Содержание

Благодарности 11

Предисловие 15

Введение 17

Терминология 18
Соглашения об именах 23
Многопоточность 24
Библиотеки TR1 и Boost 24

Глава 1. Приучайтесь к C++ 26

Правило 1: Относитесь к C++ как к конгломерату языков 26
Правило 2: Предпочитайте const, enum и inline использованию #define 28
Правило 3: Везде, где только можно, используйте const 31
Константные функции-члены 33
Как избежать дублирования в константных и неконстантных функциях-членах 37
Правило 4: Прежде чем использовать объекты, убедитесь, что они инициализированы 40

Глава 2. Конструкторы, деструкторы и операторы присваивания 47

Правило 5: Какие функции C++ создает и вызывает молча 47
Правило 6: Явно запрещайте компилфятору генерировать функции, которые вам не нужны 50
Правило 7: Объявляйте деструкторы виртуальными в полиморфном базовом классе 52
Правило 8: Не позволяйте исключениям покидать деструкторы 56
Правило 9: Никогда не вызывайте виртуальные функции в конструкторе или деструкторе 60

Правило 10: Операторы присваивания должны возвращать ссылку на *this 64
Правило 11: В operator= осуществляйте проверку на присваивание самому себе 65
Правило 12: Копируйте все части объекта 68

Глава 3. Управление ресурсами 72

Правило 13: Используйте объекты для управления ресурсами 72
Правило 14: Тщательно продумывайте поведение при копировании классов, управляющих ресурсами 76
Правило 15: Предоставляйте доступ к самим ресурсам из управляющих ими классов 79
Правило 16: Используйте одинаковые формы new и delete 83
Правило 17: Помещение в «интеллектуальный» указатель объекта, выделенного с помощью new, лучше располагать в отдельном предложении 85

Глава 4. Проектирование программ и объявления 87

Правило 18: Проектируйте интерфейсы так, что их легко было использовать правильно и трудно – неправильно 87
Правило 19: Рассматривайте проектирование класса как проектирование типа 92
Правило 20: Предпочитайте передачу по ссылке на const передаче по значению 94
Правило 21: Не пытайтесь вернуть ссылку, когда должны вернуть объект 98
Правило 22: Объявляйте данные-члены закрытыми 102
Правило 23: Предпочитайте функциям-членам функции, не являющиеся ни членами, ни друзьями класса 105
Правило 24: Объявляйте функции, не являющиеся членами, когда преобразование типов должно быть применимо ко всем параметрам 109
Правило 25: Подумайте о поддержке функции swap, не возбуждающей исключений 112

Глава 5. Реализация 119

Правило 26: Откладывайте определение переменных насколько возможно 119
Правило 27: Не злоупотребляйте приведением типов 122

Правило 28: Избегайте возвращения «дескрипторов» внутренних данных 128
Правило 29: Стремитесь, чтобы программа была безопасна относительно исключений 132
Правило 30: Тщательно обдумывайте использование встроенных функций 139
Правило 31: Уменьшайте зависимости файлов при компиляции 144

Глава 6. Наследование и объектно-ориентированное проектирование 153

Правило 32: Используйте открытое наследование для моделирования отношения «является» 153
Правило 33: Не скрывайте унаследованные имена 159
Правило 34: Различайте наследование интерфейса и наследование реализации 164
Правило 35: Рассмотрите альтернативы виртуальным функциям 171
Реализация паттерна «Шаблонный метод» с помощью идиомы невиртуального интерфейса 172
Реализация паттерна «Стратегия» посредством указателей на функции 173
Реализация паттерна «Стратегия» посредством класса tr::function .. 175
«Классический» паттерн «Стратегия» 177
Резюме 178
Правило 36: Никогда не переопределяйте наследуемые невиртуальные функции 179
Правило 37: Никогда не переопределяйте наследуемое значение аргумента функции по умолчанию 181
Правило 38: Моделируйте отношение «содержит» или «реализуется посредством» с помощью композиции 185
Правило 39: Продумывайте подход к использованию закрытого наследования 188
Правило 40: Продумывайте подход к использованию множественного наследования 193

Глава 7. Шаблоны и обобщенное программирование 200

Правило 41: Разберитесь в том, что такое неявные интерфейсы и полиморфизм на этапе компиляции 200

Правило 42: Усвойте оба значения ключевого слова typename ... 204
Правило 43: Необходимо знать, как обращаться к именам в шаблонных базовых классах ... 207
Правило 44: Размещайте независимый от параметров код вне шаблонов ... 212
Правило 45: Разрабатывайте шаблоны функций-членов так, чтобы они принимали «все совместимые типы» ... 217
Правило 46: Определяйте внутри шаблонов функции, не являющиеся членами, когда желательны преобразования типа ... 221
Правило 47: Используйте классы-характеристики для предоставления информации о типах ... 225
Правило 48: Изучите метапрограммирование шаблонов ... 231

Глава 8. Настройка new и delete ... 237

Правило 49: Разберитесь в поведении обработчика new ... 238
Правило 50: Когда имеет смысл заменять new и delete ... 244
Правило 51: Придерживайтесь принятых соглашений при написании new и delete ... 249
Правило 52: Если вы написали оператор new с размещением, напишите и соответствующий оператор delete ... 252

Глава 9. Разное ... 258

Правило 53: Обращайте внимание на предупреждения компилятора ... 258
Правило 54: Ознакомьтесь со стандартной библиотекой, включая TR1 ... 259
Правило 55: Познакомьтесь с Boost ... 264

Приложение A. За пределами «Эффективного использования C++» ... 268

Приложение B. Соответствие правил во втором и третьем изданиях ... 272

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 (Hend-

[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.).

rik 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 большин-

стве 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 выраже-

ния, 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,