оптимизация возвращаемого значения?

Часто функция должна возвращать результат по значению. Например, в классе `String` ниже реализован бинарный оператор конкатенации, который обязан возвращать вновь созданный объект `String` по значению:

```
class String {  
public:  
    String( const char * );  
    String( const String & );  
    String &operator =( const String &rhs );  
    String &operator +=( const String &rhs );  
    friend String  
        operator +( const String &lhs, const String &rhs );  
    // ...
```

```
private:
    char *s_;
};
```

Как и в случае инициализации формального аргумента, инициализация возвращаемого функцией значения выполняется путем обращения к конструктору копирования:

```
String operator +( const String &lhs, const String &rhs ) {
    String temp( lhs );
    temp += rhs;
    return temp;
}
```

Логически конструктор копирования инициализирует область, в которую функция помещает результат, значением переменной `temp`, после чего объект `temp` уничтожается. В общем случае компилятор реализует возврат, передавая целевой объект в виде неявного аргумента функции, как если бы функция изначально была написана следующим образом:

```
void
operator +( String &dest, const String &lhs, const String &rhs ) {
    String temp( lhs );
    temp += rhs;
    dest.String::String( temp ); // конструктор копирования
    temp.~String();
}
```

Обратите внимание, что компилятор может сгенерировать подобное обращение к конструктору копирования, но нам это запрещено. Простые смертные должны прибегать к ухищрениям:

```
new (&dest) String( temp ); // трюк с размещающим new, см. Совет 62
```

Одно из следствий такого преобразования состоит в том, что, вообще говоря, эффективнее инициализировать переменную класса значением, возвращаемым некоторой функцией, чем путем присваивания:

```
String ab( a+b ); // эффективно
ab = a + b; // возможно, не очень эффективно
```

В объявлении `ab` компилятор может скопировать результат вычисления `a + b` прямо в `ab`. В случае присваивания это невозможно. Оператор присваивания для объектов `String` – это функция-член, которая сначала уничтожает `ab`, а затем повторно инициализирует его; поэтому никогда не надо даже пытаться присвоить что-то неинициализированной памяти (см. «Совет 47»):

```
String &String::operator =( const String &rhs );
```

Для инициализации аргумента `rhs` функции-члена `operator =` в классе `String` компилятор обязан скопировать значение `a + b` во временный объект, инициализировать `rhs` его значением и разрушить этот объект после возврата из `operator =`. Поэтому эффективности ради пользуйтесь инициализацией, а не присваиванием.

Рассмотрим семантику инициализации копированием при возврате результата вычисления выражения, тип которого отличается от объявленного типа возвращаемого значения:

```
String promote( const char *str )
{ return str; }
```

Здесь семантика инициализации копированием требует, чтобы с помощью аргумента `str` был инициализирован временный объект типа `String`, который затем будет применен для конструирования возвращаемого значения путем копирования. И в конце временный объект должен быть уничтожен. Однако компилятору разрешено применять то же преобразование программы при инициализации возвращаемого значения, что и при инициализации в объявлениях и формальных параметрах. Поэтому вполне может статься, что `str` будет использована для прямой инициализации возвращаемого значения путем вызова обычного (некопирующего) конструктора `String`, избежав тем самым создания временного объекта. Преобразование программы, заключающееся в замене инициализации копированием прямой инициализацией в контексте возврата значения из функции, называется «оптимизация возвращаемого значения» (`return value optimization` – `RVO`).

Программисты нередко пытаются повысить эффективность за счет таких низкоуровневых конструкций:

```
String operator +( const String &lhs, const String &rhs ) {
    char *buf = new char[ strlen(lhs.s_)+strlen(rhs.s_)+1];
    String temp( strcat( strcpy( buf, lhs.s_ ), rhs.s_ ) );
    delete [] buf;
    return temp;
}
```

К сожалению, этот код может оказаться даже медленнее, чем показанная выше реализация `operator +`. Мы выделяем память для локального буфера, в котором строится конкатенация двух строк, но лишь для того, чтобы инициализировать его содержимым временный возвращаемый объект `String`. После этого буфер уже не нужен.

В подобных случаях иногда бывает полезно ввести в реализацию класса «вычислительный конструктор». Такой конструктор составляет неотъемлемую часть класса и обычно делается закрытым. По существу, это лишь вспомогательная функция, реализованная в виде конструктора, чтобы получить доступ к специальным средствам, которыми обладают только конструкторы, но не обычные функции-члены. Как правило, интерес представляет тот факт, что конструктор работает с неинициализированной памятью, а не с объектом. А это означает, что «очищать» нечего:

```
class String {
    // ...
private:
    String( const char *a, const char *b ) { // вычислительный
        s_ = new char[ strlen(a)+strlen(b)+1];
        strcat( strcpy( s_, a ), b );
    }
    char *s_;
};
```

Такой вычислительный конструктор можно применять для эффективного возврата по значению из других функций в реализации класса:

```
inline String operator +( const String &a, const String &b )
{ return String( a.s_, b.s_ ); }
```

Напомним, что инициализация возвращаемого значения копированием аналогична инициализации в объявлении:

```
String retval = String( a.s_, b.s_ );
```

Если компилятор вообще применяет преобразования к инициализации, то мы получаем функциональный аналог прямой инициализации:

```
String retval( a.s_, b.s_ );
```

Часто вычислительные конструкторы оказываются совсем простыми и могут встраиваться. Обращающаяся к конструктору функция `operator +` теперь является подходящим кандидатом на встраивание, что приводит к очень эффективной реализации, не уступающей той, что закодирована вручную. Заметим, однако, что вычислительные конструкторы обычно никак не улучшают открытый интерфейс типа. Поэтому их следует считать деталью реализации класса и объявлять в закрытом разделе. Все без исключения вычислительные конструкторы с одним аргументом должны быть объявлены `explicit`, чтобы компилятор не применял их в процессе неявного преобразования типов (см. «Совет 37»).

Компиляторы C++ часто применяют в контексте возврата из функции еще одно преобразование, известное под названием «оптимизация именованного возвращаемого значения» (named return value optimization – NRV). Оно похоже на RVO, но позволяет использовать для хранения возвращенного значения именованную локальную переменную. Рассмотрим следующую реализацию `operator +`:

```
String operator +( const String &lhs, const String &rhs ) {
    String temp( lhs );
    temp += rhs;
    return temp;
}
```

Если компилятор применит к этому коду оптимизацию NRV, то локальная переменная `temp` будет заменена ссылкой на ту переменную в вызывающей программе, куда в конечном итоге должно быть помещено возвращенное значение. Как если бы функция была написана так:

```
void
operator +( String &dest, const String &lhs, const String &rhs ) {
    dest.String::String( lhs ); // конструктор копирования
    dest += rhs;
}
```

Оптимизация NRV обычно применяется только, если компилятор может с уверенностью сказать, что все выражения, возвращаемые функцией, идентичны и ссылаются на одну и ту же локальную переменную. Чтобы повысить вероятность такой оптимизации, старайтесь, чтобы функция имела только одну точку возврата или, если это невозможно, то чтобы всегда возвращалась одна и та же локальная переменная. Чем проще, тем лучше. Отметим, что NRV – это преобразование программы, а не оптимизация, поскольку побочные эффекты, которые могли бы возникнуть при инициализации и уничтожении временного объекта, устраняются.

Выигрыш в производительности от таких преобразований может быть велик, поэтому часто имеет смысл облегчить компилятору их применение за счет применения вычислительных конструкторов или заведения их простых локальных переменных для хранения возвращенных значений.

Совет 59. Инициализация статических членов в конструкторе

Статические данные-члены существуют независимо от объектов класса и обычно начинают свое существование до создания первого объекта. (Помните о типичных ограничениях.) Как и функции-члены (равно статические и не-статические), статические данные-члены имеют внешнюю компоновку и принадлежат области видимости класса:

```
class Account {
    // ...
private:
    static const int idLen = 20;
    static const int prefixLen;
    static long numAccounts;
};
// ...
const int Account::idLen;
const int Account::prefixLen = 4;
long Account::numAccounts = 0;
```

Инициализация статических членов интегральных типов и перечислений может происходить вне класса, но только один раз. Если речь идет о целочисленных значениях, то часто разумно вместо инициализированных целых констант использовать перечисления:

```
class Account {
    // ...
private:
    enum {
        idLen = 20,
        prefixLen = 4
    };
    static long numAccounts;
};
// ...
long Account::numAccounts = 0;
```

В общем случае перечисления могут служить заменой целым константам. Однако под них не отводится память, поэтому невозможно взять их адрес. Тип перечисления отличается от `int`, что может иметь значение при выборе перегруженной функции, если перечисление выступает в роли фактического аргумента. Заметим также, что определять член `numAccounts` вне класса необходимо, а вот явно инициализировать необязательно. В таком случае он будет инициализирован «всеми нулями» или просто нулем. Тем не менее, явная инициализация нулем предпочтительнее, поскольку предотвращает желание сопровождающего

инициализировать член каким-то другим значением (почему-то часто употребляют для этой цели 1 или -1). См. также «Совет 25».

Инициализация статических данных-членов класса во время исполнения – это порочная идея. Результат инициализации статического члена может быть отменен в момент, когда во время выполнения инициализируется сам статический объект класса:

```
class Account {
public:
    Account() {
        ... calculateCount() ...
    }
    // ...
    static long numAccounts;
    static const int fudgeFactor;
    int calculateCount()
    { return numAccounts+fudgeFactor; }
};
// ...
static Account myAcct; // беда!
// ...
long Account::numAccounts = 0;
const int Account::fudgeFactor = atoi(getenv("FUDGE"));
```

Объект `myAcct` класса `Account` определен раньше статического члена `fudgeFactor`, поэтому конструктор `myAcct` будет пользоваться неинициализированным значением `fudgeFactor` при обращении к `calculateCount` (см. «Совет 55»). Член `fudgeFactor` будет равен нулю, поскольку статические данные инициализируются «всеми нулями». Если нуль – допустимое значение `fudgeFactor`, то найти эту ошибку будет трудно.

Некоторые программисты пытаются обойти эту проблему, «инициализируя» статические данные-члены в каждом конструкторе класса. Это невозможно, так как статические члены не могут находиться в списке инициализации членов, и, если программа вошла в тело конструктора, то речь уже идет не об инициализации, а о присваивании:

```
Account::Account() {
    // ...
    fudgeFactor = atoi( getenv( "FUDGE" ) ); // ошибка!
}
```

Единственная альтернатива: сделать член `fudgeFactor` неконстантным, написать код для «отложенной инициализации» (см. «Совет 3») в каждом конструкторе и надеяться, что в ходе сопровождения этот код будет синхронно модифицироваться во всех конструкторах.

Лучше относиться к статическим данным-членам как к любым другим статическим переменным. Избегайте их по мере возможности. Если без них не обойтись, то инициализируйте, но старайтесь не прибегать к инициализации во время выполнения.



Глава 6. Управление памятью и ресурсами

C++ предоставляет поразительную гибкость в управлении памятью, но лишь немногие программисты в полной мере знакомы с имеющимися возможностями. Самые разнообразные языковые средства (перегрузка, сокрытие имен, конструкторы и деструкторы, исключения, статические и виртуальные функции, операторные и не-операторные функции) используются согласованно для того, чтобы обеспечить максимальную гибкость и возможности настройки управления памятью. К сожалению, все это довольно сложно, но, наверное, тут ничего не поделаешь.

В этой главе мы познакомимся с тем, как разные средства C++ совместно применяются для управления памятью, как иногда их совместная работа приводит к неожиданным эффектам и как взаимодействие можно упростить.

Поскольку память всего лишь один из многих ресурсов, которыми управляет программа, мы посмотрим, как можно связать с памятью и другие ресурсы, чтобы изощренные механизмы управления памятью в C++ можно было применить и к управлению ими тоже.

Совет 60. Различайте выделение и освобождение памяти для скаляров и для массивов

Можно ли сказать, что `Widget` и массив `Widget` – это одно и то же? Конечно, нет. Тогда почему многие программисты на C++ так удивляются, видя, что для выделения и освобождения памяти под скаляры и массивы применяются разные операторы?

Мы знаем, как выделить и освободить память для одного `Widget`. С помощью операторов `new` и `delete`:

```
Widget *w = new Widget( arg );  
// ...  
delete w;
```

В отличие от большинства прочих операторов в C++, поведение `new` нельзя изменить путем перегрузки. Оператор `new` всегда вызывает функцию с именем `operator new`, которая должна (предположительно) получить какой-то объем памяти. Затем эта память может быть инициализирована. В показанном выше случае применение оператора `new` к `Widget` приведет к вызову функции `operator new`, которая принимает единственный аргумент типа `size_t`. Затем

к полученной неинициализированной памяти будет применен конструктор класса `Widget`, чтобы создать в ней объект этого класса.

Оператор `delete` вызывает деструктор для объекта `Widget`, а затем функцию `operator delete`, которая должна (предположительно) освободить память, уже отведенную несуществующему объекту `Widget`.

Изменить поведение механизмов выделения и освобождения памяти можно путем перегрузки, подмены или сокрытия функций `operator new` и `operator delete`, а не модификации самих операторов `new` и `delete`.

Мы знаем также, как выделять и освобождать память для массивов `Widget`. Но для этого служат не операторы `new` и `delete`.

```
w = new Widget[n];
// ...
delete [] w;
```

Вместо них применяются операторы `new []` и `delete []` (читается «new для массивов» и «delete для массивов»). Подобно `new` и `delete`, поведение операторов `new` и `delete` для массивов модифицировать невозможно. Оператор `new` для массивов сначала вызывает функцию с именем `operator new []`, чтобы получить память, а затем (если необходимо) выполняет инициализацию по умолчанию для каждого элемента массива, начиная с первого. Оператор `delete` для массивов уничтожает все элементы массива в порядке, обратном их инициализации, а затем вызывает функцию с именем `operator delete []`, чтобы освободить память.

Попутно отметим, что зачастую лучше пользоваться стандартным библиотечным классом `vector` вместо массивов. По эффективности `vector` почти не уступает массивам, но безопасен относительно типов и обеспечивает большую гибкость. Можно считать, что класс `vector` – это «интеллектуальный» массив с похожей семантикой. Однако при уничтожении вектора его элементы уничтожаются, начиная с первого, то есть в порядке, обратном тому, что применяется для массивов.

Функции управления памятью должны применяться парами. Если для выделения памяти был использован оператор `new`, то освободить ее надо оператором `delete`. Если память была получена от функции `malloc`, будьте добры освободить ее, вызвав `free`. Иногда использование комбинаций `free/new` и `malloc/delete` будет «работать» для ограниченного множества типов на конкретной платформе, но никаких гарантий не дается:

```
int *ip = new int(12);
// ...
free( ip ); // неправильно!
ip = static_cast<int *>(malloc( sizeof(int) ));
*ip = 12;
// ...
delete ip; // неправильно!
```

То же требование справедливо для выделения и освобождения памяти под массивы. Типичная ошибка – выделить память под массив с помощью `new` для массивов, а освободить с помощью скалярного `delete`. Как и в случае сочетания

`new` с `free`, такой код случайно может отработать без ошибок в конкретной ситуации, тем не менее, он неправилен и в будущем, скорее всего, откажет:

```
double *dp = new double[1];  
// ...  
delete dp; // неправильно!
```

Отметим, что компилятор не может предупредить о том, что вы пытаетесь удалить массив как скаляр, поскольку он не отличает указатель на массив от указателя на отдельный элемент. Обычно оператор `new` для массивов помещает в начало участка памяти, выделенной для массива, не только размер этого участка, но и число элементов в массиве. Эту информацию затем использует оператор `delete` для массивов.

Формат блоков, выделенных для массива и для скаляра, скорее всего, будет различаться. Если скалярный `delete` вызвать для блока, выделенного с помощью `new` для массивов, то информация о размере блока и числе элементов (а она предназначена оператору `delete` для массивов), вероятно, будет интерпретирована неверно, и результат такой операции не определен. Не исключено также, что память для скаляров и массивов выделяется из разных пулов. Тогда попытка вернуть выделенную для массива память в пул, предназначенный для скаляров, приведет к катастрофе:

```
delete [] dp; // правильно
```

Проблема, свойственная выделению памяти для скаляров и массивов, проявляется также в дизайне функций-членов для управления памятью:

```
class Widget {  
public:  
    void *operator new( size_t );  
    void operator delete( void *, size_t );  
    // ...  
};
```

Автор класса `Widget` решил настроить управление памятью, но забыл, что операторы `new` и `delete` для массивов называются иначе, чем для скаляров, поэтому они не скрыты именами функций-членов:

```
Widget *w = new Widget( arg ); // правильно  
// ...  
delete w; // ОК  
w = new Widget[n]; // беда!  
// ...  
delete [] w; // беда!
```

Поскольку в классе `Widget` не объявлены функции `operator new[]` и `operator`, то при управлении памятью для массивов `Widget` будут применяться глобальные версии этих функций. Скорее всего, такое поведение неправильно, автору класса `Widget` надо было бы предоставить также свои версии функций `new` и `delete` для массивов.

Если же такое поведение правильно, то автор должен был бы подчеркнуть этот факт для тех, кому предстоит сопровождать класс `Widget`, поскольку иначе они вполне могут «исправить» ошибку, включив «недостающие» функции. Лучше всего документировать такое решение не в комментариях, а в самом коде:

```
class Widget {
public:
    void *operator new( size_t );
    void operator delete( void *, size_t );
    void *operator new[]( size_t n )
        { return ::operator new[]( n); }
    void operator delete[]( void *p, size_t )
        { ::operator delete[]( p); }
    // ...
};
```

Поскольку версии этих функций-членов встраиваемые, то их вызов во время выполнения ничего не стоит, и даже самого невнимательного сопровождающего этот код должен убедить в том, что в намерения автора входило именно обращение к глобальным версиям функций `new` и `delete` для массивов `Widget`.

Совет 61. Контроль ошибок при выделении памяти

Есть вопросы, которые даже задавать не стоит, и один из них: удачно ли завершилось выделение памяти.

Взгляните, как раньше на C++ писали программы, выделяющие память. Вот код, в котором тщательно проверяется результат каждой такой операции:

```
bool error = false;
String **array = new String *[n];
if( array ) {
    for( String **p = array; p < array+n; ++p ) {
        String *tmp = new String;
        if( tmp )
            *p = tmp;
        else {
            error = true;
            break;
        }
    }
}
else
    error = true;
if( error )
    handleError();
```

Кодировать в таком стиле утомительно, но, быть может, усилия и были бы оправданы, если бы это позволило обнаружить все возможные ошибки выделения памяти. Увы, это не так. В самом конструкторе класса `String` может возникнуть нехватка памяти и сообщить о ней внешней программе не так-то просто. Можно было бы поступить следующим образом: конструктор переводит объект в некоторое ошибочное состояние и поднимает в нем флажок, который проверяют пользователи класса. Перспектива не из приятных. Но даже если предположить, что мы имеем доступ к классу `String` и можем реализовать такое поведение, автору исходного кода и тем, кто будет его сопровождать, предстоит проверять лишнее условие.

Или пренебречь проверкой. Код для проверки ошибок, который при этом получается, редко бывает корректным изначально, а после некоторого периода сопровождения – почти никогда. Лучше вообще ничего не проверять:

```
String **array = new String * [n];
for( String **p = array; p < array+n; ++p )
    *p = new String;
```

Этот код короче, яснее, быстрее и к тому же правилен. Стандартное поведение `new` – возбудить исключение `bad_alloc` в случае невозможности выделить память. Это позволяет инкапсулировать контроль таких ошибок и отделить его от остальной программы, в результате чего проект становится чище, яснее и, как правило, эффективнее.

Как бы то ни было, проверять значение, возвращаемое совместимым со стандартом оператором `new`, всегда бессмысленно, так как `new` либо завершается успешно, либо возбуждает исключение:

```
int *ip = new int;
if( ip ) { // это условие всегда истинно
    // ...
}
else {
    // никогда не выполняется
}
```

Можно воспользоваться стандартной `nothrow`-версией функции `operator new`, которая вернет нулевой указатель в случае ошибки:

```
int *ip = new (nothrow) int;
if( ip ) { // это условие почти всегда истинно
    // ...
}
else {
    // почти никогда не выполняется
}
```

Однако при этом возвращаются все проблемы, связанные со старой семантикой `new`, плюс мы получаем уродливый синтаксис. Лучше не прибегать к такому неуклюжему трюку ради обратной совместимости, а с самого начала проектировать и кодировать программу в расчете на то, что оператор `new` возбуждает исключение.

Исполняющая система также автоматически решает одну весьма неприятную проблему, возникающую в случае нехватки памяти. Напомним, что оператор `new` фактически обращается к двум функциям: `operator new` для выделения памяти и конструктору для ее инициализации:

```
Thing *tp = new Thing( arg );
```

Если мы перехватываем исключение `bad_alloc`, то знаем, что при выделении памяти возникла ошибка. Но где именно? Она могла произойти как во время выделения памяти для самого объекта `Thing`, так и в конструкторе `Thing`. В первом случае нам ничего не надо освобождать, поскольку указатель `tp` так и не был установлен. А во втором – следовало бы вернуть в кучу (неинициализированную) память, на которую указывает `tp`. Однако различить эти два случая очень трудно или даже вовсе невозможно.

К счастью, за нас все сделает исполняющая система. Если память для объекта Thing выделена успешно, но конструктор возбуждает исключение, то будет автоматически вызвана подходящая функция `operator delete` (см. «Совет 62») для освобождения памяти.

Совет 62. Подмена глобальных `new` и `delete`

Подменять стандартные глобальные версии функций `operator new`, `operator delete`, а также `new` и `delete` для массивов – почти всегда неудачная мысль, хотя стандарт этого и не запрещает. Стандартные версии обычно хорошо оптимизированы для универсального управления памятью, и определенные пользователем варианты вряд ли окажутся лучше. (Однако часто бывает разумно переопределить функции-члены для оптимизации механизма управления памятью в конкретном классе или иерархии наследования.)

Вместе со специализированными версиями `operator new` и `operator delete`, реализующими отличное от стандартного поведение, в программу, скорее всего, будут внесены ошибки, поскольку корректность многих функций как из стандартной, так и из других библиотек зависит от определенной в стандарте семантики.

Безопаснее перегрузить глобальный `operator new`, а не подменять его. Предположим, вам нужно заполнить только что выделенную память какой-то комбинацией символов:

```
void *operator new( size_t n, const string &pat ) {
    char *p = static_cast<char *> (::operator new( n ));
    const char *pattern = pat.c_str();
    if( !pattern || !pattern[0] )
        pattern = "\0"; // примечание: два нулевых символа
    const char *f = pattern;
    for( int i = 0; i < n; ++i ) {
        if( !*f )
            f = pattern;
        p[i] = *f++;
    }
    return p;
}
```

Эта версия `operator new` принимает в качестве аргумента строку символов `string`, которой заполняется выделенная память. Компилятор различает стандартный `operator new` и нашу версию с двумя аргументами, применяя правила разрешения перегрузки.

```
string fill( "<garbage>" );
string *string1 = new string( "Hello" ); // стандартная версия
string *string2 =
    new (fill) string( "World!" ); // перегруженная версия
```

Стандарт также определяет перегруженный `operator new`, который, помимо обязательного первого аргумента типа `size_t`, принимает еще один – типа `void *`. Его реализация просто возвращает второй аргумент. (Синтаксическая конструкция `throw()` – это спецификация исключения, которая говорит, что данная

функция не возбуждает никаких исключений. В последующем обсуждении, да и вообще, на нее можно не обращать внимания.)

```
void *operator new( size_t, void *p ) throw()
{ return p; }
```

Это стандартный «размещающий оператор new», применяемый для конструирования объекта по заданному адресу. (В отличие от стандартной функции `operator new` с одним аргументом, попытка подменить размещающий `new` некорректна.) По существу, единственная причина его применения – заставить компилятор вызвать конструктор. Например, во встроенном приложении может возникнуть необходимость сконструировать объект «регистр состояния» по конкретному аппаратному адресу:

```
class StatusRegister {
// ...
};
void *regAddr = reinterpret_cast<void *>(0xFE0000);
// ...
// разместить объект, представляющий регистр, по адресу regAddr
StatusRegister *sr = new (regAddr) StatusRegister;
```

Естественно, объекты, созданные размещающим `new`, должны быть когда-то уничтожены. Однако, поскольку для них не выделялось памяти, то не следует ее и освобождать. Напомним, что оператор `delete` сначала вызывает деструктор удаляемого объекта, а потом функцию `operator delete` для освобождения памяти. Если же объект создавался размещающим `new`, то придется прибегнуть к явному вызову деструктора, дабы избежать любых попыток освободить память:

```
sr->~StatusRegister(); // явный вызов деструктора, а не оператора delete
```

Размещающий оператор `new` и явный вызов деструктора – вещи, конечно, полезные, но могут оказаться весьма опасными, если относиться к ним без должной осторожности. (См. пример из стандартной библиотеки в «Совете 47».)

Отметим, что хотя мы и можем перегрузить `operator delete`, но перегруженная версия никогда не будет вызываться из стандартного выражения `delete`:

```
void *operator new( size_t n, Buffer &buffer ); // перегруженный new
void operator delete( void *p,
    Buffer &buffer ); // соответствующий delete
// ...
Thing *thing1 = new Thing; // используется стандартный operator new
Buffer buf;
Thing *thing2 = new (buf) Thing; // используется перегруженный
// operator new
delete thing2; // неправильно, следовало использовать перегруженную
// версию delete
delete thing1; // правильно, используется стандартный operator delete
```

Как и в случае объекта, созданного размещающим `new`, мы должны вызвать деструктор явно, а затем явно же освободить занятую уничтоженным объектом память, вызвав подходящую версию функции `operator delete`:

```
thing2->~Thing(); // правильно, уничтожить Thing
operator delete( thing2, buf ); // правильно, используется перегруженная
// версию delete
```

На практике память, выделенную перегруженной глобальной функцией `operator new`, часто ошибочно освобождают с помощью стандартной глобальной функции `operator delete`. Один из способов избежать такой ошибки – гарантировать, что любая память, выделенная перегруженной глобальной `operator new`, получена в результате обращения к стандартной глобальной `operator new`. Именно так мы и поступили в реализации первой перегруженной версии выше, и она будет корректно работать со стандартной глобальной `operator delete`:

```
string fill( "<garbage>" );
string *string2 = new (fill) string( "World!" );
// ...
delete string2; // работает
```

Перегруженная версия глобальной `operator new` должна в общем случае либо не выделять никакой памяти, либо служить простой оберткой для стандартной глобальной функции `operator new`.

Часто самое лучшее решение – ничего не делать с глобальными операторными функциями управления памятью, а вместо этого настроить механизм на уровне отдельного класса или иерархии за счет реализации функций-членов `new`, `delete`, `new[]` и `delete[]`.

В конце «Совета 61» мы упомянули, что исполняющая система вызовет «подходящую» функцию `operator delete`, если исключение распространится за пределы выражения `new`:

```
Thing *tp = new Thing( arg );
```

Если выделение памяти для `Thing` завершилось успешно, но конструктор `Thing` возбуждает исключение, то исполняющая система вызовет подходящую функцию `operator delete` для освобождения неинициализированной памяти, на которую указывает `tp`. В примере выше подходящей будет либо глобальная функция `operator delete(void *)`, либо функция-член `operator delete` с такой же сигнатурой. Однако для разных `operator new` нужны разные `operator delete`:

```
Thing *tp = new (buf) Thing( arg );
```

В данном случае подходящая функция `operator delete` – это версия с двумя аргументами, соответствующая перегруженной функции `operator new`, которую мы использовали для выделения памяти под `Thing`: `operator delete(void *, Buffer &)`, и именно эту версию вызовет исполняющая система.

C++ допускает большую гибкость в определении того, как должна вести себя система управления памятью, но платой за гибкость является сложность. Для большинства целей достаточно стандартных глобальных версий `operator new` и `operator delete`. Более сложные подходы применяйте лишь, когда возникает настоятельная необходимость.

Совет 63. Об области видимости и активации функций-членов `new` и `delete`

Функции `operator new` и `operator delete`, являющиеся членами класса, вызываются, когда соответственно создаются и уничтожаются объекты класса,

где они объявлены. Область видимости, в которой находится выражение `new` или `delete`, не имеет значения:

```
class String {
public:
    void *operator new( size_t ); // функция-член operator new
    void operator delete( void * ); // функция-член operator delete
    void *operator new[]( size_t ); // функция-член operator new[]
    void operator delete [] ( void * ); // функция-член operator delete[]
    String( const char * = "" );
    // ...
};

void f() {
    String *sp = new String( "Heap" ); // используется
                                        // String::operator new
    int *ip = new int( 12 ); // используется ::operator new
    delete ip; // используется ::operator delete
    delete sp; // используется String::delete
}
```

Повторим еще раз: область видимости, в которой встречаются выражения `new` или `delete`, не имеет значения; то, какая функция вызывается, зависит лишь от типа создаваемого или уничтожаемого объекта:

```
String::String( const char *s )
: s_( strcpy( new char[strlen(s)+1], s ) )
{}

```

Память для массива символов выделяется в области видимости класса `String`, но при этом используется глобальный оператор `new` для массивов; `char` – это не `String`. Может помочь явная квалификация:

```
String::String( const char *s )
: s_( strcpy( reinterpret_cast<char *>
    (String::operator new[]( strlen(s)+1 )), s ) )
{}

```

Было бы хорошо, если бы могли сказать нечто вроде `String::new char [strlen(s)+1]` для доступа к оператору `new[]` из класса `String` через оператор `new` (ну-ка, разберитесь в этой фразе!), но такой синтаксис недопустим. (Хотя мы можем написать `::new` для доступа к глобальным функциям операторов `new` и `operator new[]` или `::delete` – для доступа к глобальным операторам `delete` и `operator delete[]`.)

Совет 64. Строковые литералы в выражении `throw`

Авторы многих учебников по программированию на C++ демонстрируют механизм исключений, используя в выражении `throw` строковые литералы:

```
throw «Stack underflow!»;
```

Они знают, что это порочная практика, но все равно приводят ее в «педагогических целях». К сожалению, авторы часто забывают сказать читателю, что если тот воспримет этот пример, как рекомендацию поступать так и в собственных программах, то призовет на свою голову все кары небесные.

Никогда не передавайте в качестве объекта исключения строковые литералы. Принципиальная причина в том, что исключение должно быть рано или поздно перехвачено, а перехват осуществляется на основе типа, а не значения:

```
try {
    // ...
}
catch( const char *msg ) {
    string m( msg );
    if( m == "stack underflow" ) // ...
    else if( m == "connection timeout" ) // ...
    else if( m == "security violation" ) // ...
    else throw;
}
```

На практике возбуждение и перехват исключения, представленного строковым литералом, не предоставляет почти никакой информации о природе исключения в самом его типе. В результате предложение `catch` должно перехватить каждое исключение и проверить значение литерала, только тогда станет ясно, следует его обрабатывать или нет. Хуже того, сравнение значений также не дает исчерпывающей информации, а в ходе сопровождения весь механизм может «поломаться», если кто-нибудь заменит строчные буквы заглавными или изменит формат «сообщения об ошибке». В примере выше мы никогда не узнаем, что произошла попытка извлечения из пустого стека.

Это замечание относится к исключениям других встроенных и стандартных типов. Передача в качестве объекта исключения целых чисел, чисел с плавающей точкой, объектов типа `string` или (придет же в голову такая мысль!) множеств (`set`) векторов (`vector`) из чисел с плавающей точкой (`float`) влечет за собой те же самые неприятности. Проще говоря, проблема в том, что, возбуждая исключение встроенного типа, мы не сможем в перехватчике узнать, какой ошибке оно соответствует и как на нее реагировать. Код, возбудивший такое исключение, как бы дразнит нас: «Произошла такая пакость, такая пакость. Догадайся сам, какая!» И у нас нет никакого выбора, кроме как сыграть в игру, в которой мы почти наверняка проиграем.

Тип исключения – это абстрактный тип данных, представляющий исключение. Рекомендации по его проектированию такие же, как для любого другого абстрактного типа данных: определите и поименуйте назначение типа, решите, какие абстрактные операции он должен поддерживать, и реализуйте его. В ходе реализации обращайте внимание на операции инициализации, копирования и преобразования. Все просто. Применение строкового литерала для представления исключения столь же бессмысленно, как представление комплексного числа строкой. Теоретически может сработать, но утомительно и чревато ошибками.

Какую абстрактную концепцию мы пытаемся описать, когда возбуждаем исключение, представляющее попытку извлечения из пустого стека? Да буквально ее саму. Вот так:

```
class StackUnderflow {};
```

Часто тип объекта исключения несет всю необходимую информацию. Нередко можно встретить типы исключений, которые вообще не содержат членов-дан-

ных. Впрочем, наличие еще и текстового описания тоже не повредит. Другая информация об исключении присутствует в объекте гораздо реже:

```
class StackUnderflow {
public:
    StackUnderflow( const char *msg = "stack underflow" );
    virtual ~StackUnderflow();
    virtual const char *what() const;
    // ...
};
```

Если текстовое описание присутствует, то возвращающая его функция должна быть виртуальным членом с именем `what` и показанной выше сигнатурой. Это согласуется со стандартными типами исключений, в каждом из которых такая функция есть. Вообще, имеет смысл делать ваши классы исключений производными от классов стандартных исключений:

```
class StackUnderflow : public std::runtime_error {
public:
    explicit StackUnderflow( const char *msg = "stack underflow" )
        : std::runtime_error( msg ) {}
};
```

Это позволит перехватить исключение либо как `StackUnderflow`, либо как более общий тип `runtime_error`, либо как совсем уж общий тип `exception` (открытый базовый класс `runtime_error`). Также нередко создают некий общий, хотя и нестандартный тип исключения. Обычно он служит базовым классом для всех типов исключений, которые может возбуждать отдельный модуль или библиотека.

```
class ContainerFault {
public:
    virtual ~ContainerFault();
    virtual const char *what() const = 0;
    // . . .
};

class StackUnderflow
    : public std::runtime_error, public ContainerFault {
public:
    explicit StackUnderflow( const char *msg = "stack underflow" )
        : std::runtime_error( msg ) {}
    const char *what() const
        { return std::runtime_error::what(); }
};
```

И, наконец, не забывайте о корректной семантике копирования и уничтожения для типов исключений. В частности, объекты исключений должно быть разрешено копировать, поскольку именно это делает исполняющая система при возбуждении исключения (см. «Совет 65»), а скопированный объект должен быть уничтожен после обработки. Часто мы можем позволить компилятору написать эти операции за нас (см. «Совет 49»):

```
class StackUnderflow
    : public std::runtime_error, public ContainerFault {
public:
    explicit StackUnderflow( const char *msg = "stack underflow" )
```

```

    : std::runtime_error( msg ) {}
    // StackUnderflow( const StackUnderflow & );
    // StackUnderflow &operator = ( const StackUnderflow & );
    const char *what() const
    { return std::runtime_error::what(); }
};

```

Теперь пользователи сами могут решать, как обнаруживать извлечение из пустого стека: как исключение `StackUnderflow` (они знают, что пользуются нашим стеком и держат ухо востро), как более общее исключение `ContainerFault` (они знают, что пользуются нашей библиотекой контейнеров и готовы к любым ошибкам при работе с контейнерами), как `runtime_error` (они ничего не знают о нашей библиотеке, но хотели бы обрабатывать любые стандартные ошибки во время выполнения) или как `exception` (готовы к обработке любых стандартных исключений).

Совет 65. Обрабатывайте исключения правильно

Вопросы общей стратегии и архитектуры обработки исключений все еще оживленно дебатировались. Но на низком уровне принципы возбуждения и перехвата исключений понятны и редко нарушаются.

При выполнении выражения `throw` исполняющая система копирует объект исключения во временный объект, хранящийся в «надежном» месте. Где именно, зависит от платформы, но в любом случае гарантируется, что этот объект существует, пока исключение не будет обработано. Иными словами, временный объект доступен вплоть до момента завершения последнего предложения `catch`, даже если он обрабатывается несколькими перехватчиками. Это важное свойство, поскольку, если говорить прямо, при возбуждении исключения все черти срываются с привязи. Этот временный объект – око тайфуна, бушующего в океане исключения.

Вот почему не стоит возбуждать исключение в виде указателя:

```
throw new StackUnderflow( "operator stack" );
```

Адрес объекта `StackUnderflow` в куче копируется в надежное место, но сама память, на которую он ссылается, не защищена. При таком подходе остается также возможность, что указатель относится к адресу в стеке:

```
StackUnderflow e( "arg stack" );
throw &e;
```

Здесь память, на которую ссылается указатель на объект исключения (напомним, что предложению `throw` передается указатель, а не то, на что он указывает), может уже быть затерта к моменту перехвата исключения. (Кстати говоря, когда объектом исключения является строковый литерал, во временный объект копируется весь массив символов, а не просто адрес первого символа. Практическая ценность этой информации невелика, потому что строковые литералы никогда не следует использовать в качестве объектов исключений. См. «Совет 64».) Кроме

того, указатель может быть нулевым. Кому нужна вся эта дополнительная сложность? Возбуждайте исключения не в виде указателей, а в виде объектов:

```
StackUnderflow e( "arg stack" );
throw e;
```

Объект исключения немедленно копируется во временный объект, так что объявление `e` на самом деле не нужно. Традиционно объектами исключений служат анонимные временные объекты.

```
throw StackUnderflow( «arg stack» );
```

Использование анонимного временного объекта ясно дает понять, что единственное назначение объекта типа `StackUnderflow` – сыграть роль объекта исключения, поскольку время его жизни ограничено выражением `throw`. Если же переменная `e` объявлена явно, то она все равно будет уничтожена во время исполнения выражения `throw`, но остается в области видимости и доступна до конца блока, содержащего объявление. Применение анонимных временных объектов также искореняет некоторые «творческие» подходы к обработке исключений:

```
static StackUnderflow e( Эarg stackЭ );
extern StackUnderflow *argstackerr;
argstackerr = &e;
throw e;
```

Здесь «изобретательный» кодировщик решил приберечь адрес объекта исключения для использования в дальнейшем, возможно, в каком-то `catch`-обработчике. Увы, `argstackerr` указывает не на объект исключения (который представлен временным объектом, хранящимся в недоступном месте), а на уже разрушенный объект, которым тот был инициализирован. Код обработки исключений – не лучшее место для трудно обнаруживаемых ошибок. Старайтесь делать его как можно проще.

Как лучше всего перехватывать объект исключения? Не по значению:

```
try {
    // ...
}
catch( ContainerFault fault ) {
    // ...
}
```

Подумайте, что произойдет, если этот обработчик перехватит объект `StackUnderflow`. Срезка. Так как `StackUnderflow` «является разновидностью» `ContainerFault`, то мы можем инициализировать `fault` сгенерированным объектом исключения, но все данные и поведение, принадлежащие производному классу, будут срезаны. (См. «Совет 30»).

В данном случае, впрочем, проблема не в срезке, поскольку `ContainerFault`, как и подобает уважающему себя базовому классу, является абстрактным (см. «Совет 93»). Поэтому предложение `catch` вообще некорректно. Нельзя перехватить по значению такой объект исключения, как `ContainerFault`.

Перехват по значению ставит перед нами еще более хитрые задачи:

```
catch( StackUnderflow fault ) {
    // произвести частичное восстановление ...
}
```

```

    fault.modifyState(); // моя ошибка
    throw; // повторно возбудить текущее исключение
}

```

Нередко бывает, что в предложении `catch` производится частичное восстановление после ошибки, информация об этом сохраняется в объекте исключения, после чего исключение возбуждается повторно для дополнительной обработки. Увы, ничего подобного здесь не произойдет. Мы модифицировали состояние локальной копии объекта исключения, а передали следующему обработчику сам объект исключения (неизменный).

Для простоты и с целью обойти все эти сложности всегда возбуждайте исключения в виде анонимных временных объектов и перехватывайте их по ссылке.

Следите за тем, чтобы в обработчике не возникало ошибок из-за копирования. Чаше всего это случается, когда обработчик возбуждает новое исключение, а не повторно возбуждает уже существующее:

```

catch( ContainerFault &fault ) {
    // произвести частичное восстановление ...
    if( condition )
        throw; // повторно возбудить исключение
    else {
        ContainerFault myFault( fault );
        myFault.modifyState(); // все еще моя ошибка
        throw myFault; // новый объект исключения
    }
}

```

В данном случае записанные изменения не будут потеряны, зато теряется исходный тип исключения. Предположим, что первоначальное исключение имело тип `StackUnderflow`. Когда оно перехватывается по ссылке на `ContainerFault`, динамический тип объекта исключения все еще `StackUnderflow`, поэтому повторно возбужденное исключение имеет все шансы быть перехваченным как обработчиком `StackUnderflow`, так и обработчиком `ContainerFault`. Но новый объект исключения `myFault` имеет тип `ContainerFault` и, значит, не может быть перехвачен обработчиком `StackUnderflow`. Лучше повторно возбуждать уже имеющееся исключение, чем обрабатывать его и возбуждать новое:

```

catch( ContainerFault &fault ) {
    // произвести частичное восстановление ...
    if( !condition )
        fault.modifyState();
    throw;
}

```

На наше счастье, базовый класс `ContainerFault` абстрактный, поэтому в данном случае эта ошибка не проявит себя; вообще говоря, базовые классы всегда стоит делать абстрактными. Очевидно, этот совет неприменим, если вы обязаны возбудить исключение совершенно другого типа:

```

catch( ContainerFault &fault ) {
    // произвести частичное восстановление ...
    if( out_of_memory )
        throw bad_alloc(); // возбудить новое исключение
    fault.modifyState();
}

```

```
    throw; // повторно возбудить
}
```

Еще одна распространенная ошибка связана с порядком catch-обработчиков. Поскольку эти обработчики перебираются в том порядке, в котором записаны (как условия в предложении if-elseif, а не как ветви switch), то указанные в них типы должны быть упорядочены от более к менее специфичным. Если какие-то типы не связаны отношением базовый-производный, то располагайте их в логическом порядке:

```
catch( ContainerFault &fault ) {
    // произвести частичное восстановление ...
    fault.modifyState(); // не моя ошибка
    throw;
}
catch( StackUnderflow &fault ) {
    // ...
}
catch( exception & ) {
    // ...
}
```

Показанная выше цепочка обработчиков никогда не перехватит исключение StackUnderflow, так как более общий тип ContainerFault встречается в ней раньше.

Механизм обработки исключения предлагает массу возможностей усложнить себе задачу, но вовсе не обязательно соглашаться на это предложение. Возбуждая и перехватывая исключения, стремитесь к простоте.

Совет 66. Внимательно относитесь к адресам локальных объектов

Не возвращайте указатель или ссылку на локальную переменную. Большинство компиляторов предупреждает о таких попытках; отнеситесь к этому предупреждению серьезно.

Исчезающие фреймы стека

Если переменная автоматическая, то состояние памяти, которую она занимала, после возврата из функции не определено:

```
char *newLabel1() {
    static int labNo = 0;
    char buffer[16]; // см. Совет 2
    sprintf( buffer, "label%d", labNo++ );
    return buffer;
}
```

Эта функция работает от случая к случаю. После возврата фрейм, принадлежащий функции newLabel1, выталкивается из стека, и память (включая и ту, где находился массив buffer) освобождается для следующего вызова функции. Од-

нако, пока очередная функция не вызвана, возвращенный указатель, хотя и недействительный, все еще пригоден для использования:

```
char *uniqueLab = newLabel1();
char mybuf[16], *pmybuf = mybuf;
while( *pmybuf++ = *uniqueLab++ );
```

Вряд ли сопровождающий будет мириться с таким кодом долго. Возможно, он решит выделить буфер из кучи:

```
char *pmybuf = new char[16];
```

Возможно, он решит отказаться от копирования буфера «вручную»:

```
strcpy( pmybuf, uniqueLab );
```

А, может быть, он захочет воспользоваться более абстрактным типом, чем простой массив символов:

```
std::string mybuf( uniqueLab );
```

Любая из этих модификаций может привести к затиранию локальной памяти, на которую указывает `uniqueLab`:

Затирание статических переменных

Если переменная статическая, то последующий вызов той же функции отразится на результатах предыдущих вызовов:

```
char *newLabel2() {
    static int labNo = 0;
    static char buffer[16];
    sprintf( buffer, "label%d", labNo++ );
    return buffer;
}
```

Память, отведенная под буфер, доступна и после возврата из функции, но последующий вызов той же функции может изменить ее состояние:

```
// случай 1
cout << "first: " << newLabel2() << ' ';
cout << "second: " << newLabel2() << endl;

// случай 2
cout << "first: " << newLabel2() << ' '
  << "second: " << newLabel2() << endl;
```

В первом случае будут напечатаны разные значения `label`. Во втором случае мы, скорее всего (хотя и не наверняка), напечатаем одно и то же значение дважды. Вероятно, тот, кто хорошо знаком с необычной реализацией функции `newLabel2` написал первый вариант код, чтобы разбить вывод на два предложения и тем самым компенсировать дефект реализации. Сопровождающий же не был знаком со странностями `newLabel2` и объединил два предложения в одно. В результате появилась ошибка. Хуже того, не исключено, что после объединения предложений программа продолжает вести себя так же, как и раньше, но в будущем ее поведение может непредсказуемо измениться. (См. «Совет 14».)

Идиоматические трудности

Нас подстерегает и еще одна опасность. Не забывайте, что у пользователей функции обычно нет доступа к ее реализации, а есть лишь объявление, из которого они должны иметь возможность понять, как трактовать возвращаемое функцией значение. Да, необходимую информацию можно поместить в комментарий (см. «Совет 1»), но лучше проектировать функцию так, чтобы правильное использование напрашивалось само собой.

Не возвращайте ссылку на память, выделенную внутри функции. Пользователи обязательно забудут освободить эту память, что приведет к утечкам:

```
int &f()
{ return *new int( 5 ); }
// ...
int i = f(); // утечка памяти!
```

Корректная программа преобразовала бы ссылку в адрес или скопировала результат и освободила память. Спасибо, мне это не подходит:

```
int *ip = &f(); // один кошмарный способ
int &tmp = f(); // другой, ничем не лучше
int i = tmp;
delete &tmp;
```

Особенно плоха эта идея для перегруженных операторов:

```
Complex &operator +( const Complex &a, const Complex &b )
{ return *new Complex( a.re+b.re, a.im+b.im ); }
// ...
Complex a, b, c;
a = b + c + a + b; // море утечек!
```

Возвращайте вместо ссылки указатель или не выделяйте память, а возвращайте результат по значению:

```
int *f() { return new int(5); }
Complex operator +( Complex a, Complex b )
{ return Complex( a.re+b.re, a.im+b.im ); }
```

Идиоматически, пользователи функции, возвращающей указатель, готовы к тому, что от них может потребоваться освободить память, на которую он ссылается, и предпримут некоторые усилия, чтобы выяснить, так ли это на самом деле (скажем, прочтут комментарий). Если же функция возвращает ссылку, то мало кто станет уточнять детали.

Проблемы локальной области видимости

Проблемы с временем жизни локальных переменных встречаются не только на границах между функциями, но и при наличии вложенных областей видимости внутри одной функции:

```
void localScope( int x ) {
    char *cp = 0;
    if ( x ) {
        char buf1[] = "asdf";
```

```

cp = buf1; // порочная идея!
char buf2[] = "qwerty";
char *cp1 = buf2;
// ...
}
if( x-1 ) {
    char *cp2 = 0; // накладывается на buf1?
    // ...
}
if( cp )
    printf( cp ); // возможно, ошибка ...
}

```

Компиляторам предоставлена большая гибкость в определении того, как размещать в памяти локальные переменные. В зависимости от платформы и заданных при вызове флагов компилятор может разместить `buf1` и `cp2` в одной и той же памяти. Это допустимо, так как области видимости и время жизни `buf1` и `cp2` не пересекаются. Если перекрытие действительно произойдет, то `buf1` будет затерта, что отразится на поведении `printf` (возможно, она ничего не напечатает). Переносимости ради не стоит полагаться на конкретную структуру фрейма стека.

Исправление ошибки путем добавления `static`

Столкнувшись с особо вредной ошибкой, иногда удается «устранить» ее, добавив спецификатор хранения `static`:

```

// ...
char buf[MAX];
long count = 0;
// ...
int i = 0;
while( i++ <= MAX )
    if( buf[i] == '\0' ) {
        buf[i] = '*';
        ++count;
    }
assert( count <= i );
// ...

```

Цикл в этой программе написан неправильно, иногда он пишет в память за концом буфера `buf`, изменяя содержимое переменной `count`. В результате утверждение `assert` не выполняется. В суете, которая иногда сопровождает попытки исправить ошибку, программист может объявить переменную `count` локальной статической, и программа станет работать:

```

char buf[MAX];
static long count;
// ...
count = 0;
int i = 0;
while( i++ <= MAX )
    if( buf[i] == '\0' ) {
        buf[i] = '*';
        ++count;
    }
assert( count <= i );

```


Многие программисты, не желающие задаваться вопросом, почему ошибку удалось так легко исправить, на этом и остановятся. Увы, ошибка не исчезла, просто переместилась в другое место. Она затаилась и терпеливо дожидается подходящего момента, чтобы нанести удар.

Сделав локальную переменную `count` статической, мы переместили ее из стека совсем в другую область памяти, где хранятся все статические объекты. Конечно, она больше не затирается. Но теперь на `count` распространяется проблема, описанная в разделе «Затирание статических переменных». А, кроме того, затираться будет какая-то другая локальная переменная, возможно, пока еще несуществующая. Правильное решение, как обычно, состоит в том, чтобы исправить ошибку, а не спрятать ее.

```
char buf[MAX];
long count = 0;
// ...
for( int i = 1; i < MAX; ++i )
    if( buf[i] == '\0' ) {
        buf[i] = '*';
        ++count;
    }
// ...
```

Совет 67. Помните, что захват ресурса есть инициализация

Стыдно, что многие начинающие программисты на C++ не видят замечательной симметрии между конструкторами и деструкторами. По большей части, они пришли в C++ из других языков, которые пытаются защитить их от сложностей работы с указателями и управления памятью. В безопасности и под контролем. Счастливы в своем неведении. Привыкли программировать именно так, как предписал автор языка. Единственно правильным способом. Тем, который автор считает правильным.

К счастью, C++ питает больше уважения к своим поклонникам и предоставляет куда большую гибкость в способах применения языка. Я вовсе не хочу сказать, что у нас нет общих принципов и полезных идиом (см. «Совет 10»). И одна из самых важных – это идиома «захват ресурса есть инициализация». Может быть, ее название и труднопроизносимо, зато это простая и допускающая обобщение техника, смысл которой заключается в привязке ресурса к памяти, после чего управлять тем и другим можно эффективно и предсказуемо.

Порядок выполнения конструкторов и деструкторов противоположен. Объект класса всегда конструируется в одном и том порядке:

- сначала подобъекты виртуальных базовых классов («в порядке их появления в объявлении и рекурсивного обхода ациклического ориентированного графа базовых классов в глубину слева направо», как гласит стандарт);
- затем непосредственные базовые классы в порядке их появления в списке базовых классов в объявлении класса;

- затем нестатические данные-члены класса в порядке их объявления;
- затем выполняется тело конструктора.

В деструкторе порядок выполнения обратный: тело деструктора, данные-члены в порядке, обратном их следованию в объявлении класса, непосредственные базовые классы в порядке, обратном объявлению, и виртуальные базовые классы. Удобно представлять себе конструирование, как заталкивание некоторой последовательности в стек, а уничтожение как извлечение из стека в обратном порядке. Симметрия конструирования и уничтожения считается настолько важной, что все конструкторы классов выполняют инициализацию объектов в одной и той же последовательности даже, если списки инициализации членов записаны в разном порядке (см. «Совет 52»).

В качестве побочного эффекта или, если хотите, результата инициализации конструктор захватывает ресурсы, необходимые для работы объекта. Часто порядок захвата ресурсов имеет значение (например, нужно заблокировать базу данных перед тем, как писать в нее; нужно сначала получить описатель файла, а затем уже записывать в него данные). Задача деструктора – освободить ресурсы в порядке, обратном тому, в котором они захватывались. Тот факт, что конструкторов может быть много, а деструктор только один, означает, что все конструкторы должны инициализировать компоненты в одном и том же порядке.

(Кстати говоря, так было не всегда. На ранних этапах развития языка порядок инициализации в конструкторах не фиксировался, что вызывало немало проблем в проектах любой сложности. Как и большинство правил C++, это стало результатом вдумчивого проектирования и опыта практического применения.)

Симметрия между конструированием и уничтожением остается в силе даже, если мы переходим от структуры самого объекта к использованию нескольких объектов. Рассмотрим простой класс для трассировки:

➤> gotcha67/trace.h

```
class Trace {
public:
    Trace( const char *msg )
        : m_( msg ) { cout << "Entering " << m_ << endl; }
    ~Trace()
        { cout << "Exiting " << m_ << endl; }
private:
    const char *m_;
};
```

Возможно, этот класс чересчур прост, поскольку в нем предполагается, что инициализатор корректен и время его жизни не меньше времени жизни объекта Trace, но для наших целей он годится. Объект Trace выводит одно сообщение, когда создается, и другое сообщение, когда разрушается. Поэтому его можно применить для трассировки потока исполнения:

➤> gotcha67/trace.cpp

```
Trace a( "global" );
```

```

void loopy( int cond1, int cond2 ) {
    Trace b( "function body" );
it: Trace c( "later in body" );
    if( cond1 == cond2 )
        return;
    if( cond1-1 ) {
        Trace d( "if" );
        static Trace stat( "local static" );
        while( -cond1 ) {
            Trace e( "loop" );
            if( cond1 == cond2 )
                goto it;
        }
        Trace f( "after loop" );
    }
    Trace g( "after if" );
}

```

При вызове функции `loopy` с аргументами 4 и 2 получаем следующий результат:

```

Entering global
Entering function body
Entering later in body
Entering if
Entering local static
Entering loop
Exiting loop
Entering loop
Exiting loop
Exiting if
Exiting later in body
Entering later in body
Exiting later in body
Exiting function body
Exiting local static
Exiting global

```

Из этих сообщений ясно видно, что время жизни объекта `Trace` определяется текущей областью видимости. В частности, обратите внимание, какое влияние оказывают предложения `goto` и `return` на время жизни активных объектов `Trace`. Ни одну из этих ветвей не надо рассматривать как пример образцового кодирования, но во время сопровождения такие конструкции иногда появляются:

```

void doDB() {
    lockDB();
    // работа с базой данных ...
    unlockDB();
}

```

В примере выше мы позаботились о том, чтобы заблокировать базу данных перед доступом и разблокировать по окончании работы. Увы, этот тщательно написанный код может развалиться в ходе сопровождения, особенно если между блокировкой и разблокировкой находится много текста:

```

void doDB() {
    lockDB();

```

```
// ...
if( i_feel_like_it ) // if (у_меня_такое_настроение)
    return;
// ...
unlockDB();
}
```

Теперь ошибка возникает всякий раз, как у функции `doDB` возникает «подходящее настроение»: база данных остается заблокированной, и это без сомнения вызовет проблемы в каком-то другом месте. На самом деле, даже первоначальная версия кода написана неправильно, поскольку между моментами блокировки и разблокировки базы может возникнуть исключение. Эффект тот же, что и при любом другом пути исполнения, обходящем `unlockDB`: база данных остается заблокированной.

Можно попытаться исправить эту ошибку, явно обрабатывая исключения и дав суровые наставления сопровождающим:

```
void doDB() {
    lockDB();
    try {
        // работа с базой данных ...
    }
    catch( ... ) {
        unlockDB();
        throw;
    }
    unlockDB();
}
```

Но это решение многословно, нетехнологично, неудобно для сопровождения. Вы рискуете быть принятым за сотрудника «подотдела отдела департамента по борьбе с избыточностью». Правильно написанный код для обработки исключений обычно состоит из нескольких `try`-блоков. Не проще ли рассматривать захват ресурса как инициализацию:

```
class DBLock {
public:
    DBLock() { lockDB(); }
    ~DBLock() { unlockDB(); }
};

void doDB() {
    DBLock lock;
    // работа с базой данных ...
}
```

Создание объекта `DBLock` приводит к захвату ресурса, то есть к блокировке базы данных. Когда объект `DBLock` покидает область видимости по любой причине, деструктор освобождает ресурс и, значит, разблокирует базу данных. Эта идиома встречается в C++ так часто, что ее даже не замечают. Но всякий раз, создавая объект любого из стандартных классов `string`, `vector`, `list` да и многих других, вы применяете идиому захвата ресурса как инициализации.

Кстати, хочу привлечь ваше внимание к двум распространенным ошибкам, связанным с применением дескрипторных классов ресурсов, подобных `DBLock`:

```
void doDB() {
    DBLock lock1; // правильно
    DBLock lock2(); // ой ли?
    DBLock(); // так ли?
    // работа с базой данных ...
}
```

Объявление `lock1` корректно; это объект `DBLock`, который попадает в область видимости непосредственно перед завершающей объявление точкой с запятой и покидает ее в конце блока, содержащего объявление (в данном случае, в конце функции). Объявление `lock1` говорит, что это функция, которая не принимает аргументов и возвращает `DBLock` (см. «Совет 19»). Это не ошибка, но, скорее всего, не то, что вы хотите, поскольку ни блокировки, ни разблокировки не происходит.

Следующая строка содержит выражение, которое создает анонимный временный объект `DBLock`. Блокировка базы данных при этом произойдет, но, поскольку анонимный временный объект выходит из области видимости в конце выражения (непосредственно перед точкой с запятой), то база данных сразу же разблокируется. Маловероятно, что вам нужно именно это.

Для объектов, размещенных в куче, универсальным дескриптором ресурса может служить стандартный шаблонный класс `auto_ptr` (см. «Совет 10» и «Совет 68»).

Совет 68. Правильно используйте `auto_ptr`

Стандартный шаблон `auto_ptr` – это простой и полезный дескрипторный класс с необычной семантикой копирования (см. «Совет 10»). Как правило, его применение не вызывает сомнений:

```
template <typename T>
void print( Container<T> &c ) {
    auto_ptr< Iter<T> > i( c.genIter() );
    for( i->reset(); !i->done(); i->next() ) {
        cout << i->get() << endl;
        examine( c );
    }
    // неявная очистка ...
}
```

Это типичный пример использования `auto_ptr`. Смысл его в том, чтобы гарантировать освобождение памяти и ресурсов, связанных с находящимся в куче объектом в момент, когда направленный на него указатель покидает область видимости. (Более подробное обсуждение иерархии `Container` см. в «Совете 90».) Предположение заключается в том, что память для `Iter<T>`, которую возвращает функция `genIter`, была выделена из кучи. Поэтому `auto_ptr< Iter<T> >` может вызвать оператор `delete` для освобождения этой памяти при выходе `auto_ptr` из области видимости.

Однако с таким использованием `auto_ptr` связано две типичных ошибки. Во-первых, `auto_ptr` не должен ссылаться на массив. Рассмотрим пример:

```
void calc( double src[], int len ) {
```

```
double *tmp = new double[len];  
// ...  
delete [] tmp;  
}
```

В функции `calc` может возникнуть утечка памяти, выделенной для массива `tmp`, если во время ее выполнения возникнет исключение или в ходе неаккуратно сопровождаемого будет добавлен преждевременный возврат. Необходим объект для управления ресурсом, а стандарт предлагает для этой цели `auto_ptr`:

```
void calc( double src[], int len ) {  
    auto_ptr<double> tmp( new double[len] );  
    // ...  
}
```

Однако `auto_ptr` рассчитан только на управление одиночными объектами, а не массивами. Когда `tmp` покидает область видимости и вызывается деструктор, то к массиву `double`, память для которого была выделена с помощью `new` для массивов, применяется оператор `delete` для скаляров (см. «Совет 60»). Ведь компилятор не способен различить указатели на массив и на одиночный объект. И, что еще хуже, этот код на некоторых платформах может случайно отработать правильно, а ошибка проявится лишь при переносе на другую платформу или установке новой версии компилятора на текущей.

Гораздо надежнее вместо массива `double` воспользоваться стандартным классом `vector`. Он как раз и является дескриптором ресурса для массива, в каком-то роде «`auto_ptr`», к тому же обладает рядом дополнительных возможностей. Одновременно мы избавимся от примитивного и опасного использования формального аргумента-указателя, маскирующегося под массив:

```
void calc( vector<double> &src ) {  
    vector<double> tmp( src.size() );  
    // ...  
}
```

Еще одна распространенная ошибка – поместить `auto_ptr` в STL-контейнер. Контейнеры из библиотеки STL предъявляют не слишком много требований к своим элементам, но одно из них – следование традиционной семантике копирования.

Шаблон `auto_ptr` определен в стандарте так, что добавление его конкретизаций в STL-контейнер незаконно; при этом должны возникать ошибки во время компиляции (причем довольно загадочные). К несчастью, многие современные реализации отстают от стандарта.

В одной весьма распространенной реализации `auto_ptr` семантика копирования допускает помещение конкретизированных из него элементов в контейнер, ими даже можно пользоваться. Но лишь до тех пор, пока вы не установите обновленную или вообще другую версию стандартной библиотеки. В этом случае ваша программа перестанет компилироваться. Исправлять эту ошибку противно, хотя обычно и не сложно.

Хуже, когда реализация `auto_ptr` не полностью отвечает стандарту, поэтому поместить конкретизированные из него элементы в STL-контейнер можно, но се-

Совет 68. Правильно используйте `auto_ptr` 167

мантика копирования не согласуется с требованиями STL. Как описано в «Совете 10», при копировании `auto_ptr` владение передается целевому объекту, а в исходном указатель сбрасывается в нуль:

```
auto_ptr<Employee> e1( new Hourly );
auto_ptr<Employee> e2( e1 ); // e1 нулевой
e1 = e2; // e2 нулевой
```

Это свойство полезно во многих контекстах, но для элемента STL-контейнера неприемлемо:

```
vector< auto_ptr<Employee> > payroll;
// ...
list< auto_ptr<Employee> > temp;
copy( payroll.begin(), payroll.end(), back_inserter(temp) );
```

На некоторых платформах этот код будет компилироваться и работать, но делать не то, что вам надо. Вектор (`vector`) указателей на `Employee` будет скопирован в список (`list`), но после копирования все указатели в векторе окажутся нулевыми!

Не помещайте `auto_ptr`-элементы в STL-контейнер, даже если текущая платформа это позволяет.

Глава 7. Полиморфизм

Наряду с абстрагированием данных, наследование и полиморфизм относятся к числу фундаментальных средств объектно-ориентированного программирования. Полиморфизм реализован в С++ эффективно и гибко, но механизм отличается сложностью.

В этой главе мы увидим, что гибкостью полиморфизма в С++ нередко злоупотребляют, и предложим рекомендации, которые позволят «обуздать» сложность реализации. Попутно мы узнаем, как реализованы в С++ наследование и виртуальные функции, и как эта реализация отражается на самом языке.

Совет 69. Кодирование типов

Один из верных признаков «моей первой программы на С++» – наличие кода типа в качестве одного из данных-членов. (В моей первой программе на С++ они присутствовали и принесли мне немало горестей.) В объектно-ориентированном программировании тип объекта определяется его поведением, а не состоянием. Лишь в редких случаях в хорошо спроектированной программе на С++ возникает необходимость в конкретных кодах типа, но хранить его в качестве члена данных не нужно никогда.

```
class Base {
public:
    enum Tcode { DER1, DER2, DER3 };
    Base( Tcode c ) : code_( c ) {}
    virtual ~Base();
    int tcode() const
    { return code_; }
    virtual void f() = 0;
private:
    Tcode code_;
};
class Der1 : public Base {
public:
    Der1() : Base( DER1 ) {}
    void f();
};
```

В приведенном выше коде мы видим типичное проявление проблемы. Дело в том, что проектировщик еще не достаточно уверен в себе, чтобы полностью перейти на объектно-ориентированный дизайн, при котором во всей иерархии последовательно используется динамическое связывание. Код типа оставлен (по мысли проектировщика) на случай, если когда-нибудь потребуется предложение `switch` (конструкция, унаследованная от С и выглядящая в С++ чужеродной) или возникнет необходимость точно определить, с объектом какого из подклассов

Base мы имеем дело. Но применять код типа в объектно-ориентированном проекте все равно, что пытаться нырнуть, держась одной ногой за вышку: ничего не выйдет, но будет больно.

В C++ никогда не встречается ветвление по коду типа в объектно-ориентированных частях программы. Никогда. Суть проблемы в перечислении `Tcode` в классе `Base`. При добавлении любого производного класса приходится изменять исходный текст, поскольку базовый класс знает о своих потомках и тесно связан с ними. Нет никакой гарантии, что все части кода, в которых проверяются элементы перечисления `Tcode`, будут обновлены. При сопровождении программ на C неизменно модифицируется лишь 98% предложений `switch`, в которых проверяются коды типов. При использовании же виртуальных функций такой проблемы просто не возникает, так зачем же тратить силы на то, чтобы искусственно внести ее.

Хранение кодов типов в виде данных-членов приводит и к более тонким проблемам. Не исключено, что код типа копируется из одного подкласса `Base` в другой. В большой и сложной программе может появиться такой код:

```
Base *bp1 = new Der1;
Base *bp2 = new Der2;
*bp2 = *bp1; // катастрофа!
```

Обратите внимание, что объект `Der2` не изменился. Тип определяется своим поведением, а поведение `Der2` в значительной мере определяется тем, как его настроил конструктор во время инициализации. Возьмем, например, указатель на таблицу виртуальных функций, который был неявно вставлен компилятором и определяет, какие именно функции объекта вызывать при динамическом связывании. Приведенный выше код его не изменил, но явно объявленные в `Base` данные-члены изменились (см. «Совет 50» и «Совет 78»). На рис. 7.1 закрашены те части объекта, на который указывает `bp2`, что будут изменены в результате присваивания.

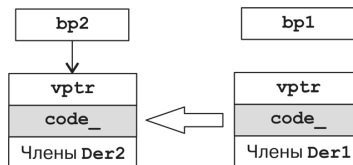


Рис. 7.1. Что происходит с подобъектом базового класса в результате присваивания объекта одного производного класса объекту другого класса? Копируются лишь данные-члены, объявленные в базовом классе. Неявно добавленный компилятором механизм виртуализации не копируется

После того как тип объекта был задан во время конструирования, он уже не изменяется. Однако объект `Der2`, на который указывает `bp2`, заявляет, что он является объектом `Der1`. Любой код, основанный на предложении `switch`, поверит

этому заявлению, тогда как код, в котором используется динамическое связывание, его проигнорирует. Не объект, а шизофреник какой-то.

Если вы все же столкнулись с очень редкой ситуацией, когда код типа необходим, придерживайтесь следующих рекомендаций. Во-первых, не храните код в члене данных. Заведите вместо этого виртуальную функцию, поскольку она более тесно ассоциирует код типа с настоящим типом (поведением) объекта. Тем самым вы сумеете избежать шизофренического поведения, свойственного слабой ассоциации:

```
class Base {
public:
    enum Tcode { DER1, DER2, DER3 };
    Base();
    virtual ~Base();
    virtual int tcode() const = 0;
    virtual void f() = 0;
    // ...
};
class Der1 : public Base {
public:
    Der1() : Base() {}
    void f();
    int tcode() const
        { return DER1; }
};
```

Во-вторых, будет лучше оставить базовый класс в неведении относительно своих потомков, так как это уменьшает число зависимостей внутри иерархии и позволяет легко добавлять и убирать производные классы в ходе сопровождения программы. Отсюда следует, что набор кодов типов следует вынести из самой программы, быть может, включив в официальный документ, где перечислены все коды типов или описан алгоритм порождения новых кодов. Каждый производный класс может знать о своем коде, но от остальной программы эта информация должна быть скрыта.

Ситуация, когда проектировщик вынужден прибегнуть к кодам типов, возникает, например, при взаимодействии с каким-то модулем, который не следует объектно-ориентированным принципам. Например, из внешнего источника читается какое-то «сообщение», а его тип определяется целочисленным кодом. Длина и структура остальной части сообщения зависят от типа. Что делать проектировщику?

Лучше всего воздвигнуть брандмауэр. Это означает, что та часть программы, которая имеет дело с внешним представлением сообщения, ветвится по его коду и генерирует объект, в котором кода типа уже нет. Для прочих частей программы код типа уже не важен, они работают на уровне динамического связывания. Отметим, что при необходимости совсем несложно восстановить исходное сообщение из объекта, поскольку объект может знать о соответствующем ему коде, не храня его в члене данных.

Один из недостатков такой схемы заключается в том, что необходимо изменять и перекомпилировать предложение switch (пусть даже единственное) при

изменении набора возможных сообщений. Но такая модификация ограничена лишь кодом самого брандмауэра:

```
Msg *firewall( RawMsgSource &src ) {
    switch( src.msgcode ) {
        case MSG1:
            return new Msg1( src );
        case MSG2:
            return new Msg2( src );
        // и т.д.
    }
}
```

В некоторых случаях неприемлема даже такая ограниченная перекомпиляция. Например, если новые типы сообщений необходимо добавить, не останавливая работу приложения. Тогда можно воспользоваться «взаимозаменяемостью» управляющих структур и подставить вместо компилируемого условного кода структуру данных, интерпретируемую во время выполнения. В нашем примере с сообщениями мы могли обойтись простой последовательностью объектов, каждый из которых представляет сообщение одного типа:

➤ `gotcha69/firewall.h`

```
class MsgType {
public:
    virtual ~MsgType() {}
    virtual int code() const = 0;
    virtual Msg *generate( RawMsgSource & ) const = 0;
};
class Firewall { // паттерн Monostate
public:
    void addMsgType( const MsgType * );
    Msg *genMsg( RawMsgSource & );
private:
    typedef std::vector<MsgType *> C;
    typedef C::iterator I;
    static C types_;
};
```

В данном случае интерпретатор тривиален: мы просто просматриваем всю последовательность, пока не найдем интересующий нас код. Если код найден, порождается объект, представляющий нужное сообщение:

➤ `gotcha69/firewall.cpp`

```
Msg *Firewall::genMsg( RawMsgSource &src ) {
    int code = src.msgcode;
    for( I i( types_.begin() ); i != types_.end(); ++i )
        if( code == i->code() )
            return i->generate( src );
    return 0;
}
```

Структуру данных легко пополнять новыми типами сообщений:

```
void Firewall::addMsgType( const MsgType *mt )
{ types_.push_back(mt); }
```

Реализация типов индивидуальных сообщений тривиальна:

```
class Msg1Type : public MsgType {
```

```
public:
    MsgType()
    { Firewall::addMsgType( this ); }
    int code() const
    { return MSG1; }
    Msg *generate( RawMsgSource &src ) const
    { return new Msg1( src ); }
};
```

Заполнить список объектами `MsgType` можно по-разному. Самый простой путь: объявить статическую переменную этого типа. Его конструктор в качестве побочного эффекта добавит объект `MsgType` в статический список `list`, являющийся частью брандмауэра `Firewall`:

```
static MsgType msgtype;
```

Отметим, что порядок инициализации этих статических объектов несуществен. Если бы это было не так, вступил бы в действие «Совет 55». Новые объекты `MsgType` можно добавлять в список во время выполнения путем динамической загрузки.

Раз уж речь зашла о статических объектах, обращаю ваше внимание на то, что класс `Firewall` выше имеет только статические данные-члены, но манипулируют ими нестатические функции-члены. Это пример паттерна `Monostate`. Он может служить альтернативой паттерну `Singleton` в качестве способа избежать использования глобальных переменных. Паттерн `Singleton` вынуждает пользователей обращаться к единственному объекту через статическую функцию-член `instance`. Если бы класс `Firewall` был реализован как `Singleton`, то пришлось бы писать примерно такой код:

```
Firewall::instance().addMessageType( mt );
```

Паттерн же `Monostate` допускает наличие неограниченного числа объектов, но все они должны ссылаться на один и тот же статический член данных. Никакого специального протокола доступа не требуется:

```
Firewall fw;
fw.genMsg( rawsource );
FireWall().genMsg( rawsource ); // другой объект, то же состояние
```

Совет 70. Невиртуальный деструктор базового класса

Эта тема рассматривалась почти в каждом учебнике по программированию на C++ на протяжении прошедших пятнадцати лет. Во-первых, самый надежный способ узнать, предназначен класс для использования в качестве базового или нет, – посмотреть, есть ли в нем виртуальный деструктор. И это лучшая документация. Если деструктор не виртуален, то классу, скорее всего, нельзя наследовать.

Неопределенное поведение

После опубликования стандарта этот аргумент стал еще более веским. Во-первых, при уничтожении объекта производного класса через интерфейс его базового

вого класса возникает неопределенное поведение, если деструктор базового класса не виртуален:

```
class Base {
    Resource *br;
    // ...
    ~Base() // note: nonvirtual
        { delete br; }
};
class Derived : public Base {
    OtherResource *dr;
    // ...
    ~Derived()
        { delete dr; }
};
Base *bp = new Base;
// ...
delete bp; // fine . . .
bp = new Derived;
// ...
delete bp; // скрытая ошибка!
```

Скорее всего, будет просто вызван деструктор базового класса для объекта производного класса, а это ошибка. Но компилятор может в этом случае сделать все, что ему угодно (сбросить дамп памяти? отправить электронное сообщение вашему начальнику? оформить вам пожизненную подписку на «Еженедельник объектно-ориентированного COBOL'a?»).

Виртуальные статические функции-члены

С другой стороны, наличие виртуального деструктора в базовом классе позволяет добиться эффекта вызова виртуальной статической функции-члена. Спецификаторы `virtual` и `static` взаимно исключающие, а операторные функции для управления памятью (операторы `new`, `delete`, `new []` и `delete []`) – это статические функции-члены. Но, как и в случае виртуального деструктора, во время удаления объекта, необходимо вызывать наиболее специализированную функцию `operator delete`, особенно когда существует соответствующая функция-член `operator new` (см. «Совет 63»):

```
class B {
public:
    virtual ~B();
    void *operator new( size_t );
    void operator delete( void *, size_t );
};
class D : public B {
public:
    ~D();
    void *operator new( size_t );
    void operator delete( void *, size_t );
};
// ...
B *bp = getABofSomeSort();
// ...
delete bp; // вызывается delete из производного класса!
```

Благодаря наличию виртуального деструктора в базовом классе, стандарт гарантирует, что функция-член `operator delete` будет вызвана в «области видимости динамического типа класса». Это означает, что из деструктора производного класса будет вызвана функция-член `operator delete`. Поскольку деструктор производного класса находится в области видимости производного класса (естественно!), то берется именно та функция `operator delete`, которая определена в производном классе.

Резюмируя, можно сказать, что, хотя `operator delete` – это статическая функция-член, но наличие виртуального деструктора в базовом классе гарантирует, что при удалении объекта через указатель на базовый класс будет вызвана версия `operator delete` из производного класса. Например, в примере выше при удалении указателя `bp` вызывается деструктор `D`, и в результате происходит обращение к функции `operator delete` из класса `D`, и вторым аргументом ей передается `sizeof(D)`, а не `sizeof(B)`. Элегантно. Виртуальные статические функции.

Всех обманем

Раньше программы на C++ часто писались в предположении, что при одиночном наследовании адрес подобъекта базового класса совпадает с адресом всего объекта (см. «Совет 29»).

```
class B {
    int b1, b2;
};
class D : public B {
    int d1, d2;
};
D *dp = new D;
B *bp = dp;
```

Хотя стандарт таких гарантий не дает, но в данном случае при размещении объекта `D` в памяти подобъект `B` почти наверняка окажется в начале, как на рис. 7.2.

Однако, если в производном классе объявлена виртуальная функция, то компилятор, скорее всего, вставит скрытый указатель на таблицу виртуальных функций (см. «Совет 78»). На рис. 7.3 показаны две наиболее распространенных схемы размещения объекта в памяти для этого случая.

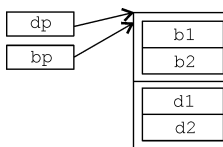


Рис. 7.2. Вероятное размещение в памяти объекта, не содержащего виртуальных функций, при одиночном наследовании. При данной реализации адреса начала полного объекта `D` и подобъекта `B` совпадают

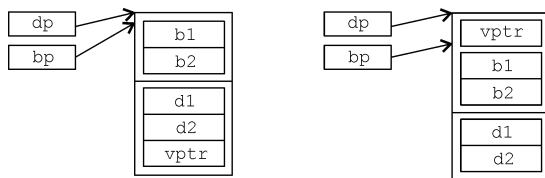


Рис. 7.3. Две возможных схемы размещения в памяти объекта при одиночном наследовании, когда в производном классе объявлена виртуальная функция, отсутствующая в базовом классе. На схеме слева указатель на таблицу виртуальных функций находится в конце полного объекта, а на схеме справа — в начале. В результате во втором случае подобъект базового класса смещен относительно начала полного объекта

В первом случае молчаливое предположение о том, что объект производного класса и его подобъект базового класса имеют один и тот же адрес, все еще верно, а во втором — уже нет. Конечно, правильный способ решения этой проблемы состоит в том, чтобы переписать код, основанный на неподтверждаемом стандартом допущении. На практике это обычно означает, что надо перестать использовать `void *` для хранения указателей на объекты классов (см. «Совет 29»). Если это сложно, то можно поместить виртуальную функцию в базовый класс. Тогда есть больше шансов, что объект будет размещен в памяти в соответствии с нестандартным предположением об эквивалентности адресов (см. рис. 7.4).

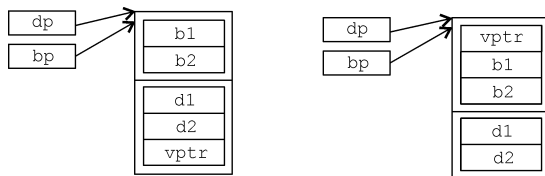


Рис. 7.4. Вероятное размещение в памяти объекта при одиночном наследовании, когда виртуальная функция объявлена в производном классе

Обычно самым лучшим кандидатом на роль такой виртуальной функции в базовом классе является деструктор.

Исключения из правил

Даже у самых фундаментальных идиом бывают исключения. Например, иногда удобно поместить набор имен типов, статических функций-членов и статических данных-членов в один «пакет»:

```
namespace std {
    template <class Arg, class Res>
    struct unary_function {
        typedef Arg argument_type;
```

```

        typedef Res result_type;
    };
}

```

В таком случае виртуальный деструктор не обязателен, так как классы, конкретизируемые из этого шаблона, не управляют ресурсами, которые надо освободить. Класс был тщательно спроектирован таким образом, чтобы использование его в качестве базового не отражалось ни на потреблении памяти, ни на времени работы:

```

struct Expired : public unary_function<Deal *, bool> {
    bool operator ()( const Deal *d ) const
        { return d->expired(); }
};

```

Наконец, шаблон `unary_function` входит в состав стандартной библиотеки. Опытные программисты на C++ знают, что его не надо применять в качестве полнофункционального базового класса и потому не будут пытаться манипулировать объектами производных классов через интерфейс `unary_function`. Это особый случай.

Вот еще один пример из хорошо известной, но не стандартной библиотеки. Проектные ограничения те же, что для рассмотренного выше стандартного базового класса, но, поскольку этот класс стандартным все же не является, автор не мог полагаться на то, что программисты будут с ним знакомы:

```

namespace Loki {
    struct OpNewCreator {
        template <class T>
        static T *Create() { return new T; }
    protected:
        ~OpNewCreator() {}
    };
}

```

В данном случае автор решил объявить защищенный, встраиваемый, не виртуальный деструктор. Эффективность по времени и памяти при этом сохраняется, использовать деструктор не по назначению становится трудно, и имеется явное указание на то, что класс предназначен только для употребления в качестве базового.

Это исключительные случаи, а, вообще говоря, при проектировании лучше всего включать в любой базовый класс виртуальный деструктор.

Совет 71. Скрытие не виртуальных функций

Невиртуальная функция определяет инвариант относительно иерархии (или ее части) с корнем в базовом классе. Проектировщики производных классов не могут переопределять не виртуальные функции и не должны скрывать их (см. «Совет 77»). Обоснование этого правила фундаментально и вместе с тем очевидно: любой другой подход ведет к нарушению полиморфизма.

У полиморфного объекта есть одна реализация (класс), но много типов. Будучи знакомы с абстрактными типами данных, мы знаем, что тип – это набор операций, и эти операции представлены в открытом интерфейсе. Например, класс

Совет 71. Скрытие неvirtуальных функций 177

Circle (круг) «является разновидностью» Shape (геометрическая фигура) и должен работать предсказуемо и согласованно при обращении через любой из своих интерфейсов:

```
class Shape {
public:
    virtual ~Shape();
    virtual void draw() const = 0;
    void move( Point );
    // . . .
};
class Circle : public Shape {
public:
    Circle();
    ~Circle();
    void draw() const;
    void move( Point );
    // . . .
};
```

Проектировщик класса Circle решил скрыть функцию move (быть может, в базовом классе предполагается, что Point описывает верхний угол, а в Circle это центр). Теперь объект Circle может вести себя по-разному в зависимости от того, через какой интерфейс к нему обращаются:

```
void doShape( Shape *s, void (Shape::*op)(Point), Point p )
{ (s->*op)( p ); }
Circle *c = new Circle;
Point pt( x, y );
c->move( pt );
doShape( c, &Shape::move, pt ); //беда!
```

Скрытие неvirtуальной функции из базового класса лишь усложняет пользование иерархией, не давая ничего взамен:

```
class B {
public:
    void f();
    void f( int );
};
class D : public B{
public:
    void f(); // неудачная мысль!
};
```

```
B *bp = new D;
bp->f(); // неожиданность! вызывается B::f() для объекта D
D *dp = new D;
dp->f( 123 ); // ошибка! B::f(int) скрыта
```

Вirtуальные и чисто virtуальные функции – это механизмы определения зависимости от типа реализации. Когда мы переопределяем virtуальную функцию в производном классе, то гарантируем, что во время выполнения данный объект будет обладать только одной реализацией и, следовательно, единственным поведением. Стало быть, поведение объекта не зависит от того, через какой интерфейс к нему обратились.

Попутно заметим, что виртуальные функции можно вызывать и, прибегая к помощи механизма виртуализации, если воспользоваться оператором разрешения области видимости, но это особенность использования интерфейса, а не его дизайна. Однако, в этом смысле переопределенная виртуальная функция из базового класса все-таки доступна в производных от него классах:

```
class Msg {
public:
    virtual void send();
    // . . .
};
class XMsg : public Msg {
public:
    void send();
    // . . .
};
// . . .
XMsg *xmsg = new XMsg;
xmsg->send(); // вызывается XMsg::send
xmsg->Msg::send(); // вызывается скрытая/переопределенная Msg::send
```

Это необходимый иногда трюк, но не часть проекта. Впрочем, возможность не виртуально обратиться к переопределенной виртуальной функции из базового класса может использоваться проектировщиком и осознанно. Такое обращение обычно используется, чтобы предоставить общую реализацию из базового класса в распоряжение переопределенных функций в производных классах.

Иллюстрацией этого подхода может служить стандартная реализация паттерна Decorator, который применяется, чтобы дополнить, а не подменить существующие в иерархии функции:

```
gotcha71/msgdecorator.h
class MsgDecorator : public Msg {
public:
    void send() = 0;
    // ...
private:
    Msg *decorated_;
};
inline void MsgDecorator::send() {
    decorated_->send(); // перенаправить вызов
}
```

Класс `MsgDecorator` является абстрактным, поскольку в нем объявлена чисто виртуальная функция `send`. Конкретные классы, производные от `MsgDecorator`, должны переопределить `MsgDecorator::send`. Но, хотя вызывать `MsgDecorator::send` как виртуальную функцию невозможно (разве что необычным, нестандартным и чреватым ошибками способом; см. «Совет 75»), разрешено обращаться к ней не виртуально, с помощью оператора разрешения области видимости. Реализация `MsgDecorator::send` представляет ту общую часть, которую должны включать все переопределенные функции `send` в производных классах. Воспользоваться ей можно с помощью не виртуального вызова:

➤ `gotcha71/msgdecorator.cpp`

```
void BeepDecorator::send() {
```

```
MsgDecorator::send(); // вызвать функцию из базового класса
cout << '\a' << flush; // дополнительно поведение ...
}
```

Есть и альтернатива: в классе `MsgDecorator` можно было бы объявить защищенную не виртуальную функцию, реализующую общее поведение, но применение имеющей определение чисто виртуальной функции более четко указывает на ее предназначение: быть вызванной из функций производных классов.

Совет 72. Не делайте шаблонные методы слишком гибкими

Паттерн **Template Method** (Шаблонный Метод) разбивает алгоритм на постоянную и переменную части. Постоянная часть алгоритма определяется как не виртуальный член базового класса. Но эта не виртуальная функция позволяет настроить некоторые шаги алгоритма в производных классах. Обычно алгоритм вызывает защищенные виртуальные функции, которые можно переопределить в производных классах. (Обращаю ваше внимание, что паттерн `Template Method` не имеет ничего общего с шаблонами в смысле языка `C++`.)

Это позволяет проектировщику базового класса сохранить общую структуру алгоритма для всех производных классов, обеспечив в то же время необходимую настройку:

```
class Base {
public:
    // . . .
    void algorithm();
protected:
    virtual bool hook1() const;
    virtual void hook2() = 0;
};
void Base::algorithm() {
    // . . .
    if( hook1() ) {
        // . . .
        hook2();
    }
    // . . .
}
```

Паттерн `Template Method` дает возможность управлять разделением обязанностей между виртуальными и не виртуальными функциями. Интересно посмотреть, сколько проектных ограничений мы можем навязать проектировщикам производных классов, пользуясь лишь этой идиомой:

```
class Base {
public:
    virtual ~Base(); // я базовый класс
    virtual bool verify() const = 0; // проверять вы должны сами
    virtual void doit(); // можете сделать это, как я, или иначе
    long id() const; // пользуйтесь этой функцией или идите в другое место
    void jump(); // если я говорю "прыгай", вы можете спросить лишь ...
}
```

```
protected:
    virtual double howHigh() const; // ... насколько высоко, и ...
    virtual int howManyTimes() const = 0; // ... сколько раз.
};
```

Многие начинающие проектировщики ошибочно полагают, что проект должен быть максимально гибким. Поэтому они объявляют виртуальным и сам алгоритм шаблонного метода, считая, что лишняя гибкость никому не мешает. Неверно. Проектировщики производных классов больше всего выиграют от недвусмысленного контракта, «подписанного» базовым классом. Если вызывающая программа ожидает от шаблонного метода конкретного общего поведения, то производные классы должны учитывать это пожелание и реализовывать его.

Совет 73. Перегрузка виртуальных функций

Что не так в следующем фрагменте базового класса?

```
class Thing {
public:
    // ...
    virtual void update( int );
    virtual void update( double );
};
```

Рассмотрим производный класс, разработанный проектировщиком, который решил, что только версия функции с параметром типа `int` должна вести себя по-другому:

```
class MyThing : public Thing {
public:
    // ...
    void update( int );
};
```

Здесь мы имеем несчастливое сочетание перегрузки и переопределения – понятий, которые не имеют между собой ничего общего. Результат аналогичен тому, что случается при сокрытии невиртуальной функции из базового класса: поведение объекта `MyThing` будет меняться в зависимости от того, через какой интерфейс к нему обращаются:

```
MyThing *mt = new MyThing;
Thing *t = mt;
t->update( 12.3 ); // правильно, из базового класса
mt->update( 12.3 ); // ой, из производного класса!
```

При вызове `mt->update(12.3)` будет найдено имя `update` в производном классе, которое соответствует параметрам после преобразования аргумента типа `double` в `int`. Вероятно, программист имел в виду не это. Но даже если программист с необычным взглядом на мир хотел получить именно такое поведение, то вряд ли этот код придется по вкусу будущим сопровождающим с более традиционным умонастроением.

Чем возражать против перегрузки виртуальных функций, мы могли бы, как часто предлагалось в книгах по C++, вышедших до принятия стандарта, настоять

на том, чтобы проектировщики производных классов переопределяли все функции из набора перегруженных с одним и тем же именем. Но это не практичный подход, так как требует, чтобы проектировщики всех производных классов следовали некоему единому правилу. Однако многие производные классы, в частности, создаваемые для расширения каркаса, разрабатываются в контексте, очень далеком от базового класса и тех соглашений о проектировании и кодировании, в рамках которого он создавался.

Так или иначе, но отказ от перегрузки виртуальных функций не налагает серьезных ограничений на интерфейс базового класса. Если перегрузка так важна для конкретного базового класса, никто не мешает перегрузить неvirtуальные функции, которые обращаются к виртуальным функциям с разными именами:

```
class Thing {
public:
    // ...
    void update( int );
    void update( double );
protected:
    virtual void updateInt( int );
    virtual void updateDouble( double );
};
inline void Thing::update( int a )
{ updateInt( a ); }
inline void Thing::update( double d )
{ updateDouble( d ); }
```

Теперь производный класс может независимо переопределять любую виртуальную функции, не опасаясь нарушить полиморфизм. Конечно, в производном классе не следует объявлять функцию-член с именем `update`; запрет на сокрытие неvirtуальных членов базового класса по-прежнему действует!

У этого правила есть исключения, но встречаются они сравнительно редко. Одно из них связано с традиционной реализацией паттерна Visitor (Посетитель) (см. «Совет 77»).

Совет 74. Виртуальные функции с аргументами по умолчанию

Это по существу та же проблема, что и перегрузка виртуальных функций. Как и перегрузка, аргументы по умолчанию – не более чем синтаксическое удобство, позволяющее изменить интерфейс функции, не добавляя новое поведение:

```
class Thing {
    // ...
    virtual void doitNtimes( int numTimes = 12 );
};
class MyThing : public Thing {
    // ...
    void doitNtimes( int numTimes = 10 );
};
```

Возникающая проблема проистекает из несоответствия между статическим и динамическим поведением объекта. Найти источник ошибки часто бывает довольно трудно:

```
Thing *t = new MyThing;  
t->doitNtimes();
```

Предположение состоит в том, что для объекта `MyThing` функция `doitNtimes` должна по умолчанию сделать то, что ей положено 10 раз, тогда как для других типов, производных от `Thing`, 12 раз. К сожалению, значение аргумента по умолчанию применяется статически, а в статически определенном базовом классе оно равно 12 и таковым и останется во всех динамически связанных с ним производных классах.

Можно было бы попытаться обойти эту проблему, потребовав, чтобы проектировщики всех производных классов в точности повторяли значения аргументов по умолчанию, заданные для переопределяемой функции в базовом классе. Но по ряду причин это порочная идея.

Во-первых, многие разработчики не прислушаются к этому совету, такова уж их природа. (Возможно, они утратили доверие к базовому классу, увидев, как в нем инициализируются аргументы по умолчанию, и решили поступить по-своему.)

Во-вторых, такого рода рекомендации делают производные классы излишне уязвимыми по отношению к изменениям в базовом классе. Если в базовом классе меняется значение аргумента по умолчанию, то придется провести аналогичные изменения во всех производных классах. Как правило, это невозможно.

В-третьих, семантика аргумента по умолчанию может меняться в зависимости от того, в каком месте исходного текста он появляется. Синтаксически идентичные инициализаторы могут иметь разный смысл в контексте базового и производного класса:

```
// В файле thing.h ...  
const int minim = 12;  
namespace SCI {  
class Thing {  
    // ...  
    virtual void doitNtimes( int numTimes = minim );  
    // используется ::minim  
};  
}  
  
// В файле mything.h ...  
namespace SCI {  
const int minim = 10;  
class MyThing : public Thing {  
    // ...  
    void doitNtimes( int numTimes = minim );  
    // используется ::minim  
};  
}
```

Трудно винить проектировщика производного класса за то, что он выбрал не тот `minim`, особенно если объявление `SCI::minim` было добавлено после того, как был написан класс `MyThing`.

Простейшее и самое безопасное решение: вообще не употреблять аргументы по умолчанию в виртуальных функциях. Как и в случае перегрузки виртуальных функций, реализовать нужный нам интерфейс можно с помощью нехитрого трюка со встраиванием:

```
class Thing {
    // ...
    void doitNtimes( int numTimes = minim )
    { doitNtimesImpl( numTimes ); }
protected:
    virtual void doitNtimesImpl( int numTimes );
};
```

Пользователи иерархии Thing будут работать со значением аргумента по умолчанию, статически определенным в базовом классе, а производные классы могут модифицировать поведение функции, не заботясь о том, каково значение инициализатора.

Совет 75. Вызовы виртуальных функций из конструкторов и деструкторов

Конструкторы служат для того, чтобы захватить ресурсы, необходимые объекту для выполнения работы, а назначение деструктора – освободить эти ресурсы. Так почему же не выразить это архитектурное решение явно в проекте базового класса?

```
class B {
public:
    B() { seize(); }
    virtual ~B() { release(); }
protected:
    virtual void seize() {}
    virtual void release() {}
};
```

Затем производные классы могли бы переопределить функции seize и release и тем самым настроить способ захвата ресурсов:

```
class D : public B {
public:
    D() {}
    ~D() {}
    void seize() {
        B::seize(); // получить ресурсы для базового класса
        // получить ресурсы для производного класса ...
    }
    void release() {
        // освободить ресурсы производного класса ...
        B::release(); // освободить ресурсы базового класса
    }
};
// . . .
D x; // никакие ресурсы не захвачены и не освобождены!
```

На первом шаге инициализации x конструктор производного класса вызывает конструктор базового класса, который, в свою очередь, вызывает виртуальную

функцию `seize`. На последнем шаге уничтожения x деструктор производного класса вызывает деструктор базового класса, который обращается к виртуальной функции `release`. Однако ни захвата, ни освобождения ресурсов не происходит.

Проблема в том, что в точке, где из конструктора производного класса вызывается конструктор базового, объект x еще не имеет тип `D`. Конструктор базового класса инициализирует подобъект `B` в объекте x , в результате чего он будет вести себя как `B`. Поэтому при вызове виртуальной функции `seize` происходит динамическая привязка к `B::seize`. То же самое, только в обратном порядке происходит при уничтожении объекта. Когда деструктор производного класса вызывает деструктор базового, объект x уже не принадлежит типу `D`, а подобъект `B` ведет себя, как положено объекту класса `B`. Поэтому вызов виртуальной функции `release` связывается с `B::release`.

В данном случае простейшее решение: воспользоваться встроенным механизмом реализации конструирования и уничтожения сложных объектов. Код, осуществляющий захват и освобождение ресурсов для подобъектов базового класса, должен присутствовать соответственно в конструкторах и деструкторах:

```
class B {
public:
    B() {
        // получить ресурсы для базового класса...
    }
    virtual ~B() {
        // освободить ресурсы для базового класса ...
    }
};
class D : public B {
public:
    D() {
        // получить ресурсы для производного класса ...
    }
    ~D() {
        // освободить ресурсы для производного класса ...
    }
};
// ...
D x; // работает!
```

Кстати говоря, таким способом иногда можно вызвать чисто виртуальную функцию с помощью виртуальной, а не статической последовательности вызова:

```
class Abstract {
public:
    Abstract();
    Abstract( const Abstract & );
    virtual bool validate() const = 0;
    // ...
};
bool Abstract::validate() const
{ return true; }
Abstract::Abstract() {
    if( validate() ) // попытка вызвать чисто виртуальную функцию
        // ...
};
```


Однако согласно стандарту поведение такого вызова не определено. На некоторых платформах подобный виртуальный вызов функции просто аварийно завершает программу, поскольку предпринимается попытка обратиться к функции по нулевому указателю. А иногда (и это самое опасное) действительно вызывается `Abstract::validate`. Но даже если вы этого и хотели, такой код нестабилен и не переносим.

Отметим, что мы здесь говорим только о вызове виртуальной функции для объекта, который находится в процессе конструирования или уничтожения. Никто не запрещает вызывать из конструктора или деструктора виртуальные функции другого, полностью сконструированного объекта:

```
Abstract::Abstract( const Abstract &that ) {
    if ( that.validate() ) // правильно
        // ...
}
```

Совет 76. Виртуальное присваивание

Присваивание может быть виртуальным, но использование такой возможности редко бывает оправдано. Например, можно построить иерархию контейнеров, поддерживающих виртуальное присваивание через интерфейс базового класса:

```
template <typename T>
class Container {
public:
    virtual Container &operator =( const T & ) = 0;
    // . . .
};
template <typename T>
class List : public Container<T> {
    List &operator =( const T & );
    // . . .
};
template <typename T>
class Array : public Container<T> {
    Array &operator =( const T & );
    // . . .
};
// . . .
Container<int> &c( getCurrentContainer() );
c = 12; // понятно ли, что имеется в виду?
```

Обратите внимание, что это копирующее присваивание, так как тип аргумента отличается от типа контейнера. (О том, почему тип значения, возвращаемого переопределенными в производных классах операторами присваивания, может отличаться от того типа, который возвращают операторы присваивания в базовом классе, см. «Совет 77».) Назначение этого оператора присваивания: присвоить всем элементам контейнера `Container` одно и то же значение. Увы, опыт показывает, что такое применение присваивания часто интерпретируют неверно; некоторые пользователи полагают, что должен измениться размер контейнера, а другие, что задается значение только первого элемента (см. «Совет 84»). Безо-

паснее отказаться от перегрузки оператора в пользу недвусмысленной не-операторной функции:

```
template <typename T>
class Container {
public:
    virtual void setAll( const T &newElementValue ) = 0;
    // . . .
};
// . . .
Container<int> &c( getCurrentContainer() );
c.setAll( 12 ); // семантика ясна
```

Копирующий оператор присваивания тоже может быть виртуальным, но эта идея редко осмыслена, так как определенный в производном классе копирующий оператор присваивания не переопределяет такой же оператор из базового класса:

```
template <typename T>
class Container {
public:
    virtual Container &operator =( const Container & ) = 0;
    // ...
};
template <typename T>
class List : public Container<T> {
    List &operator =( const List & ); // не переопределяет!
    List &operator =( const Container<T> & ); // переопределяет...
    // ...
};
// ...
Container<int> &c1 = getMeAList();
Container<int> &c2 = getMeAnArray();
c1 = c2; // присвоить массив списку???
```

Виртуальный копирующий оператор присваивания позволил бы присвоить объект одного производного класса объекту производного класса совсем другого типа! Мало найдется случаев, когда это имеет смысл. Избегайте виртуальных копирующих операторов присваивания.

Можно попытаться отыскать место для виртуального копирующего оператора присваивания в иерархии `Container` выше, так как, возможно, имеет смысл присвоить содержимое одного контейнера (массива) другому контейнеру (списку). При этом предполагается, что каждый тип контейнера знает все об остальных типах (что обычно считается неправильным подходом к проектированию) или что в проекте участвует довольно сложный каркас. Проще, а значит, и лучше, было бы иметь не виртуальную функцию `copyContent`, являющуюся или не являющуюся членом класса `Container`, и написанную в терминах виртуальных функций или итераторов, которые извлекают элементы из исходного контейнера и вставляют их в целевой:

```
Container<int> &c1 = getMeAList();
Container<int> &c2 = getMeAnArray();
c1.copyContent( c2 ); // скопировать содержимое массива в список
```

Пример такого подхода можно найти в контейнерах из стандартной библиотеки, которые допускают инициализацию контейнера последовательностью, взятой из существующего контейнера другого типа:

```
vector<int> v;
// ...
list<int> el( v.begin(), v.end() );
```

Часто вместо виртуального присваивания лучше применить виртуальный конструктор копирования. Разумеется, в С++ нет никаких виртуальных конструкторов, зато есть идиома «виртуального конструктора», которая теперь больше известна как паттерн **Prototype** (Прототип). Вместо того чтобы присваивать объект неизвестного типа, мы его клонируем. Базовый класс предоставляет чисто виртуальную операцию `clone`, которая переопределяется в производных классах так, чтобы объект возвращал точную копию самого себя. Обычно копия создается конструктором копирования производного класса, так что операцию `clone` можно считать в некотором смысле виртуальным конструктором.

➤ gotcha90/container.h

```
template <typename T>
class Container {
public:
    virtual Container *clone() const = 0;
    // . . .
};
template <typename T>
class List : public Container<T> {
    List( const List & );
    List *clone() const
    { return new List( *this ); }
    // . . .
};
template <typename T>
class Array : public Container<T> {
    Array( const Array & );
    Array *clone() const
    { return new Array( *this ); }
    // . . .
};
// . . .
Container<int> *cp = getCurrentContainer();
Container<int> *cp2 = cp->clone();
```

Применяя паттерн **Prototype**, мы по сути дела говорим: «Я точно не знаю, на что указываю, но хочу получить точно такое же!»

Совет 77. Различайте перегрузку, переопределение и сокрытие

Всегда шокирует, когда после долгой технической дискуссии вы вдруг обнаруживаете, что ваш собеседник не видит разницы между перегрузкой и переопределением. Добавьте еще нечеткое представление о том, что такое сокрытие имен, и

вы поймете, что я называю бессмысленной беседой. Так быть не должно, различать эти понятия необходимо.

В C++ перегрузка – это просто использование одного и того же идентификатора для разных функций, объявленных в одной и той же области видимости. Последнее важно:

```
bool process( Credit & );
bool process( Acceptance & );
bool process( OrderForm & );
```

Ясно, что эти три глобальные функции перегружены. У них общий идентификатор `process`, и они объявлены в одной и той же области видимости. Компилятор отличает одну от другой по фактическому аргументу, переданному при вызове `process`. Это разумно. Если я прошу обработать объект типа `Acceptance`, то ожидаю, что будет вызвана вторая из перечисленных выше функций, а не первая и не третья. В C++ имя функции состоит из комбинации ее идентификатора (в данном случае `process`) и типов формальных аргументов в ее объявлении. А теперь погрузим эти три функции в класс:

```
class Processor {
public:
    virtual ~Processor();
    bool process( Credit & );
    bool process( Acceptance & );
    bool process( OrderForm & );
    // ...
};
```

Они по-прежнему перегружены, и компилятор еще может отличить одну от другой по типу фактического аргумента. Наличие виртуального деструктора в классе `Processor` говорит о том, что проектировщик намеревался использовать его в качестве базового класса, так что мы вправе расширить функциональность путем наследования:

```
class MyProcessor : public Processor {
public:
    bool process( Rejection & );
    // ...
};
```

Только не так. Функция `process` из производного класса не перегружает функции `process` из базового. Она скрывает их:

```
Acceptance a;
MyProcessor p;
p.process( a ); // ошибка!
```

Когда компилятор ищет имя `process` в области видимости производного класса, он находит единственную функцию-кандидат. В соответствии с объявлением она принимает аргумент типа `Rejection`, так что типы аргументов не совпадают (если только нет какого-нибудь преобразования из `Acceptance` в `Rejection`). Конец дискуссии. Компилятор не станет продолжать поиск функций `process` в объемлющих областях видимости. Функция `process` из произ-

водного класса объявлена в области видимости производного, а не базового класса и, следовательно, не может перегружать функции из базового класса.

Можно импортировать объявления из базового класса в область видимости производного с помощью using-объявления:

```
class MyProcessor : public Processor {
public:
    using Processor::process;
    bool process( Rejection & );
    // ...
};
```

Теперь все четыре функции находятся в одной и той же области видимости, поэтому функция process из производного класса перегружает три функции, явно импортированные в его область видимости. Отметим, что такой метод проектирования не является образцовым, поскольку он слишком сложен, а сложный дизайн всегда хуже простого, если только не компенсирует сложность чем-то еще.

В данном случае, объект Rejection можно обработать только через интерфейс класса MyProcessor, а при попытке обработать его через интерфейс Processor возникнет ошибка компиляции. Однако, если Rejection можно преобразовать в Acceptance, OrderForm или Credit, то вызов завершится успешно через любой интерфейс, только поведение будет различным.

Переопределение возможно лишь при наличии в базовом классе виртуальной функции. Точка. Переопределение не имеет ничего общего с перегрузкой. Невиртуальная функция из базового класса не может быть переопределена, а только скрыта:

```
class Doer {
public:
    virtual ~Doer();
    bool doit( Credit & );
    virtual bool doit( Acceptance & );
    virtual bool doit( OrderForm & );
    virtual bool doit( Rejection & );
    // ...
};
class MyDoer : public Doer {
private:
    bool doit( Credit & ); // #1, скрывает
    bool doit( Acceptance & ); // #2, переопределяет
    virtual bool doit( Rejection & ) const; // #3, не переопределяет
    double doit( OrderForm & ); // #4, ошибка
    // ...
};
```

(Обратите внимание, что классы Doer выше приведены для иллюстрации, а не как примеры правильного проектирования. На самом деле, перегружать виртуальные функции, как правило, вредно. См. «Совет 73».)

Функция doit с меткой #1 не переопределяет одноименную функцию из базового класса, поскольку последняя не-виртуальна. Однако она скрывает все четыре функции doit из базового класса.

Функция с меткой #2 переопределяет одноименную функцию из базового класса. Отметим, что уровень доступа на переопределении никак не отражается. Независимо, что в базовом классе функция открыта, а в производном закрыта или наоборот. По принятому соглашению, переопределенная функция в производном классе имеет тот же уровень доступа, что и соответствующая ей функция в базовом классе. Впрочем, в некоторых случаях бывает полезно отклониться от стандартной практики:

```
class Visitor {
public:
    virtual void visit( Acceptance & );
    virtual void visit( Credit & );
    virtual void visit( OrderForm & );
    virtual int numHits();
};
class ValidVisitor : public Visitor {
    void visit( Acceptance & ); // переопределяет
    void visit( Credit & ); // переопределяет
    int numHits( int ); // #5, не виртуальная
};
```

В данном случае проектировщик иерархии решил позволить настройку поведения базового класса, но хотел бы, чтобы пользователи иерархии все же ограничились интерфейсом базового класса. Чтобы добиться своей цели, проектировщик объявил функции-члены базового класса открытыми, но в производных классах переопределил их, сделав закрытыми.

Обратите также внимание на то, что употреблять ключевое слово `virtual` при переопределении функций в производных классах совершенно необязательно. Семантика объявления функции в производном классе будет одной и той же вне зависимости от наличия этого слова:

```
class MyValidVisitor : public ValidVisitor {
    void visit( Credit & ); // переопределяет
    void visit( OrderForm & ); // переопределяет
    int numHits(); // #6, виртуальная, переопределяет Visitor::numHits
};
```

Часто думают, что если при переопределении функции в производном классе опустить слово `virtual`, то будет запрещено переопределять эту функцию в классах ниже по иерархии. Это не так, `MyValidVisitor::visit(Credit &)` переопределяет соответствующие функции в классах `ValidVisitor` и `Visitor`.

Кроме того, никто не запрещает переопределять в производном классе функции, расположенные в далеко отстоящем базовом классе. `MyValidVisitor::visit(OrderForm &)` переопределяет соответствующую функцию в классе `Visitor`.

В производном классе разрешено даже переопределять функцию из далеко отстоящего базового класса, которая не видна в области видимости производного класса. Например, функция с меткой #5 `ValidVisitor::numHits` не переопределяет функцию `Visitor::numHits` из базового класса, но скрывает ее от классов, расположенных ниже по иерархии. Тем не менее, функция `MyValidVisitor::numHits` переопределяет `Visitor::numHits`.

Функция-член класса `MyDoer` с меткой #3 заслуживает более пристального внимания. Она не переопределяет виртуальную функцию из базового класса, поскольку является константной, а в базовом классе соответствующей константной функции нет. Константность – это часть сигнатуры функции (см. «Совет 82»).

Функция-член класса `MyDoer` с меткой #4 ошибочна. Она переопределяет соответствующую виртуальную функцию из базового класса, но типы возвращаемых ими значений несовместимы; функция из базового класса возвращает `bool`, а функция из производного класса – `double`. Результат: ошибка компиляции.

В общем случае, если функция в производном классе переопределяет функцию из базового, то она должна возвращать значение того же типа. Это гарантирует статическую безопасность типов при связывании во время исполнения. Виртуальная функция из производного класса обычно вызывается через интерфейс базового класса (ведь для этого мы и пишем виртуальные функции). Компилятор должен сгенерировать код в предположении, что тип возвращенного функцией значения (все равно, будет она связана во время исполнения с функцией из базового или производного класса,) совпадает с тем, что объявлен в базовом классе.

В случае некорректного объявления #4 функция из производного класса попытается скопировать объект размером `sizeof(double)` байтов в область памяти, зарезервированную для значения размером всего `sizeof(bool)` байтов. Даже если эти размеры совместимы (то есть длина `bool` не меньше длины `double`), маловероятно, что интерпретация `double` как `bool` даст разумный результат.

У этого правила есть исключение, известное под название «ковариантные возвращаемые типы». (Не путайте ковариантность с контравариантностью! См. «Совет 46».) Типы значений, возвращаемых функцией-членом базового класса и перегружающей ее функцией из производного класса, ковариантны, если оба являются указателями или ссылками на объекты классов, и тип, возвращаемый функцией из производного класса, связан отношением «является» с типом, возвращаемым функцией из базового класса. Сразу и не выговоришь, поэтому рассмотрим два канонических примера ковариантных возвращаемых типов.

```
class B {
    virtual B *clone() const = 0;
    virtual Visitor *genVisitor() const;
    // ...
};
class D : public B {
    D *clone() const;
    ValidVisitor *genVisitor() const;
};
```

Функция `clone` возвращает указатель на копию объекта, попросившего о клонировании (это пример паттерна `Prototype`, см. «Совет 76»). Обычно такой запрос делается через интерфейс базового класса, и точный тип клонируемого объекта не известен:

```
B *aB = getObjectDerivedFromB();
B *anotherLikeThat = aB->clone();
```

Но иногда у нас имеется более определенная информация о типе, и мы не хотим бы ее терять или прибегать к понижающему приведению:

```
D *aD = getObjectThatIsAtLeastD();
D *anotherLikeThatD = aD->clone();
```

Если бы не ковариантный возвращаемый тип, нам пришлось бы приводить

```
D *anotherLikeThatD = static_cast<D *>(aD->clone());
```

Отметим, что в данном случае мы можем воспользоваться эффективным оператором `static_cast` вместо `dynamic_cast`, так как знаем, что операция `clone` в классе `D` возвращает объект `D`. При других обстоятельствах безопаснее и предпочтительнее было бы применить `dynamic_cast` (или обойтись без приведения вовсе).

Функция `genVisitor` (пример паттерна `Factory Method`, см. «Совет 90») иллюстрирует тот факт, что ковариантные возвращаемые классы не обязательно должны быть как-то связаны с той иерархией, в которой имел место вызов функции.

Механизм переопределения в `C++` – гибкий и полезный инструмент. Но платой за пользование им является сложность. Этот и другие советы из настоящей главы показывают, как можно справиться со сложностью, не отказываясь от применения этого механизма, когда в нем возникает нужда.

Совет 78. О реализации виртуальных функций и механизма переопределения

Многие начинающие программисты на `C++` имеют лишь поверхностное представление о том, как в языке реализован механизм переопределения. А ведь иногда это знание помогает яснее понять, что происходит в программе. Есть несколько разных эффективных способов реализации виртуальных функций и переопределения. Ниже описывается один из наиболее распространенных.

Сначала рассмотрим простую реализацию для одиночного наследования.

```
class B {
public:
    virtual int f1();
    virtual void f2( int );
    virtual int f3( int );
};
```

Каждой виртуальной функции, объявленной в классе, компилятор назначает некоторый индекс. Например, функция `B::f1` получит индекс 0, `B::f2` – индекс 1 и так далее. Эти индексы служат для доступа к таблице указателей на функции. Элемент таблицы с индексом 0 содержит адрес `B::f1`, элемент с индексом 1 – адрес `B::f2` и так далее. В каждом объекте класса хранится вставленный компилятором указатель на эту таблицу. Объект типа `B` может размещаться в памяти, как показано на рис. 7.5.

В разговорной речи таблица указателей на функции называется «`vtbl`», а указатель на нее – «`vptr`». Конструкторы класса `B` инициализируют `vptr` так, что он

указывает на `vtbl` (см. «Совет 75»). Обращение к виртуальной функции производится косвенно через таблицу `vtbl`. Так, вызов

```
B *bp = new B;
bp->f3(12);
```

транслируется в нечто такое:

```
(* (bp->vptr) [2]) (bp, 12)
```

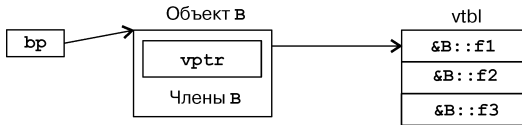


Рис. 7.5. Простая реализация виртуальных функций при одиночном наследовании

Адрес функции, которую нужно вызвать, мы получаем из той записи `vtbl`, которая соответствует индексу функции. После этого выполняется косвенный вызов, и функции передается адрес самого объекта в виде неявного аргумента `this`. Механизм виртуальных функций в C++ эффективен. Косвенные вызовы обычно оптимизированы для конкретной аппаратной архитектуры, а все объекты одного типа, как правило, пользуются одной и той же таблицей `vtbl`. При одиночном наследовании у каждого объекта есть только один `vptr`, вне зависимости от того, сколько в классе объявлено виртуальных функций.

Посмотрим на реализацию производного класса, в котором некоторые виртуальные функции из базового класса переопределены:

```
class B {
public:
    virtual int f1();
    virtual void f2( int );
    virtual int f3( int );
};
class D : public B {
    int f1();
    virtual void f4();
    int f3( int );
};
```

Объект типа `D` содержит подобъект типа `B`. Обычно, хотя и не всегда (см. «Совет 70»), подобъект базового класса находится в начале объекта производного класса (со смещением 0), а все дополнительные члены, специфичные только для производного класса, располагаются после базовой части, как на рис. 7.6.

Рассмотрим вызов той же виртуальной функции-члена, что и раньше, но на этот раз вместо объекта `B` воспользуемся объектом `D`:

```
B *bp = new D;
bp->f3(12);
```

Компилятор сгенерирует ту же самую последовательность вызова, но теперь во время выполнения она будет связана с функцией `D::f3`, а не `B::f3`:

```
(* (bp->vptr) [2]) (bp, 12)
```

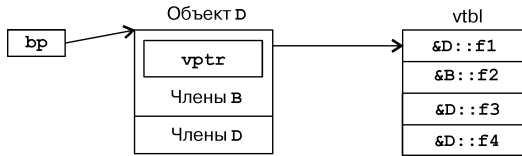


Рис. 7.6. Простая реализация виртуальных функций в объекте производного класса при одиночном наследовании. Подобъект базового класса по-прежнему содержит `vptr`, но теперь он указывает на таблицу, настроенную на производный класс

Ценность механизма виртуальных функций становится более наглядной в полиморфном коде, когда точный тип объекта, с которым мы работаем, неизвестен:

```
B *bp = getSomeSortOfB();
bp->f3(12);
```

Код виртуального вызова, генерируемый компилятором, позволяет вызывать без всякой перекомпиляции функцию `f3` из любого класса, производного от `B`, даже если такой класс еще не существует.

С точки зрения механизма, переопределение – это процедура замены адреса функции-члена базового класса адресом функции-члена производного класса во время конструирования таблицы виртуальных функций для производного класса. В примере выше в классе `D` переопределены виртуальные функции `f1` и `f3`, унаследована реализация `f2` и добавлена новая виртуальная функция `f4`. Это точно отражено в структуре виртуальной таблицы для класса `D`.

Детали механизма виртуальных функций при множественном наследовании сложнее, но принцип остается тем же самым. Дополнительная сложность – результат того, что у одного объекта может быть несколько подобъектов базовых классов, а значит, и несколько действительных адресов. Рассмотрим следующую иерархию:

```
class B1 { /* . . . */ };
class B2 { /* . . . */ };
class D : public B1, public B2 { /* . . . */ };
```

Объектом производного класса можно манипулировать через интерфейс любого из его открытых базовых классов; именно в этом смысле отношения «является». Следовательно, на объект типа `D` можно сослаться с помощью указателей на `D`, `B1` или `B2`:

```
D *dp = new D;
B1 *b1p = dp;
B2 *b2p = dp;
```

Со смещением `0` от начала производного класса может быть расположен подобъект только одного базового класса, поэтому подобъекты базовых классов обычно следуют один за другим в том порядке, в котором объявлены в списке базовых классов. В случае объекта `D` сначала идет подобъект `B1`, потом `B2`, как показано на рис. 7.7 (см. «Совет 38»).

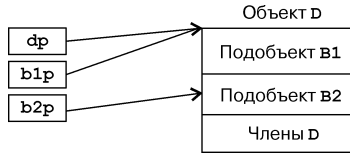


Рис. 7.7. Типичное размещение объекта в памяти при множественном наследовании

Наполним эту простую иерархию с множественным наследованием виртуальными функциями:

```
class B1 {
public:
    virtual void f1();
    virtual void f2();
};
class B2 {
public:
    virtual void f2();
    virtual void f3( int );
    virtual void f4();
};
```

В каждом из классов B1 и B2 есть виртуальные функции, поэтому в объектах этих классов есть указатели vptr на соответствующие таблицы vtbl (рис. 7.8).

Объект D является разновидностью как B1, так и B2, поэтому у нас есть два указателя vptr и две таблицы vtbl (рис. 7.9):

```
class D : public B1, public B2 {
public:
    void f2();
    void f3( int );
    virtual void f5();
};
```

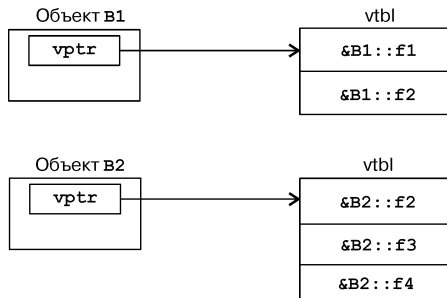


Рис. 7.8. Два потенциальных базовых класса

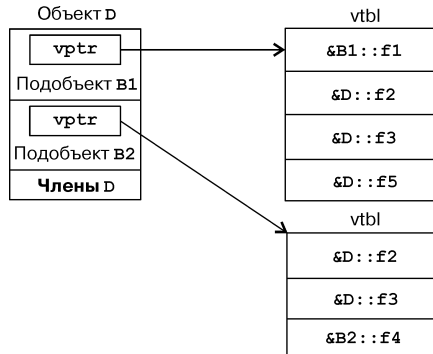


Рис. 7.9. Возможная реализация виртуальных функций при множественном наследовании. Полный объект переопределяет виртуальные функции для каждого из двух своих подобъектов базовых классов

Отметим, что `D::f2` переопределяет `f2` в обоих базовых классах. Переопределенная в производном классе функция замещает виртуальные функции с тем же именем и сигнатурой (числом и типами аргументов) во всех базовых классах – как непосредственных, так и отдаленных (базовый класс базового класса ...). Даже если в класс `D` добавлена новая виртуальная функция (`D::f5`), компилятор не вставляет `vptr` в специфичную для `D` часть объекта. Обычно новые виртуальные функции, объявленные в производном классе, добавляются в таблицу виртуальных функций какого-то из базовых классов.

Но тут возникает проблема. Рассмотрим следующий код:

```
B2 *b2p = new D;
b2p->f3(12);
```

Мы собираемся манипулировать объектом производного класса через интерфейс одного из его базовых классов, как это всегда и делается. Однако, если компилятор сгенерирует ту же последовательность вызова, что и для одиночного наследования, то указатель `this` будет иметь некорректное значение:

```
(* (b2p->vptr) [1]) (b2p, 12)
```

Причина в том, что этот вызов динамически связывается с функцией `D::f3`, которая ожидает, что неявный аргумент `this` будет указывать на начало объекта `D`. К сожалению, `b2p` указывает на начало (под)объекта `B2`, который смещен от начала объекта `D` (рис. 7.7). Необходимо «подправить» значение `this`, передаваемое при этом вызове, так чтобы `b2p` указывал на начало объекта `D`.

На наше счастье, при конструировании `vtbl` для производного класса компилятор точно знает величины поправочных слагаемых, так как ему известно, для какого класса строится `vtbl` и каковы смещения подобъектов базовых классов относительно начала объекта производного класса. Для включения поправочной информации есть несколько способов: от небольших фрагментов кода (они непра-

вильно называются «think» – переходник), выполняемого перед собственно обращением к функции, до реализации функций-членов с несколькими точками входа. Концептуально, самый чистый способ выполнить эту операцию заключается в том, чтобы просто записать необходимое смещение в таблицу `vtbl` и модифицировать последовательность вызова, так чтобы оно учитывалось (см. рис. 7.10).

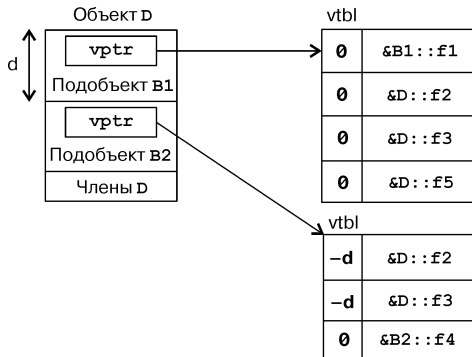


Рис. 7.10. Одна из многих возможных реализаций виртуальных функций при множественном наследовании. Здесь поправочные значения для указателя `this` хранятся в самой таблице виртуальных функций

Теперь запись в таблице `vtbl` представляет собой небольшую структуру, содержащую адрес функции-члена (`fptr`) и смещение (`delta`), которое нужно добавить к значению `this`. Последовательность вызова приобретает такой вид:

```
(* (b2p->vptr) [1].fptr) (b2p+(b2p->vptr) [1].delta, 12)
```

Этот код отлично оптимизируется, так что обходится он не так дорого, как кажется на первый взгляд.

Совет 79. Вопросы доминирования

Возможно, вы недоумеаете, как вас угораздило заняться программированием на языке, в котором есть такие понятия, как «друзья», «закрытые части», «связанные друзья» и «доминирование». В этом разделе мы рассмотрим понятие доминирования в проектировании иерархий наследования: почему оно столь таинственно и когда в нем возникает необходимость. Ну, конечно, легко сказать, что при вашем образе жизни такая проблема вообще не возникает, но рано или поздно большинство опытных программистов на C++ сталкиваются с доминированием (со своей стороны или со стороны коллег). Так что лучше заранее подготовиться. Предупрежден – значит вооружен.

Доминирование возникает только в контексте виртуального наследования. Проиллюстрировать его лучше графически. На рис. 7.11 идентификатор `B::name` доминирует над `A::name`, если `A` – базовый класс `B`. Отметим, что доминирование

обобщается и на другие пути поиска. Например, если компилятор ищет идентификатор `name` в области видимости класса `D`, то найдет и `B : name`, и – по другому пути – `A : name`. Однако из-за доминирования неоднозначности не возникает. Доминирует идентификатор `B : name`.

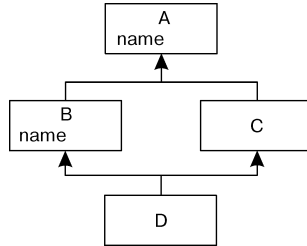


Рис. 7.11. Идентификатор `B : name` доминирует над `A : name`

В аналогичной ситуации, но без виртуального наследования неоднозначность имеет место. На рис. 7.12 поиск `name` в области видимости класса `D` приводит к неоднозначности, так как `B : name` не доминирует над `A : name` в базовом классе `C`.

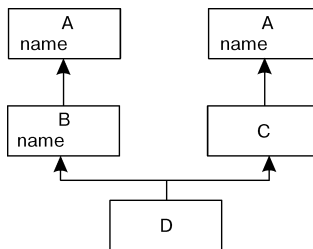


Рис. 7.12. Здесь доминирования нет. Идентификатор `B : name` скрывает `A : name` на одном пути, но не на другом

Это правило языка может показаться странным, но без доминирования во многих случаях было бы невозможно построить таблицы виртуальных функций для классов с виртуальным наследованием. Короче говоря, сочетание динамического связывания и виртуального наследования неумолимо влечет за собой концепцию доминирования.

Рассмотрим простую иерархию с виртуальным наследованием, представленную на рис. 7.13. Объект `D` состоит из трех подобъектов базовых классов, причем доступ к разделяемому подобъекту `V` осуществляется по указателям, как показано на рис. 7.14. (Возможны разные реализации. Эта несколько устарела, но ее легко нарисовать, а логически она эквивалентна прочим подходам.)

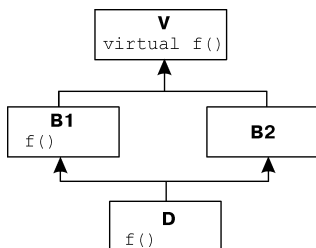


Рис. 7.13. Функция `D::f` переопределяет и `B1::f`, и `V::f`. Виртуальные таблицы для подобъектов `B1` и `V` содержат информацию, необходимую для вызова `D::f`

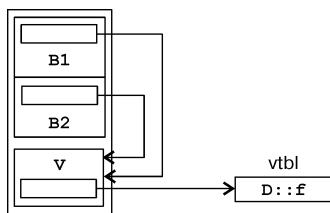


Рис. 7.14. Возможно размещение полного объекта `D`. Показана таблица виртуальных функций для подобъекта `V`

Можно ожидать, что объявление функции-члена `D::f` (рис. 7.13) переопределяет и `B1::f`, и `V::f`:

```

B2 *b2p = new D;
b2p->f(); // вызывается D::f
  
```

Рассмотрим другой случай, показанный на рис. 7.15. Он просто некорректен, так как для переопределения `V::f` в `D` можно использовать и `B1::f`, и `B2::f`. Возникает неоднозначность и, как следствие, ошибка компиляции.

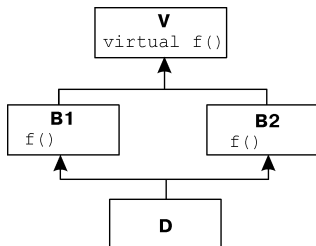


Рис. 7.15. Неоднозначность. И `B1::f`, и `B2::f` могут переопределить `V::f` в таблице виртуальных функций подобъекта `V`

И, наконец, рассмотрим случай, когда в игру вступает доминирование:

```
B2 *b2p = new D;  
b2p->f(); // вызывается B1::f()!
```

На рис. 7.16 идентификатор $B : : f$ доминирует над идентификатором $V : : f$ на всех путях, и в таблице виртуальных функций для подобъекта V объекта D будет установлен указатель на $B1 : : f$. Если бы не правило доминирования, то и этот случай был бы неоднозначен, так как для реализации $V : : f$ в объекте D годились бы как $V : : f$, так и $B1 : : f$. Доминирование разрешает неоднозначность в пользу $B1 : : f$.

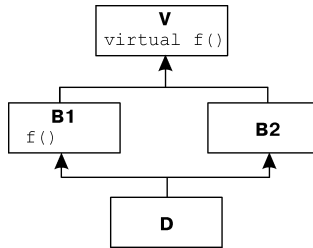


Рис. 7.16. Доминирование устраняет неоднозначность при построении таблицы виртуальных функций. $B1 : : f$ доминирует над $V : : f$, поэтому в виртуальной таблице подобъекта V содержится информация, необходимая для вызова $B1 : : f$



Глава 8. Проектирование классов

Проектирование эффективных абстрактных типов данных – это одновременно наука и искусство. Для создания хороших интерфейсов нужно обладать техническими знаниями, разбираться в социальной психологии и иметь опыт. Но только ясный, интуитивно понятный интерфейс может гарантировать, что программу будет легко понять и сопровождать.

В этой главе мы рассмотрим ряд типичных ошибок при проектировании интерфейсов классов и покажем, как их избежать. Мы также остановимся на некоторых вопросах реализации, влияющих на интерфейс класса.

Совет 80. Интерфейсы get/set

В абстрактном типе данных все данные-члены должны быть закрыты. Однако класс, состоящий только из закрытых данных-членов и открытых функцией get/set, трудно назвать абстрактным типом данных.

Напомним, что цель абстрагирования данных – отвлечься от обсуждения конкретной реализации и дать авторам и читателям кода возможность говорить на языке предметной области. Для этого абстрактный тип данных определяется просто как набор операций, соответствующих абстрактному представлению о том, для чего этот тип нужен. Рассмотрим стек:

```
template <class T>
class UnusableStack { // непригодный для использования стек
public:
    UnusableStack();
    ~UnusableStack();
    T *getStack();
    void setStack( T * );
    int getTop();
    void setTop( int );
private:
    T *s_;
    int top_;
};
```

Хорошего в этом шаблоне разве что имя. Здесь нет никакой абстракции, просто слегка замаскированный набор данных. Открытый интерфейс не обеспечивает абстракции стека для пользователей и даже не защищает от изменений во внутренней реализации. Хорошая реализация стека должна дать ясную абстракцию и независимость от внутренних деталей:

```
template <class T>
class Stack {
public:
```

```

Stack();
~Stack();
void push( const T & );
T &top();
void pop();
bool empty() const;
private:
T *s_;
int top_;
};

```

Вообще-то никакой проектировщик никогда не создал бы такой избыливающей ошибки интерфейса, как `UnusableStack`. Каждый компетентный программист знает, какие операции требуются от стека, и пишет эффективный интерфейс почти автоматически. Но так обстоит дело далеко не для всех абстрактных типов данных, особенно если вы занимаетесь проектированием в предметной области, где не являетесь специалистом. В таком случае важно тесно сотрудничать с экспертами, которые помогут определить, какие абстрактные типы данных требуются, и какие в них должны быть операции. Характерным признаком проекта, который создавался без адекватной подготовки в предметной области, является большая доля классов с интерфейсами `get/set`.

Тем не менее, часто в интерфейс класса нужно включать какое-то количество функций-аксессоров `get/set`. Как лучше их представить? Есть несколько распространенных вариантов:

```

class C {
public:
    int getValue1() const           // get/set, вариант 1
    { return value_; }
    void setValue1( int value )
    { value_ = value; }
    int &value2()                 // get/set, вариант 2
    { return value_; }
    int setValue3( int value ) // get/set, вариант 3
    { return value_ = value; }
    int value4( int value ) { // get/set, вариант 4
    int old = value_;
    value_ = value;
    return old;
    }
private:
    int value_;
};

```

Второй вариант самый краткий, самый гибкий, но и самый опасный. Возвращая дескриптор закрытого члена реализации класса, функция `value2` оказывается немногим лучше открытых данных. Пользователи класса могут написать код, зависящий от текущей реализации, и напрямую обращаться к внутренним данным класса. Эта форма плоха даже, если предоставляется доступ только для чтения. Рассмотрим класс, реализованный с помощью стандартного библиотечного контейнера:

```

class Users {
public:

```

```
const std::map<std::string, User> &getUserContainer() const
{ return users_; }
// ...
private:
std::map<std::string, User> users_;
};
```

Функция `get` выставляет на всеобщее обозрение закрытую информацию о том, что пользовательский контейнер реализован посредством стандартного контейнера `map`. Любой код, который вызывает эту функцию, может (и, скорее всего, так и будет) оказаться зависимым от конкретной реализации `Users`. В случае (вполне вероятном), если профилирование покажет, что контейнер `vector` эффективнее, придется переписывать весь код, в котором используется класс `Users`. Таких функций-аксессоров просто не должно быть.

Третий вариант несколько необычен, поскольку он, строго говоря, не дает доступа к текущему значению члена-данных, а устанавливает и возвращает новое значение. (Предполагается, что вы где-то запомните старое значение. В конце концов, ведь это вы же его и установили, правда?) Пользователи класса получают возможность писать выражения типа `a += setValue3(12)` вместо двух коротких предложений `setValue1(12); a +=getValue1();`. Проблема в том, что многие пользователи такого интерфейса будут считать, что возвращается предыдущее значение, а это может повлечь за собой трудно обнаруживаемые ошибки.

Четвертый вариант привлекателен тем, что позволяет с помощью одной функции получить текущее значение и установить новое. Однако; если нужно только получить текущее значение, приходится прибегать к ухищрениям:

```
int current = c.value4( 0 ); // получить и установить
c.value4( current ); // восстановить
```

Чтобы получить текущее значение, мы должны записать в `value4` какое-то фиктивное значение, а затем заменить его предыдущим. Это выглядит не вполне естественно, но такая техника характерна для C++ и применяется в функциях `set_new_handler`, `set_unexpected` и `set_terminate` из стандартной библиотеки, которые служат для задания функций обратного вызова для управления памятью и обработки исключений. Как правило, такой механизм используется для работы с функциями обратного вызова по принципу стека, но без самого стека:

```
typedef void (*new_handler)(); // тип функции обратного вызова
// ...
new_handler old_handler = set_new_handler( handler ); // затолкнуть
// что-то сделать ...
set_new_handler( old_handler ); // вытолкнуть
```

Получить с помощью этого подхода текущий обработчик не так-то просто. В C++ применяется такая идиома:

```
new_handler handler = set_new_handler( 0 ); // получить текущий
set_new_handler( handler ); // восстановить
```

Но не стоит применять это решение в качестве общего механизма `get/set`. Оно увеличивает стоимость и сложность простого доступа для чтения члена данных, затрудняет обеспечение безопасности относительно исключений и написание многопоточного кода. Кроме того, его легко спутать с описанным выше вариантом 3.

Лучше всего для написания аксессоров подходит вариант 1. Он самый простой, эффективный и, что самое важное, не допускает неоднозначного толкования:

```
int a = c.getValue1(); // разумеется, получить
c.setValue( 12 ); // установить, что же еще
```

Если проект вашего класса предусматривает использование функций get/set, остановитесь на варианте 1.

Совет 81. Константные и ссылочные данные-члены

Одна из самых лучших рекомендаций общего характера: «если что-то может быть константным, пусть оно будет константным». И наоборот, «если что-то не всегда является константным, не объявляйте это константным». В сочетании эти два совета означают, что нужно принимать во внимание как текущее, так и ожидаемое использование конструкции и делать ее «настолько константной, насколько возможно, но не более того».

В этом разделе я попытаюсь убедить вас в том, что редко имеет смысл объявлять данные-члены класса константными или ссылками. Наличие константных и ссылочных данных-членов затрудняет работу с классом, требует неестественной семантики копирования и оставляет сопровождающим возможность внести опасные ошибки.

Рассмотрим пример простого класса с константными и ссылочными данными-членами:

```
class C {
public:
    C();
    // ...
private:
    int a_;
    const int b_;
    int &ra_;
};
```

Конструктор должен инициализировать константные и ссылочные данные-члены:

```
C::C()
: a_( 12 ), b_( 12 ), ra_( a_ )
{}
}
```

Пока все хорошо. Мы можем объявлять объекты класса C и инициализировать их:

```
C x; // конструктор по умолчанию
C y( x ); // конструктор копирования
```

Оп! А откуда взялся этот конструктор копирования? Его написал за нас компилятор, и по умолчанию он выполняет почленную инициализацию членов y соответствующими членами x (см. «Совет 49»). К несчастью, эта реализация по умолчанию установит ссылку ra_ в y равной члену a_ в x. Раз уж мы заговорили

об полезных рекомендациях, то вот вам еще одна: «Подумайте, не стоит ли самостоятельно написать операции копирования для любого класса, содержащего описатель (обычно указатель или ссылку) каких-то других данных»:

```
C::C( const C &that )
    : a_( that.a_ ), b_( that.b_ ), ra_( a_ )
    {}
```

Продолжим пользоваться нашими объектами класса €:

```
x = y; // ошибка!
```

Здесь проблема в том, что компилятор не может сгенерировать оператор присваивания. По умолчанию он пытается написать оператор, который просто присваивает значение члена данных `y` соответствующему члену данных `x`. Но для объектов класса `C` это невозможно, так как членам `b_` и `ra_` нельзя ничего присваивать. И это хорошо, поскольку такой оператор присваивания вел бы себя так же некорректно, как и конструктор копирования по умолчанию.

А написать правильный оператор присваивания не так просто. Вот первая попытка:

```
C &C::operator =( const C &that ) {
    a_ = that.a_; // правильно
    b_ = that.b_; // ошибка!
    return *this;
}
```

Присваивать константе запрещено. Опасность в том, что «творчески настроенный» сопровождающий может попытаться выполнить присваивание во что бы то ни стало. Для начала обычно пробуют привести:

```
int *pb = const_cast<int *>(&b_);
*pb = that.b_;
```

Такой код вряд ли вызовет проблемы во время исполнения, так как маловероятно, что член `b_` находится в сегменте памяти, предназначенном только для чтения, коль скоро он является частью неконстантного объекта `C`. Однако такую реализацию трудно назвать естественной, а для члена-ссылки этот трюк просто не работает. (Отметим, что в данном конкретном операторе присваивания не обязательно было привязывать другое значение к ссылочному члену данных `C`, так как он уже ссылается на член `a_` своего собственного объекта.)

Некоторые особо ревностные сопровождающие могут подойти к проблеме с другой стороны. Вместо того чтобы присвоить объект `y` объекту `x`, они просто уничтожают `x` и заново инициализируют его с помощью `y`:

```
C &C::operator =( const C &that ) {
    if( this != &that ) {
        this->~C(); // вызывать деструктор
        new (this) C(that); // конструктор копирования
    }
    return *this;
}
```

Немало перьев было затуплено за прошедшие годы в спорах об этом подходе, но, в конце концов, он был отвергнут. Хотя в данном частном случае он, возможно, и будет работать (некоторое время), но такой подход слишком сложен, не масшта-

бируется и, скорее всего, приведет к проблемам в будущем. Посмотрим, что произойдет, если `C` когда-нибудь будет использоваться в качестве базового класса. Вероятно, оператор присваивания в производном классе будет вызывать оператор присваивания из класса `C`. Деструктор, если он виртуальный, уничтожит весь объект, а не только часть `C`. Если же деструктор не виртуальный, то мы получим неопределенное поведение. Так что лучше от такого решения держаться подальше.

Самый простой и прямолинейный способ – не употреблять константные и ссылочные члены вовсе. Поскольку все наши данные-члены закрыты (ведь так, не правда ли?), то мы уже и так защитили их от случайной модификации. Если же цель использования константных или ссылочных членом том, чтобы не дать компилятору сгенерировать оператор присваивания, то есть и более идиоматичный способ решить эту задачу (см. «Совет 49»).

```
class C {
    // . . .
private:
    int a_;
    int b_;
    int *pa_;
    C( const C & ); // запретить конструирование копированием
    C &operator =( const C & ); // запретить присваивание
};
```

Константные и ссылочные данные-члены редко бывают необходимы. Избегайте их.

Совет 82. В чем смысл константных функций-членов?

Синтаксис

Первое, на что обращаешь внимание при работе с константными функциями-членами, – непривычный синтаксис. Необходимость размещать слово `const` в конце объявления выглядит странно. На самом деле ничего странного нет. Как и другие части синтаксиса объявлений, унаследованные от `C`, синтаксис объявления константной функции по-своему логичен, хотя и может привести в замешательство:

```
class BoundedString {
public:
    explicit BoundedString( int len );
    // ...
    size_t length() const;
    void set( char c );
    void wipe() const;
private:
    char * const buf_;
    int len_;
    size_t maxLen_;
};
```

Сначала взгляните на объявление закрытого члена данных `buf_`. Это константный указатель на символ (пример приведен только для иллюстрации, см. «Совет 81»). Константным является указатель, а не символы, на которые он указывает, поэтому квалификатор типа `const` поставлен после звездочки. Если мы бы поместили его перед звездочкой, то он относился бы к встроенному типу `char`, то есть речь шла бы о неконстантном указателе на константные символы.

То же относится и к константной функции-члену `length`. Если бы мы поместили слово `const` перед именем функции, то объявили бы функцию, которая не принимает аргументов и возвращает константное значение типа `size_t`. Размещение же `const` после имени говорит о том, что константна сама функция, а не возвращаемое ей значение.

Простая семантика и механизм работы

В чем смысл константности функции-члена? Обычно дают такой ответ: «константная функция-член не изменяет своего объекта». Это простое утверждение, и компилятору несложно его реализовать.

Каждой не-статической функции-члену передается скрытый аргумент – указатель на объект, с помощью которого она была вызвана. Внутри функции к этому указателю можно обратиться с помощью ключевого слова `this`:

```
BoundedString bs( 12 );
cout << bs.length(); // "this" – это &bs
BoundedString *bsp = &bs;
cout << bsp->length(); // "this" – это bsp
```

Для неконстантной функции-члена класса `X` указатель `this` имеет тип `X *` `const`; иными словами, это константный указатель на неконстантный объект `X`. Сам указатель модифицировать нельзя (следовательно, `this` всегда будет ссылаться на один и тот же объект `X`), но члены `X` модифицировать можно. Внутри неконстантной функции-члена любой доступ к не-статическому члену класса осуществляется через указатель на не-`const`:

```
void BoundedString::set( char c ) {
    for( int i = 0; i < maxlen_; ++i )
        buf_[i] = c;
    buf_[maxlen_] = '\0';
}
```

Для константной функции-члена класса `X` указатель `this` имеет тип `const X *` `const`; это константный указатель на константный объект `X`. Теперь нельзя модифицировать ни указатель, ни объект, на который он указывает:

```
size_t BoundedString::length() const
{ return strlen( buf_ ); }
```

По существу, константная функция-член позволяет задать константность своего скрытого аргумента `this`. Рассмотрим, например, объявление не являющегося членом класса оператора сравнения на равенство объектов `BoundedString`:

```
bool operator ==( const BoundedString &lhs,
                  const BoundedString &rhs );
```

Эта функция не изменяет своих аргументов, а только анализирует их, следовательно, и правый, и левый аргумент можно объявить как ссылки на `const`. То же должно быть верно и для случая, когда такая функция делается членом класса:

```
class BoundedString {
    // . . .
    bool operator <( const BoundedString &rhs );
    bool operator >=( const BoundedString &rhs ) const;
};
```

Напомним, что левый операнд перегруженной функции, представляющей бинарный оператор, передается неявно в виде аргумента `this`. Правый же аргумент инициализируется явно объявленным формальным аргументом (в обеих операторных функциях выше он назван `rhs`). Оператор `>=` объявлен правильно, и функция обещает не изменять ни левый, ни правый аргумент. Оператор же `<` некорректен, поскольку гарантирует лишь неизменность правого, но не левого аргумента. Эта небрежность даст о себе знать, если мы попытаемся реализовать `>=` самым прямолинейным способом:

```
bool BoundedString::operator >=( const BoundedString &rhs ) const
{ return !(*this < rhs); }
```

Мы получаем ошибку при компиляции `operator <`. Передавая выражение `* this` в качестве первого аргумента `operator <`, мы пытаемся инициализировать указатель `this` для неконстантной функции-члена адресом константного объекта.

Семантика константной функции-члена

Мы только что описали механизм работы константных функций-членов, но семантика таких функций определяется в значительной степени сообществом компетентных программистов на C++. Рассмотрим реализацию члена `wipe` класса `BoundedString`:

```
void BoundedString::wipe() const
{ buf_[0] = '\0'; }
```

Это допустимый код, но из законности действия еще не следует, что оно ожидаемо или морально оправдано. Функция `wipe` не изменяет объект, то есть не модифицирует никакие данные-члены объекта `BoundedString`. Тем не менее, она изменяет некоторые данные, формально расположенные вне объекта, но влияющие на его поведение. После вызова `wipe` логическое состояние `BoundedString` станет другим. Константность указателя `this` что-то гарантирует лишь для данных-членов объекта `BoundedString`. Данные же вне объекта оказываются беззащитными, хотя логически они составляют его часть.

Большинство пользователей класса `BoundedString` неприятно удивятся, увидев, что поведение объекта изменилось после обращения к константной функции. Поскольку `wipe` изменяет логическое состояние, ее не следовало бы объявлять константной. Вот почему в объявлении выше функция `set` не была сделана константной, хотя компилятор ничего против константности не возразил бы.

А теперь взгляните на реализацию функции-члена `length`. Очевидно, она должна быть константной, поскольку вычисление длины строки `BoundedString` не изменяет ее логического состояния. Проще всего было бы воспользоваться библиотечной функцией `strlen`, как мы выше и поступили. Вероятно, это наилучшая реализация, так как она проста, достаточно эффективна и дает правильный результат. Предположим, однако, что, по нашим наблюдениям, многие строки никогда не меняют длину, тогда как длина других – и при том весьма длинных – вычисляется часто. В таком случае предпочтительной может оказаться другая реализация:

```
size_t BoundedString::length() const {
    if( len_ < 0 )
        len_ = strlen( buf_ );
    return len_;
}
```

Мы решили хранить текущую длину строки в самом объекте `BoundedString` и вычислять ее только при поступлении запроса. Накладные расходы в случае, когда длина строки никогда не запрашивается, невелики, зато выигрыш при частых обращениях к `length` весьма ощутим. К сожалению, при попытке присвоить значение члену `len_` компилятор выдаст сообщение об ошибке. Это константная функция, ей запрещено изменять состояние объекта.

С этой проблемой можно было бы справиться, сделав функцию `length` неконстантной, но это вступает в противоречие с ее логическим смыслом. К тому же мы уже не сумеем объявить максимальную длину `BoundedString` константной (не важно, является она такой в действительности или нет; см. «Совет 6» и «Совет 31»). Следовательно, мы делаем `length` неконстантной в угоду удобству реализации, тогда как вопросы реализации не должны влиять на интерфейс абстрактного типа данных.

Распространенный и достойный порицания образ действий в такой ситуации – «отбросить `const`» в константной функции-члене:

```
size_t BoundedString::length() const {
    if( len_ < 0 )
        const_cast<int &>(len_) = strlen( buf_ );
    return len_;
}
// ...
BoundedString a(12);
int alen = a.length(); // будет работать ...
const BoundedString b(12);
int blen = b.length(); // неопределенность!
```

Любая попытка модифицировать константный объект вне конструктора или деструктора приводит к неопределенному поведению. Поэтому вызов функции-члена `length` для `b` может сработать, а может таинственным образом закончиться ошибкой спустя длительное время после завершения тестирования программы и поставки ее заказчику. И то, что мы воспользовались новомодным оператором `const_cast`, не спасает.

Правильный выход – объявить член данных `len_` как `mutable`. Спецификатор класса хранения `mutable` можно применять к не-статическим, неконстант-

ным, нессылочным данным-членам; он показывает, что член может быть модифицирован константным (равно как и неконстантными) функциями-членами.

```
class BoundedString {
    // ...
private:
    char * const buf_;
    mutable int len_;
    size_t maxLen_;
};
```

Для программистов на C++ константная функция-член реализует «логическую» константность. То есть наблюдаемое состояние объекта не должно измениться в результате вызова такой функции, хотя физическое состояние может быть модифицировано.

Совет 83. Различайте агрегирование и использование

Различить отношения владения (агрегирования) и использования («знакомства») в самом языке C++ невозможно. Это может приводить к самым разным ошибкам, в том числе утечкам памяти и появлению псевдонимов:

```
class Employee {
public:
    virtual ~Employee();
    void setRole( Role *newRole );
    const Role *getRole() const;
    // ...
private:
    Role *role_;
    // ...
};
```

Из этого интерфейса не ясно, то ли объект `Employee` владеет своей ролью `Role`, то ли просто ссылается на объект `Role`, который может использоваться и другими объектами `Employee`. Проблемы возникают, когда пользователь класса `Employee` делает предположение о владении, отличающееся от того, что имел в виду проектировщик класса:

```
Employee *e1 = getMeAnEmployee();
Employee *e2 = getMeAnEmployee();
Role *r = getMeSomethingToDo();
e1->setRole( r );
e2->setRole( r ); // ошибка #1!
delete r; // ошибка #2!
```

Если проектировщик решил, что `Employee` – владелец `Role`, то строка с меткой ошибка #1 приведет к тому, что два объекта `Employee` будут ссылаться на один и тот же объект `Role`. Тогда при удалении `e2` и `e1` произойдет двойное удаление объекта `Role`.

Строка с меткой ошибка #2 тоньше. Здесь пользователь класса `Employee` решил, что функция `setRole` делает копию своего аргумента `Role`, и ранее выде-

ленный из кучи объект `Role` надо освободить. Если истинная семантика не такова, то и `e1`, и `e2` будут содержать «висячие» указатели.

Опытный разработчик мог бы поискать ответ в тексте функции `setRole` и наткнуться на одну из следующих реализаций:

```
void Employee::setRole( Role *newRole ) // версия 1
{ role_ = newRole; }

void Employee::setRole( Role *newRole ) { // версия 2
delete role_;
role_ = newRole;
}

void Employee::setRole( Role *newRole ) { // версия 3
delete role_;
role_ = newRole->clone();
}
```

Версия 1 говорит о том, что объект `Employee` не владеет объектом `Role`, поскольку не предпринимает никаких попыток удалить существующий экземпляр перед тем, как установить указатель на новый. (Мы предполагаем, что это осознанное намерение проектировщика, а не просто ошибка.)

Из текста версии 2 следует, что объект `Employee` владеет своим объектом `Role` и становится владельцем того объекта, на который указывает аргумент `setRole`. Версия 3 также свидетельствует о том, что `Employee` – владелец `Role`. Однако в данном случае он не просто берет на себя контроль, но и делает копию переданного аргумента. Отметим, что в третьей версии было бы лучше объявить аргумент как `const Role *`, а не `Role *`. Клонирование – всегда константная операция, поскольку она не модифицирует объект, а лишь создает его копию. Кроме того, для версии 1, в которой `Role` – разделяемый объект, странно было бы передавать его как указатель на `non-const`.

Однако у пользователей абстрактного типа данных обычно нет доступа к его реализации, поскольку это противоречило бы идее сокрытия данных и могло бы привести к зависимости от особенностей конкретной реализации. Например, тот факт, что версия 1 функции `setRole` не удаляет существующий объект `Role`, необязательно означает, что проектировщик класса `Employee` действительно хотел разделять `Role`; это могла быть просто ошибка. Однако после того как во многих пользовательских программах принято допущение о разделении объектов `Role`, ошибка перестает быть таковой и становится особенностью реализации.

Поскольку язык `C++` не позволяет явно описать отношение владения, то приходится прибегать к соглашениям об именах, типах формальных аргументов и комментариям (да, в этом случае они оправданы):

```
class Employee {
public:
virtual ~Employee();
void adoptRole( Role *newRole ); // вступить во владение
void shareRole( const Role *sharedRole ); // не владеет
void copyRole( const Role *roleToCopy ); // Role копируется
const Role *getRole() const;
// ...
};
```

Имена `adoptRole`, `shareRole` и `copyRole` достаточно необычны, чтобы побудить пользователей класса `Employee` прочитать комментарии. Если комментарий короткий и ясный, то, возможно, он даже будет сопровождаться (см. «Совет 1»).

Распространенный пример неправильного понимания владения – это контейнеры, содержащие указатели. Рассмотрим список указателей:

```
>> gotcha83/ptrlist.h
```

```
template <class T> class PtrList;
template <> class PtrList<void> {
    // ...
};
template <class T>
class PtrList : private PtrList<void> {
public:
    PtrList();
    ~PtrList();
    void append( T *newElem );
    // ...
};
```

И здесь проблема в том, что проектировщик и пользователь контейнера могут интерпретировать его по-разному:

```
PtrList<Employee> staff;
staff.append( new Techie );
```

В этом фрагменте пользователь списка `PtrList`, вероятно, предполагает, что контейнер вступает во владение объектом, на который указывает аргумент функции `append`. Это означает, что деструктор `PtrList` удалит все объекты, на которые указывают элементы списка. Если контейнер ничего подобного не делает, произойдет утечка памяти. Автор же кода, показанного ниже, подумал иначе:

```
PtrList<Employee> management;
Manager theBoss;
management.append( &theBoss );
```

Здесь предполагается, что контейнер не будет удалять объекты, на которые указывают хранящиеся в нем элементы. Если это допущение неверно, то `PtrList` попытается освободить память, не выделенную из кучи.

Самый лучший способ избежать неправильной интерпретации идеи владения при работе с контейнерами: пользоваться стандартными контейнерами. Поскольку они описаны в стандарте C++, то все опытные программисты знают об особенностях их поведения. Если элементами являются указатели, то стандартный контейнер не станет удалять объекты, на которые они указывают:

```
std::list<Employee *> management;
Manager theBoss;
management.push_back( &theBoss ); // правильно
```

Если же мы хотим, чтобы контейнер все же удалил объекты, на которые ссылаются его элементы, то есть два пути. Самый прямолинейный – почистить за собой самостоятельно:

```
template <class Container>
```

```
void releaseElems( Container &c ) {
    typedef typename Container::iterator I;
    for( I i = c.begin(); i != c.end(); ++i )
        delete *i;
}
// ...
std::list<Employee *> staff;
staff.push_back( new Techie );
// ...
releaseElems( staff ); // очистка
```

К сожалению, про написанный вручную код очистки часто забывают, а во время сопровождения могут переместить в неподходящее место или вовсе удалить. Кроме того, он неустойчив по отношению к исключениям. Лучше вместо обычного указателя воспользоваться для этой цели интеллектуальным. (Подчеркнем, что стандартный шаблон `auto_ptr` нельзя использовать в качестве элемента контейнера, поскольку его семантика копирования для этого не подходит. См. «Совет 68».) Вот простой пример:

➤➤ `gotcha83/cptr.h`

```
template <class T>
class Cptr {
public:
    Cptr( T *p ) : p_( p ), c_( new long( 1 ) ) {}
    ~Cptr() { if( !-*c_ ) { delete c_; delete p_; } }
    Cptr( const Cptr &init )
        : p_( init.p_ ), c_( init.c_ ) { ++*c_; }
    Cptr &operator =( const Cptr &rhs ) {
        if( this != &rhs ) {
            if( !-*c_ ) { delete c_; delete p_; }
            p_ = rhs.p_;
            ++*(c_ = rhs.c_);
        }
        return *this;
    }
    T &operator *() const
        { return *p_; }
    T *operator ->() const
        { return p_; }
private:
    T *p_;
    long *c_;
};
```

При конкретизации контейнера указывается тип не обычного, а интеллектуального указателя (см. «Совет 24»). Когда контейнер удаляет свои элементы, деструктор интеллектуального указателя «подчищает» объект, на который тот указывает:

```
std::vector< Cptr<Employee> > staff;
staff.push_back( new Techie );
staff.push_back( new Temp );
staff.push_back( new Consultant );
// явная очистка не нужна ...
```

Такое употребление интеллектуальных указателей обобщается и на более сложные случаи:

```
std::list< Cptr<Employee> > expendable;
expendable.push_back( staff[2] );
expendable.push_back( new Temp );
expendable.push_back( staff[1] );
```

Когда контейнер `expendable` покинет область действия, он корректно удалит свой второй элемент `Temp` и уменьшит счетчики ссылок в первом и третьем элементах, которые разделяет с контейнером `staff`. Когда из области действия выходит `staff`, он удаляет все три своих элемента.

Совет 84. Не злоупотребляйте перегрузкой операторов

Можно обойтись и вовсе без перегрузки операторов:

```
class Complex {
public:
    Complex( double real = 0.0, double imag = 0.0 );
    friend Complex add( const Complex &, const Complex & );
    friend Complex div( const Complex &, const Complex & );
    friend Complex mul( const Complex &, const Complex & );
    // ...
};
// ...
Z = add( add( R, mul( mul( j, omega ), L ) ),
        div( 1, mul( j, omega ), C ) );
```

Перегрузка операторов – это не более чем синтаксическое удобство, но она облегчает написание и чтение кода и проясняет намерения проектировщика:

```
class Complex {
public:
    Complex( double real = 0.0, double imag = 0.0 );
    friend Complex operator +( const Complex &, const Complex & );
    friend Complex operator *( const Complex &, const Complex & );
    friend Complex operator /( const Complex &, const Complex & );
    // ...
};
// ...
Z = R + j*omega*L + 1/(j*omega*C);
```

Вариант формулы для вычисления сопротивления переменному току, написанный с использованием инфиксных операторов, правилен, в отличие от предыдущего варианта, где применялся синтаксис вызова функции. Однако найти и исправить ошибку без перегрузки операторов будет сложнее.

Перегрузка операторов оправдана также для расширения существующего синтаксиса, как, например, в библиотеках `iostream` и `STL`:

```
ostream &operator <<( ostream &os, const Complex &c )
{ return os << '(' << c.r_ << ", " << c.i_ << ')'; }
```

Видя такой успешный пример применения, начинающие разработчики часто начинают пользоваться перегрузкой операторов к месту и не к месту:

```

template <typename T>
class Stack {
public:
    Stack();
    ~Stack();
    void operator +( const T & ); // затолкнуть
    T &operator *( ); // вершина
    void operator -(); // вытолкнуть
    operator bool() const; // не пуст?
    // ...
};
// ...
Stack<int> s;
s + 12;
s + 13;
if( s ) {
    int a = *s;
    -s;
    // ...
}

```

Умно? Нет, легкомысленная чушь. Перегрузка операторов служит для того, чтобы сделать код более понятным читателю, а не чтобы проектировщик мог порисоваться. Наличие перегруженного оператора должно вызывать к укоренившимся привычкам читателя; любое разумное предположение, которое опытный читатель может сделать относительно смысла оператора, должно быть верным. В хорошей реализации стека следует употреблять общепринятые, не-операторные имена операций со стеком:

```

template <typename T>
class Stack {
public:
    Stack();
    ~Stack();
    void push( const T & );
    T &top();
    void pop();
    bool isEmpty() const;
    // ...
};
// ...
Stack<int> s;
s.push( 12 );
s.push( 13 );
if( !s.isEmpty() ) {
    int a = s.top();
    s.pop();
    // ...
}

```

Отметим, что перегрузка оператора допустима только, если ее смысл не оставляет места для двояких толкований. Пусть семантика перегруженного оператора понятна вам и еще 75% ваших коллег, но в 25% случаев возможны неверная интерпретация и применение. При таком раскладе от перегрузки следует отказаться, поскольку она создаст больше проблем, чем решит.

На ум приходит пример из моего личного опыта. Я проектировал простой шаблон для массивов:

>> gotcha05/array.h

```

template <class T, int n>
class Array {
public:
    Array();
    explicit Array( const T &val );
    Array &operator =( const T &val ); // всем очевидно?
    // . . .
private:
    T a_[n];
};
// . . .
Array<float,100> ary( 0 );
ary = 123; // очевидно?

```

Я был абсолютно убежден, что смысл этого присваивания очевиден. Ясно же, что я хочу присвоить значение 123 каждому элементу массива. Правильно? Оказывается, немалая доля пользователей класса `Array` так не считала. Некоторые опытные программисты решили, что речь идет о задании нового размера массива – для 123 элементов. Другие полагали, что я хочу присвоить значение 123 только первому элементу. Я-то знал, что прав я, а все, кто думает иначе, ошибаются, но из практических соображений был вынужден отказаться от своего решения и ввести недвусмысленную не-операторную функцию:

```
ary.setAll( 123 ); // скучно, зато ясно
```

Если перегрузка оператора не дает ощутимых преимуществ по сравнению с не-операторной функцией, не перегружайте.

Совет 85. Приоритеты и перегрузка

Приоритет оператора – это часть того, что пользователь ожидает от его поведения. Если ожидания не оправдываются, оператор будет использоваться неправильно. Рассмотрим нестандартную реализацию комплексных чисел:

```

class Complex {
public:
    Complex( double = 0, double = 0 );
    friend Complex operator +( const Complex &, const Complex & );
    friend Complex operator *( const Complex &, const Complex & );
    friend Complex operator ^( const Complex &, const Complex & );
    // ...
};

```

Мы хотели бы определить для комплексных чисел оператор возведения в степень, но в C++ такого оператора нет. Поскольку вводить новые операторы мы не можем, то решаем задействовать один из существующих, который для комплексных чисел никакой predefined семантики не имеет, а именно: «исключающее или».

Но тут же сталкиваемся с проблемой, потому что опытный программист на C или C++ будет считать (совершенно разумно), что a^b – это результат применения «исключающего или» к a и b , а вовсе не a , возведенное в степень b . Однако есть и более неприятная проблема:


```
a = -1 + e ^ (i*pi);
```

В математике и в большинстве языков программирования, имеющих явную поддержку для возведения в степень, соответствующий оператор имеет очень высокий приоритет. Вероятно, автор этого кода ожидал, что при разборе выражения возведение в степень будет выполняться первым:

```
a = -1 + (e ^ (i*pi));
```

На самом деле компилятор ничего не знает о том, что пользователь думает о приоритете операции возведения в степень. Он видит «исключающее или» и производит разбор так, как считает правильным:

```
a = (-1 + e) ^ (i*pi);
```

В данном случае лучше отказаться от перегрузки оператора и воспользоваться более понятной не-операторной функцией:

```
a = -1 + pow( e, (i*pi) );
```

Приоритет оператора – это часть его интерфейса. Следите за тем, чтобы приоритет перегруженного оператора не обманывал ожиданий пользователей.

Совет 86. Операторы, являющиеся членами и друзьями класса

Перегруженный оператор должен допускать применение любых преобразований, поддерживаемых типами его аргументов:

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

Например, конструктор класса `Complex` допускает преобразование из встроенных числовых типов в `Complex`. Не являющаяся членом функция `add` позволяет неявно применить это преобразование к любому из своих аргументов:

```
Complex add( const Complex &, const Complex & );
Complex c1, c2;
double d;

add(c1,c2);
add(c1,d); // add( c1, Complex(d,0.0) )
add(d,c1); // add( Complex(d,0.0), c1 )
```

Не являющаяся членом функция `operator +` (см. ниже) допускает те же самые неявные преобразования:

```
Complex operator +( const Complex &, const Complex & );
c1 + c2;
operator +(c1,c2); // то же, что и выше
c1 + d;
operator +(c1,d); // то же, что и выше
d + c1;
operator +(d,c1); // то же, что и выше
```

Однако, если реализовать бинарное сложение двух объектов `Complex` с помощью функции-члена, то появляется асимметрия по отношению к неявному преобразованию:

```
class Complex {
public:
    // операторы-члены класса
    Complex operator +( const Complex & ) const; // бинарный
    Complex operator -( const Complex & ) const; // бинарный
    Complex operator -() const; // унарный
    // ...
};
// ...
c1 + c2; // правильно.
c1.operator +(c2); // правильно.
c1 + d; // правильно.
c1.operator +(d); // правильно.
d + c1; // ошибка!
d.operator +(c1); // ошибка!
```

Компилятор не может применить неявное определенное пользователем преобразование к первому аргументу функции-члена. Если такое преобразование должно быть частью интерфейса, то реализация бинарного оператора в виде функции-члена не годится. Друзья класса, не являющиеся его членами, допускают применение преобразования к первому аргументу. Функции-члены способны только на преобразования типа «является разновидностью» (см. также «Совет 42»).

Совет 87. Проблемы инкремента и декремента

Даже самые лучшие программисты на C обычно применяют префиксную и постфиксную формы инкремента и декремента на равных в ситуациях, где подходит любая форма:

```
int j;
for( j = 0; j < max; j++ ) /* нормально, в языке C. */
```

Однако в C++ такая практика считается старомодной. Если годится любая форма, то предпочтение следует отдать префиксной. Причина связана с перегрузкой операторов.

Операторы инкремента и декремента часто перегружают для поддержки операций с итераторами или интеллектуальными указателями. Они могут как являться, так и не являться членами класса, но обычно все же реализуются в виде функций-членов:

```
class Iter {
public:
    Iter &operator ++(); // префиксный
    Iter operator ++(int); // постфиксный
    Iter &operator --(); // префиксный
    Iter operator --(int); // постфиксный
    // ...
};
```

Префиксная форма должна возвращать модифицируемое lvalue, это согласуется с поведением встроенных операторов. На практике это означает, что оператор должен возвращать ссылку на свой аргумент:

```
Iter &Iter::operator ++() {
    // инкрементировать *this ...
    return *this;
}
// ...
int j = 0;
++++j; // правильно, но j+=2 лучше
Iter i;
+++++++i; // правильно, хотя выглядит странно
```

Постфиксные формы отличаются от префиксных наличием неиспользуемого целочисленного аргумента. Компилятор просто передает нулевой фактический аргумент, чтобы отличить одну форму от другой:

```
Iter i;
+i; // то же, что i.operator ++();
i++; // то же, что i.operator ++(0);
i.operator ++(1024); // допустимо, но странно
```

Как правило, реализации постфиксных операторов игнорируют этот целочисленный аргумент. Чтобы имитировать поведение встроенных операторов инкремента и декремента, перегруженная версия должна возвращать копию объекта, содержащего то значение, которое было до применения операции. Обычно постфиксный оператор реализуется посредством соответствующего префиксного:

```
Iter Iter::operator ++(int) {
    Iter temp( *this );
    ++*this;
    return temp;
}
```

По существу, требуется, чтобы постфиксный оператор возвращал результат по значению. Даже если будет применено какое-то преобразование программы, например, оптимизация именованного возвращаемого значения (см. «Совет 58»), все равно постфиксный оператор инкремента или декремента, скорее всего, будет работать медленнее соответствующего префиксного, если аргумент принадлежит типу класса. Рассмотрим типичное применение стандартной библиотеки:

```
vector<T> v;
// ...
vector<T>::iterator end( v.end() );
for( vector<T>::iterator vi( v.begin() ); vi != end;
    vi++ ) { // плохо!
    // ...
}
```

Итератор для класса `vector` может быть простым указателем, и в этом случае применение постфиксного инкремента не снижает производительности, но может быть и объектом типа класса, а тогда эффект будет ощутимым. Поэтому в C++ всегда при прочих равных следует пользоваться префиксной, а не постфиксной формой. В реализации многих обобщенных алгоритмов эту рекомендацию

принимают слишком буквально и стараются избежать постфиксных операторов инкремента и декремента любым путем:

```
template <typename In, typename Out>
Out myCopy( In b, In e, Out r ) {
    while( b != e ) {
        // а не *r++ = *b++
        *r = *b;
        ++r;
        ++b;
    }
    return r;
}
```

Отметим, что встроенные постфиксные операторы инкремента и декремента возвращают `rvalue`. Это означает, что результат операции не имеет адреса и не может передаваться операторам, которым нужно `lvalue` (см. «Совет 6»):

```
int a = 12;
++a = 10; // правильно
++++a; // правильно
a++ = 10; // ошибка!
a++++; // ошибка!
```

К сожалению, показанная выше реализация постфиксного `++` возвращает анонимный временный объект, сгенерированный компилятором. Согласно стандарту, это не `lvalue`, но мы можем вызывать функции-члены такого объекта, а, следовательно, его можно инкрементировать и присваивать. Однако инкрементированный временный объект, которому даже присвоено значение, уничтожается в конце выражения!

```
Iter i;
Iter j;
++i = j; // правильно
i++ = j; // допустимо, но должно быть ошибочным!
```

На значение `i` не влияет присваивание `j`, так как новое значение было присвоено анонимному временному объекту, который (предположительно) содержал значение `i` до инкремента. Более безопасная реализация определенного пользователем постфиксного оператора инкремента или декремента должна была бы возвращать константный объект:

```
class Iter {
public:
    Iter &operator ++(); // префиксный
    const Iter operator ++(int); // постфиксный
    Iter &operator --(); // префиксный
    const Iter operator --(int); // postfix
    // ...
};
// ...
i++ = j; // ошибка!
i++++; // ошибка!
```

Это предотвратит большинство случаев некорректного использования значения, возвращаемого операторами инкремента и декремента, но не защитит от

преднамеренного злоупотребления. Возвращаемое значение не модифицируемо, но адрес у него все же есть:

```
const Iter *ip = &i++;
```

Этот «умный» программист умудрился взять адрес не `i`, а сгенерированного компилятором временного объекта, который будет уничтожен сразу после инициализации указателя. Это мошенничество со злым умыслом, и оно будет непременно наказано (см. «Совет 11»).

Выше мы упомянули, что определенные пользователем операторы инкремента и декремента обычно реализуются в виде функций-членов. Но когда речь идет об инкременте и декрементах перечислений, это уже не так, поскольку перечисления не могут иметь функций-членов:

```
enum Sin { pride, covetousness, lust, anger,
          gluttony, envy, sloth, future_use, num_sins };
// гордыня, скудость, сладострастие, гнев, чревоугодие, зависть,
// праздность, зарезервировано_на_будущее, число_грехов

inline Sin &operator ++( Sin &s )
{ return s = static_cast<Sin>(s+1); }

inline const Sin operator ++( Sin &s, int ) {
    Sin ret( s );
    s = ++s;
    return ret;
}
```

Обратите внимание на отсутствие контроля выхода за границы в этих функциях. Программист, который решил для представления некоторой концепции воспользоваться перечислением, а не более изощренным классом, наверное, поступил так по причинам эффективности. Любая попытка контролировать диапазоны для такого типа, вероятно, противоречит исходному замыслу. Кроме того, мы получим в результате множество лишних двойных проверок граничных условий:

```
for( Sin s = pride; s != num_sins; ++s ) // ...
```

Совет 88. Неправильная интерпретация шаблонных операций копирования

Шаблонные функции-члены часто применяются для реализации конструкторов. Например, во многих стандартных контейнерах есть шаблонный конструктор, который позволяет инициализировать контейнер последовательностью:

```
template <typename T>
class Cont {
public:
    template <typename In>
        Cont( In b, In e );
    // ...
};
```

Применение такого шаблонного конструктора дает возможность инициализировать контейнер последовательностью, взятой из любого источника, в ре-

зультате чего контейнер становится намного полезнее. В стандартном шаблоне `auto_ptr` также применяются шаблонные функции-члены:

```
template <class X>
class auto_ptr {
public:
    auto_ptr( auto_ptr & ); // copy ctor
    template <class Y>
        auto_ptr( auto_ptr<Y> & );
    auto_ptr &operator =( auto_ptr & ); // copy assignment
    template <class Y>
        auto_ptr &operator =( auto_ptr<Y> & );
    // ...
};
```

Отметим, однако, что `auto_ptr`, помимо шаблонного конструктора и оператора присваивания, еще и явно объявляет собственные операции копирования. Это необходимо для обеспечения корректного поведения, поскольку шаблонные функции-члены никогда не применяются для копирования. Как всегда в отсутствие явно объявленного конструктора копирования или копирующего оператора присваивания, компилятор сгенерирует их автоматически. Об этом исключении из правил конкретизации шаблонов часто забывают, что иногда приводит к ошибкам:

➤ `gotcha88/money.h`

```
enum Currency { CAD, DM, USD, Yen };

template <Currency currency>
class Money {
public:
    Money( double amt );
    template <Currency otherCurrency>
        Money( const Money<otherCurrency> & );
    template <Currency otherCurrency>
        Money &operator =( const Money<otherCurrency> & );
    ~Money();
    double get_amount() const
        { return amt_; }
    // ...
private:
    Curve *myCurve_;
    double amt_;
};
// ...
Money<Yen> acct1( 1000000.00 );
Money<DM> acct2( 123.45 );
Money<Yen> acct3( acct2 ); // шаблонный конструктор
Money<Yen> acct4( acct1 ); // сгенерированный компилятором
                           // конструктор копирования!
acct3 = acct2; // шаблонный оператор присваивания
acct4 = acct1; // сгенерированный компилятором оператор
               // присваивания!
```

Это просто еще одно проявление очень старой проблемы, характерной для проектирования классов в C++. Если класс содержит указатель или какой-либо

иний дескриптор ресурса, которым сам не управляет, то нужно очень внимательно относиться к операции копирования в этом классе, чтобы избежать утечки ресурсов или совмещения имен. Рассмотрим фрагмент реализации вышеупомянутого шаблонного оператора присваивания:

➤➤ gotcha88/money.h

```
template <Currency currency>
template <Currency otherCurrency>
Money<currency> &
Money<currency>::operator =( const Money<otherCurrency> &rhs ) {
    amt_ = myCurve->
        convert( currency, otherCurrency, rhs.get_amount() );
}
```

Ясно, что при реализации класса Money важно следить за тем, чтобы объект класса Curve, на который ссылается myCurve, не модифицировался и не поступал в общее владение в ходе присваивания. Однако именно это совершит компилятор, если позволить ему генерировать операции копирования:

```
template <Currency currency>
Money<currency> &
Money<currency>::operator =( const Money<currency> &that ) {
    myCurve_ = that.myCurve; // утечка, совмещения и изменение curve!
    amt_ = myCurve->
        convert( currency, otherCurrency, rhs.get_amount() );
}
```

В шаблоне Money операции копирования должны быть реализованы явно.

Операции копирования никогда не реализуются с помощью шаблонных функций-членов. При проектировании любого класса обязательно уделяйте пристальное внимание операциям копирования (см. «Совет 49»).



Глава 9. Проектирование иерархий

Проектировать иерархии трудно. Иерархия классов должна быть достаточно гибкой, чтобы допускать разумное расширение, но в то же время настолько фиксированной, чтобы не пропала выражаемая ей проектная идея. Она должна быть по возможности простой, но при этом эффективно абстрагировать предметную область. В отличие от проектирования большинства других программных компонентов, иерархия классов будет расширяться и модифицироваться еще долгое время после того, как разработчик спроектировал, откомпилировал и распространил ее. Поэтому проектировщик должен решить, в какой мере пользователям разрешено расширять иерархию и подстраивать ее под свои нужды.

При проектировании иерархий приходится искать оптимальное решение с учетом различных, порой противоречивых требований. Однако, как и в линейном программировании, оптимальных решений может быть несколько. Поэтому эффективный дизайн – это скорее результат опыта и способности к предвидению, нежели рутинного применения заранее сформулированных правил. Поэтому и рекомендации в этой главе не такие жесткие, как в предыдущих, и выражают личное мнение автора.

Тем не менее, при проектировании иерархий встречаются подводные камни, на которые натываются особенно часто. Некоторые из них являются результатом переноса опыта проектирования для других языков. Другие – результат отсутствия всякого опыта. А третьи – некоторые новые, порочные, где-то подхваченные разработчиком идеи. Все это мы рассмотрим.

Совет 89. Массивы объектов класса

Остерегайтесь массивов, состоящих из элементов типа класса, особенно если это базовый класс. Рассмотрим следующую функцию-«применитель», которая вызывает некоторую другую функцию для каждого элемента массива:

>> [gotcha89/apply.cpp](#)

```
void apply( B array[], int length, void (*f)( B & ) ) {
    for( int i = 0; i < length; ++i )
        f( array[i] );
}
// ...
D *dp = new D[3];
apply( dp, 3, somefunc ); // катастрофа!
```

Проблема в том, что тип первого формального аргумента `apply` – это «указатель на B», а не «массив B». С точки зрения компилятора, мы инициализируем `B *` с помощью `D *`. Это допустимо, если `B` – открытый базовый класс `D`, поскольку

в этом случае `D` «является разновидностью» `B`. Однако массив `D` – не то же самое, что массив `B`, и программа поведет себя совершенно неправильно, если мы попытаемся применить арифметику указателей, пользуясь размерами `B` вместо `D`.

Эта ситуация показана на рис. 9.1. Функция `apply` ожидает, что указатель `array` ссылается на массив объектов `B` (слева на диаграмме), хотя на самом деле он ссылается на массив объектов `D` (справа на диаграмме). Напомним, что индексирование – это просто сокращенная запись арифметических операций над указателями (см. «Совет 7»), так что выражение `array[i]` эквивалентно `*(array+i)`. К несчастью, компилятор выполнил сложение с указателем в предположении, что `array` указывает на объект базового класса. Если объект производного класса больше или по-другому размещен в памяти, то в результате индексирования мы получим некорректный адрес.

Попытки заставить массив вести себя разумно к успеху не приводят. Если бы базовый класс `B` был объявлен абстрактным (вообще говоря, неплохая мысль), то мы вообще не смогли бы создать массив из объектов `B`, но функция `apply` все равно была бы допустима (хотя и некорректна), поскольку она имеет дело с указателями на `B`, а не с самими объектами класса `B`. Объявление формального аргумента ссылкой на массив (например, `B (&array) [3]`) эффективно, но непрактично, так как мы должны были бы зафиксировать длину массива (в данном случае 3) и не смогли бы передать указатель (скажем, на выделенный из кучи массив) в качестве фактического аргумента.

Пользоваться массивами объектов базового класса не рекомендуется ни при каких обстоятельствах, да и вообще к массивам, состоящим из объектов, следует относиться настороженно.

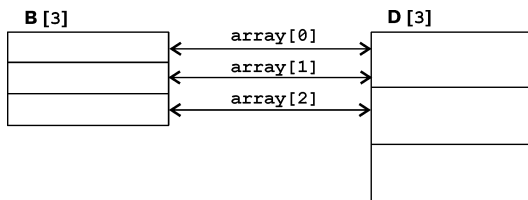


Рис. 9.1. Арифметические операции над указателями, применяемые для доступа к элементам массива объектов базового класса, обычно не работают для массива объектов производного класса

Применение обобщенного алгоритма вместо функции, настроенной на конкретный тип, может улучшить ситуацию:

```
for_each( dp, dp+3, somefunc );
```

Употребление стандартного алгоритма `for_each` позволяет компилятору вывести типы аргументов шаблонной функции. Неявное преобразование из производного класса в открытый базовый не составляет проблемы, поскольку такое преобразование просто не выполняется. Компилятор конкретизирует `for_each`

сразу для производного класса D. К сожалению, это решение отличается от первоначального, так как мы подменили полиморфизм времени исполнения полиморфизмом на этапе компиляции.

Лучше было бы воспользоваться массивом указателей на объекты классов, а не массивом самих объектов. С таким массивом можно работать полиморфно, не опасаясь возникновения проблем из-за арифметики указателей:

```
void apply_prime( B *array[], int length, void (*)( B * ) ) {
    for( int i = 0; i < length; ++i )
        f( array[i] );
}
```

Часто еще лучше вообще отказаться от массивов и пользоваться стандартными контейнерами, обычно в этом качестве выступает `vector`. Применение строго типизированных контейнеров позволит полностью уйти от проблем, связанных с арифметикой указателей на объекты классов. Кроме того, контейнер, содержащий указатели, допускает полиморфное использование:

```
vector<B> vb; // объекты D запрещены!
vector<B *> vbp; // полиморфно
```

Совет 90. Не всегда один контейнер можно подставить вместо другого

STL-контейнеры – это естественный выбор для программистов на C++. Однако они не могут удовлетворить все нужды, так как их сильные стороны неотъемлемы от определенных ограничений. Один из плюсов STL-контейнеров в том, что, будучи реализованы с помощью шаблонов, они позволяют принять большую часть решений о структуре и поведении на этапе компиляции. В результате получается компактная и эффективная реализация, точно настроенная с учетом статического контекста использования.

Однако не всю важную информацию можно предоставить во время компиляции. Рассмотрим, например, упрощенную структуру, ориентированную на применение в каркасах, которая поддерживает «открыто-закрытый принцип», то есть может модифицироваться и расширяться без перекомпиляции каркаса как такового. Каркас состоит из иерархии контейнеров и параллельной иерархии итераторов:

➤> `gotcha90/container.h`

```
template <typename T>
class Container {
public:
    virtual ~Container();
    virtual Iter<T> *genIter() const = 0; // Фабричный метод
    virtual void insert( const T & ) = 0;
    // ...
};
template <typename T>
class Iter {
public:
    virtual ~Iter();
```

```
virtual void reset() = 0;
virtual void next() = 0;
virtual bool done() const = 0;
virtual T &get() const = 0;
};
```

Мы можем написать код в терминах этих абстрактных базовых классов, откомпилировать его, а затем расширить возможности путем добавления новых производных классов контейнеров и итераторов:

>> gotcha90/container.cpp

```
template <typename T>
void print( Container<T> &c ) {
    auto_ptr< Iter<T> > i( c.genIter() );
    for( i->reset(); !i->done(); i->next() )
        cout << i->get() << endl;
}
```

Вообще говоря, применение параллельных иерархий при проектировании - вещь не бесспорная, так как изменения в одной иерархии требует согласованных изменений и в другой. Лучше бы иметь только одну точку изменения. Однако использование паттерна Factory Method (Фабричный Метод) в реализации класса Container сглаживает эту проблему в частном случае иерархий Container/Iter.

Factory Method – это механизм, с помощью которого пользователь интерфейса абстрактного базового класса может создать объект, соответствующий фактическому типу объекта производного класса, оставаясь в неведении относительно типа объекта. Если в качестве абстрактного базового класса выступает Container, то обращение к его фабричному методу genIter означает: «Создай объект класса, производного от Iter, который тебя устраивает, но избавь меня от деталей». Часто Factory Method дает альтернативу не рекомендуемой практике применения условного кода на базе проверки типов (см. «Совет 96»). Другими словами, мы ни в коем случае не хотим писать код такого рода: «Слушай, Container, если ты на самом деле Array, дай мне итератор ArrayIter. А если ты Set, дай мне SetIter. А если...»

Довольно легко спроектировать взаимозаменяемые типы Container. Тогда вместо Container<T> можно было бы подставить Set<T>, и при этом было бы допустимо стандартное преобразование из Set<T> * в Container<T> *. Наличие чисто виртуального фабричного метода genIter в базовом классе Container<T> – это явное напоминание проектировщику типа конкретного контейнера вспомнить и об иерархии Iter:

```
template <typename T>
SetIter<T> *Set<T>::genIter() const
{ return new SetIter<T>( *this ); } // не забудь написать SetIter!
```

Однако, к сожалению, многие предполагают, что взаимозаменяемость элементов влечет за собой и взаимозаменяемость контейнеров, содержащих эти элементы. Мы знаем, что это не так для массивов (контейнеров, встроенных в язык C++). Массив объектов базового класса не всегда можно подставить вместо массива

объектов производного класса (см. «Совет 89»). То же относится и к определенным пользователями контейнерам, содержащим взаимозаменяемые элементы. Рассмотрим следующую простую иерархию контейнеров для оценки стоимости финансовых инструментов:

➤➤ gotcha90/bondlist.h

```
class Object
{ public: virtual ~Object(); };
class Instrument : public Object
{ public: virtual double pv() const = 0; };
class Bond : public Instrument
{ public: double pv() const; };
class ObjectList {
public:
void insert( Object * );
Object *get();
// ...
};
class BondList : public ObjectList { // bad idea!!!
public:
void insert( Bond *b )
{ ObjectList::insert( b ); }
Bond *get()
{ return static_cast<Bond *>(ObjectList::get()); }
// ...
};
```

➤➤ gotcha90/bondlist.cpp

```
double bondPortfolioPV( BondList &bonds ) {
double sumpv = 0.0;
for( each bond in list ) {
Bond *b = current bond;
sumpv += b->pv();
}
return sumpv;
}
```

Нет ничего плохого в том, чтобы реализовать список указателей на `Bond` в виде списка указателей на `Object` (хотя лучше было бы воспользоваться списком `void *`, а весь класс `Object` отправить на свалку, см. «Совет 97»). Ошибка в том, что мы использовали открытое наследование, а не закрытое наследование или агрегирование, и тем самым постулировали наличие отношения «является» между типами, которые не взаимозаменяемы. Однако, в отличие от случая, когда у нас имеется указатель на указатель (или массив указателей), компилятор не может попенять нам за проявленное легкомыслие (см. «Совет 33»).

➤➤ gotcha90/bondlist.cpp

```
class UnderpaidMinion : public Object {
public:
virtual double pay()
{ /* положить миллион долларов на счет minion */ }
};
void sneaky( ObjectList &list )
```

```
{ list.insert( new UnderpaidMinion ); }  
void victimize() {  
    BondList &blist = getBondList();  
    sneaky( blist );  
    bondPortfolioPV( blist ); //готово!  
}
```

Здесь мы ухитрились подставить один класс-потомок вместо другого, то есть мы подсунули каркасу `UnderpaidMinion` вместо ожидаемого им `Bond`. На большинстве платформ в результате будет вызвана `UnderpaidMinion::pay` вместо `Bond::pay` (необнаруживаемая ошибка, которая проявится во время выполнения). Как нельзя подставлять массив объектов производного класса вместо массива объектов базового класса, так пользовательские контейнеры, содержащие объекты производного класса или указатели на них, нельзя подставлять вместо контейнеров, содержащих объекты или указатели на объекты базового класса.

Взаимозаменяемость контейнеров, если она вообще имеет место, должна ограничиваться структурой самого контейнера, а не содержащимися в нем элементами.

Совет 91. Что такое защищенный доступ?

Вопрос о доступности членов класса иногда зависит от того, с какой точки зрения их рассматривать. Например, открытые члены базового класса выглядят закрытыми с точки зрения закрыто наследующего ему производного класса:

```
class Inst {  
public:  
    int units() const  
        { return units_; }  
    // ...  
private:  
    int units_;  
    // ...  
};
```

```
class Sbond : private Inst {  
    // ...  
};  
// ...  
void doUnits() {  
    Sbond *bp = getNextBond();  
    Inst *ip = (Inst *)bp; // приведение в старом стиле необходимо ...  
    bp->units(); // ошибка!  
    ip->units(); // допустимо  
}
```

Эта конкретная ситуация забавна, но на практике встречается нечасто. Как правило, мы пользуемся открытым наследованием, если хотим раскрыть интерфейс базового класса через интерфейс производного. Закрытое наследование применяется главным образом для наследования реализации. Если для преобразования указателя на производный класс в указатель на базовый класс возникает необходимость в приведении, мы можем точно сказать, что проект неудачен.

Кстати, обратите внимание, что для преобразования указателя на производный класс в указатель на закрытый базовый класс требуется приведение в старом стиле. Лучше бы воспользоваться более безопасным оператором `static_cast`, но, увы... Не может `static_cast` привести указатель к недоступному базовому классу. А приведение в старом стиле способно замаскировать ошибку, которая возникнет, если позже отношение между классами `Sbond` и `Inst` изменится (см. «Совет 40» и «Совет 41»). Лично я считаю, что от этого приведения следует избавиться, а всю иерархию перепроектировать.

Давайте снабдим базовый класс виртуальным деструктором, сделаем функцию-акцессор защищенной и создадим подходящие производные классы:

```
class Inst {
public:
    virtual ~Inst();
    // . . .
protected:
    int units() const
    { return units_; }
private:
    int units_;
};
class Bond : public Inst {
public:
    double notional() const
    { return units() * faceval_; }
    // . . .
private:
    double faceval_;
};

class Equity : public Inst {
public:
    double notional() const
    { return units() * shareval_; }
    bool compare( Bond * ) const;
    // . . .
private:
    double shareval_;
};
```

Функция-член базового класса, возвращающая число единиц финансового инструмента, теперь защищена. Отсюда с очевидностью следует, что она предназначена для использования в производных классах. Эта информация используется при вычислении условной суммы и для облигаций, и для непривилегированных акций.

Однако в наши дни считается разумным сравнивать непривилегированные акции (`equity`) с облигациями (`bond`), поэтому в классе `Equity` объявлена функция `compare` как раз для этой цели:

```
bool Equity::compare( Bond *bp ) const {
    int bunits = bp->units(); // ошибка!
    return units() < bunits;
}
```

Многие программисты удивляются, что эта первая попытка воспользоваться защищенной функцией `units` приводит к нарушению защиты. Причина в том, что произведена попытка доступа из члена производного класса `Equity` к объекту класса `Bond`. Если речь идет о не-статических членах, то для защищенного доступа требуется не только, чтобы функция, выполняющая доступ, была членом или другом производного класса, но и чтобы объект, к которому осуществляется доступ, имел тот же тип, что и класс, членом которого является функция (или, что эквивалентно, был объектом открытого производного класса) или который объявил ее своим другом.

В данном случае ни членам класса `Equity`, ни его друзьям нельзя доверять. Они могут неправильно интерпретировать смысл функции `units` в объекте `Bond`. Класс `Inst` предоставляет своим подклассам функцию `units`, но каждый подкласс должен интерпретировать ее правильно. Маловероятно, что простое сравнение числа единиц каждого инструмента с помощью функции `compare` будет иметь какой-то смысл без дополнительной (и закрытой) информации о номинальной стоимости облигации или цене акции. А такая информация для каждого класса специфична. Это дополнение к правилам контроля доступа к защищенным членам благотворно влияет на общий дизайн, так как поощряет разрыв зависимостей между производными классами.

Попытка обойти защиту, передав `Bond` как `Inst`, не поможет:

```
bool Equity::compare( Inst *ip ) const {
    int bunits = ip->units(); // ошибка!
    return units() < bunits;
}
```

Доступ к унаследованным защищенным членам разрешен только объектам производного класса (и открыто наследующим им). Если функция `compare` действительно необходима, то ее надо переместить вверх по иерархии туда, где ее наличие не станет причиной появления лишних связей между производными классами:

```
bool Inst::unitCompare( const Inst *ip ) const
{ return units() < ip->units(); }
```

Если такой вариант вам не подходит, и вы не возражаете против введения некоторой зависимости между классами `Equity` и `Bond` (а должны бы возражать), то на помощь придет «обоюдный друг»:

```
class Bond : public Inst {
public:
    friend bool compare( const Equity *, const Bond * );
    // . . .
};
class Equity : public Inst {
public:
    friend bool compare( const Equity *, const Bond * );
    // . . .
};
bool compare( const Equity *eq, const Bond *bond )
{ return eq->units() < bond->units(); }
```

Совет 92. Применение открытого наследования для повторного использования кода

Иерархии классов способствуют повторному использованию кода двояко. Во-первых, код, используемый в разных производных классах, можно поместить в их общий базовый класс. Во-вторых, все открыто наследующие производные классы могут разделять интерфейс своего базового класса. И разделение кода, и разделение интерфейса – это похвальные цели, но разделение интерфейса важнее.

Применение открытого наследования только ради повторного использования реализации базового класса часто приводит к неестественному, неудобному для сопровождения и, в конечном итоге, неэффективному проекту. Причина в том, что априорная установка на наследование с целью повторного использования кода может наложить на интерфейс базового класса такие ограничения, что подставить вместо него производные окажется затруднительно. А в результате производным классам будет сложно воспользоваться общим кодом, написанным во исполнение «контракта», заключенного базовым классом. Обычно, гораздо большей степени повторного использования удастся добиться за счет написания большого объема кода общего назначения, а не разделения мелких фрагментов, находящихся в базовом классе.

Преимущества использования кода общего назначения, написанного в соответствии с контрактом базового класса, настолько велики, что зачастую имеет смысл облегчить этот процесс, спроектировав иерархию с интерфейсным классом в корне. «Интерфейсным классом» называется базовый класс, который не содержит никаких данных и имеет виртуальный деструктор. Обычно все функции-члены в нем виртуальны, а конструктора нет вовсе. Иногда интерфейсные классы называют «классами-протоколами», поскольку они определяют лишь протокол работы с иерархией, не предлагая никакой реализации. («Присоединяемый» (mix-in) класс похож на интерфейсный, но все-таки содержит какие-то данные и реализацию.)

Размещение интерфейсного класса в корне иерархии облегчает ее последующее сопровождение, позволяя применять такие паттерны, как **Decorator** (Декоратор), **Composite** (Компоновщик), **Proxy** (Заместитель) и другие. (Кроме того, интерфейсные классы упрощают решение ряда технических проблем, связанных с использованием виртуальных базовых классов; см. «Совет 53».)

Канонический пример интерфейсного класса – это применение паттерна **Command** (Команда) для реализации иерархии абстрактных функций обратного вызова. Например, в графическом интерфейсе может быть класс **Button**, представляющий кнопку, которая при нажатии исполняет функцию **Action**.

➤> [gotcha92/button.h](#)

```
class Action {
public:
    virtual ~Action();
```



```

    virtual void operator ()() = 0;
    virtual Action *clone() const = 0;
};
class Button {
public:
    Button( const char *label );
    ~Button();
    void press() const;
    void setAction( const Action * );
private:
    string label_;
    Action *action_;
};

```

Паттерн **Command** инкапсулирует операцию в виде объекта. Мы увидим ниже, что этот паттерн позволяет задействовать в проекте и некоторые другие.

Обратите внимание на использование перегруженной функции `operator ()` в реализации `Action`. Мы могли бы воспользоваться не-операторной функцией-членом `execute`, но перегрузка оператора вызова функции является в C++ идиомой, которая говорит, что `Action` – это абстракция функции, точно так же, как перегрузка `operator ->` означает, что объект представляет собой «интеллектуальный указатель» (см. «Совет 24» и «Совет 83»). В классе `Action` находит применение также паттерн **Prototype** (Прототип), о чем свидетельствует наличие функции-члена `clone`, которая создает дубликат объекта, не зная его типа (см. «Совет 76»).

В нашем первом конкретном воплощении типа `Action` используется паттерн **Null Object** (Пустой Объект) для создания объекта `Action`, который не делает ничего, но при этом соблюдает требования интерфейса `Action`. Класс `NullAction` «является разновидностью» `Action`:

➤ gotcha92/button.h

```

class NullAction : public Action {
public:
    void operator ()()
    {
    }
    NullAction *clone() const
    { return new NullAction; }
};

```

Имея каркас `Action`, уже совсем просто создать безопасную и гибкую реализацию `Button`. Применение паттерна `Null Object` гарантирует, что `Button` всегда будет делать что-то при нажатии, даже если это «что-то» на деле означает «не делать ничего» (см. «Совет 96»).

➤ gotcha92/button.cpp

```

Button::Button( const char *label )
    : label_( label ), action_( new NullAction ) {}
void Button::press() const
    { (*action_)(); }

```

Паттерн **Prototype** позволяет классу `Button` иметь собственную копию `Action`, ничего не зная о точном типе копируемого объекта `Action`:

➤ gotcha92/button.cpp

```
void Button::setAction( const Action *action )
{ delete action_; action_ = action->clone(); }
```

Тем самым заложена основа каркаса Button/Action и, как видно из рис. 9.2., мы можем наполнить его конкретными операциями (которые, в отличие от NullAction, все же что-то делают), не перекомпилируя при этом весь каркас.

Наличие интерфейсного класса в корне иерархии Action позволяет расширять возможности этой иерархии. Например, можно было бы воспользоваться паттерном **Composite**, чтобы наделить Button способностью исполнять целую группу действий Action:

gotcha92/moreactions.h

```
class Macro : public Action {
public:
    void add( const Action *a )
        { a_.push_back( a->clone() ); }
    void operator ()() {
        for( I i(a_.begin()); i != a_.end(); ++i )
            (**i)();
    }
    Macro *clone() const {
        Macro *m = new Macro;
        for( CI i(a_.begin()); i != a_.end(); ++i )
            m->add((*i).operator ->());
        return m;
    }
private:
    typedef list< Cptr<Action> > C;
    typedef C::iterator I;
    typedef C::const_iterator CI;
    C a_;
};
```

Присутствие «легкого» интерфейсного класса в корне иерархии Action дало нам возможность применить паттерны **Null Object** и **Composite**, как показано на рис. 9.3. А будь в базовом классе Action какая-то содержательная реализация, все производные классы должны были бы наследовать ее вместе со всеми побочными эффектами, связанными с инициализацией и уничтожением объектов. Это не позволило бы нам эффективно воспользоваться паттернами Null Object, Composite и другими.

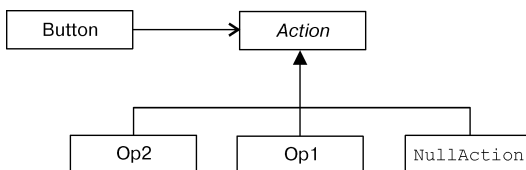


Рис. 9.2. Применение паттернов Command и Null Object для реализации операций обратного вызова при нажатии кнопки

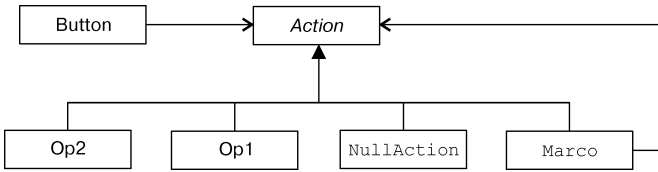


Рис. 9.3. Расширение иерархии *Action* за счет применения паттерна Composite

Однако надо признать, что, отдавая предпочтение гибкости интерфейсного класса, мы все же жертвуем степенью разделения кода и некоторым повышением производительности, которого можно было бы достичь, если бы базовый класс был более содержательным. Например, часто бывает, что реализации многих конкретных классов, производных от *Action*, в чем-то дублируются, и этот общий код можно было бы поместить в базовый класс. Но тогда мы утратили бы возможность вводить в иерархию дополнительную функциональность, как в случае применения паттерна Composite выше. В таких ситуациях можно попытаться взять лучшее от каждого варианта за счет заведения искусственного базового класса с одной-единственной целью предоставить общую реализацию (рис. 9.4).

Однако злоупотребление таким подходом может породить иерархии, в которых слишком много искусственных классов, не имеющих отношения к предметной области. Подобные иерархии сложны для понимания и сопровождения.

В общем случае лучше уделить основное внимание наследованию интерфейса. Правильное и эффективное повторное использование кода станут автоматическим следствием.

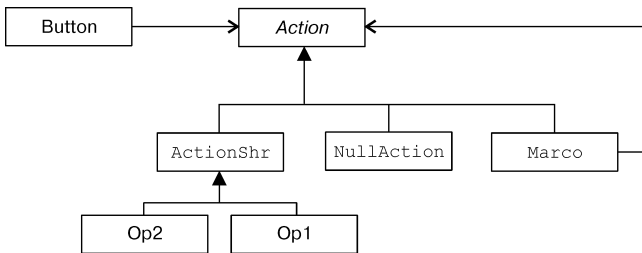


Рис. 9.4. Искусственный базовый класс, позволяющий наследовать как интерфейс, так и общую реализацию

Совет 93. Конкретные открытые базовые классы

С точки зрения проектирования открытые базовые классы, вообще говоря, должны быть абстрактными, поскольку служат для представления абстрактных понятий из предметной области. Мы не ожидаем и не желаем видеть абстракции

в нашем родном материальном пространстве (представьте, как мог бы выглядеть абстрактный служащий, фрукт или устройство ввода/вывода). Точно также, нам ни к чему объекты абстрактных интерфейсов в пространстве программы.

В C++ есть также и практические соображения, относящиеся к реализации. Нужно принимать во внимание срезку и связанные с ней проблемы, в частности, реализацию операций копирования (см. «Совет 30», «Совет 49» и «Совет 65»). В общем, делайте открытые базовые классы абстрактными.

Совет 94. Не пренебрегайте вырожденными иерархиями

Эвристика базовых и автономных классов в проектировании весьма различна. Клиентский код работает с базовыми классами совсем не так, как с автономными. Поэтому желательно заранее решить, в какой роли будет выступать проектируемый вами класс.

Осознание на ранних этапах проектирования того, что некоторый класс позже может стать базовым, и преобразование его в простую двухуровневую иерархию. Это пример «проектирования с расчетом на будущее». Тем самым вы заставляете пользователей иерархии писать код в соответствии с требованиями абстрактного интерфейса и облегчаете последующее расширение возможностей иерархии. Альтернативное решение – начать с конкретного класса, а производные ввести позже – вынудит пользователей переписывать уже готовый код. Такие простые иерархии можно назвать «вырожденными» (это термин не следует трактовать ни в математическом, ни в этическом смысле).

Автономные классы, которые позже становятся базовыми, могут внести хаос в программу. Автономные классы часто реализуются с семантикой значения, то есть могут эффективно копироваться по значению; пользователи принимают как данность, что их можно передавать по значению в качестве аргументов, возвращать по значению и присваивать один другому.

Когда такой класс превращается в базовый, каждое копирование грозит срезкой (см. «Совет 30»). Объекты автономных классов могут быть элементами массива; позже это может привести к ошибкам при арифметических операциях над указателями (см. «Совет 89»). Если код пишется в предположении, что некоторый тип имеет фиксированный размер или жестко заданное поведение, то при нарушении этого допущения могут возникать и более тонкие ошибки. Делайте потенциальные базовые классы абстрактными.

И наоборот. Многие, быть может, даже большинство классов никогда не будут базовыми, и не надо их проектировать как таковые. Разумеется, это относится к мелким типам, которые должны быть максимально эффективны. Типичные примеры типов, которые очень редко являются частью иерархии: числовые типы, даты, строки и так далее. Мы проектировщики и должны пытаться следовать этой рекомендации, применяя свой опыт, проницательность и способность к предвидению.

Совет 95. Не злоупотребляйте наследованием

Слишком широкие или глубокие иерархии, возможно, свидетельствуют о неудачном дизайне. Часто такие иерархии возникают из-за неправильного распределения обязанностей между иерархиями. Рассмотрим следующую простую иерархию геометрических фигур (рис. 9.5).

По умолчанию все фигуры рисуются синим цветом. Предположим, какому-то новообращенному в веру С++ программисту, который еще находится под впечатлением от открывшихся ему возможностей наследования, дали задание расширить иерархию, так чтобы фигуры можно было рисовать и красным цветом. Легко!

На рис. 9.6 показано, что теперь у нас имеется классический пример «экспоненциально» растущей иерархии. Чтобы добавить еще один цвет, придется ввести в иерархию новый класс для каждой фигуры. А если мы захотим добавить новую фигуру, понадобится заводить класс для каждого цвета. Это глупо, а правильное решение напрашивается. Нужно вместо наследования воспользоваться композицией (рис. 9.7).

Square (квадрат) является Shape (фигура) и содержит Color (цвет). Но не все примеры злоупотребления наследованием настолько очевидны. Рассмотрим иерархию, представляющую опционы для разных типов финансовых инструментов (рис. 9.8).

В ней есть единственный базовый класс опциона, а каждый его конкретный подкласс – это комбинация типа опциона и типа финансового инструмента, к которому он применяется. И снова мы получаем расширяющуюся иерархию, в которой добавление одного нового типа опциона или финансового инструмента приводит к порождению многих классов. Обычно правильное решение – разработать не одну монолитную иерархию, а композицию простых иерархий, как показано на рис. 9.9.

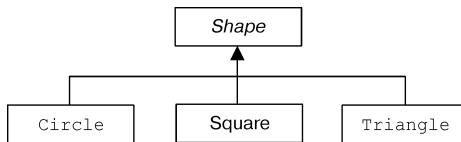


Рис. 9.5. Иерархия геометрических фигур

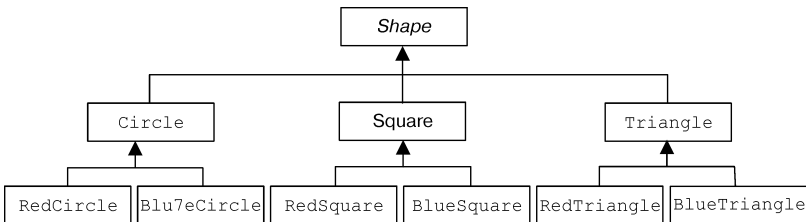


Рис. 9.6. Неправильная экспоненциально растущая иерархия

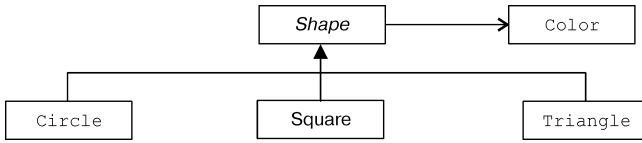


Рис. 9.7. Правильное решение, в котором применяется сочетание наследования и композиции

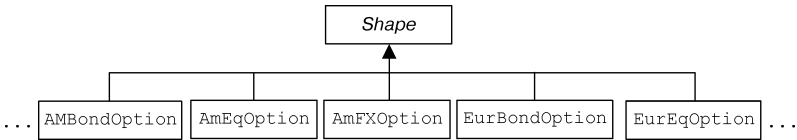


Рис. 9.8. Плохо спроектированная монолитная иерархия

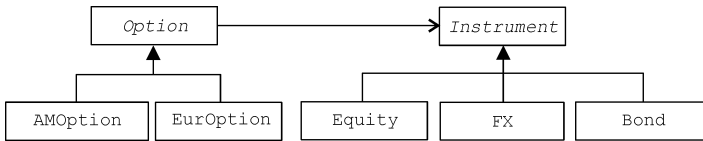


Рис. 9.9. Правильный дизайн – композиция простых иерархий

`Option` содержит `Instrument`. В данном случае трудности при проектировании иерархий возникли из-за некачественного анализа предметной области, но не так уж редко громоздкие иерархии появляются, несмотря на безупречно проведенный анализ. Продолжая пример с финансовыми инструментами, рассмотрим упрощенную реализацию класса облигаций:

```

class Bond {
public:
    // ...
    Money pv() const; // вычислить текущую стоимость
};
  
```

Функция-член `pv` вычисляет текущую стоимость облигации (`Bond`). Но может существовать несколько алгоритмов расчета. Один из вариантов – поместить все алгоритмы в одну функцию и выбирать нужный, указывая его код:

```

class Bond {
public:
    // ...
    Money pv() const;
    enum Model { Official, My, Their };
    void setModel( Model );
private:
    // ...
    Model model_;
  
```

```

};
Money Bond::pv() const {
    Money result;
    switch( model_ ) {
        case Official:
            // ...
            return result;
        case My:
            // ...
            return result;
        case Their:
            // ...
            return result;
    }
}

```

Однако при таком подходе трудно добавить новую модель расчета цены, поскольку придется изменять исходные тексты и перекомпилировать всю программу. Стандартная практика объектно-ориентированного проектирования подсказывает, что для реализации нового поведения нужно воспользоваться наследованием и динамическим связыванием (рис. 9.10).

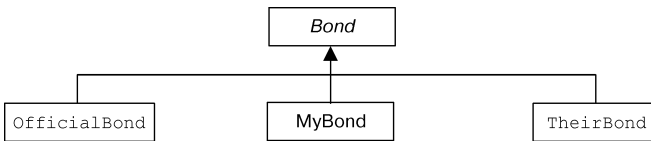


Рис. 9.10. Неправильное применение наследования; оно используется для изменения поведения единственной функции-члена

К сожалению, при таком подходе поведение функции `pv` фиксируется во время создания объекта `Bond` и в дальнейшем не может быть изменено. Кроме того, другие аспекты реализации `Bond` могут изменяться независимо от функции `pv`. В результате мы получим комбинаторный рост числа производных классов.

Например, в классе `Bond` может быть функция-член для вычисления волатильности цены облигации. Если алгоритм ее вычисления не зависит от способа вычисления текущей цены, то при добавлении нового способа вычисления той или другой величины придется добавить в иерархию новые производные классы для каждого сочетания алгоритмов расчета цены и волатильности. В общем случае наследование используется для реализации разновидности поведения объекта в целом, а не отдельных его операций.

Как и в предыдущем примере с цветными фигурами, правильное решение – применить композицию. В частности, мы можем воспользоваться паттерном **Strategy** (Стратегия) для преобразования монолитной иерархии `Bond` в композицию простых иерархий (рис. 9.11).

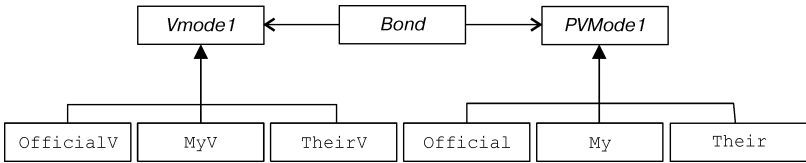


Рис. 9.11. Правильное использование паттерна Strategy для выражения идеи независимого поведения двух функций-членов

Паттерн **Strategy** позволяет вынести реализацию алгоритма из тела функции-члена в отдельную иерархию:

```

class PVMode1 { // Strategy
public:
    virtual ~PVMode1();
    virtual Money pv( const Bond * ) = 0;
};
class VModel { // Strategy
public:
    virtual ~VModel();
    virtual double volatility( const Bond * ) = 0;
};
class Bond {
    // . . .
    Money pv() const
    { return pvmodel_->pv( this ); }
    double volatility() const
    { return vmodel_->volatility( this ); }
    void adoptPVMode1( PVMode1 *m )
    { delete pvmodel_; pvmodel_ = m; }
    void adoptVModel( VModel *m )
    { delete vmodel_; vmodel_ = m; }
private:
    // . . .
    PVMode1 *pvmodel_;
    VModel *vmodel_;
};
  
```

Использование паттерна Strategy помогло нам одновременно упростить структуру иерархии Bond и изменить поведение функций pv и volatility во время выполнения.

Совет 96. Управление на основе типов

В объектно-ориентированных программах никогда не следует принимать решения на основе анализа кодов типов:

```

void process( Employee *e ) {
    switch( e->type() ) { // плохой код!
    case SALARY: fireSalary( e ); break;
    case HOURLY: fireHourly( e ); break;
    case TEMP: fireTemp( e ); break;
    default: throw UnknownEmployeeType();
  }
}
  
```



```
}  
}
```

Гораздо лучше полиморфный подход:

```
void process( Employee *e )  
{ e->fire(); }
```

Преимущества полиморфного подхода неоспоримы. Он проще. Не придется перекомпилировать программу при добавлении новых типов служащих. Невозможны ошибки из-за неправильного определения типа во время выполнения. Кроме того, код, скорее всего, будет быстрее и компактнее. Принимайте решения путем динамического связывания, а не с помощью условных управляющих конструкций. (См. также «Совет 69», «Совет 90» и «Совет 98».)

Замена условного кода динамическим связыванием настолько эффективна, что часто имеет смысл переделать условную конструкцию, чтобы воспользоваться этими преимуществами. Рассмотрим код, который просто хочет обработать некоторый объект `Widget`. В открытом интерфейсе класса `Widget` есть функция `process`, но в зависимости от того, где находится объект `Widget`, может потребоваться предварительная обработка:

```
if( Widget в локальной памяти )  
    w->process();  
else if( Widget в разделяемой памяти )  
    сделать для обработки что-то хитрое  
else if( Widget на другой машине )  
    сделать что-то еще более хитрое  
else  
    error();
```

Эта условная конструкция может не только оказаться непригодной («Я хочу обработать `Widget`, но не знаю, где он находится»), она еще и повторяться в программе может многократно. И все эти независимые фрагменты придется синхронно обновить, если набор возможных местоположений `Widget` сужается или расширяется. Лучше закодировать местоположение `Widget` в его типе, как на рис. 9.12.

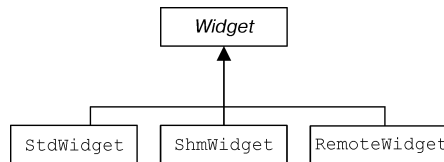


Рис. 9.12. Отказ от условной конструкции; для кодирования протокола доступа к объекту в его типе применяется паттерн `Proxy` (Заместитель)

Эта ситуация возникает настолько часто, что даже получила название. Речь идет о паттерне **Proxy** (Заместитель). Разные механизмы доступа к объекту `Widget` в зависимости от того, где он находится, теперь закодированы прямо в типе `Widget`, так что различить разные объекты позволит простая виртуальная функция. Ко всему прочему, код больше не дублируется, а виртуальная функция не может ошибиться при выборе способа доступа:

```
Widget *w = getNextWidget();
w->process();
```

Еще одно существенное достоинство отказа от условного кода настолько очевидно, что его можно даже не заметить: чтобы избежать принятия неправильного решения, лучше не принимать решения вовсе. Проще говоря, чем меньше в программе условного кода, тем меньше вероятность допустить в нем ошибку.

Одно из воплощений этой рекомендации – паттерн **Null Object** (Пустой Объект). Рассмотрим функцию, которая возвращает указатель на «устройство» (device), которым нужно «управлять» (handle):

```
class Device {
public:
    virtual ~Device();
    virtual void handle() = 0;
};
// ...
Device *getDevice();
```

Абстрактный базовый класс `Device` представляет разные виды устройств. Вполне возможно, что функция `getDevice` не сможет вернуть объект `Device`, поэтому код для получения и управления устройством будет выглядеть так:

```
if( Device *curDevice = getDevice() )
    curDevice->handle();
```

Он несложен, но приходится принимать решение. А не забудет ли сопровождающий проверить значение, возвращенное `getDevice`, прежде чем пытаться вызвать для него функцию `handle`?

Согласно паттерну **Null Object**, мы должны создать искусственный подкласс `Device`, который удовлетворяет всем требованиям интерфейса `Device` (то есть в нем есть функция `handle`), но управление таким устройством – это пустая операция. В самом буквальном смысле:

```
class NullDevice : public Device {
public:
    void handle() {}
};
// ...
Device &getDevice();
```

Вот теперь `getDevice` всегда что-то возвращает, мы можем удалить проверку условия и избежать потенциальной ошибки в будущем:

```
getDevice().handle();
```

Совет 97. Космические иерархии

Более десяти лет назад сообщество программистов на C++ пришло к выводу, что использование «космических» иерархий (в которых все классы являются производными от некоторого корневого, обычно имеющего имя `Object`) – это неэффективная в контексте C++ методика проектирования. Причин отвергнуть такой подход накопилось немало (как с точки зрения проектирования, так и реализации).

Если говорить о проектировании, то космические иерархии нередко порождают обобщенные контейнеры объектов. Содержимое этих контейнеров часто не-

предсказуемо и ведет к неожиданному поведению во время выполнения. В классическом контрпримере Бьярна Страуструпа рассматривалась возможность поместить боевой корабль в стаканчик для карандашей – космическая иерархия это позволяет, но стаканчик был бы немало удивлен.

Среди неопытных проектировщиков распространено опасное заблуждение, будто архитектура должна быть максимально гибкой. Это неверно. Архитектура должна быть максимально приближена к предметной области и при этом сохранять достаточную гибкость для будущего расширения. Если наступает момент, когда новые требования с трудом вписываются в существующую архитектуру, то проект и код нужно перерабатывать. Стремление создать архитектуру, годящуюся на все случаи жизни, подобны попыткам получить максимально эффективный код без профилирования: и полезной архитектуры не создадите, и в эффективности проиграете (см. также «Совет 72»).

Неправильное понимание смысла архитектурного проектирования в сочетании с нежеланием заниматься трудным делом абстрагирования предметной области, часто приводит к появлению особо пагубных форм космических иерархий:

```
class Object {
public:
    Object( void *, const type_info & );
    virtual ~Object();
    const type_info &type();
    void *object();
    // ...
};
```

Здесь проектировщик вообще не попытался понять и надлежащим образом абстрагировать предметную область, а просто создал оболочку, которая позволит включить совершенно несвязанные типы в космическую иерархию. В `Object` можно обернуть объект любого типа. Никто не мешает создавать контейнеры объектов класса `Object`, в которые можно поместить все что угодно (и часто так и поступают).

Проектировщик может также предоставить средства для безопасного преобразования из типа обертки `Object` в тип обернутого объекта:

```
template <class T>
T *dynamicCast( Object *o ) {
    if ( o && o->type() == typeid(T) )
        return reinterpret_cast<T *>(o->object());
    return 0;
}
```

На первый взгляд, это приемлемый (хотя и несколько неуклюжий) подход, но давайте внимательно рассмотрим задачу извлечения и использования содержимого контейнера, в котором может находиться все что угодно:

```
void process( list<Object *> &cup ) {
    typedef list<Object *>::iterator I;
    for( I i(cup.begin()); i != cup.end(); ++i ) {
        if( Pencil *p =
            dynamicCast<Pencil>(*i) )
            p->write();
        else if( Battleship *b =
            dynamicCast<Battleship>(*i) )
            b->anchorsAweigh();
    }
```

```

else
    throw InTheTowel();
}
}

```

Любой пользователь космической иерархии будет вынужден играть в глупую «угадайку», цель которой восстановить информацию о типе, которую изначально не следовало терять. Иными словами, тот факт, что стаканчик для карандашей не может вместить боевой корабль, это не ошибка при проектировании стаканчика. Изъян следует искать в той части программы, которая полагает, что втискивать туда корабль разумно. Маловероятно, что такая возможность соответствует чему-то в предметной области, поэтому мы не должны поощрять подобный способ кодирования. Если возникает локальная необходимость в космической иерархии, значит, где-то в проекте есть упущение.

Поскольку абстракции стаканчика и корабля – упрощенные модели реального мира (что бы слово «реальный» ни означало в этом контексте), имеет смысл прикинуть, как эта ситуация развивалась бы на практике. Предположим, что вы, проектировщик (физического) стаканчика для карандашей, получили от клиентов претензию: мол, не помещается туда корабль. Что бы вы предложили: исправить проект стаканчика или обратиться за помощью совсем в другое место?

Последствия нежелания брать на себя ответственность за проект многообразны и серьезны. Любое использование контейнера, включающего объекты `Object`, грозит бесчисленными ошибками, связанными с типами. Всякое изменение множества типов объектов, которые можно обернуть в `Object`, потребует модифицировать сколь угодно много кода, а этот код может оказаться и недоступен. И, наконец, поскольку никакой эффективной архитектуры не предложено, любой пользователь контейнера оказывается перед проблемой: как получить точную информацию об анонимных объектах?

Любой из указанных аспектов проекта приводит к различным несовместимым между собой способам обнаружения и диагностики ошибок. Например, пользователь контейнера может счесть вопросы типа: «Ты кто? Карандаш? Нет? Корабль? Тоже нет?..» глупыми и предпочтет опрашивать возможности объекта. Результат будет немногим лучше (см. «Совет 99»).

Бывает, что присутствие космической иерархии не так бросается в глаза, как в рассмотренном выше случае. На рис. 9.13 изображена иерархия активов:

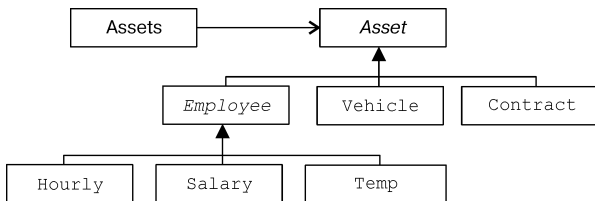


Рис. 9.13. Неопределенная иерархия.

Непонятно, является иерархия `Asset` излишне общей или нет

Сразу не ясно, является ли иерархия *Asset* излишне общей или нет, особенно когда перед нами такая высокоуровневая диаграмма проекта. Часто качество проекта трудно оценить, пока дело не дойдет до гораздо более низкого уровня или даже до кодирования. Если чрезмерно общий характер иерархии приводит к порочным методам кодирования (см. «Совет 98» и «Совет 99»), то, вероятно, это некая разновидность космической иерархии, и от нее надо избавляться.

Иногда достаточно некоторого переосмысления иерархии, даже без переделки исходного кода. Многие проблемы космических иерархий вызваны выбором слишком общего базового класса. Если начать рассматривать базовый класс как интерфейсный (рис. 9.14) и довести такое изменение концепции до пользователей, то можно будет избежать многих разрушительных приемов кодирования.

Из проекта исчезла единая космическая иерархия, зато появились три отдельных иерархии, которые соответствуют независимым подсистемам со своими интерфейсами. Это лишь концептуальное изменение, но оно очень важно. Теперь подсистема учета активов может манипулировать служащими, транспортными средствами и контрактами как активами, но, не зная, какие классы наследует *Asset*, она не будет пытаться получить точную информацию об объектах, которыми манипулирует. То же рассуждение применимо и к другим интерфейсным классам, так что вероятность натолкнуться во время исполнения на ошибку, связанную с типами, невелика.

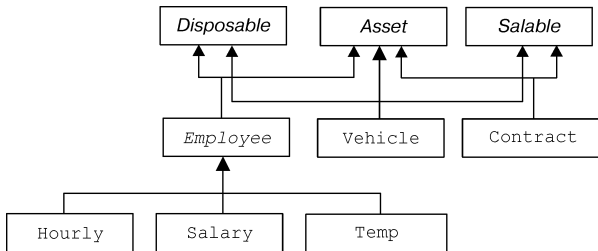


Рис. 9.14. Эффективный пересмотр концепции.

Отношение «является» будет ослаблено, если мы станем рассматривать *Asset* не как базовый класс, а в качестве класса-протокола

Совет 98. Задание «интимных» вопросов объекту

В этом разделе рассматривается возможность объектно-ориентированного проектирования, которой часто злоупотребляют: получение информации о типе во время выполнения. В языке C++ запросы о типе стандартизованы, и это как бы придает легитимность их использованию. Конечно, задавать вопросы о типе объекта во время выполнения разрешено, но пользоваться этим средством следует редко и уж ни в коем случае не закладывать его в основу проекта. Как это ни

печально, но опыт, накопленный сообществом C++ в плане правильной и эффективной работы с иерархиями типов, часто игнорируют, отдавая предпочтение подходам на основе получения информации о типе во время выполнения. А это свидетельство непроработанного, излишне общего, сложного, не поддающегося сопровождению проекта, который к тому же чреват ошибками.

Ниже показан базовый класс для представления служащих. Иногда приходится добавлять какие-то функции уже после того, как значительная часть системы разработана и протестирована. Так, в интерфейсе класса `Employee` явно кое-чего не хватает:

```
class Employee {
public:
    Employee( const Name &name, const Address &address );
    virtual ~Employee();
    void adoptRole( Role *newRole );
    const Role *getRole( int ) const;
    // . . .
};
```

Правильно. Нам нужно еще определить оптимальное количество этих активов. (Кстати, им еще придется платить, но это подождет до следующей версии.) Руководство требует, чтобы служащего можно было уволить, имея лишь указатель на его базовый класс, причем без перекомпиляции и каких-либо переделок иерархии `Employee`. Ясно, что увольнять служащих-почасовиков и получающих оклад надо по-разному:

```
void terminate( Employee * );
void terminate( SalaryEmployee * );
void terminate( HourlyEmployee * );
```

Самый прямолинейный способ решить эту задачу – прибегнуть к трюку. Мы просто зададим серию вопросов о точном типе служащего:

```
void terminate( Employee *e ) {
    if( HourlyEmployee *h = dynamic_cast<HourlyEmployee *>(e) )
        terminate( h );
    else if( SalaryEmployee *s = dynamic_cast<SalaryEmployee *>(e) )
        terminate( s );
    else
        throw UnknownEmployeeType( e );
}
```

Но у этого подхода есть очевидные недостатки с точки зрения эффективности и возможности ошибок в случае неизвестного типа. Вообще говоря, поскольку C++ – язык со статической системой типов, а механизм динамического связывания (виртуальные функции) контролируется статически, должна быть возможность полностью уйти от такого рода ошибок во время выполнения. Это достаточное основание для того, чтобы счесть приведенную выше реализацию функции `terminate` временной заплатой, а не основой для расширяемого дизайна.

Слабость этого подхода проявится еще более наглядно, если мы попробуем интерпретировать этот код в контексте предметной области, которую он призван моделировать:

Вице-президент врывается в свой кабинет в страшном гневе. Его место на парковке уже в третий раз за месяц занял какой-то задрипанный драндулет, принадлежащий разработчику-летуну, нанятому на работу месяц назад. «Подать сюда этого Дьюхерста!» – рычит он в интерком.

Не прошло и нескольких секунд, как вице-президент сверлит злополучного разработчика пронзительным взглядом и нараспев произносит: «Если вы почасовик, то будете уволены как почасовик. Иначе, если вы на окладе, то будете уволены как служащий, получающий оклад. Иначе... убирайтесь из моего кабинета и создавайте проблемы кому-нибудь другому.»

Я консультант и никогда не лишался контракта из-за менеджера, который использует информацию о типах, получаемую во время выполнения, для решения своих проблем. Конечно же, правильное решение – включить необходимые операции в базовый класс `Employee` и пользоваться стандартным, безопасным динамическим связыванием для разрешения всех вопросов о типах, возникающих во время выполнения:

```
class Employee {
public:
    Employee( const Name &name, const Address &address );
    virtual ~Employee();
    void adoptRole( Role *newRole );
    const Role *getRole( int ) const;
    virtual bool isPayday() const = 0;
    virtual void pay() = 0;
    virtual void terminate() = 0;
    // ...
};
```

... он сверлит злополучного разработчика пронзительным взглядом и нараспев произносит: «Вы уволены!»

Иногда запрашивать тип во время выполнения необходимо, бывает даже, что это лучший из всех вариантов. Мы видели, что эта техника может быть удобной в качестве временной заплатки при столкновении с плохо спроектированной программой из стороннего источника. К ней приходится прибегать и в случае, когда предъявляется требование модифицировать существующий код без перекомпиляции, но код этот спроектирован без учета новых требований, и никак иначе встроиться в него невозможно. Полезны запросы о типе во время выполнения и при отладке программ. Хотя и нечасто, но это средство применяется в таких специфических инструментах, как отладчики, обозреватели классов и так далее. И, наконец, когда моделируемая предметная область внутренне не согласована, то эта ее особенность может проявиться и в виде опросов типа во время выполнения.

Но поскольку механизм опроса типа во время выполнения утвержден стандартом C++, то многие проектировщики стали использовать его вместо более простых, более эффективных и более пригодных для сопровождения решений. Как правило, опрос типа призван компенсировать недостатки архитектуры, являющиеся следствием плохо выполненного анализа предметной области и ошибочно-го мнения, будто архитектура должна быть максимально гибкой.

На практике редко бывает необходимо задавать объекту «интимные» вопросы о его типе.

Совет 99. Опрос возможностей

На практике столь очевидное злоупотребление запросами информации о типе во время выполнения, как в функции `terminate` из предыдущего совета, обычно являются результатом попытки исправить структурные огрехи и плохого управления проектом, нежели неудачного дизайна. Однако некоторые «прогрессивные» способы применения динамического приведения типов в иерархиях с множественным наследованием часто подаются как основа архитектуры:

Служащий приходит в отдел кадров, чтобы сообщить о первом выходе на работу, а ему говорят: «В очередь, вместе с остальными активами». И отправляют в конец длинной очереди прочих служащих, в которой, как ни странно, стоят разные виды офисного оборудования, транспортные средства, мебель и контракты.

Когда, наконец, до него доходит черед, его атакуют серией странных вопросов: «Вы потребляете бензин? Вы умеете программировать? Можно снять с вас копию?» Получив отрицательные ответы на все эти вопросы, его отпускают домой, а он недоумевает, почему никто не спросил, умеет ли он мыть полы, раз именно для этого его и взяли.

Странновато звучит, не правда ли? (Если вы работали в крупной корпорации, то, может быть, все это вам знакомо.) Но иначе и быть не может, ведь мы неправильно воспользовались механизмом опроса возможностей.

Оставим пока отдел кадров и обратимся к иерархии финансовых инструментов. Предположим, что мы торгуем ценными бумагами. В нашем распоряжении подсистема определения цены и подсистема персистентности, мы можем воспользоваться ими для реализации иерархии. Требования каждой подсистемы четко выражены в интерфейсном классе, которому должен наследовать ее пользователь:

```
class Saveable { // интерфейс подсистемы персистентности
public:
    virtual ~Saveable();
    virtual void save() = 0;
    // ...
};
class Priceable { // интерфейс подсистемы определения цены
public:
    virtual ~Priceable();
    virtual void price() = 0;
    // ...
};
```

Некоторые конкретные классы иерархии `Deal` (сделка) согласуются с контрактами этих подсистем и могут пользоваться их кодом. Это стандартное, эффективное и правильное использование множественного наследования:

```
class Deal {
public:
    virtual void validate() = 0;
```



```
// . . .
};
class Bond
    : public Deal, public Priceable
    { /* . . . */ };
class Swap
    : public Deal, public Priceable, public Saveable
    { /* . . . */ };
```

Теперь нам нужно добавить возможность «обработать» сделку, имея только указатель на базовый класс `Deal`. Наивный подход заключается в том, чтобы начать задавать прямые вопросы о типе объектов, но это ничем не лучше первой попытки реализации функции увольнения сотрудников `terminate` (см. «Совет 98»):

```
void processDeal( Deal *d ) {
    d->validate();
    if( Bond *b = dynamic_cast<Bond *>(d) )
        b->price();
    else if( Swap *s = dynamic_cast<Swap *>(d) ) {
        s->price();
        s->save();
    }
    else
        throw UnknownDealType( d );
}
```

Другой, к сожалению, не менее популярный подход – спрашивать у объекта, не кто он, а что он умеет делать. Это называется «опросом возможностей»:

```
void processDeal( Deal *d ) {
    d->validate();
    if( Priceable *p = dynamic_cast<Priceable *>(d) )
        p->price();
    if( Saveable *s = dynamic_cast<Saveable *>(d) )
        s->save();
}
```

Каждый базовый класс определяет некоторый набор возможностей. Приведение с помощью `dynamic_cast` поперек иерархии или «поперечное приведение» эквивалентно заданию вопроса о том, может ли объект выполнить конкретную функцию или некоторый набор функций (рис. 9.15). Вторую версию `processDeal` можно выразить словами так: «`Deal`, проверь свою корректность. Если для тебя можно вычислить цену, оцени себя. Если тебя можно сберечь, сбереги себя.».

Этот подход немного изощреннее предыдущего. Пожалуй, он даже более стабилен, так как может адаптироваться к новым типам сделок, не возбуждая исключений. Но вместе с тем он недостаточно эффективен и сложен для сопровождения. Взгляните, что произойдет, если в иерархии `Deal` появится новый интерфейсный класс (рис. 9.16).

Появление в иерархии новой возможности не обнаруживается. Программа даже не догадывается спросить, законна ли сделка (впрочем, это довольно реалистичный анализ предметной области). Как и решение задачи об увольнении служащего, подход на основе опроса возможностей – это лишь временный выход, а не основа для построения архитектуры.

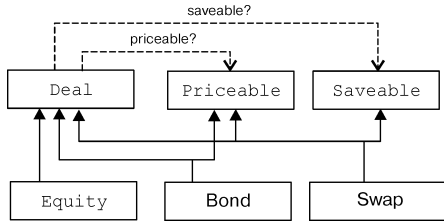


Рис. 9.15. Использование поперечного приведения для реализации опроса возможностей

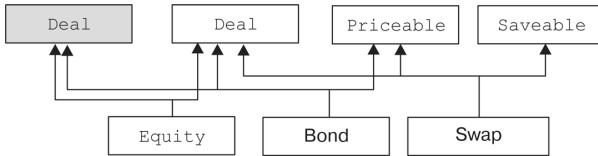


Рис. 9.16. Хрупкость методики опроса возможностей. Что если мы забудем задать нужный вопрос?

Корень зла и в случае запросов о типе, и в случае опроса возможностей в том, что некоторый важный аспект поведения объекта определяется внешним по отношению к самому объекту образом. Этот подход противоречит принципу абстрагирования данных, возможно, самому фундаментальному из всех принципов объектно-ориентированного программирования. Семантика абстрактного типа данных теперь уже не инкапсулирована в классе, где он реализован, а расплзлась по всей программе.

Как и в случае иерархии `Employee`, самый безопасный и эффективный способ добавить некоторую возможность в иерархию `Deal` одновременно является и самым простым:

```

class Deal {
public:
    virtual void validate() = 0;
    virtual void process() = 0;
    // . . .
};
class Bond : public Deal, public Priceable {
public:
    void validate();
    void price();
    void process() {
        validate();
        price();
    }
};
class Swap : public Deal, public Priceable, public Saveable {

```

```
public:
    void validate();
    void price();
    void save();
    void process() {
        validate();
        price();
        save();
    }
};
// и так далее ...
```

Есть и другие способы отказаться от опроса возможностей без модификации иерархии, если только их поддержка заложена в исходном проекте. Паттерн **Visitor** (Посетитель) позволяет добавлять в иерархию новые возможности, но он хрупок по отношению к изменениям иерархии. Паттерн **Acyclic Visitor** (Ациклический Посетитель) более устойчив, но требует (одного) запроса о возможностях, который может завершиться неудачно. Как бы то ни было, любое из этих решений лучше систематического применения опроса возможностей.

Вообще говоря, если возникает необходимость в опросе возможностей, значит, проект неудачен. Предпочтень следует простую, эффективную, безопасную виртуальную функцию, которая всегда завершается успешно.

Служащий приходит в отдел кадров, чтобы сообщить о первом выходе на работу. Его отправляют в конец длинной очереди прочих служащих. Когда, наконец, до него доходит черед, ему говорят: «За работу!» Поскольку его приняли уборщиком, то он берет тряпку и остаток дня моет полы.



Список литературы

- Andrei Alexandrescu. Modern C++ Design, Addison-Wesley, 2001.
- Association for Computing Machinery. ACM Code of Ethics and Professional Conduct, www.acm.org/constitution/code.html.
- . Software Engineering Code of Ethics and Professional Practice, www.acm.org/serving/se/code.htm.
- Marshall P. Cline, Greg A. Lomow, and Mike Girou. C++ FAQs, Second Edition, Addison-Wesley, 1999.
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns, Addison-Wesley, 1995.
- Nicolai Josuttis. The C++ Standard Library, Addison-Wesley, 1999.
- Robert Martin. Agile Software Development, 2nd ed., Prentice Hall, 2003.
- Scott Meyers. Effective C++, 2nd ed. Addison-Wesley, 1998.
- Scott Meyers. Effective STL, Addison-Wesley, 2001.
- Scott Meyers. More Effective C++, Addison-Wesley, 1996.
- Stephen C. Dewhurst and Kathy T. Stark. Programming in C++, 2nd ed., Prentice-Hall, 1995.
- William Strunk and E. B. White. The Elements of Style, 3d ed., Macmillan, 1979.
- Herb Sutter. More Exceptional C++, Addison-Wesley, 2002.
- E. B. White. Writings from The New Yorker, HarperCollins, 1990.

Предметный указатель

#

- #define, и пространства имен, 63
- #define, определение литералов, 63
- #define, определение
 - псевдофункций, 65
- #if, директива
 - выбор реализации класса, 70
 - на практике, 71
 - независимость от платформы, 69
 - переносимость, 69
 - применение для отладки, 67

&

- && (логический оператор), 47

,

- , (оператор запятая), 47

?

?

- (условный оператор), 47

[

- [] (выделение и освобождение памяти для массивов), 43, 145

|

- || (логический оператор), 47

<

- <<< (сержантский оператор), 54

>

- > (оператор стрелка), 61

A

- Acyclic Visitor, паттерн, 252
- assert, макрос, 72
- auto_ptr, шаблон, 38, 166

B

- Bridge, паттерн, 32

C

- Catch-обработчики, порядок, 158
- Command, паттерн, 233
- Composite, паттерн, 233, 235
- const, квалификатор типа
 - для ссылок, 24
 - перестановка, 56
- const_cast, оператор, 95, 103, 206, 210
- cv-квалификаторы. См. const, квалификатор типа; volatile, квалификатор типа

D

- Decorator, паттерн, 179, 233
- delete [], оператор, 145
 - область видимости и активация, 151
 - подмена, 149
- dynamic_cast, оператор
 - для указателя на подобъект виртуального базового класса, 125
 - задание интимных вопросов о типе, 247
 - и преобразования, 105
 - инициализация виртуального базового класса по умолчанию, 129
 - как предпочтительная альтернатива static_cast, 85
 - неоднозначность, 105
 - опрос возможностей, 250

F

- Factory Method, паттерн, 192, 228
- for, предложение
 - ограничение области видимости переменной, 51
 - сравнение с while, 53
- free и delete, 145

L

lvalue

- возврат из функции, 24
- как результат вычисления
условного оператора, 27
- неизменяемое, 26, 64
- определение, 26
- связывание ссылки с, 24, 102

M

- malloc и new, 145
- Monostate, паттерн, 173

N

- NDEBUG, таинственные ошибки, 67
- Null Object, паттерн, 234, 235, 243

O

- operator delete, 175
- operator new
 - выделение памяти для скаляров, 144
 - область видимости и активация, 151
 - ошибки при выделении памяти, 148
 - подмена, 149

P

- POD-классы (добрые старые
данные), 121
- Prototupe, паттерн, 188, 192, 234
- Proxy, паттерн, 242

R

- reinterpret_cast, 75, 93, 129

S

- set/get, интерфейсы, 202
- Singleton, паттерн, 21, 133, 173
- static_cast, 95, 231
- Strategy, паттерн, 241
- switch, предложение, 29

T

- Template Method, паттерн, 180

V

- vector, сравнение с массивом, 44,
145, 167
- Visitor, паттерн, 182, 191, 252
- void *, 74
- volatile, квалификатор типа
 - перестановка, 56
 - ссылки, 24
- vptr (указатель на vtbl), 193
- vtbl (таблица виртуальных функций), 194

W

- while, предложение, сравнение с for, 53

A

- Абстрагирование данных
 - для типов исключений, 153
 - цель, 202
- Автоинициализация, 57
- Агрегирование и использование, 211
- Адреса
 - арифметические ошибки, 75, 85,
94, 237
 - возвращенного не-lvalue, 222
 - подобъектов базовых классов, 175
 - функций-членов. См. Указатели,
на члены класса
- Акронимы, 36
- Акцессоры. См. Интерфейсы get/set
- Алгоритмы
 - переменная и постоянная частью, 180
- Анонимные временные объекты
 - время жизни, 101
 - инициализация ссылки на const, 102
 - инициализация формального
параметра-ссылки, 98
 - как объекты исключений, 156
 - как результат инициализации
копированием, 134
 - как результат постфиксных ++
и —, 221
 - передача функциональных
объектов по значению, 113
- Ассоциативность
 - и приоритеты, 48

проблемы, 50

Б

Базовый язык C++

- логические операторы, 27
- оператор взятия индекса, 28
- предложение switch, 29
- проваливание в switch, 29
- условный оператор, 28

Безымянное пространство имен, 59

Брандмауэр, 171

В

Взятия индекса оператор,
встроенный, 28

Видимость, сравнение
с доступностью, 30

Виртуальное присваивание, 186

Виртуальные статические
функции-члены, 174

Виртуальные функции
вызов из конструкторов
и деструкторов, 184
невиртуальный вызов, 178
перегрузка, 181
с аргументами по умолчанию, 182
чистые, вызов, 179

Виртуальный конструктор, идиома.
См. также Prototype, паттерн

Внешние типы, 59

Временные объекты, 101
преобразования, 102

Выражение throw, 155

Вырожденные иерархии, 237

Вычислительные конструкторы, 140

Г

Глобальные переменные, 20

Д

Декремент, оператор, 219

Дельта-арифметика
адреса объектов классов, 92
корректировка значения this
при вызове виртуальной

функции, 197

понижающее приведение, 85
приведение неполных типов, 94

Деструкторы

вызов виртуальных функций из, 184

Динамическое связывание, 169

Доминирование, 198

З

Заголовочные файлы
счетчик Шварца, 133

Захват ресурса как инициализация,
38, 162

И

Идиомы

auto_ptr, 38

pimpl. *См.* Паттерны, Bridge
виртуальный конструктор, 188
захват ресурса

как инициализация, 162
интеллектуальный указатель, 62, 234
операции копирования, 37
функциональный объект, 234

Иерархии классов.

См. Проектирование иерархий

Имена массивов и константные
указатели, 83

Индексирование
имени массива, 28
указателей, 28
целых указателями, 28

Инициализаторы

путаница с объявлением массивов, 43

Инициализация

автоинициализация, 57
аргументов по умолчанию
применение, 21
сравнение с перегрузкой, 22
аргументов, прямая, 136
базового класса в конструкторе
копирования, 129
виртуальных базовых классов, 125
и присваивание, 112, 122
область видимости переменных, 115
операции копирования, 117

- оптимизация возвращаемого значения, 138
- паттерн Singleton, 21
- передача аргументов, 113
- побитовое копирование объектов классов, 120
- порядок членов в списке инициализации, 124
- прямая, сравнение с копированием, 134
- ссылка, 102
- статических данных во время выполнения, порядок, 132
- статических членов в конструкторе, 142
- формальных аргументов временными объектами, 98
- Инкремент, оператор, 219
- Интеллектуальный указатель, идиома, 62, 234
- Интерфейсные классы, 127, 233, 235
- Интерфейсы get/set, 202
- Инфиксная нотация, 60, 215
- История изменений, 17
- К**
- Квалификаторы типов
 - const
 - перестановка, 56
 - ссылки, 24
 - volatile
 - перестановка, 56
 - ссылки, 24
- Квалификаторы, преобразование, 80
- Классы
 - POD (добрые старые данные), 121
 - интерфейсные, 127
 - ограничение доступа, 30
 - чисто виртуальный базовый класс, 34
- Ковариантные возвращаемые типы, 192
- Кодирование, краткость, 16
- Коды типов, 169, 241
- Комментарии. См. также
 - Сопровождение; Удобочитаемость и сопровождение, 16
 - излишние, 16
 - о проваливании, 29
 - самодокументированный код, 16
- Конкретные открытые базовые классы, 236
- Константное выражение, 68
- Константные данные-члены, 205
- Константные объекты и литералы, 26
- Константные указатели и имена объектов, 84
- и указатели на const, 79
- определение, 55
- преобразования, повышающие степень константности, 79
- Константные функции-члена механизм работы, 208
- отбрасывание const, 210
- семантика, 209
- синтаксис, 207
- Константы
 - и литералы, 20
 - присваивание, 206
- Конструкторы
 - вызов виртуальных функций из, 184
 - вычислительные, 140
 - и преобразования, 89
 - идиома виртуального конструктора, 188
 - инициализация статических членов, 142
 - инициализация и присваивание, 122
 - реализация с помощью шаблонных функций-членов, 222
- Контейнеры, содержащие указатели, 213
- Контравариантность, 109
- Космические иерархии, 243
- Л**
- Лексический анализ, 54
- Литералы
 - и константные объекты, 26
 - и константы, 19
 - определение с помощью #define, 63
- Логические операторы, 27, 47

Локальные объекты
время жизни, 160
затиранье статических
переменных, 159
идиоматические трудности, 160
исчезающие фреймы стека, 158

М

Магические числа, 18
Макросы
побочные эффекты, 28, 65
Максимальный кусок
определение, 54
примеры, 39, 54
Массивы
и векторы, 145
и синтаксис инициализатора, 43
массивов, 56, 84
объектов класса, 225
освобождение памяти, 145
перестановка квалификаторов
типа, 56
преобразование указателя
на многомерный массив, 83
ссылки на, 25
Мудрствование, излишнее, 39

Н

Наследование
и проектирование иерархий, 233, 238
ограничение доступа, 34
Невиртуальный деструктор базового
класса адреса подобъектов базовых
классов, 175
и виртуальный статические
функции-члены, 174
исключения из правил, 176
неопределенное поведение, 173
Неполные типы
для уменьшения числа
зависимостей, 31
приведение, 93
Неявные преобразования
и ссылки, 102
из производного класса в открытый
базовый, 110

контравариантность, 109
неоднозначность результата, 85
при инициализации формальных
аргументов, 98
с помощью конструктора, 89

Нуль

нулевые ссылки, 24
нулевые указатели, 35
результат `dynamic_cast`, 107

О

Область видимости
литералов, определенных
с помощью `#define`, 63
локальная, проблемы, 160
переменных, ограничение, 51
Обработка исключений, 148, 152, 165
Объекты классов, побитовое
копирование, 120
Ограничение доступа
и абстрагирование данных, 202
и видимость, 30
наследование, 34, 230
описание, 30
паттерн `Bridge`, 32
Операторы
&& (логический), 47
, (запятая), 47
?
(условный), 47
|| (логический), 47
<<<< (сержантский), 54
>> (стрелка), 61
в базовом языке C++, 27
взятия индекса, встроенный, 28
логические, 27
поиск операторной функции, 59
порядок вычисления, 45
преобразования типов, 35
приведения типов, 35
размещающий `new`, 46
Операторы преобразования
альтернативы, 89
назначение, 87
неоднозначные, 86

- Операторы преобразования и приведения, 34
- Операции копирования
 - запрет, 119
 - идиома, 37
 - инициализация, 117
 - шаблонные, 222
- Опережающее объявление класса.
 - См. Неполное объявление
- Опрос возможностей, 249
- Оптимизация возвращаемого значения, 138
- Оптимизация именованного возвращаемого значения, 141
- Освобождение
 - выделенных из кучи объектов, 166
 - скаляров и массивов, 144
- Отбрасывание const, 210
- Открытое наследование, 233
- Отладка
 - директива #if, 67
 - недостижимый код, 68
- П**
- Параллельные иерархии, 227
- Паттерны
 - Acyclic Visitor, 252
 - Bridge, 32
 - Command, 233
 - Composite, 233, 235
 - Decorator, 179, 233
 - Factory Method, 192, 228
 - Monostate, 173
 - Null Object, 234, 235, 243
 - Prototype, 188, 192, 234
 - Proxy, 242
 - Singleton, 21, 133, 173
 - Strategy, 241
 - Template Method, 180
 - Visitor, 182, 191, 252
- Перегрузка
 - виртуальных функций, 181
 - и инициализация аргументов по умолчанию, 22
 - инфиксная нотация, 60
 - оператора ->, 61
 - оператора взятия индекса, 28
 - операторов, 215
 - операторов инкремента и декремента, 219
 - поиск операторной функции, 59
 - порядок вычисления, 48
 - сравнение с переопределением и сокрытием, 188
- Передача аргументов, 113
- Перекомпиляция, как избежать, 32, 235
- Переменные
 - исключение типа в имени, 34
 - ограничение области видимости, 51
- Переносимость
 - директива #if, 69
 - нулевые указатели, 35
- Переопределение
 - механизм, 193
 - невидимых функций, 191
 - определение, 195
 - сравнение с перегрузкой и сокрытием, 188
- Перехват
 - исключений, 156
 - строковых литералов, 153
- Переходник (thunk), 197
- Перечисляемые константы
 - и литералы, определенные с помощью #define, 64
 - инициализация статических членов, 142
 - магические числа, 19
 - точка объявления, 58
- Платформенная независимость литералы и константы, 19
- преобразования, 74
- Повторное использование
 - глобальных переменных, 20
 - кода, 258
 - прозрачный ящик.
 - См. Наследование
- Повторное использование кода, 233
- Подвыражения, порядок вычисления, 45
- Подобъект базового класса
 - адрес, 175

- инициализация, 125, 129, 185
- Подстановка контейнеров, 227
- Подсчет ссылок, 215
- Полиморфизм
 - алгоритм с переменной и постоянной частью, 180
 - брандмауэр, 171
 - ветвление по коду типа, 170
 - виртуальные функции
 - вызов из конструкторов и деструкторов, 184
 - невиртуальный вызов, 178
 - перегрузка, 181
 - с аргументами по умолчанию, 182
 - виртуальный конструктор копирования, 188
 - гибкость шаблонных методов, 180
 - динамическое связывание, 169
 - доминирование, 198
 - кодирование типов, 169
 - связи между компонентами, 171
 - сокрытие
 - виртуальной присваивание, 186
 - невиртуальный деструктор базового класса адреса
 - подобъектов базовых классов, 175
 - виртуальные статические функции-члены, 174
 - исключения из правил, 176
 - неопределенное поведение, 173
 - перегрузка, 188
 - виртуальных функций, 181
 - сокрытие неvirtуальных функций, 177
- Понижающее приведение, 85
- Поперечное приведение, 250
- Порядок вычислений. См. Приоритеты
- Преобразование
 - `reinterpret_cast`, 75, 94
 - временные объекты, 101
 - время жизни временных объектов, 101
 - дельта-арифметика
 - для адресов объектов классов, 92
 - корректировка значения `this` при вызове виртуальной функции, 197
 - понижающие приведения, 85
 - приведение неполных типов, 94
 - зависимость от платформы, 74
 - имени массива в указатель, 83
 - инициализация
 - ссылок, 102
 - формальных аргументов, 98
 - использование функций вместо, 86
 - квалификаторов, 80
 - константные указатели и имена массивов, 84
 - контравариантность, 109
 - неявное
 - для инициализации формальных аргументов, 98
 - из производного класса
 - в открытый базовый, 110
 - контравариантность, 109
 - неоднозначность, 85
 - с помощью конструкторов, 89
 - оператор `const_cast`, 95, 103, 206, 210
 - оператор `dynamic_cast`
 - для задания интимных вопросов о типе, 247
 - для опроса возможностей, 250
 - для приведения к указателю на виртуальный базовый класс, 129
 - как предпочтительная альтернатива `static_cast`, 85, 125
 - неоднозначность, 107
 - повышающие степень константности, 79
 - понижающие приведения, 85
 - посредством `void *`, 74
 - срезка объектов производного класса, 77
 - статические приведения, 95
 - указателей, 80
 - на неполные типы, 93
 - на указатели на производный класс, 83

- на члены класса, 110
- указателя на многомерный массив, 83
- формальные аргументы
- инициализация временными объектами, 98
- передача по значению, 100
- передача по ссылке, 100
- Препроцессор
 - NDEBUG, таинственные ошибки, 67
 - выбор реализации класса с помощью #if, 70
 - константные выражения, 68
 - макрос assert, 71
 - область видимости литералов, 63
 - определение литералов с помощью #define, 63
 - отладка, 67
 - отладочные версии, 67
 - псевдофункции, 65
 - утверждения, побочные эффекты, 71
- Приведение. См. также void *
 - reinterpret_cast, 75, 93
 - в случае множественного наследования, 92
 - в старом стиле, 94, 231
 - нединамическое. См. Статические приведения
 - неполных типов, 93
 - проблемы в ходе сопровождения, 81, 95, 96
 - статическое, 95
 - указателя на базовый класс к указателю на производный класс. См. Понижающее приведение
- Приоритеты операторов, 217
 - && (логический оператор), 47
 - , (оператор запятой), 47
 - ?
 - (условный оператор), 47
 - || (логический оператор), 47
 - и ассоциативность, 48, 50
 - обзор, 44
 - оператор взятия индекса, 28
 - оператор указания, 50
 - перегрузка операторов, 48
 - размещающий оператор new, 46
 - уровни приоритетов, 49
 - фиксированный порядок вычислений, 47
- Присваивание и инициализация, 112, 122
- Присоединяемые классы, 233
- Проваливание, 29
- Проектирование иерархий
 - арифметические ошибки при вычислении адреса, 237
 - ветвление по кодам типов, 241
 - вырожденные иерархии, 237
 - запросы о типе во время выполнения, 246
 - защищенный доступ, 230
 - интерфейсные классы, 233
 - классы-протоколы, 233
 - конкретные открытые базовые классы, 236
 - космические иерархии, 243
 - массивы объектов класса, 225
 - наследование, 238
 - опрос возможностей, 249
 - открытое наследование, 233
 - повторное использование кода, 233
 - подстановка контейнеров, 227
 - семантика значения, 237
 - срезка, 237
 - управление на основе типов, 241
- Проектирование классов
 - агрегирование и использование, 211
 - интерфейсы get/set, 202
 - константные данные-члены, 205
 - константные функции-члены механизма работы, 208
 - отбрасывание const, 210
 - семантика, 209
 - синтаксис, 207
 - оператор декремента, 219
 - оператор инкремента, 219

- операторы-члены и друзья класса, 218
- перегрузка операторов, 215
- приоритеты операторов, 217
- ссылочные данные-члены, 205
- шаблонные операции
 - копирования, 222
- Пространство имен
 - безымянное, 59
 - и #define, 63
- Прямая инициализация аргументов, 136
- Псевдонимы
 - и отношение агрегирования / использования, 211
 - ссылки как, 24, 102
- Псевдофункции, определение, 65
- Р**
- Размещающий оператор new
 - вызов конструктора, 113, 122, 136
 - подмена глобальных new и delete, 149
 - порядок вычисления аргументов, 46
- Реализация класса, выбор с помощью #if, 70
- С**
- Связи между компонентами
 - глобальные переменные, 20
 - обход защиты доступа, 232
 - полиморфизм, 171
- Связывание
 - динамическое, 169
 - ссылки с lvalue, 24, 102
 - ссылки с функцией, 25
- Семантика значения, 237
- Сержантский оператор, 54
- Синтаксис
 - const, квалификатор типа, перестановка, 56
 - volatile, квалификатор типа, перестановка, 56
 - ассоциативность и предшествование, 48
 - проблемы, 50
 - внешние типы, 59
 - выделение лексем, 54
 - инициализаторы
 - не путать с массивами, 43
 - инфиксная нотация, 60
 - конкретизация шаблонов, 54
 - константные указатели, 55
 - константные функции-члены, 207
 - лексический анализ, 54
 - максимальный кусок, 54
 - массивы
 - не путать с инициализаторами, 43
 - перестановка квалификаторов типа, 56
 - ограничение области видимости переменных, 51
 - перегрузка
 - инфиксная нотация, 60
 - оператора ->, 61
 - операторов, 59
 - предшествование
 - автоинициализация, 57
 - и ассоциативность, 48, 50
 - оператор указания, 50
 - уровни приоритетов, 49
 - перестановка квалификаторов типов, 56
 - поиск операторной функции, 59
 - порядок вычислений
 - && (логический оператор), 47
 - , (оператор запятая), 47
 - ? (условный оператор), 47
 - || (логический оператор), 47
 - обзор, 44
 - перегрузка операторов, 48
 - подвыражений, 45
 - размещающий оператор new, 46
 - порядок
 - вычислений фиксированный, 47
 - предложение for
 - ограничение области видимости переменной, 51
 - сравнение с while, 53
 - спецификатор компоновки, 59
 - спецификаторы в объявлениях, порядок следования, 55

- статические типы, 59
- функция или объект, 56
- Соглашения об именовании
 - кодирование типа переменной в ее имени, 34
 - мнемонические имена, 18
 - ограничение доступа, 34
 - описание отношений владения, 212
 - простота, 34
 - самодокументированный код, 18
- Скрытие
 - невиртуальных функций, 177
 - сравнение с перегрузкой и переопределением, 188
- Скрытие данных. *См.* Ограничение доступа
- Сопровождение. *См.* также Комментарии; Удобочитаемость и приведения типов, 75, 81, 95, 96
- способы облегчить
 - взаимозаменяемые контейнеры, 228
- владение в контейнерах, 213
- выделение памяти для скаляров и массивов, 146
- идиомы, 37
- инициализация статических членов, 143
- коды типов, 171
- мнемонические имена, 18
- неполные объявления, 31
- порядок квалификаторов в объявлениях, 56
- предложение `for`, 51, 53
- приоритеты операторов, 49
- проваливание, 29
- соглашения об именовании, 17
- стандарты кодирования, 41, 47
- управление на основе типов, 242
- утверждения, 72
- Спецификаторы объявлений
 - порядок следования, 55
- Список инициализации членов,
 - порядок следования, 124

- Срезка
 - и проектирование иерархий, 237
 - объектов производных классов, 77

- Ссылки
 - возврат из функции, 24
 - и указатели, 23
 - инициализация, 102
 - как псевдонимы, 24, 102
 - квалификаторы `const` и `volatile`, 24
 - на локальные переменные, 158
 - на массивы, 25
 - на неполные типы, 93
 - нулевые, 24
 - приведение типа, 25
 - привязка к функции, 25
- Статические переменные,
 - инициализация во время выполнения, 132
- Статические типы, 59
- Строковые литералы, как объекты исключений, 152
- Счетчик Шварца, 133

T

- Темные закоулки языка C++
 - адрес возвращенного `no-lvalue`, 222
 - вызов чисто виртуальной функции, 179
 - гарантии относительно `reinterpret_cast`, 93
 - генерируемый компилятором оператор присваивания
 - подобъектам виртуального базового класса, 127
 - индексирование целого, 28
 - копирование строкового литерала во временный объект
 - при употреблении в выражении `throw`, 155
 - неприменимость квалификаторов к ссылкам, 24
 - область видимости при вызове функции-члена `operator delete`, 175
 - переопределение невидимых функций, 191

- применение квалификатора
 - к typedef функции, 57
- результат условного оператора — lvalue, 27
- структура предложения switch, 29
- точка объявления перечисляемой константы, 58
- Типы исключений, 153
- У**
- Удобочитаемость. См. Комментарии;
- Сопровождение форматирование кода, 39
- Указатели
 - владение, 213
 - возбуждение исключений, 155
 - и ссылки, 23
 - контейнеры, содержащие указатели, 213
 - на локальные переменные, 158
 - на неполные типы, 93
 - на указатель на производный класс, 83
 - на функции, 25
 - на члены класса, 23, 109
 - преобразование, 80
 - приоритеты операторов, 50
- Улыбка Чеширского кота. См. Bridge, паттерн
- Управление памятью и ресурсами
 - auto_ptr, 166
 - возбуждение исключений
 - в виде указателей, 156
 - временные объекты в качестве объектов исключений, 156
 - захват ресурса есть
 - инициализация, 162
 - исправление ошибки путем добавления static, 161
 - область видимости и активация new и delete, 151
 - обработка исключений, 152
 - освобождение памяти
 - для выделенных из кучи объектов, 166
 - для скаляров и массивов, 144
 - ошибки при выделении памяти, 147
 - перехват исключений, 156
 - перехват строковых литералов, 153
 - подмена глобальных new и delete, 149
 - порядок catch-обработчиков, 158
 - типы исключений, 153
 - утечки памяти, 160
- Утверждения, побочные эффекты, 71
- Утечки памяти, 160, 211
- Ф**
- Формальные аргументы
 - инициализация временными объектами, 98
 - описание прав владения, 212
 - передача по значению, 100
 - передача по ссылке, 100
- Функции
 - возврат ссылки на значение, 24
 - для преобразований, 86
 - невидимые, переопределение, 191
 - связывание ссылки с, 25
 - ссылки на, 25
 - указатели на, 25
- Функции преобразования, явные, 86
- Функции-члены
 - виртуальные статические, 174
 - шаблонные, 39
- Функциональный объект, идиома, 234
- Ч**
- Чисто виртуальные функции,
 - вызов, 179
- Члены класса
 - нуждающиеся в инициализации, 122
 - указатели на, 23, 109
- Ш**
- Шаблонные методы, гибкость, 180
- Шаблонные операции копирования, 222
- Э**
- Этика программиста, 41

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.**

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.aliants-kniga.ru**.

Оптовые закупки: тел. +7 (499) 782-38-89.

Электронный адрес: **books@aliants-kniga.ru**.

Стефан К. Дьюхэрст

Скользкие места С++

**Как избежать проблем
при проектировании и кодировании**

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Литературный редактор *Готлиб О. В.*

Верстка *Страмоусова О. И.*

Дизайн обложки *Мовчан А. Г.*

Формат 60×90^{1/16}.

Гарнитура «Петербург». Печать офсетная.

Усл. печ. л. 18,62. Тираж 100 экз.

Web-сайт издательства: www.dmkpress.com