Это блестящая книга, проясняющая многие аспекты языка C++, начиная с редко используемых его свойств и заканчивая разделами, которые программисты считают простыми и недвусмысленными. Только глубоко понимая, каким образом компилятор C++ обрабатывает исходные тексты программ, можно надеяться на создание надежного программного обеспечения. Эта книга является бесценным источником такого понимания. Прочитав книгу, я как будто вместе с большим специалистом по C++ отредактировал огромное количество исходных текстов и получил от него массу очень ценных наставлений.

Фред Вайлд (Fred Wild), вице-президент по технологиям,
Advantage Software Technologies

Эта книга описывает множество важных приемов, позволяющих писать эффективные программы на C++. В ней объясняется, как придумывать и реализовывать идеи и как не попасться впросак, используя ту или иную архитектуру программы. В книге также подробнейше рассматриваются новые свойства, недавно добавленные к C++. Любой программист, желающий использовать эти свойства, обязательно захочет иметь под рукой такую книгу.

Кристофер Дж. Ван Вык (Christopher J. Van Wyk),
профессор, подразделение математики
и компьютерных наук, Университет Дрю

В пособии представлены возможности промышленного применения языка C++ в лучшем смысле этого слова. Превосходная книга для тех, кто читал предыдущую – «Эффективное использование C++».

Эрик Наглер (Eric Nagler), преподаватель и автор книг,
Калифорнийский университет, отделение в Санта Круз

«Наиболее эффективное использование C++» – ценное продолжение первой книги Скотта «Эффективное использование C++». Я считаю, что каждый профессиональный разработчик на C++ должен прочесть и постоянно держать в памяти советы из этих двух книг. Все они, по моему мнению, касаются очень важных, но плохо понимаемых аспектов языка. Я настоятельно рекомендую эту книгу, также как и предыдущую, разработчикам, бета-тестерам и руководителям проектов; глубокие знания автора и превосходный стиль изложения делают ее полезной для всех.

Стив Беркетт (Steve Burkett),
консультант по программному обеспечению

Содержание

Благодарности 9
Источники идей 9
Об этой книге 11
Люди, которые мне помогали 13

Введение 14

Глава 1. Основы 23
Правило 1. Различайте указатели и ссылки 23
Правило 2. Предпочитайте приведение типов в стиле C++ 25
Правило 3. Никогда не используйте полиморфизм в массивах 30
Правило 4. Избегайте неоправданного использования конструкторов по умолчанию 33

Глава 2. Операторы 38
Правило 5. Опасайтесь определяемых пользователем функций преобразования типа 38
Правило 6. Различайте префиксную и постфиксную формы операторов инкремента и декремента 45
Правило 7. Никогда не перегружайте операторы &&, || и , ... 48
Правило 8. Различайте значение операторов new и delete 51

Глава 3. Исключения 57
Правило 9. Чтобы избежать утечки ресурсов, используйте деструкторы 58
Правило 10. Не допускайте утечки ресурсов в конструкторах 63
Правило 11. Не распространяйте обработку исключений за пределы деструктора 71
Правило 12. Отличайте генерацию исключения от передачи параметра или вызова виртуальной функции 73
Правило 13. Перехватывайте исключения, передаваемые по ссылке 80
Правило 14. Разумно используйте спецификации исключений 84
Правило 15. Оценивайте затраты на обработку исключений 90

Глава 4. Эффективность 94

Правило 16. Не забывайте о правиле «80–20» 95
Правило 17. Используйте отложенные вычисления 97
Правило 18. Снижайте затраты на ожидаемые вычисления 106
Правило 19. Изучите причины возникновения временных объектов 110
Правило 20. Облегчайте оптимизацию возвращаемого значения 113
Правило 21. Используйте перегрузку, чтобы избежать неявного преобразования типов 116
Правило 22. По возможности применяйте оператор присваивания вместо отдельного оператора 118
Правило 23. Используйте разные библиотеки 121
Правило 24. Учитывайте затраты, связанные с виртуальными функциями, множественным наследованием, виртуальными базовыми классами и RTTI 124

Глава 5. Приемы 134

Правило 25. Делайте виртуальными конструкторы и функции, не являющиеся членами класса 134
Правило 26. Ограничивайте число объектов в классе 140
Правило 27. В зависимости от ситуации требуйте или запрещайте размещать объекты в куче 154
Правило 28. Используйте интеллектуальные указатели 167
Правило 29. Используйте подсчет ссылок 190
Правило 30. Применяйте proxy-классы 218
Правило 31. Создавайте функции, виртуальные по отношению более чем к одному объекту 231

Глава 6. Разное 254

Правило 32. Программируйте, заглядывая в будущее 254
Правило 33. Делайте нетерминальные классы абстрактными 259
Правило 34. Умейте использовать в одной программе C и C++ 270
Правило 35. Ознакомьтесь со стандартом языка 276

Приложение 1. Список рекомендуемой литературы 284

Приложение 2. Реализация шаблона auto_ptr 289

Алфавитный указатель 293

Благодарности

В создании этой книги принимало участие множество людей. Одни предложили важные технические идеи, другие помогли подготовить ее к печати, а третьи просто скрашивали мою жизнь, пока я работал над ней.

Часто, когда количество людей, принимавших участие в работе над книгой, достаточно велико, появляется соблазн отказаться от перечисления участников проекта, ограничившись стандартной фразой «Список людей, работавших над книгой, слишком длинен, чтобы быть приведенным здесь». Я, однако, предпочитаю подход Джона Л. Хеннеси (John L. Hennessey) и Дэвида А. Петерсона (David A. Patterson) – см. «Компьютерные архитектуры: численный подход», изд. Морган Кауфман (Morgan Kaufman), 1-ое издание, 1990. Один из аргументов за включение полного списка благодарностей, приведенного ниже, – статистические данные для закона «80–20», на который я ссылаюсь в правиле 16.

Источники идей

За исключением прямого цитирования, весь текст этой книги принадлежит мне. Тем не менее, многие описанные в ней идеи были придуманы другими. Я всячески пытался отслеживать авторство нововведений, но мне все же пришлось включить информацию из источников, названия которых я уже не могу вспомнить, в основном это сообщения из конференций Usenet comp.lang.c++ и comp.std .c++.

Многие идеи в сообществе C++ зарождаются почти одновременно и совершенно независимо в головах многих людей. Ниже я указываю только, где услышал ту или иную мысль, что не всегда совпадает с тем, где она была озвучена впервые.

Брайан Керниган (Brian Kernighan) предложил использовать макроопределения для приближения к синтаксису новых операторов приведения типа, описанных в правиле 2.

Предупреждение по поводу удаления массива объектов производного класса с помощью указателя на базовый класс, изложенное в правиле 3, основано на материалах лекции Дэна Сакса (Dan Saks), прочитанной им на нескольких конференциях и торговых выставках.

Техника использования proxy-классов из правила 5, позволяющая избежать нежелательного вызова конструкторов с одним аргументом, основана на материалах колонки Эндрю Кенига (Andrew Koenig) в журнале C++ Report за январь 1994 года.

Джеймс Канце (James Kanze) прислал сообщение в comp.lang.c++ относительно реализации постфиксных декрементных и инкрементных операторов через соответствующие префиксные операторы. Этот прием рассматривается в правиле 6.

Дэвид Кок (David Cok), написав мне по одному вопросу, затронутому в «Эффективном использовании C++», привлек мое внимание к различию между `operator new` и оператором `new`, положенному в основу правила 8. Даже прочитав письмо, я не в полной мере осознал существующую разницу, но если бы не этот первый толчок, то, скорее всего, не понимал бы ее до сих пор.

Метод записи деструкторов, позволяющий избежать утечки ресурсов (см. правило 9), взят из раздела 15.3 книги Маргарет А. Эллис (Margaret A. Ellis) и Бьерна Страуструпа (Bjarne Stroustrup) The Annotated C++ Reference Manual. Там этот метод имеет название «Выделение ресурса – инициализация». Том Каргилл (Tom Cargill) предложил перенести акцент с выделения ресурсов на их освобождение.

Часть рассуждений в разделе, посвященном правилу 11, была навеяна содержимым главы 4 книги Taligent's Guide to Designing Programs, изд. Addison-Wesley, 1994.

Описание предварительного выделения памяти для класса `DynArray` в правиле 18 основано на статье Тома Каргилла «Динамический вектор сложнее, чем кажется», опубликованной в журнале C++ Report за июнь 1992 года. Информацию о более сложной архитектуре для класса динамического массива можно найти в заметке того же автора (номер C++ Report за январь 1994 года).

Правило 21 появилось благодаря докладу Брайана Кернигана «AWK для транслятора C++» на конференции USENIX по C++ в 1991 году. Его идея использовать перегруженные операторы (общим числом 67!) для выполнения арифметических операций с операндами разных типов хотя и не была связана с проблемой, обсуждаемой в правиле 21, но заставила меня рассмотреть множественную перегрузку операторов в качестве решения задачи по созданию временных объектов.

Мой вариант шаблона класса для подсчета объектов, рассмотренный в правиле 26, основан на сообщении Джамшида Афшара (Jamshid Afshar) в конференцию comp.lang.c++.

Идея смешанного класса, позволяющего отслеживать указатели, созданные с помощью `operator new` (см. правило 27), базируется на предложении Дона Бокса (Don Box). Стив Клемидж (Steve Clamage) придал этой идее практическое значение, объяснив, как можно использовать `dynamic_cast` для нахождения начала области памяти, занимаемой объектом.

Описание smart-указателей в правиле 28 основано: частично на заметке Стивена Буроффа (Steven Buroff) и Роба Мюррея (Rob Murray) C++ Oracle в журнале C++ Report за октябрь 1993 года, на классической работе Даниэла Р. Эдельсона (Daniel R. Edelson) «Интеллектуальные (smart) указатели: интеллектуальные, но не указатели» в материалах конференции USENIX по C++ от 1992 года, на содержимом раздела 15.9.1 книги Бьерна Страуструпа «Архитектура и развитие C++», на докладе Грегори Колвина (Gregory Colvin) «Управление памятью в C++» на учебном семинаре «Решения для C/C++ '95» и на заметке Кея Хорстманна (Cay Horstmann) в мартовском и апрельском номерах C++ Report за 1993 год. Но кое-что сделал и я сам.

Использованный в правиле 29 метод хранения в базовом классе счетчиков ссылок и smart-указателей для работы с этими счетчиками основан на идее Роба Мюррея (см. разделы 6.3.2 и 7.4.2 его книги «Стратегия и тактика в C++»). Прием, позволяющий добавлять счетчики ссылок к существующим классам, аналогичен тому, что был предложен Кеем Хорстманном в заметке, опубликованной в мартовском и апрельском номерах журнала C++ Report за 1993 год.

Источником для правила 30, касающегося контекстов `lvalue`, послужили комментарии к заметке Дэна Сакса в журнале C User's Journal (теперь C/C++

User's Journal) за январь 1993 года. Наблюдение, что методы классов, которые не являются методами proxy-классов, не доступны при вызове по proxy-механизму, взято из неопубликованной работы Кея Хорстманна.

Способ, как использовать динамическую информацию о типах для того, чтобы построить похожие на vtbl массивы указателей функций (в правиле 31), основан на идеях Бьерна Страуструпа, выдвинутых им в сообщениях в конференцию comp.lang.c++ и разделе 13.8.1 его книги «Архитектура и развитие C++».

Сведения, на базе которых появилось правило 33, частично были опубликованы в моих колонках журнала C++ Report за 1994 и 1995 года. Эти колонки, в свою очередь, включали замечания об использовании `dynamic_cast` для реализации виртуального оператора `operator=`, определяющего наличие аргументов некорректного типа, которые я получил от Клауса Крефта (Klaus Kreft).

Большая часть рассуждений в правиле 34 вызвана статьей Стива Клемиджа «Связывание C++ с другими языками» в мартовском номере журнала C++ Report за 1992 год. Мой подход к решению проблем, вызванных использованием таких функций, как `strdup`, был инициирован замечаниями читателя, не сообщившего своего имени.

Об этой книге

Просмотр черновых вариантов книги – работа неблагодарная, но жизненно необходимая. Мне повезло, что так много людей пожелали вложить в нее свое время и энергию. Хочу особенно поблагодарить: Джил Хатчитэл (Jill Huchital), Тима Джонсона (Tim Johnson), Брайана Кернигана, Ерика Наглера и Криса Ван Вык (Chris Van Wyk), потому что они прочли мою книгу (или ее значительную часть) более одного раза. Кроме этих любителей неприятной работы полностью черновик книги прочли: Катрина Эвери (Katrina Avery), Дон Бокс, Стив Буркетт (Steve Burkett), Том Каргилл, Тони Дэвис (Tony Davis), Кэролин Даби (Carolyn Duby), Брюс Экель (Bruce Eckel), Рид Флеминг (Read Fleming), Кей Хорстманн, Джеймс Канце, Расс Пейли (Russ Paielly), Стив Розенталь (Steve Rosenthal), Робин Руйе (Robin Rowe), Дэн Сакс, Крис Селлз (Chris Sells), Уэбб Стейси (Webb Stacy), Дэйв Свифт (Dave Swift), Стив Виноски (Steve Vinosky) и Фред Уайлд (Fred Wild). Частично черновики прочли: Боб Бьючейн (Bob Beauchaine), Герд Хойрен (Gerd Hoeren), Джефф Джексон (Jeff Jackson) и Нэнси Л. Урбано (Nancy L. Urbano). Замечания каждого из них помогли представить материал более точно и доступно.

После выхода книги я получил исправления и предложения от множества людей. Ниже эти наблюдательные читатели перечислены в порядке получения от них сообщений: Льюис Кида (Luis Kida), Джон Поттер (John Potter), Тим Уттормарк (Tim Uttormark), Майк Фелькерсон (Mike Fulkerson), Дэн Сакс, Вольфганг Глунц (Wolfgang Glunz), Кловис Тондо (Clovis Tondo), Майкл Лофтус (Michael Loftus), Лиз Хэнкс (Liz Hanks), Вил Эверс (Wil Evers), Стефан Кухлинз (Stefan Kuhlins), Джим МакКракен (Jim McCracken), Элан Дучан (Alan Duchan), Джон Джекобсма (John Jacobsma), Рамеш Нагабушнам (Ramesh Nagabushnam), Эд Виллинк (Ed Willink), Кирк Свенсон (Kirk Swenson), Джек Ривз (Jack Reeves), Дуг

Шмидт (Doug Schmidt), Тим Бучовски (Tim Buchowski), Пол Чисхолм (Paul Chisholm), Эндрю Клейн (Andrew Klein), Эрик Наглер, Джеффри Смит (Jeffrey Smith), Сэм Бент (Sam Bent), Олег Штейнбук (Oleg Shteynbuk), Антон Доблмайер (Anton Doblmaier), Ульф Михаэлис (Ulf Michaelis), Секхар Муддана (Sekhar Muddana), Майкл Бейкер (Michael Baker), Йечил Кимчи (Yechiel Kimchi), Дэвид Папюрт (David Papurt), Йан Хаггард (Ian Haggard), Роберт Шварц (Robert Schwartz), Дэвид Хэлпин (David Halpin), Грэхам Марк (Graham Mark), Дэвид Баретт (David Barett), Дэмьен Канарек (Damian Kanarek), Рон Коуттс (Ron Coutts), Ланс Витцель (Lance Whitesel), Йон Лачелт (Jon Lachelt), Шерил Фергюсон (Cheryl Ferguson), Мунир Махмуд (Munir Mahmood), Клаус-Георг Адами (Klaus-Georg Adams), Дэвди Гох (David Goh), Крис Морли (Chris Morley), Рейнер Баумшлагер (Rainer Baumschlager), Брайан Керниган, Чарльз Грин (Charles Green), Марк Роджерс (Mark Rodgers), Бобби Шмидт (Bobby Schmidt), Шиварамахришан Дж. (Sivaramakrishnan J.) и Эрик Андерсон (Eric Anderson). Их предложения позволили мне улучшить книгу, и я очень благодарен им за помощь.

При подготовке этой книги я сталкивался с множеством вопросов, связанных с появлением стандарта ISO/ANSI для языка C++, решить которые мне помогли Стив Клэмидж и Дэн Сакс. Они не пожалели времени, отвечая на мои беспрестанные вопросы по электронной почте.

Джон Макс Скаллер (John Max Skaller) и Стив Рамсби (Steve Rumsby) помогли мне получить текст ANSI-стандарта C++ в формате HTML до его публикации. Вивиан Неу (Vivian Neou) подсказала мне, что для просмотра HTML-документов в 16-битной системе Microsoft Windows можно использовать браузер Netscape. Я глубоко благодарен сотрудникам компании Netscape Communications за бесплатное распространение своего браузера для этой системы.

Брайан Хоббс (Bryan Hobbs) и Хачеми Зенад (Hachemi Zenad) предоставили мне предварительную версию компилятора MetaWare C++, что позволило проверить тексты программ, приведенных в этой книге, с использованием самых новых свойств языка. Кей Хорстманн помог мне с установкой и запуском компилятора в чуждых для меня мирах DOS и защищенного режима DOS. Корпорация Borland (теперь Inprise) предоставила мне последнюю бета-версию своего компилятора, а Эрик Наглер и Крис Селлз обеспечили неоценимую помощь, проверив тексты программ на недоступных для меня компиляторах.

Книга не могла бы появиться на свет без помощи сотрудников отдела корпоративной и специальной литературы издательства Addison-Wesley. Я очень обязан: Ким Доули (Kim Dawley), Лане Лэнглуа (Lana Langlois), Симоне Пэймент (Simone Payment), Марти Рабинович (Marty Rabinowitz), Прадипе Сива (Pradeepa Siva), Джону Уэйту (John Wait) и другим сотрудникам за их терпение, поддержку и помощь в подготовке этой работы.

Крис Гузиковски (Chris Guzikovsky) помогал проектировать обложку книги, а Тим Джонсон (Tim Johnson) уделил часть своего времени, обычно всецело посвященного исследованиям в области низкотемпературной физики, для критических замечаний по последним версиям этого текста.

Том Каргилл благородно согласился на размещение его статьи по исключениям из журнала C++ Report на сайте издательства Addison-Wesley в Internet.

Люди, которые мне помогали

Кэти Рид (Kathy Reed) ввела меня в мир программирования. Дональд Френч (Donald French) поверил в мою способность разрабатывать и представлять учебные материалы по C++ при отсутствии у меня значительного опыта в этой области. Он также представил меня редактору издательства Addison-Wesley Джону Вэйту (John Wait), за что я всегда буду ему благодарен. Троица в Бивер Ридж – Джейни Бесо (Jayni Besaw), Лорри Филдс (Lorry Fields) и Бет МакКи (Beth McKee) позволяла мне развлечься и отдохнуть в перерывах между работой над книгой.

Моя жена, Нэнси Л. Урбано, стоически перенесла все этапы подготовки книги. Сколько раз она слышала, что мы обязательно сделаем что-нибудь, после того как книга будет опубликована! Теперь работа завершена, и я выполню все свои обещания. Она удивительная. Я люблю ее.

И наконец, я должен вспомнить собаку Персефону, чье появление навсегда изменило наш мир. Без нее эта книга была бы закончена быстрее, и спал бы я больше, но значительно меньше смеялся.

#

Сейчас у программистов C++ горячие денечки. Хотя коммерческие версии компиляторов языка C++ появились менее чем десять лет назад, за это время C++ стал стандартным языком для создания сложных систем почти на всех вычислительных платформах. Компании и программисты, решая серьезные задачи по разработке программного обеспечения, постоянно расширяют круг пользователей языка. Перед теми, кто пока не имел дело с C++, чаще стоит вопрос «*Когда* начать использование языка?», а не «*Что будет*, если мы начнем применять этот язык?». Стандартизация C++ завершена, а богатая функциональность и разнообразие тематик сопровождающих язык библиотек, которые включают и расширяют библиотеки C, позволяют создавать сложные, многофункциональные программы, не теряющие при этом переносимости, а также реализовывать стандартные алгоритмы и структуры данных «с нуля». Компиляторы C++ продолжают совершенствоваться, их возможности расширяются, а качество генерируемого кода постоянно улучшается. Среды и средства для разработки на C++ становятся все более многочисленными, мощными и полнофункциональными. Библиотеки программного обеспечения, распространяемые на коммерческой основе, во многом устранили саму необходимость написания исходных текстов.

По мере «взросления» языка и роста опытности его пользователей изменилась и потребность в информации о нем. В 1990 году специалисты хотели знать, *что* представляет собой язык C++. К 1992 году их интересовало, *как* его применять. Сейчас программисты на C++ задают вопросы более высокого уровня. Как создавать программное обеспечение с учетом его адаптации к будущим потребностям? Как сделать программный код более эффективным, при этом не усложняя его и сохраняя корректность работы? Как реализовать ту или иную функцию, не поддерживаемую языком непосредственно?

В книге приводятся ответы на эти и многие похожие вопросы.

Книга показывает, как разрабатывать и внедрять *более эффективное*, чем то, которое вы создавали до сих пор, программное обеспечение на языке C++: содержащее меньшее количество ошибок, более надежное в экстремальных ситуациях, более производительное, более переносимое, более полно использующее возможности языка, требующее меньших затрат при поддержке, более пригодное для работы в системах, где задействовано несколько языков программирования, более простое при правильном использовании, затрудняющее неправильное использование. Короче, программное обеспечение, которое просто *лучше*.

Содержание этой книги разделено на 35 правил. В каждом разделе собраны накопленные сообществом C++ сведения по какому-то определенному вопросу. Большинство правил сформулированы как рекомендации, а объяснение, сопутствующее каждому правилу, содержит информацию о том, почему эта рекомендация

имеет право на существование, что происходит, если не следовать ей, и при каких условиях стоит все же ее нарушать.

Правила можно разбить на несколько категорий. Одни относятся к отдельным свойствам языка, по преимуществу недавно появившимся, для которых еще не накоплено опыта по применению. Например, правила с 9 по 15 посвящены исключениям. Другие правила объясняют, как объединить возможности языка для выполнения нестандартных задач. В эту группу входят правила с 25 по 31, которые описывают, как ограничить количество или размещение объектов, как создавать функции, являющиеся виртуальными по отношению к объектам разных типов, как создавать интеллектуальные указатели и т.п. Некоторые правила касаются более сложных случаев, так, правила с 16 по 24 связаны с проблемами эффективности. Но чему бы ни было посвящено правило, вопрос обсуждается серьезно и всесторонне. Эта книга учит, *как использовать* C++ наиболее эффективно. Описание конструкций языка, что составляет львиную долю текста других книг по C++, здесь является вспомогательной информацией.

Поэтому, приступая к чтению данной книги, вы должны быть уже знакомы с языком C++. Вы должны знать, что такое классы, уровни изоляции, виртуальные и невиртуальные функции и т.п., а также должны иметь представление о шаблонах и исключениях. Но пусть эти требования не смущают тех, кто не является специалистом по языку: исследуя закоулки C++, я всегда буду объяснять, как и что происходит.

Язык C++ в этой книге

Язык C++, представленный в этой книге, соответствует документу Final Draft International Standard (Финальный проект международного стандарта), выпущенному комитетом по стандартизации ISO/ANSI в ноябре 1997 года. Поэтому некоторые свойства языка, представленные в книге, ваши компиляторы, возможно, еще не поддерживают. Не волнуйтесь. Предполагается, что единственное «новое» свойство, которое вам потребуется, – шаблоны, а шаблоны реализованы почти везде. Я также использую исключения, но это использование в значительной мере ограничено правилами с 9 по 15, которые как раз и посвящены исключениям. Если у вас нет доступа к компилятору, поддерживающему исключения, ничего страшного. Это не повлияет на вашу работу с остальными частями книги. Но правила с 9 по 15 вам все же стоит почитать, потому что эти разделы помогут вам получить информацию, которой вы еще не владеете, но которую должны знать.

Допускаю, что благословение комитета по стандартам для какого-либо свойства языка или введение его в общепринятую практику не дает гарантии, что ваши компиляторы поддерживают это свойство и что известные способы применимы к существующим средам программирования. В тех случаях, когда возникает расхождение между теорией (утверждено комитетом по стандартам) и практикой (должно работать), я обсуждаю обе возможности, хотя склоняюсь к практическому решению. Так как рассматриваются обе стороны вопроса, у вас будет повод заглянуть в эту книгу всякий раз, когда ваши компиляторы на очередной шаг приблизятся к требованиям стандарта. Она покажет вам, как использовать

существующие конструкции языка для реализации новых свойств, не поддерживаемых пока вашими компиляторами, и даст рекомендации, как преобразовать эти обходные пути, когда ваши компиляторы начнут поддерживать новые свойства.

Заметьте, что я ссылаюсь на *ваши компиляторы* – во множественном числе. Различные компиляторы реализуют различные приближения к стандарту, поэтому я призываю вас вести разработку программного обеспечения с использованием как минимум двух компиляторов. Такая практика поможет вам избежать ненужной зависимости от нестандартных расширений языка, поддерживаемых только одним производителем компиляторов, или отклонений от стандарта. Это также позволит вам держаться подальше от переднего края компьютерных технологий, то есть от новых свойств языка, поддерживаемых только одним производителем компиляторов. Реализация таких свойств часто имеет значительные недостатки (наличие ошибок, низкая производительность, а иногда и то и другое). Кроме того, сообщество С++ еще не накопило достаточно опыта, чтобы предоставить разработчикам информацию, как правильно использовать новейшие свойства. Прокладывать дорогу – это здорово, но если ваша цель – создание надежного кода, нащупывать путь предоставьте другим.

В книге описаны две конструкции, с которыми вы можете быть незнакомы. Обе являются сравнительно новыми расширениями языка. Некоторые компиляторы поддерживают их, но если ваши компиляторы не настолько современны, вы легко можете воспроизвести эти расширения имеющимися в наличии средствами.

Первая конструкция – это тип `bool`, имеющий в качестве значений константы `true` и `false`. Даже если ваши компиляторы не поддерживают данный тип, существуют два способа реализовать его. Один состоит в использовании глобального оператора enum:

```
enum bool { false, true };
```

Такой прием позволяет перегружать функции, руководствуясь типом входящего аргумента `bool` или `int`. Однако у этого способа есть недостаток: встроенные операторы сравнения (такие как ==, <, >= и т.п.) все равно возвращают тип `int`. В результате приведенный ниже код не будет работать так, как это задумывалось:

```
void f(int);
void f(bool);
int x, y;
...
f(x<y);            // Вызывает f(int), а должен
                   // вызывать f(bool).
```

Применение оператора `enum` может нарушить работу программы после перехода на компилятор, поддерживающий тип `bool`.

Другой вариант состоит в том, чтобы использовать `typedef` для типа `bool` и константы для `true` и `false`:

```
typedef int bool;
const bool false = 0;
const bool true = 1;
```

Такой подход соответствует традиционной семантике C и C++, а поведение программ, использующих данный прием, не изменится с переходом на компиляторы, обеспечивающие поддержку `bool`. Недостаток же этого подхода состоит в том, что при перегрузке функций тип `bool` нельзя отличить от `int`. Оба варианта неплохи. Выбирайте тот из них, который лучше всего подходит в каждом конкретном случае.

Вторая новая конструкция на самом деле состоит из четырех, это операторы приведения типа: `static_cast`, `const_cast`, `dynamic_cast` и `reinterpret_cast`. Если вы не знакомы с перечисленными конструкциями, обратитесь к правилу 2, где содержится полная информация о них. Они не только делают больше, чем операторы приведения типов в стиле C, но и лучше выполняют все поставленные программистом задачи. Поэтому для приведения типов в данной книге я использовал именно их.

C++ – это не только язык. В него входит также и стандартная библиотека. Везде, где возможно, вместо указателей `char*` в книге использован стандартный тип `string`, и я призываю вас поступать также. Применение объектов `string` не создает дополнительных трудностей по сравнению со строками, обработка которых производится с помощью указателей `char*`, но значительно облегчает управление памятью. Кроме того, когда используются объекты `string`, снижается риск утечек памяти при возникновении исключений (см. правила 9 и 10). Хорошо реализованный тип `string` по эффективности не уступит своему эквиваленту `char*`, а, возможно, и превзойдет его (чтобы понять, как это сделать, см. правило 29). Если стандартный тип `string` не реализован в вашем компиляторе, то в нем почти наверняка реализован *какой-либо* класс, подобный `string`. Обязательно используйте его. Это всегда предпочтительней, чем включение в код указателей `char*`.

В книге при любой возможности используются структуры данных, взятые из Standard Template Library (STL – стандартная библиотека шаблонов, см. правило 35). STL содержит битовые наборы, векторы, списки, очереди, стеки, карты, множества и т.д., и лучше иметь дело со стандартизованными объектами, чем пытаться разработать их эквиваленты самостоятельно. В состав ваших компиляторов STL может быть не включена, но благодаря компании Silicon Graphics бесплатная копия библиотеки доступна на Web-сайте SGI STL: http://www.sgi.com/Technology/STL/.

Если вы довольны используемой вами в настоящее время библиотекой алгоритмов и структур данных, то не нужно переходить на STL только потому, что она «стандартная». Однако если вы стоите перед выбором, взять ли компонент из STL или написать какой-либо код самому «с нуля», то вариант с STL, конечно, предпочтительнее. Помните о принципе повторного применения кода? STL (и остальная часть стандартной библиотеки) содержит множество модулей, которые очень и очень стоит использовать повторно.

Соглашения и терминология

Когда в книге упоминается наследование, я всегда имею в виду открытое наследование. Использование закрытого наследования каждый раз оговаривается

специально. При изображении иерархии наследования стрелки проводятся от производных классов к базовому. Например, на рис. 1 приведена иерархия классов для правила 31:

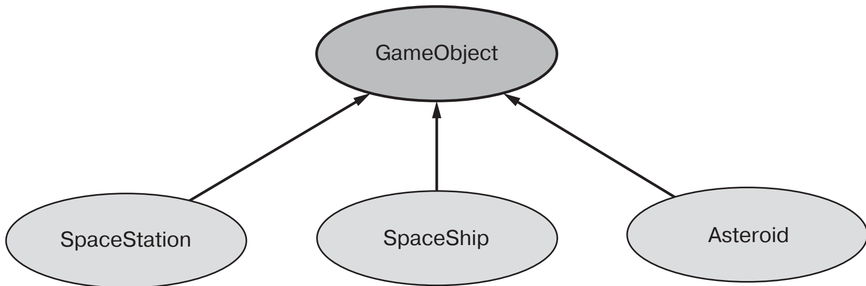

Рис. 1

Эта запись обратна той, которую я использовал в первом (но не во втором) издании книги «Эффективное использование C++». Теперь я убежден, что большинство пользователей C++ проводят стрелки наследования от производного класса к базовому, и с удовольствием соглашаюсь с ними. На диаграммах подобного типа абстрактные классы (например, `GameObject` (Игровой объект)) закрашены темно-серым цветом, а конкретные классы (такие как `SpaceShip` (Космический корабль)) заштрихованы более светлым оттенком.

Наследование автоматически приводит к появлению указателей и ссылок двух различных типов: *статического* и *динамического*. Статический тип указателя или ссылки соответствует типу *объявления*. Динамический тип – типу объекта, на который в данный момент указывает указатель или ссылка. Вот несколько примеров, основанных на приведенных выше классах:

```
GameObject *pgo =                // Статический тип pgo -
  new SpaceShip;                 // GameObject, динамический тип -
                                 // SpaceShip*.
Asteroid *pa = new Asteroid;     // Статический тип pa - Asteroid*.
                                 // Динамический тип - тоже
                                 // Asteroid*.
pgo = pa;                        // Статический тип pgo
                                 // не изменился (и не изменится),
                                 // он по-прежнему равен
                                 // GameObject*. Его
                                 // динамический тип теперь -
                                 // Asteroid*.
GameObject& rgo = *pa;           // Статический тип rgo -
                                 // GameObject, динамический тип -
                                 // Asteroid.
```

Эти примеры также демонстрируют соглашение об именах переменных, используемое в этой книге. Переменная `pgo` – это указатель на `GameObject`,

pa – указатель на Asteroid, rgo – ссылка на GameObject. Я часто составляю имена переменных и ссылок подобным образом.

Два моих любимых имени для параметров – это lhs и rhs, сокращения для «слева» (left-hand side) и «справа» (right-hand side) соответственно. Чтобы понять, что стоит за этими именами, рассмотрим класс для представления действительных чисел:

```
Class Rational { ...... };
```

Функция для попарного сравнения объектов класса Rational может быть объявлена следующим образом:

```
bool operator==(const Rational& lhs, const Rational& rhs);
```

Такое объявление позволяет писать программы следующим образом:

```
Rational r1, r2;
...
if (r1 == r2) ...
```

В операторе сравнения r1 стоит слева от оператора == и соответствует аргументу lhs при вызове operator==, r2 расположено справа от оператора == и соответствует аргументу rhs.

Другие сокращения, использованные в этой книге: ctor обозначает конструктор (constructor), dtor – деструктор (destructor), RTTI – динамическое определение типов в C++ (оператор dynamic_cast – наиболее часто используемый компонент этого механизма).

Если выделить память, а затем не освободить ее, то происходит утечка памяти. Утечки могут возникать как в C, так и в C++, но в последнем они приводят к более серьезным последствиям. Это связано с тем, что в C++ при создании объектов автоматически вызываются конструкторы, которые могут сами выделять ресурсы. Посмотрите на этот пример:

```
class Widget { ...... };     // Некий класс - неважно,
                             // что он делает.
Widget *pw = new Widget;     // Динамическое выделение
                             // памяти под объект Widget.
...                          // Предположим, что область,
                             // на которую указывает pw,
                             // никогда не освобождается.
```

Этот код вызывает утечку памяти, потому что объект Widget, на который указывает переменная pw, никогда не будет удален. Возможна ситуация, когда конструктор Widget потребует выделения дополнительных ресурсов (таких как дескрипторы, семафоры, манипуляторы окон, блокираторы баз данных* и т.п.),

* В других книгах вы можете встретить иные варианты перевода английских понятий window handle и database lock, поскольку в русском языке компьютерная терминология еще недостаточно устоялась. Обычно программисты обобщенно называют эти инструменты хэндлами (от англ. handle). (*Прим. ред.*)

которые должны освобождаться при удалении объекта `Widget`. Если `Widget` не будет удален, то доступ к этим ресурсам будет утерян так же, как и к занимаемой ими области памяти. Чтобы подчеркнуть, что утечки памяти в C++ часто приводят к утечкам других ресурсов, я буду использовать выражение «утечка ресурсов» во всех случаях.

Примеры кода, приведенные в книге, редко включают встраиваемые функции. Это не значит, что я их не люблю. Встраиваемые функции, безусловно, являются важной чертой языка C++. Однако критерии для определения, должна ли функция объявляться как встраиваемая, могут быть довольно сложными, нечеткими и зависеть от платформы. В результате я избегаю использовать встраиваемые функции, если только это не связано напрямую с обсуждаемой темой. Когда вы встречаете в примерах невстраиваемую функцию, это не означает, что ее объявление как `inline` создаст дополнительные проблемы, просто вопрос, объявлять ли функцию встраиваемой или нет, не влияет на изложение темы в данном месте книги.

Некоторые свойства C++ комитет по стандартам объявил *устаревшими*. В них больше нет необходимости, потому что к языку были добавлены новые свойства, которые намного лучше выполняют функции прежних. В этой книге я специально обращаю внимание на устаревшие конструкции и объясняю, чем их можно заменить. Программистам не следует работать с такими конструкциями, хотя испытывать особые угрызения совести от их использования также не стоит. Для сохранения обратной совместимости производители компиляторов, скорее всего, будут поддерживать устаревшие свойства еще много лет.

Клиент (или пользователь) – это кто-то (возможно, программист) или что-то (обычно класс, функция), использующий написанный вами код. Например, вы создали класс `Date` (для описания дней рождения, календарных сроков, дня второго пришествия и т.п.). Тогда всякий, обращающийся к этому классу, является вашим клиентом. Далее, любые модули, использующие класс `Date`, также окажутся вашими клиентами. Ради клиентов и ведется разработка! Если ваше программное обеспечение никому не требуется, зачем его создавать? Читая книгу, вы заметите, что я прикладываю массу усилий, чтобы облегчить жизнь клиентам, часто за ваш счет, поскольку хорошее программное обеспечение должно быть «клиентоцентричным»: оно должно «вращаться» вокруг клиентов. Если это кажется вам излишней филантропией, посмотрите на проблему с точки зрения собственных интересов. Используете ли вы ваши классы и функции повторно? Если да, то вы – ваш собственный клиент, и облегчая жизнь клиентам вообще, вы облегчаете ее самому себе.

Рассуждая о шаблонах классов или функций и сгенерированных по этим шаблонам объектах, я позволю себе быть несколько неряшливым и особо не подчеркивать разницу между шаблонами и созданными по ним объектами. Например, если `Array` – это шаблон класса с параметром `T`, то я могу ссылаться на шаблонный класс как на `Array`, хотя на самом деле его правильное имя – `Array<T>`. Аналогично, если `swap` – шаблон функции с параметром типа `T`, то имя `swap` (вместо `swap<T>`) также может обозначать и шаблонную функцию. Разумеется, если

такая сокращенная запись может привести к недоразумению, я записываю полные имена объектов.

Принятые обозначения

Для более простого восприятия материала в книге приняты следующие условные обозначения.

Все `листинги`, приведенные в книге, напечатаны моноширинным шрифтом.

Имена `классов`, `объектов`, `переменных`, констант и т.д., встречающиеся непосредственно в тексте, также даны `моноширинным шрифтом`.

Информация, которую необходимо обязательно принять к сведению, выделена *курсивом*.

Как сообщить об ошибках, внести предложения, получить обновления книги

Насколько это возможно, я старался сделать книгу точной, удобной для чтения и полезной, однако нет предела совершенству. Если вы обнаружите в ней какую-либо ошибку: техническую, грамматическую, опечатку, *какую-нибудь* – пожалуйста, сообщите мне об этом. Я постараюсь исправить допущенную оплошность в последующих изданиях книги, а если вы окажетесь первым, кто сообщит об ошибке, с удовольствием добавлю ваше имя в список благодарностей. Если у вас появятся другие предложения по улучшению книги, также буду вам очень признателен.

Я по-прежнему продолжаю собирать рекомендации по эффективному программированию на C++. Если у вас есть какие-нибудь идеи на этот счет, буду очень благодарен, если вы поделитесь ими со мной. Шлите ваши рекомендации, комментарии, замечания и сообщения об ошибках по адресу:

Scott Meyers
c/o Editor-in-Chief, Corporate and Professional Publishing
Addison-Wesley Publishing Company
1 Jacob Way
Reading, MA 01867
U.S.A.

Вы также можете послать сообщение электронной почты по адресу: mec++@awl.com.

Я веду список изменений, таких как исправления ошибок, пояснения и обновления, внесенных в книгу с первого издания. Этот список, а также другие материалы, связанные с данной книгой, размещен на Web-сайте издательства Addison-Wesley по адресу: http://www.awl.com/cp/mec++.html. Он также находится на FTP-сайте по адресу: ftp.awl.com в каталоге cp/mec++. Если у вас нет доступа в Internet, то для получения списка изменений пошлите запрос по одному из приведенных выше двух адресов, и я прослежу, чтобы список был вам выслан.

В настоящее время существует также список рассылки, подписаться на который можно, послав сообщение по адресу scott_meyers-subscribe@egroups.com. Архив списка рассылки находится на странице http://www.egroups.com/messages/scott_meyers. Он предназначен для рассылки объявлений программистам, интересующимся языком C++. Объем рассылки небольшой, обычно не более двух сообщений в месяц. Более подробные сведения о списке рассылки можно получить на странице http://www.aristeia.com/MailingList/index.html.

Глава 1. Основы

Указатели, ссылки, приведение типов, массивы, конструкторы – это то, что составляет основу языка. Все программы на языке C++, за исключением самых простых, используют большую часть названных понятий, а многие программы используют их все.

Даже самые знакомые вещи иногда могут нас удивлять. Особенно это справедливо для программистов, переходящих с языка C на C++, так как концепции, на которых базируются понятия ссылок, динамического приведения типов, конструкторов по умолчанию и других, не принадлежащих языку C, обычно не всегда очевидны.

Эта глава объясняет разницу между указателями и ссылками и содержит советы, когда следует использовать каждое из этих понятий. В ней также описан новый синтаксис языка C++ для приведения типов и объясняется, чем новый стиль превосходит заменяемый стиль языка C. Кроме того, рассматривается концепция массивов в языке C и концепция полиморфизма в языке C++, а также говорится, почему их никогда не стоит использовать одновременно. Наконец, в ней рассказано о плюсах и минусах конструкторов по умолчанию и предложены пути для обхода ограничений языка, которые требуют существования такого конструктора, даже если это не имеет практического смысла.

Следуя советам, приведенным в нижеизложенных правилах, вы сможете создавать такое программное обеспечение, где ваш замысел будет реализован ясно и правильно.

Правило 1. Различайте указатели и ссылки

Указатель (`pointer`) и ссылка (`reference`) *существенно* отличаются по внешнему виду (указатели используют операторы `*` (умножить) и `->` (стрелка), ссылки используют оператор `.` (точка)), но применяются для решения одних и тех же задач. И указатели и ссылки позволяют неявно ссылаться на другие объекты. Как же тогда решить, когда применять указатели, а когда ссылки?

Во-первых, запомните, что не существует нулевых ссылок. Ссылка должна *всегда* ссылаться на какой-либо объект. Если ваша переменная обеспечивает доступ к объекту, которого может и не быть, вы должны использовать указатель, потому что это позволит приравнять его нулю. С другой стороны, если переменная должна *всегда* ссылаться на существующий объект, то есть не должна иметь нулевого значения, то, скорее всего, лучше использовать в качестве такой переменной ссылку.

«Но подождите!», – воскликнет читатель, – «а как же будет работать следующий кусок кода?»:

```
char *pc = 0;            // Присвоить указателю значение null.
char& rc = *pc;          // Установить ссылку на содержимое
                         // нулевого указателя.
```

Надо сказать, это пример самого настоящего безобразия. Результаты работы такой программы не определены: компиляторы могут генерировать программный код, который будет делать все, что угодно. Если у вас возникают подобные проблемы, то лучше вообще отказаться от использования ссылок. В качестве другого выхода вы можете поискать для сотрудничества программистов более высокого класса. В дальнейшем мы не будем считаться с возможным существованием нулевых ссылок.

Так как ссылка должна ссылаться на объект, C++ требует ее инициализации:

```
string& rs;              // Ошибка! Ссылки должны быть
                         // проинициализированы.
string s("xyzzy");
string& rs = s;          // Нормально, rs ссылается на s.
```

На указатели таких ограничений не налагается:

```
string *ps;              // Неинициализированный указатель:
                         // допустимо, но рискованно.
```

Невозможность существования нулевых ссылок подразумевает, что использование ссылок более эффективно, чем использование указателей. Корректность ссылки не нужно предварительно проверять:

```
void printDouble(const double& rd)
{
   cout << rd;           // Нет необходимости проверять rd;
}                        // она должна ссылаться на double.
```

Указатели же, наоборот, обычно должны проверяться на равенство нулю:

```
void printDouble(const double *pd)
{
   if (pd) {             // Проверка на значение null.
      cout << *pd;
   }
}
```

Другое важное различие между указателями и ссылками состоит в возможности присваивать указателям различные значения для доступа к разным объектам. Ссылка же всегда указывает на один и тот же объект, заданный при ее инициализации:

```
string s1("Nancy");
string s2("Clancy");
string& rs = s1;         // rs ссылается на s1.
string *ps = &s1;        // ps указывает на s1.
rs = s2;                 // rs все еще ссылается на s1,
                         // но теперь s1 имеет значение
                         // "Clancy".
```

```
ps = &s2;                 // ps указывает на s2;
                          // значение s1 не изменилось.
```

Вообще говоря, указатель следует использовать, если есть вероятность, что объект, связанный с указателем, отсутствует (в этом случае ему присваивается нулевое значение) или периодически возникает необходимость доступа к разным объектам (в этом случае изменяется значение указателя). Ссылку же следует использовать, если объект, к которому необходимо обеспечить доступ, будет существовать всегда, и не потребуется получить доступ к другому объекту с помощью все той же ссылки.

Существует еще одна ситуация, в которой вы должны задействовать ссылки – при реализации некоторых операторов; из них наиболее часто встречается оператор [](скобки). Обычно этот оператор должен вернуть некое значение, которое затем будет использовано как принимающее в операторе присваивания:

```
vector<int> v(10);   // Создаем вектор целых значений
                     // размерности 10;
                     // вектор является шаблоном
                     // из стандартной библиотеки C++
                     // (см. правило 35).
v[5] = 10;           // Значение, возвращаемое
                     // оператором [], является
                     // принимающей стороной оператора
                     // присваивания.
```

Если бы оператор [] возвращал указатель, то последнюю строку этого кода надо было бы записать так:

```
*v[5] = 10;
```

Но создавалось бы ложное впечатление, что v является вектором указателей. Поэтому почти всегда желательно, чтобы оператор [] возвращал ссылку. (Интересное исключение из этого правила приведено в правиле 30.)

Итак, использование ссылок оправдано, когда доподлинно известно, что объект ссылки существует, когда нет необходимости изменять значение ссылки и при реализации операторов, в которых применение указателей нежелательно из-за синтаксических требований. Во всех других случаях используйте указатели.

Правило 2. Предпочитайте приведение типов в стиле C++

Рассмотрим прямое приведение типов. Это почти такой же изгой, как и оператор goto, но тем не менее оно продолжает использоваться, потому что, когда ситуация становится «хуже некуда», может оказаться необходимым.

Приведение типов в стиле языка C не применяется так широко, как могло бы. Во-первых, это довольно грубый инструмент, практически позволяющий привести произвольный тип к любому другому. Было бы неплохо более точно определять цель каждого приведения. Например, существует большая разница между приведением

указателя на объект const к указателю на объект, не являющийся const (то есть меняющим только атрибут const объекта), и приведением указателя на объект базового класса к указателю на объект производного класса (то есть приведение типа, которое полностью меняет тип указателя). Традиционное приведение типа в стиле языка C не различает эти два случая (и неудивительно – приведение типов в стиле языка C было разработано для C, а не для C++).

Вторая проблема с приведением типов заключается в том, что случаи его применения трудно обнаружить. Синтаксически приведение типов состоит всего-навсего из пары скобок и идентификатора, а скобки и идентификаторы используются в C++ повсеместно. Таким образом, нелегко ответить даже на основной вопрос: «Использует ли программа приведение типов?». Это происходит потому, что человеческий глаз не всегда замечает код приведения типов, а такие утилиты, как grep, не могут отличить его от других, синтаксически схожих, конструкций.

Чтобы преодолеть недостатки приведения типов в стиле языка C, в язык C++ введены четыре новых *оператора приведения типов*: static_cast, const_cast, dynamic_cast и reinterpret_cast. Для большинства задач программист должен знать только, что там, где он привык писать

```
(type) expression
```

теперь следует писать

```
static_cast<type>(expression)
```

Допустим, вам необходимо привести выражение типа int к типу double, чтобы получить число с плавающей точкой в результате вычисления целочисленного выражения. Используя приведение типов в стиле языка C, это можно было сделать следующим образом:

```
int firstNumber, secondNumber;
...
double result = ((double)firstNumber)/secondNumber;
```

С новыми операторами приведения типа это делается так:

```
double result = static_cast<double>(firstNumber)/secondNumber;
```

Теперь у нас есть приведение типов, которое легко обнаружат как человеческий глаз, так и программы.

Оператор static_cast обладает теми же возможностями, что и приведение типов общего назначения в стиле языка C. На него налагаются аналогичные ограничения. Например, также как и в языке C, с помощью static_cast вы не можете преобразовать переменную типа struct в int или переменную типа double в указатель. Более того, с помощью оператора static_cast нельзя убрать атрибут const в выражении, для этого служит специальный оператор const_cast.

Другие операторы приведения типов, введенные в C++, используются для более узкого круга задач. Оператор const_cast предназначен для работы с атрибутами const и volatile в выражениях. Используя оператор const_cast, вы

подчеркиваете (как для человека, так и для компьютера), что собираетесь только изменить атрибут const или volatile какого-либо объекта. Это значение оператора поддерживают и компиляторы. Если попытаться использовать оператор const_cast для других задач, отличных от изменения атрибутов const или volatile, то такое приведение типов будет отвергнуто. Вот некоторые примеры:

```
class Widget { ... };
class SpecialWidget: public Widget { ... };
void update(SpecialWidget *psw);
SpecialWidget sw;                        // sw не const объект,
const SpecialWidget& csw = sw;           // но csw ссылается на него,
                                         // как на const объект.
update(&csw);                            // Ошибка! Нельзя передавать const
                                         // указатель SpecialWidget*
                                         // функции, которая принимает
                                         // указатель SpecialWidget*.
update(const_cast<SpecialWidget*>(&csw));
                                         // Нормально,
                                         // атрибут const у &csw
                                         // удален в результате
                                         // преобразования типа
                                         // (и csw и sw могут быть изменены
                                         // в теле функции update).
update((SpecialWidget*)&csw);
                                         // То же самое, но используя
                                         // более трудное для обнаружения
                                         // преобразование типа в стиле
                                         // языка C.
Widget *pw = new SpecialWidget;
update(pw);                              // Ошибка! pw имеет тип Widget*,
                                         // а функция update принимает
                                         // аргумент типа SpecialWidget*.
update(const_cast<SpecialWidget*>(pw));
                                         // Ошибка! Оператор const_cast
                                         // можно использовать только
                                         // для изменения атрибутов
                                         // const или volatile
                                         // и нельзя применять для приведения
                                         // наследования.
```

В настоящее время оператор const_cast чаще всего используется для изменения атрибута const.

Второй специализированный оператор – dynamic_cast – для безопасного приведения типа между уровнями иерархии наследования. Это означает, что оператор dynamic_cast позволяет приводить указатели или ссылки на объекты базового класса к указателям или ссылкам на объекты производных или дочерних классов таким образом, чтобы можно было определить, была ли попытка приведения

типа успешной*. В результате неудачной попытки возвращается нулевой указатель (при преобразовании указателей) или возникает исключение (при преобразовании ссылок):

```
Widget *pw;
......
update(dynamic_cast<SpecialWidget*>(pw));
                                     // Нормально, функции update
                                     // передается указатель
                                     // на объект класса SpecialWidget,
                                     // если pw действительно указывает
                                     // на этот объект, в противном
                                     // случае передается нулевой
                                     // указатель.
void updateViaRef(SpecialWidget& rsw);
updateViaRef(dynamic_cast<SpecialWidget&>(*pw));
                                     // Нормально, передача функции
                                     // updateViaRef ссылки на объект
                                     // класса SpecialWidget, если pw
                                     // действительно указывает на этот
                                     // объект, в противном случае
                                     // возникает исключение.
```

Применение операторов `dynamic_cast` ограничено возможностью навигации по иерархии наследования. Операторы нельзя использовать для приведения типов, не имеющих виртуальных функций (см. также правило 24), или для работы с атрибутом `const`:

```
int firstNumber, secondNumber;
...
double result =  dynamic_cast<double>(firstNumber)/secondNumber;
                                     // Ошибка! int не имеет
                                     // виртуальных функций.
const SpecialWidget sw;
...
update(dynamic_cast<SpecialWidget*>(&sw));
                                     // Ошибка! dynamic_cast не может
                                     // работать с атрибутом const.
```

Если вы хотите выполнить преобразование ненаследуемых типов, то лучше всего подойдет оператор `static_cast`. Чтобы удалить атрибут `const`, всегда применяется оператор `const_cast`.

Последний из четырех новых операторов – оператор `reinterpret_cast`. Он используется для приведения типов в тех случаях, когда результат приведения почти всегда зависит от реализации. Из-за этого переносимость операторов `reinterpret_cast` существенно ограничена.

---

* Оператор `dynamic_cast` всегда используется для того, чтобы найти начало памяти, занимаемой объектом. Подробнее об этом рассказано в правиле 27.