Оглавление

Предисловие Шона Эдвардса

Я профессионально пишу программы на C++ уже более 25 лет, даже еще до стандартизации этого языка. Язык C++, по своей сущности предназначенный обеспечивать нулевые издержки и максимальную производительность, неизбежно создает некоторые ограничения – настолько далеко продвинулись синтаксис и безопасность типов. Использовать возможности C++ неразумным образом и получать ошеломляющие критические ошибки всегда было легко. Но поскольку язык был относительно стабильным, хорошие разработчики со временем научились писать надежное программное обеспечение на C++.

Первая стандартизированная версия C++98 формализовала то, что многие уже знали об этом языке. Вторая версия стандарта C++03 включала небольшие исправления и улучшения, но принципиально не меняла способ написания программ. Однако знание того, как программировать на C++, существенно изменилось после публикации стандарта C++11. Впервые за много лет Комитет по стандартам ISO C++ (WG21) добавил совершенно новую функциональность, а также кое-что удалил. Например, были введены `noexcept` и `std::unique_ptr`, а дни использования спецификаций динамических исключений и `std::auto_ptr` были сочтены.

В то же время Комитет по стандартам объявил о своем беспрецедентном намерении выпускать новую версию стандарта C++ каждые три года. Для крупной компании, занимающейся разработкой программного обеспечения, такой как Bloomberg, жизненный цикл программных активов которой измеряется десятилетиями, опора на стандарт языка, который обновляется с такой частотой, особенно проблематична. Компания Bloomberg надежно и точно предоставляет необходимую информацию профессиональному финансовому сообществу уже почти 40 лет, предлагая услуги, которые охватывают такие разнообразные потребности, как финансовая аналитика, маркетинговые решения и рыночные данные в режиме реального времени.

Для поддержки нашего глобального бизнеса компания Bloomberg разработала высокопроизводительные программные системы, работающие с поддержкой масштабирования, и уже более двух десятилетий пишет их в основном на C++. Как вы понимаете, внедрение и проверка новых цепочек инструментальных средств, лежащих в основе всей кодовой базы нашей компании, – непростая задача. Каждое обновление ставит под угрозу стабильность самих программных продуктов, от которых зависят наши клиенты.

Современный C++ предлагает многое – как хорошее, так и плохое. Многие из его новых функций открывают перспективу повышения производительности, выразительности, удобства сопровождения и т. д. С другой стороны, многие из этих функциональных возможностей таят в себе потенциальные ловушки – некоторые очевидны, другие менее очевидны. С каждой новой версией C++ – теперь уже раз в три года – язык постоянно расширяется, соответственно, расширяются и возможности для некорректного использования его функциональности из-за недостатка знаний и опыта.

Использование новых функций и без того сложного языка программирования, такого как C++, с которым многие разработчики могут быть не вполне знакомы, создает особую категорию риска. Менее опытные инженеры могут непреднамеренно ввести новые функции в зрелую кодовую базу, что в этом контексте, возможно, добавит заведомо отрицательную ценность. Как всегда, только время и опыт могут подтвердить, при каких условиях использование новой функции языка C++ будет вполне разумным. Мы как старшие разработчики, руководители групп и технические менеджеры ведущей финансово-технологической компании несем ответственность за защиту нашего основного актива в виде программного обеспечения от неоправданного риска.

Мы не можем оправдывать нестабильность при переписывании всего нашего программного обеспечения каждый раз, когда появляется новая версия языка, но мы не можем оставить наше ПО в навечно законсервированном состоянии и отказаться от важных преимуществ, которые предлагает современный C++. Поэтому мы движемся вперед, но со знанием дела и осторожностью, внедряя новые функциональные возможности только после того, как полностью поймем их смысл. Компания Bloomberg стре-

мится извлечь все возможные преимущества из современного C++, но как промышленное предприятие мы должны делать это безопасно.

Компания Bloomberg спонсировала создание этой книги «Современный C++: безопасное использование», потому что мы чувствовали, что, несмотря на все книги, конференции, блоги и т. д., описывающие функциональные возможности C++11/14, необходимо взглянуть на каждую функцию с точки зрения ее безопасного и эффективного применения в контексте большой, зрелой базы промышленного программного кода. Поэтому в книге представлены подробные объяснения каждой функциональной возможности языка C++11/14, примеры эффективного использования и подводные камни, которых следует избегать. Кроме того, эта книга могла быть написана только сейчас, после долгих лет накопления реального практического опыта. Более того, мы знали, что для написания такой книги у нас есть нужные люди – одни из лучших инженеров и авторов в мире.

Как и было обещано, Комитет по стандартам C++ придерживался своего графика, иногда с учетом крупных мировых событий, и были опубликованы две дополнительные версии стандарта, C++17 и C++20. По мере накопления сообществом опыта практического использования новых функциональных возможностей, представленных в этих стандартах, я полагаю, что будущие издания этой книги будут содержать соответствующие рекомендации и критические замечания.

Если вы писали программы на этом языке более десяти лет, то, несомненно, заметили, что стать опытным программистом на C++ стало сложнее, чем раньше. Эта книга поможет вам ориентироваться в современном мире C++, чтобы вы тоже почувствовали себя уверенно, применяя C++11/14 способами, которые действительно повышают ценность без неоправданного риска для дорогостоящих инвестиций вашей организации в программное обеспечение.

– *Шон Эдвардс* (Shawn Edwards),
технический директор (CTO) компании Bloomberg LP,
август 2021 г.

Предисловие Андрея Александреску

Вам нравятся системы управления версиями – Git, Perforce, Mercurial и им подобные? Мне очень нравятся. Я понятия не имею, как любая из современных сложных программных систем могла быть создана без использования системы управления версиями.

Одним из самых полезных свойств программного обеспечения для управления версиями является представление различий (diff view), наиболее существенное параллельное представление изменений в большой системе по сравнению с предыдущей, хорошо знакомой версией той же системы. Такое представление различий часто является наилучшим способом обзора кода, оценки сложности функциональных возможностей, поиска ошибок и, что наиболее важно, знакомства с новой системой. Почти каждый рабочий день я внимательно изучаю представления различий, исследуя одно или несколько достигнутых преимуществ. Такое представление различий является доказательством того, что мы действительно можем иметь в своем распоряжении часто упоминаемые в наших желаниях полезные средства.

В этой книге представлена новая концепция – сравнение между классическим C++, т. е. C++03, базовой версией языка, и современным C++, т. е. C++ после 2011 года, с его дополнительными функциональными возможностями. Различное представление функциональных возможностей языка программирования – потрясающая идея с интересными последствиями.

Книга «Современный C++: безопасное использование» предназначена для крупной категории программистов: тех, кто ежедневно работает со сложными системами C++ с длительным жизненным циклом, и кто знаком с C++03, поскольку указанные системы были написаны с использованием этой технологии. Классы. Наследование. Полиморфизм. Шаблоны. Стандартная библиотека STL. Да, они хорошо знают эти понятия и работают с ними каждый день в сложных предметных областях. Переопределять эти классические функциональные возможности нет необходимости. Но некоторые программисты, возможно, менее довольны изобилием новых функциональных возможностей, стандартизируемых каждые три года, начиная с C++11. У них нет времени на отслеживание деятельности Комитета по стандартам C++. Каждый час, потраченный на изучение новых функциональных возможностей C++, – это время, отнимаемое у основной функциональности системы, поэтому новая «крутая» функция должна оправдывать эти потери. Книга «Современный C++: безопасное использование» разумно оптимизирована для сохранения наилучшего соотношения полезности в производственной среде и времени, затрачиваемого на обучение.

С педагогической точки зрения эта книга решает почти невыполнимую задачу: описание «частной разности» (уж позвольте мне, как истинному ботанику, использовать математически обоснованную метафору) для каждой отдельной новой функциональной возможности, добавленной в C++ после 2003 года. Что я под этим подразумеваю? Когда книга учит особенностям языка, неизбежны споры: при обсуждении какой-либо одной отдельно взятой особенности обязательно привлекается большинство других характеристик. Как однажды сказал мне Скотт Мейерс (Scott Meyers): «Когда вы изучаете язык, то одновременно постигаете сущность всех его функциональных возможностей». Авторы тщательно разделили процессы обучения каждой новой функциональной возможности, поэтому если вы хотите узнать, скажем, об общих лямбда-выражениях, вы можете прочитать о них с минимальным воздействием со стороны любого другого нового функционального свойства языка. При необходимости взаимодействие между обсуждаемой функциональной возможностью и другими подробно оговаривается, документируется, и дается соответствующая перекрестная ссылка. В результате получается детально согласованная сама по себе книга, которую можно читать от корки до корки или разделить на части по глобальным темам, взаимосвязанным функциональным возможностям или по отдельным частным вопросам.

Главы 1, 2 и 3 напоминают своего рода противоположно направленную «Божественную комедию», в которой, как вы помните, поэта Данте надежные проводники ведут через ад (Inferno), чистилище (Purgatorio) и рай (Paradiso). Соответствующие главы по-

могут вам перейти от безопасных (Safe) к условно безопасным (Conditionally Safe) и небезопасным (Unsafe) функциональным возможностям современного C++.

Нет никаких сомнений в том, что безопасные функциональные возможности (см. главу 1) существенно улучшат ваш код, где бы вы их ни использовали. Изучение и применение рекомендаций, изложенных в главе 1, – это самый быстрый способ для группы разработчиков начать практическое использование современного C++ в производственной среде. `override`? Пользуйтесь на здоровье. Разделители цифр? Налетайте. Явные операторы преобразования? Потрясающе. Такие функциональные возможности рекомендуются в полной мере и безоговорочно. В главе 2 обсуждаются условно безопасные функциональные возможности, которые вам подходят, но с некоторыми оговорками. Списки инициализаторов? Давайте обсудим. Диапазон цикла `for`? Пара вещей, о которых следует помнить. Ссылки на Rvalue? Обсуждение будет долгим – запасайтесь кофе. И последнее по списку, но не по важности: небезопасные функциональные возможности, которые могут оказаться весьма сложными и требуют практических навыков и предельного внимания при использовании. Их применение должно быть максимально ограничено и завернуто в интерфейсы. Типы стандартной компоновки? Гораздо сложнее, чем может показаться на первый взгляд. Спецификатор `noexcept`? Применять с осторожностью, на собственный страх и риск. Встроенные пространства имен? Лучше отказаться. Подробная информация, примеры и обсуждения доступны для каждой отдельной функциональной возможности, добавленной после стандарта C++03.

Здесь авторы не без иронии используют слово «небезопасно». Ничто из того, о чем рассказывается в этой книге, не является небезопасным в традиционном понимании информатики. Вместо этого подумайте о несколько другом значении слова «безопасность», когда оно используется, скажем, в хозяйственном магазине. Какова безопасность различных инструментов для тех, кто только начинает ими пользоваться? Отвертка безопасна, электродрель условно безопасна, а сварочный аппарат небезопасен.

Возможно, вы задумаетесь над этими словами: «Звучит внушительно. Но что лежит в основе такой классификации?» На самом деле книга «Современный C++: безопасное использование» вовсе не является непререкаемым сводом догм и правил, она объективна и основана на обширном опыте сообщества, который собрали и обработали авторы. Они намеренно, иногда с болью в душе, скрывают свои мнения и точки зрения. Разделы «Варианты использования» и «Потенциальные ловушки», взятые из промышленного кода, представляют собой не только поучительные примеры, но и практические доказательства.

Только с течением времени можно выделить практический опыт сообщества программистов в отношении каждой функциональной возможности и ее применимости, поэтому в книге обсуждаются функциональные возможности, добавленные в C++14, несмотря на то что стандарт C++20 уже вышел. Использование функциональных возможностей в течение многих лет может заменить горячие споры об идеях дизайна языка беспристрастным, трудно заработанным опытом, который определяет строго объективный подход, примененный в этой книге. По словам Джона Лакоса (John Lakos), «мы описываем степень ожога, который вы можете получить, если положите руку на горячую плиту, но не запрещаем вам этого делать». В результате получается совершенно не оторванный от реальной жизни материал для чтения, не более предвзятый, чем книга по экспериментальной физике. Последовательное исключение внесения собственного эго и мнения в анализ требует, как ни странно, огромного объема работы. Латинская пословица ars est celare artem (искусство – в умении скрыть искусство) звучит типично (для латыни) кратко, загадочно и немного запутанно. (Является ли латынь своего рода языком APL по сравнению с современными естественными языками?) Это выражение дословно переводится как «искусство должно скрывать искусство», но глубокий смысл ближе к фразе «истинное искусство не является подчеркнуто вычурным». Настоящие художники никогда не оставляют отпечатков пальцев на своих работах. В самом конкретном смысле это и было целью написания книги «Современный C++: безопасное использование», поскольку вы не найдете в ней никаких особых мнений, проповедей или даже неоправданно витиеватых словесных конструкций. (Ожесточенные споры возникали по поводу идеального, максимально спартанского выбора слов в том или ином абзаце.) Я уверен, что такой отточенный язык книги оценит любой читатель.

Особая дополнительная привлекательность

«Единственный вид письма – это переписывание», – гласит известная цитата. Это вдвойне верно для технических книг. Качество любого (технического) учебника заключается в готовности его авторов переделывать свою работу, а также в глубине и широте группы его рецензентов, поддерживающей процесс пересмотра. Но переписать книгу не просто! Вы когда-нибудь писали код, а затем отказывались от его пересмотра, потому что он вам чрезвычайно нравился? Умножьте это чувство на 1024, и вы узнаете, как авторы книг относятся к переписыванию тех фрагментов, в которые они уже вложили свою душу. Вы действительно должны быть приверженцем повышения качества, чтобы сохранить спокойствие во время такого испытания.

То, что авторы настаивают на качестве, напоминает о том, что мне больше всего нравится в этой книге и что в то же время труднее всего объяснить. Я называю это дополнительной привлекательностью.

Я заметил кое-что в действительно великих трудах – будь то инженерное дело, искусство, спорт или любая другая сложная человеческая деятельность. Почти всегда отличная работа – это результат усилий талантливых людей, выходящих за рамки того, что можно было бы считать разумным. Высоко оценивая такую работу, мы неявно признаем огромные возможности в сочетании с соответствующими невероятными усилиями по их реализации. Хорошую работу можно сделать быстро, но великий труд – невозможно.

По странному стечению обстоятельств я в конце концов увлекся этой книгой – во-первых, из-за одной рецензии. Потом читал снова и снова – всего четыре полных прохода по всей книге. Стремление к совершенству так же заразительно, как склонность к неряшливости, но несравненно интереснее. («Уничтожить!» Джон Лакос (John Lakos) терпеливо присылал мне по электронной почте каждую новую редакцию. Мои рецензии, часто весьма едкие, мотивировали его как ничто другое.) Другие рецензенты – представители Комитета по стандартам C++, отраслевые эксперты по C++, эксперты по C++03, ранее незнакомые с версиями стандарта C++1x, эксперты по программной архитектуре, эксперты по многопоточности, эксперты по процессам, даже эксперты по LaTeX – сделали то же самое, в результате чего каждое предложение, которое вы прочтете, подвергалось критическому анализу десятки раз и, вероятно, столько же раз переписывалось. Со своей стороны я настолько увлекся проектом и бескомпромиссным подходом авторов к качеству, что в итоге написал для книги полноценный материал. (Любая ошибка в разделе 2.1 «Шаблоны функций с переменным числом аргументов» – моя вина.) Создание этой книги потребовало огромного объема работы, большего, чем я мог ожидать, – это все, на что я мог надеяться. Я считал, что стал слишком стар, чтобы продолжать работать всю ночь, но, очевидно, ошибся.

При таком активном участии я могу утверждать: в этой книге действительно заложена дополнительная привлекательность. Обсуждение идет, как и должно, никакой лишней чепухи, примеры кода точны и красноречивы. Я считаю, что «Современный C++: безопасное использование» – великолепная работа. Надеюсь, что помимо извлечения практических уроков из этой книги вы черпаете из нее вдохновение, для того чтобы добавить больше привлекательности в собственную работу. Я добавил – это я точно знаю.

– *Андрей Александреску* (Andrei Alexandrescu),
май 2021 г.

Об авторах

Джон Лакос (John Lakos), автор книг «Large-Scale C++ Software Design» (Addison-Wesley, 1996) и «Large-Scale C++ Volume I: Process and Architecture» (Addison-Wesley, 2020), работает в компании Bloomberg в Нью-Йорке в качестве старшего архитектора и инструктора по разработке программного обеспечения на C++ по всему миру. Он также является активным членом с правом голоса рабочей группы по развитию языка Комитета по стандартам C++. С 1997 по 2001 г. доктор Лакос руководил проектированием и разработкой инфраструктурных библиотек для собственных аналитических финансовых приложений в Bear Stearns. С 1983 по 1997 г. доктор Лакос трудился в компании Mentor Graphics, где разрабатывал крупные программные платформы и продвинутые приложения ICCAD, при создании которых он получил несколько патентов на программное обеспечение. Джон Лакос получил докторскую степень в области информатики в 1997 г. и степень доктора наук по электротехнике (1989 г.) в Колумбийском университете. Доктор Лакос получил степень бакалавра в Массачусетском технологическом институте по математике (1982 г.) и информатике (1981 г.).

Витторио Ромео (Vittorio Romeo) (бакалавр информатики, 2016 г.) – старший инженер-программист в компании Bloomberg в Лондоне, где он создает особо важное ПО среднего звена на C++ и обучает сотни сотрудников программированию на современном C++. Он начал программировать в возрасте 8 лет и сразу увлекся C++. Витторио создал несколько библиотек и игр на C++ с открытым исходным кодом, опубликовал множество видеокурсов и учебных пособий и активно участвует в процессе стандартизации ISO C++. Он является активным членом сообщества C++ с горячим желанием делиться своими знаниями и учиться у других: более 20 раз выступал на международных конференциях C++ (включая CppCon, C++Now, ++it, ACCU, C++ On Sea, C++ Russia и Meeting C++) по разнообразным темам: от разработки игр до метапрограммирования шаблонов. У Витторио есть собственный веб-сайт (**https://vittorioromeo.info/**) с продвинутыми статьями и материалами по C++ и канал на YouTube (**https://www.youtube.com/channel/UC1XihgHdkNOQd5IBHnlZWbA**), где размещены широко известные современные руководства по C++11/14. Витторио является активным участником StackOverfow, уделяя большое внимание ответам на интересные вопросы по C++ (его репутация 75k+). В свободное от написания кода время Витторио увлекается тяжелой атлетикой и фитнесом, а также компьютерными играми и научно-фантастическими фильмами.

Ростислав Хлебников (Rostislav Khlebnikov) является руководителем группы BDE Solutions, которая работает над различными библиотеками BDE, такими как библиотека для связи по протоколу HTTP/2, и участвует в других проектах, включая улучшение взаимодействия библиотек BDE с типами словарей стандартной библиотеки. Он является активным членом Комитета по стандартам C++ и участвовал в конференции CppCon 2019. До работы в Bloomberg доктор Хлебников получил степень бакалавра в области прикладной математики и информатики в Санкт-Петербургском государственном политехническом университете (Россия) и степень доктора информатики в Технологическом университете Граца (Австрия). Ростислав профессионально работал инженером-программистом на C++ более 15 лет.

Алисдар Мередит (Alisdair Meredith) был членом Комитета по стандартам C++ с момента создания C++11 на заседании комитета в Оксфорде в 2003 г., уделяя особое внимание интеграции функциональных возможностей и активному поиску и устранению несоответствий в языке. Алисдар был председателем LWG, когда были опубликованы стандарты C++11 и C++14, и он отдает должное напряженной работе предыдущего председателя, Ховарда Хиннанта (Howard Hinnant). Алисдар был постоянным докладчиком на конференциях в течение почти 15 лет, представляя новые работы комитета по стандартам C++. Алисдар присоединился к группе Bloomberg BDE в 2009 г. В течение десяти лет до этого Алисдар работал профессиональным программистом приложений на C++ в автогонках Формула-1 с командами Benetton и Renault, выиграв вместе с ними два чемпионата мира. Между чемпионатами Алисдар около года проработал менеджером по выпуску ПО в Borland, занимаясь маркетингом продуктов компании на C++. Алисдар любит путешествия, вкусные обеды и подводное плавание с дыхательной трубкой.

Глава 0

Экскурс в компьютерные системы

Приветствуем всех читателей! Книга «Современный C++: безопасное использование» – справочное руководство, предназначенное для профессионалов, разрабатывающих и сопровождающих крупномасштабные сложные программные системы, написанные на языке C++, и желающих в полной мере овладеть современными функциональными возможностями C++.

В этой книге основное внимание уделяется продуктивной ценности каждой новой функциональной возможности языка, начиная с C++11, особенно когда рассматриваемые системы и организации разбираются с точки зрения масштабирования. Мы намеренно оставили в стороне основные принципы и идиомы языка, – какими бы остроумными и интеллектуально привлекательными они ни выглядели, – которые могут повредить конечному результату применительно к крупномасштабным системам. Вместо этого мы сосредоточились на принятии разумных экономических и проектных решений с учетом неизбежных компромиссов, возникающих в любой инженерной дисциплине. При этом мы стараемся избегать субъективных мнений и рекомендаций.

Как известно, Ричард Фейнман (Richard Feynman) сказал: «Если что-либо не соответствует эксперименту, это неправильно. В этом простом утверждении – ключ к науке»[1]. Нет лучшего способа поэкспериментировать с функциональной возможностью языка, чем позволить времени сделать свою работу. Мы серьезно обдумали этот принцип и решили рассматривать только те функциональные возможности современного C++, которые были частью стандарта не менее пяти лет, что, по нашему мнению, дает достаточную перспективу для правильной оценки практического влияния новых функциональных возможностей. Таким образом, мы можем опираться на практический опыт, чтобы обеспечить тщательное и всестороннее изложение материала, учитывая ограниченное время для профессионального развития большинства читателей. Если вы ищете способы повышения производительности с применением надежных и действительно современных функциональных возможностей C++, то мы надеемся, что эта книга станет необходимым источником информации, к которому вы будете обращаться снова и снова.

То, чего нет в книге, так же важно, как и то, что в ней есть. Книга «Современный C++: безопасное использование» («Embracing Modern C++ Safely», известная также в виде аббревиатуры EMC++S) не является учебным пособием по программированию на C++ или даже справочником по новым функциональным возможностям C++. Мы предполагаем, что вы опытный разработчик, руководитель группы или менеджер, а также что вы уже в полной мере освоили классический C++98/03 и теперь ищете четко определенные, целенаправленные способы интеграции современных функциональных возможностей C++ в свой набор инструментов.

0.1. Чем эта книга отличается от других

Цель книги, которую вы сейчас читаете, – всегда оставаться объективной, эмпирической и практической. Мы просто представляем функциональные возможности, их при-

[1] Ричард Фейнман, лекция в Корнеллском университете (Cornell University), 1964 г. Видеозапись и комментарии доступны здесь: **https://fs.blog/2009/12/mental-model-scientific-method**.

менимость и потенциальные подводные камни, что отражено в анализе миллионов человеко-часов использования C++11 и C++14 при разработке различных крупномасштабных программных систем, а вопросы личных предпочтений были оставлены в стороне по мере наших возможностей. Мы изложили оставшуюся в итоге чистую истину, которая должна сформировать ваше понимание того, что может предложить современный C++, без воздействия наших субъективных мнений или склонностей, зависящих от какой-либо предметной области.

Окончательный анализ и интерпретация того, что подходит для вашего конкретного случая, остается за вами, читатель. Эта книга по своему замыслу не является руководством по стилю C++ или стандартам кодирования, тем не менее она обеспечивает ценный вклад для любой организации, разрабатывающей программное обеспечение, стремящейся создать или улучшить процесс разработки.

Практичность важна для нас в реальном экономическом смысле. Мы рассматриваем современные возможности C++ с точки зрения крупной компании, разрабатывающей и использующей программное обеспечение в конкурентной среде. Помимо того что мы показываем вам, как лучше всего использовать конкретную функциональную возможность языка C++ на практике, наш анализ учитывает накладные расходы, связанные с постоянным использованием этой функциональной возможности в экосистеме организации, занимающейся разработкой программного обеспечения. В большинстве работ не учитываются накладные расходы на использование особенностей языка. Другими словами, мы сопоставляем преимущества успешного использования функциональной возможности со скрытыми накладными расходами на ее постоянное неэффективное использование (или некорректное использование) и/или с накладными расходами, связанными с обучением и проверкой кода, необходимыми для разумной гарантии того, что такого неэффективного и некорректного использования не произойдет. Мы прекрасно понимаем: то, что подходит для одного человека или небольшой команды единомышленников, весьма отличается от того, что применимо к большой распределенной группе. Результатом этого анализа является наша сигнатурная классификация функциональных возможностей, основанная на степени безопасности их практического применения, а именно: безопасные, условно безопасные и небезопасные функциональные возможности.

Нам неизвестна ни одна подобная работа из обширного набора предлагаемых учебных и справочных руководств по C++. Мы написали эту книгу, потому что она была нам нужна.

0.2. Общее содержание первого издания

Учитывая огромный объем и без того обширных и быстрорастущих стандартизированных библиотек C++, мы решили ограничить тематику этой книги только функциональными возможностями самого языка. Дополняющая комплект книга «Безопасное использование современных стандартных библиотек C++» – это отдельный проект, которым мы надеемся заняться в будущем. Но чтобы быть действительно эффективной, эта книга должна оставаться сосредоточенной на том, что опытные разработчики C++ должны хорошо знать, чтобы добиться успеха здесь и сейчас.

Мы решили ограничить объем этого первого издания только теми функциональными возможностями, которые были включены в стандарт языка начиная с C++11 и широко доступны для практического применения в течение как минимум пяти лет. Данное ограничение позволяет нам лучше оценить действительное воздействие этих функциональных возможностей и особо выделить любые затруднения, которые, возможно, не предполагались до стандартизации и постоянного, активного и широкомасштабного использования в производственной среде.

Предполагается, что читатель хорошо знаком практически со всеми основными и важными специальными возможностями классического C++98/03, поэтому в данной книге мы сосредоточим внимание только на подмножестве возможностей языка, представленных в стандартах C++11 и C++14. Эта книга лучше всего подходит для тех, кому необходимо узнать, как в современных условиях с сохранением безопасности вклю-

чить функциональные возможности языка C++11/14 в уже существующую кодовую базу C++98/03.

Мы планируем подробно рассматривать материалы, предшествующие C++11, в будущих версиях издания. Однако в настоящее время мы настоятельно рекомендуем книгу «Effective C++» Скотта Мейерса (Scott Meyers)[2] как краткое практическое описание многих важных и полезных возможностей C++ 98/03.

0.3. Основополагающие принципы изложения материала в этой книге

При написании книги «Современный C++: безопасное использование» мы соблюдали ряд основополагающих принципов, которые в совокупности определяют стиль и содержание этой книги.

0.3.1. Факты, а не мнения

Эта книга описывает только полезное практическое применение и потенциальные опасности современных функциональных возможностей C++. Представленное здесь содержание основано на объективно проверяемых фактах, полученных либо из документов стандартов, либо из обширного практического опыта. Мы недвусмысленно избегаем субъективных мнений об относительных преимуществах компромиссов проектирования (сдержанность, которая является хорошим упражнением в смирении). Хотя подобные мнения часто бывают ценными, они по своей сущности имеют тенденцию отклоняться в сторону области знаний автора.

Обратите внимание: безопасность – критерий, который мы используем для разделения функциональных возможностей по главам, – является единственным исключением из этого основополагающего принципа объективности. Хотя анализ каждой функциональной возможности стремится к полной объективности, классификация каждой функциональной возможности по главам, указывающая на относительную безопасность ее повседневного использования в крупной среде разработки программного обеспечения, отражает наш общий реальный практический опыт, накопленный в течение десятилетий разработки разнообразных крупномасштабных программных систем на языке C++.

0.3.2. Объяснение, а не указание

Мы намеренно избегаем строго регламентированных предписаний каких-либо решений для устранения затруднений и потенциальных опасностей конкретных функциональных возможностей. Вместо этого мы просто описываем и характеризуем подобные проблемы достаточно подробно, чтобы предоставить вам возможность разработать решение, подходящее для вашей собственной среды разработки. В некоторых случаях мы можем ссылаться на некоторые методики или общедоступные библиотеки, которые другие разработчики использовали для обхода таких «лежачих полицейских», но мы не выносим собственный вердикт о том, какой обходной путь следует считать наилучшим.

0.3.3. Скрупулезно, а не поверхностно

Книга «Современный C++: безопасное использование» не была задумана и не предназначена для введения в современный C++. Эта книга является удобным справочным руководством для опытных программистов на C++, знакомых с более ранними версиями языка (C++98/03). Наша цель – предоставить вам факты, подробный объективный анализ и убедительные примеры из реальной практики. Тем самым мы избавляем вас от необходимости во всех подробностях изучать материал, который, как мы предполагаем, вы уже знаете. Если вы совсем незнакомы с языком C++, мы предлагаем вам на-

[2] **meyers92**. Есть переводы на русский язык в виде серии книг «Эффективное использование C++».

чать с более простой книги, которая подробно описывает именно этот язык, например «The C++ Programming Language» Бьярна Страуструпа (Bjarne Stroustrup)[3].

0.3.4. Примеры из реальной практики, а не придуманные

Мы надеемся, что вы найдете примеры, приведенные в этой книге, полезными во многих отношениях. Основная цель примеров – продемонстрировать эффективное использование каждой функциональной возможности, как это может происходить на практике. Мы избегаем искусственно придуманных примеров, которые придают одинаковое значение как редко используемым аспектам, так и предполагаемому естественному использованию функциональной возможности. Таким образом, многие из наших примеров основаны на упрощенных фрагментах кода, извлеченных из реальных кодовых баз. Несмотря на то что мы обычно меняем имена идентификаторов, чтобы они больше соответствовали сокращенному примеру (а не контексту и процессу, являющимся источником конкретного примера), мы сохраняем структуру кода каждого примера как можно ближе к исходному, реальному аналогу.

0.3.5. Масштабируемые, но не слишком упрощенные программы

Как и во многих аспектах разработки программного обеспечения, то, что работает для небольших программ и групп, часто не масштабируется при более крупных объемах разработки. Мы пытаемся одновременно охватить две различные характеристики размера: (1) чистый размер программного продукта (например, в байтах, строках исходного кода, отдельных модулях релиза): программ, систем и библиотек, разработанных и сопровождаемых организацией, выпускающей программное обеспечение, и (2) размер самой организации, измеряемый количеством разработчиков программного обеспечения, инженеров по обеспечению качества, инженеров по обеспечению надежности, операторов и т. п., работающих в конкретной организации.

Более того, мощные новые функциональные возможности языка в руках нескольких опытных программистов, работающих вместе над прототипом для собственного только что созданного стартапа, не всегда работают так же успешно, когда бессмысленно используются десятками или сотнями разработчиков в крупной организации, занимающейся разработкой программного обеспечения. Таким образом, когда мы рассматриваем относительную безопасность функциональной возможности в соответствии с определением, приведенным в следующем разделе, мы делаем это с полным осознанием того, что любая конкретная функциональная возможность может быть использована – не всегда правильно – в больших программах и большим числом программистов, обладающих широким диапазоном знаний, умений и навыков.

0.4. Что в действительности означает слово «безопасно»

Комитет ISO по стандартам C++, членами которого мы являемся, поступил бы весьма неосмотрительно – и был бы совершенно неправ, – если бы позволил стандартизировать какую-либо функциональную возможность языка C++, не являющуюся абсолютно безопасной при использовании по назначению. Тем не менее мы выбрали слово «безопасно» в качестве обозначения весьма значимой характеристики нашей книги и методики, с помощью которой мы оцениваем соотношение риска и выгоды при использовании данной функциональной возможности в крупномасштабной среде разработки. Употребляя в этом контексте термин «безопасный», мы применяем его к реальной экономике, в которой все имеет свою стоимость в нескольких измерениях: риск неправильного использования, дополнительная обязанность по сопровождению, связанная с применением новой функциональной возможности в старой кодовой базе, и необходимое обучение разработчиков, которые, возможно, незнакомы с новым свойством.

[3] **stroustrup13**. Есть перевод этой книги на русский язык: *Страуструп Б.* Язык программирования C++.

Несколько факторов влияют на повышение ценности вследствие принятия и широкомасштабного применения любой новой функциональной возможности языка, тем самым снижая уровень ее внутренней безопасности. Классифицируя функциональные возможности с точки зрения безопасности, мы стремимся найти разумно сбалансированную комбинацию следующих факторов:

- количество и реальную опасность известных недостатков;
- сложность обучения устойчивому правильному использованию;
- уровень опыта, необходимый для устойчивого правильного использования;
- риски, связанные с широкомасштабным неправильным использованием.

В этой книге степень безопасности конкретной функциональной возможности определяется как относительная вероятность того, что ее широкомасштабное использование окажет положительное воздействие и не повлияет отрицательно на кодовую базу крупной компании-разработчика программного обеспечения.

0.5. Безопасная функциональная возможность

Некоторые из новых функциональных возможностей современного C++ обладают значительной полезностью, просты в использовании, и ими чрезвычайно трудно воспользоваться некорректно по неосторожности, следовательно, повсеместное внедрение таких функциональных возможностей вполне оправдано – относительно маловероятно, что они станут причинами проблем в контексте крупномасштабной организации, занимающейся разработкой ПО, – и, как правило, поощряется – даже без обучения. Мы определяем такие стабильно полезные и надежные функции C++ как безопасные.

Например, мы классифицируем контекстное ключевое слово `override` как безопасную функциональную возможность, поскольку она предотвращает ошибки, служит в качестве средства документирования, не может быть с легкостью использована не по назначению и вообще не имеет серьезных недостатков. Если кто-то слышал что-либо об этой функциональной возможности и пытался ее использовать, и при этом программное обеспечение нормально компилируется, то кодовая база, вероятнее всего, благодаря этому улучшается. Использование `override` везде, где это применимо, всегда является разумным инженерным решением.

0.6. Условно безопасная функциональная возможность

Подавляющее большинство новых функциональных возможностей, доступных в современном C++, имеют важное, часто встречающееся и полезное применение, но правильный способ их использования, не говоря уже об оптимальности, может оказаться неочевидным. Более того, в некоторых из этих функциональных возможностей скрываются внутренние опасности и недостатки, требующие тщательного обучения и особой осторожности, чтобы обойти эти подводные камни.

Например, мы считаем инициализаторы членов по умолчанию условно безопасной функциональной возможностью, потому что, несмотря на то что сами по себе они просты в использовании, возможно, менее очевидные непредвиденные последствия их применения (например, сильная связанность во время компиляции) могут становиться непомерно дорогостоящими при определенных условиях (например, может помешать установке исправлений в случае выполнения только повторной компоновки (линковки) в реальной рабочей среде).

0.7. Небезопасная функциональная возможность

Когда опытный программист надлежащим образом использует какую-либо функциональную возможность C++, она обычно не наносит прямого вреда. Но другие разработчики, увидев ее использование в кодовой базе, однако будучи не в состоянии оценить узкоспециализированные или конкретизированные обоснования ее применения, мо-

гут попытаться использовать ее аналогичным образом, как им может показаться, но с гораздо менее ожидаемыми результатами. Точно так же специалисты, занимающиеся сопровождением ПО, могут изменить способ использования такой не вполне надежной функциональной возможности, нарушив ее семантику малозаметным, но разрушительным образом.

Для функциональных возможностей, которые классифицируются как небезопасные, могут существовать допустимые и даже важные варианты использования, но наш практический опыт показывает, что их постоянное и/или повсеместное использование было бы нецелесообразным в типовой крупномасштабной организации, занимающейся разработкой программного обеспечения.

Например, мы считаем контекстное ключевое слово `final` небезопасной функцией, поскольку ситуаций, в которых оно может быть использовано не по назначению, в подавляющем большинстве случаев больше, чем всего лишь нескольких отдельных вариантов, в которых оно уместно, не говоря уже о его полезности. Кроме того, его широкомасштабное применение будет препятствовать детализированному (например, иерархическому) повторному использованию, которое весьма важно для успешной работы крупной организации.

0.8. Каталог функциональных возможностей современного C++

Это первое издание книги «Современный C++: безопасное использование» было разработано таким образом, чтобы служить исчерпывающим каталогом функциональных возможностей языка C++11 и C++14, предоставляя весьма важную информацию для каждой из них в ясном, последовательном и предсказуемом формате, к которому опытные инженеры могут с легкостью обращаться во время разработки или обсуждения технических вопросов.

0.8.1. Как организована эта книга

Эта книга разделена на четыре главы, из которых последние три составляют каталог функциональных возможностей современного языка C++, сгруппированных по соответствующей классификационной степени их безопасности:

- глава 0 «Введение»;
- глава 1 «Безопасные функциональные возможности»;
- глава 2 «Условно безопасные функциональные возможности»;
- глава 3 «Небезопасные функциональные возможности».

В этом первом издании главы о функциональных возможностях языка (1, 2 и 3) включают два раздела, содержащих, соответственно, функции C++11 и C++14, имеющие уровень безопасности (безопасные, условно безопасные или небезопасные), соответствующий конкретной главе. Но еще раз напомним, что функциональные возможности стандартной библиотеки не относятся к теме этой книги.

Каждая функция представлена в отдельном разделе, оформленном в традиционном стандартном формате:

- *Описание* – краткое, но исчерпывающее введение в синтаксис и семантику функциональной возможности, дополненное подробными фрагментами исходного кода. Мы делаем все возможное, чтобы не использовать другие новые функциональные возможности одновременно с описываемой, поэтому описание каждой функциональной возможности можно читать независимо от других и не в порядке их расположения в книге. Иногда это может привести к тому, что код станет менее понятным, чем мог бы быть при последовательном чтении. Обязательно ознакомьтесь с разделом «Смотрите также» (который описан ниже), чтобы узнать о взаимном влиянии и потенциальных конфликтах между функциональными возможностями;

- *Варианты использования* – совокупность проверенных на практике вариантов использования, взятых из библиотек и приложений;
- *Потенциальные опасности* – случаи некорректного использования функциональной возможности, которые могут привести к серьезным ошибкам и другим проблемам;
- *Недостатки* – недостатки функциональной возможности и ее отрицательные свойства, которые могут сделать ее менее приемлемой для использования;
- *Смотрите также* – перекрестные ссылки на другие связанные функциональные возможности, описанные в этой книге, с кратким описанием данной взаимосвязи;
- *Материал для дополнительного чтения* – ссылки на внешние источники, описывающие данную функциональную возможность.

Ограничение представления каждой отдельной функциональной возможности этим стандартизированным форматом облегчает быстрый переход к любым конкретным особенностям данной функциональной возможности языка, которые вы ищете.

Мы ссылаемся на каждую функциональную возможность в соответствующей главе и разделе: например, в разделе 1.1 «Синтаксис атрибутов» указано, что функциональная возможность «Атрибуты» находится в главе 1 (т. е. она безопасная) и в разделе 1 (C++11).

Стиль комментариев заслуживает особого внимания, потому что он предоставляет полезную информацию в лаконичной форме. Обратите внимание: «описание» или «подробности» предоставляют дополнительную описательную информацию. Заполнители для неприменимого и/или нерекомендуемого кода отображаются особым стилем комментариев – одним из следующих способов:

```
/*...*/
// ...
// ...                          (<описание>)
```

Код, который не компилируется, будет помечен с помощью одного из следующих типов комментариев:

```
// Ошибка компиляции.
// Ошибка компиляции: <подробности>
```

Код, который не линкуется (не связывается), будет помечен с помощью одного из следующих типов комментариев:

```
// Ошибка на этапе связывания.
// Ошибка на этапе связывания: <подробности>
```

Код, который во время выполнения ведет себя не так, как ожидалось, будет помечен с помощью одного из следующих типов комментариев:

```
// Ошибка при выполнении.
// Ошибка при выполнении: <подробности>
```

Код, который выполняется так, как ожидалось, будет помечен с помощью одного из следующих типов комментариев:

```
// OK.
// OK: <подробности>
```

Код, который может выдать предупреждение, но ведет себя ожидаемым образом, будет помечен как «OK, но может выдавать предупреждение» или аналогичным образом. Например, если функциональная возможность объявлена устаревшей в версии C++17 и удалена в версии C++20, мы можем комментировать ее следующим образом:

```
// OK: не рекомендуется[4] (возможен вывод предупреждения)
```

[4] Удалена в версии C++20.

0.9. Как использовать эту книгу

В зависимости от ваших потребностей пользу из чтения книги «Современный C++: безопасное использование» можно извлечь несколькими способами.

- Последовательное чтение всей книги от корки до корки. Если вы знакомы с классическим C++, то последовательное чтение всей книги обеспечит полное и глубокое практическое понимание каждой из функциональных возможностей языка, введенных в версиях C++11 и C++14.
- Чтение глав по порядку, но в более медленном темпе. Также возможен и даже рекомендуется поэтапный подход, основанный на приоритетах тем, особенно если вы чувствуете себя в них неуверенно. Понимание и применение в первую очередь безопасных функциональных возможностей, описанных в главе 1, не вызовет особых затруднений. Со временем условно безопасные функциональные возможности из главы 2 позволят вам освоить все разнообразие полезных современных возможностей языка C++, отдавая приоритет тем, которые с наименьшей вероятностью окажутся проблематичными.
- В первую очередь чтение разделов C++11 в каждой из трех глав-каталогов. Если вы являетесь разработчиком в организации, которая использует C++11, но пока еще не перешла на C++14, то можете сосредоточиться на изучении всего, что можно применять прямо сейчас, а затем вернуться назад и изучить остальной материал позже, когда он станет актуальным для вашей развивающейся организации.
- Использование книги как краткого справочного руководства, когда это необходимо. Произвольный порядок тоже вполне приемлем, особенно теперь, когда вы прочли главу 0. Если вы предпочитаете не штудировать книгу целиком (или просто хотите периодически обращаться к ней, освежая знания), то выборочное чтение любого раздела по отдельным темам в любом порядке обеспечит оперативный доступ ко всем важным подробностям о любой функциональной возможности, которая интересна для вас в данный момент.

Мы верим, что вы извлечете пользу из знаний, которые мы вложили в книгу «Современный C++: безопасное использование», вне зависимости от того, как вы ее читаете. Помимо помощи в том, чтобы помочь вам стать более знающим и, следовательно, более безопасным разработчиком, эта книга предназначена для выяснения (при этом не важно, являетесь ли вы разработчиком, руководителем или менеджером), какие функции требуют дополнительной подготовки, внимания к подробностям, опыта, экспертной оценки и т. п. Основанный на реальных фактах, объективный стиль изложения также вносит немалый вклад в подготовку стандартов кодирования и руководств по стилю, которые соответствуют конкретным потребностям компании, проекта, группы или даже единственного весьма разборчивого разработчика (которым, разумеется, каждый из нас хочет стать). Наконец, любая организация, занимающаяся разработкой программного обеспечения на C++, которая принимает на вооружение эту книгу, сделает первые шаги к использованию современного C++ таким образом, чтобы извлечь максимальную выгоду при минимальных рисках, т. е. путем безопасного применения современного C++.

И последнее по порядку изложения, но определенно не менее важное: это ваша книга во всех смыслах этого слова. Это была совместная работа с участием многих инженеров, таких как вы, и она была «разработана с учетом дальнейшего сопровождения», потому что в будущем мы планируем пересмотренные редакции с новыми функциональными возможностями и улучшенным описанием ныне существующих. Эти будущие редакции могли бы извлечь огромную пользу из вашего участия. Вы обнаружили что-то неправильное или пропущенное? Знаете рациональный вариант использования? Нашли скрытую потенциальную опасность? Недостаток, с которым невозможно смириться? Мы с удовольствием добавим вашу информацию в книгу. Перейдите в браузере на сайт **http://emcpps.com** и следуйте инструкциям, чтобы отправить нам отзыв. Ваше участие всегда приветствуется. Более подробную информацию о книге вы найдете на том же сайте. Благодарим вас и желаем удачного кодирования!

Глава 1

Безопасные функциональные возможности

Современный C++ может многое предложить. Многие современные функциональные возможности языка имеют реальную практическую ценность. Они обеспечивают повышенную производительность и надежность, их легко понять и применить, и их трудно случайно использовать не по назначению. В этой главе представлены те возможности C++11 и C++14, которые имеют немногочисленные, малозначительные и легко узнаваемые скрытые потенциальные опасности, что позволяет с легкостью избежать их неправильного использования. Кроме того, эти функциональные возможности вносят минимальный системно значимый риск при широком внедрении преимущественно в кодовую базу C++03, что позволяет организовать обучение как факультативное, а не обязательное. Руководство организации может чувствовать себя достаточно спокойно, ориентируя всех своих инженеров на использование функциональных возможностей, представленных в этой главе.

В первую очередь безопасные функциональные возможности характеризуются низкой степенью риска. Вспомните, что в разделе «Безопасная функциональная возможность» в главе 0 функция `override` (раздел 0.5) была особо выделена как воплощение безопасной функциональной возможности. Хотя она применима только в контексте наследования и виртуальных функций, ее практически невозможно использовать некорректно, не получая никакой пользы. Еще один пример функциональной возможности с низким уровнем риска, но с исключительно высокой положительной отдачей – `static_assert` (раздел 1.15). Эта конкретная функциональная возможность настолько полезна и защищена от непреднамеренного неправильного использования, что мы широко используем ее на протяжении всей книги, чтобы продемонстрировать важные свойства времени компиляции, почти так же, как если бы это была функциональная возможность C++03. Хотя не все функциональные возможности, представленные в этой главе, столь же полезны или широко применимы, как эти две, все они могут использоваться с минимальным риском и поэтому считаются безопасными.

Короче говоря, широкое внедрение безопасных функциональных возможностей – это предложение с низким уровнем риска. Все эти функциональные возможности легко понять и применять с пользой, их трудно использовать не по назначению, следовательно, формальное обучение обычно не требуется. Организации не нужно беспокоиться о внедрении безопасных функциональных возможностей в кодовую базу с преобладанием C++03, поддерживаемую в основном теми, кто почти незнаком с современными функциональными возможностями. Если вы плохо знакомы с возможностями современного C++, обязательно начните с этой главы.

1.1. Поддержка обобщенных атрибутов

Новый синтаксис для описательного комментирования кода с использованием атрибутов обеспечивает переносимое представление дополнительной информации для различных реализаций компилятора и внешних инструментальных средств.

1.1.1. Описание

Разработчикам часто известна информация, которую невозможно легко вывести непосредственно из исходного кода в конкретном элементе трансляции. Часть этой информации может быть полезна для некоторых компиляторов, например для диагностики или оптимизации, но обычные атрибуты не предназначены для того, чтобы как-либо влиять на семантику правильно написанной программы. Под семантикой здесь мы обычно подразумеваем любое наблюдаемое поведение, за исключением производительности во время выполнения. Как правило, игнорирование атрибута является допустимым и безопасным вариантом выбора для компилятора. Но иногда атрибут не воздействует на поведение корректной программы, но может повлиять на поведение программы с правильным синтаксисом, но некорректной по смыслу (см. подраздел 1.1.2.3). Также могут оказаться полезными специализированные аннотации, ориентированные на внешние инструментальные средства.

1.1.1.1. *Синтаксис атрибутов C++*

C++ поддерживает стандартный синтаксис для атрибутов, представляемый с помощью согласованных пар квадратных скобок [[и]]. Самым простым является одиночный атрибут, представленный с использованием простого идентификатора, например `attribute_name`:

```
[[attribute_name]]
```

Одна аннотация может состоять из нуля и более атрибутов:

```
[[]]           // Разрешен в каждой позиции, где допустимым является любой атрибут.
[[foo, bar]]   // Равнозначен [[foo]] [[bar]].
```

Любой атрибут может иметь список аргументов, состоящий из произвольной последовательности элементов (токенов):

```
[[attribute_name()]]            // Атрибут с нулем аргументов.
[[deprecated("bad API")]]       // Атрибут с одним аргументом.
[[theoretical(1, "two", 3.0)]]  // Атрибут с несколькими аргументами.
[[complicated({1, 2, 3} + 5)]]  // Произвольные элементы[1].
```

Следует отметить, что некорректное число аргументов или несовместимый тип аргумента – это ошибка времени компиляции для всех атрибутов, определяемых по стандарту, но поведение для всех прочих атрибутов определяется реализацией (implementation-defined) (см. подраздел 1.1.3.1).

Любой атрибут может быть уточнен с помощью пространства имен атрибута[2], т. е. с использованием одного произвольного идентификатора:

```
[[gnu::const]] // (Особенность GCC) Атрибут const, уточненный с помощью пространства имен gnu.
[[my::own]]    // (Определенный пользователем) атрибут own, уточненный с помощью
               // пространства имен my.
```

1.1.1.2. *Размещение атрибутов C++*

Атрибуты можно размещать в различных локациях в соответствии с правилами грамматики C++. Для каждой такой локации стандарт определяет сущность или инструкцию, которой должен принадлежать атрибут. Например, атрибут перед простой инструкцией объявления принадлежит каждой объявляемой в ней сущности, тогда как атрибут, размещенный непосредственно после объявляемого имени, принадлежит только этой сущности:

---

[1] Компилятор GCC не обеспечивал поддержку произвольных элементов в атрибутах до версии GCC 9.3 (2020 г.).

[2] Атрибуты, имеющие имя, уточненное с помощью пространства имен, например `[[gnu::const]]`, получили только условную поддержку в стандартах C++11 и C++14, но и ранее поддерживались всеми основными компиляторами, включая Clang и GCC. Все компиляторы, соответствующие стандарту C++17, обязательно должны поддерживать пространства имен атрибутов.

```
[[foo]] void f(), g();      // Атрибут foo принадлежит обеим функциям f() и g().
void u(), v [[foo]] ();     // Атрибут foo принадлежит только v().
```

Атрибуты можно применять к сущности без имени (например, к анонимному объединению `union` или перечислению `enum`):

```
struct S { union [[attribute_name]] { int a; float b; }; };
enum [[attribute_name]] { SUCCESS, FAIL } result;
```

Допустимые позиции для любого конкретного атрибута ограничены только теми локациями, где этот атрибут принадлежит сущности, к которой он применяется. То есть, например, атрибут `noreturn`, который можно применить только к функциям, должен считаться корректным синтаксически, но не семантически, если используется для аннотации любого другого типа сущности или синтаксического элемента. Неправильное размещение стандартного атрибута приводит к грамматически неверной программе[3]:

```
void [[noreturn]] x() {}       // Ошибка, нельзя применять к типу.
     [[noreturn]] int i;       // Ошибка, нельзя применять к переменной.
     [[noreturn]] { throw; }   // Ошибка, нельзя применять к инструкции.
```

Последовательность спецификатора пустого атрибута `[[]]` допускается размещать в любом месте, где грамматика C++ разрешает атрибуты.

1.1.1.3. *Общие атрибуты, зависимые от компилятора*

До версии C++11 не существовало доступного стандартизированного синтаксиса для атрибутов, и вместо этого приходилось использовать непереносимые встроенные функции компилятора, такие как `__attribute__((fallthrough))` – это специфический синтаксис компилятора GCC. Учитывая новый стандартный синтаксис, разработчики компиляторов теперь могут выражать эти расширения синтаксически согласованным способом. Если во время компиляции встречается неизвестный атрибут, он игнорируется, и, вероятнее всего[4], будет выведено диагностическое сообщение о некритической ошибке.

В табл. 1.1.1.3.1 представлено несколько примеров широко распространенных атрибутов, зависимых от конкретного компилятора, которые уже были включены в стандарт или перенесены в стандартный синтаксис. Дополнительные атрибуты, зависимые от конкретного компилятора, см. в разделе 1.1.5.

Таблица 1.1.1.3.1. Некоторые стандартизированные атрибуты, зависимые от компилятора

Компилятор	Зависимый от компилятора	Соответствующий стандарту
GCC	`__attribute__((pure))`	`[[gnu::pure]]`
Clang	`__attribute__((no_sanitize))`	`[[clang::no_sanitize]]`
MSVC	`declspec(deprecated)`	`[[deprecated]]`

Переносимость – это самое большое преимущество практического использования стандартного синтаксиса, если он доступен для атрибутов, специально предназначенных для компилятора и внешних инструментальных средств. Поскольку большинство компиляторов просто игнорируют неизвестные атрибуты, использующие стандартный синтаксис (а начиная с версии C++17 они обязаны это делать), условная компиляция больше не требуется.

---

[3] На момент написания этой книги компилятор GCC работает небрежно и просто предупреждает, когда обнаруживает стандартный атрибут `noreturn` в недопустимой с точки зрения синтаксиса позиции, в то время как компилятор Clang вполне справедливо отказывается компилировать исходный код. Таким образом, использование даже стандартного атрибута может привести к различному поведению разных компиляторов.

[4] До версии C++17, соответствующей требованиям стандарта реализации компилятора, разрешалось рассматривать неизвестный атрибут как некорректно сформированный и немедленно завершить трансляцию, но, насколько известно авторам, никто из разработчиков этого не сделал.

1.1.2. Варианты использования

1.1.2.1. *Дополнительная полезная диагностика компилятора*

Добавление к сущностям определенных атрибутов может передавать компиляторам дополнительный контекст, достаточный для предоставления более подробной диагностики. Например, специфичный для GCC атрибут `[[gnu::warn_unused_result]]`[5] можно использовать для информирования компилятора и разработчиков о том, что возвращаемое функцией значение не должно игнорироваться[6]:

```
struct UDPListener
{
    [[gnu::warn_unused_result]] int start();
        // Инициализация фонового потока детектора-слушателя UDP, который может завершиться
        // аварийно по различным причинам. Возвращает 0 при успешном выполнении,
        // иначе – ненулевое значение.

    void bind(int port);
        // Поведение не определено, если инициализация не была завершена успешно.
};
```

Такая аннотация объявления, специально предназначенного для клиента, может предотвратить ошибки, возникающие, когда клиент забывает проверить результат функции[7]:

```
void init()
{
    UDPListener listener;
    listener.start();       // Возможен критический сбой; возвращаемое значение обязательно
                            // должно быть проверено.
    listener.bind(27015);   // Возможно неопределенное поведение. (ЭТО НЕВЕРНЫЙ ПОДХОД)
}
```

Для приведенного выше кода GCC выводит информативное предупреждающее сообщение:

```
warning: ignoring return value of 'int UDPListener::start()' declared
         with attribute 'warn_unused_result' [Wunusedresult]
         (игнорируется возвращаемое значение функции 'int UDPListener::start()',
         объявленной с атрибутом 'warn_unused_result' [Wunusedresult])
```

1.1.2.2. *Совет по дополнительным возможностям оптимизации*

Некоторые аннотации могут влиять на оптимизацию, выполняемую компилятором, в результате которой создаются более эффективные или имеющие меньший размер бинарные (выполняемые) файлы. Например, дополнение приведенной ниже функции `reportError` специфичным для GCC атрибутом `[[gnu::cold]]` (также доступным в компиляторе Clang) сообщает компилятору об уверенности разработчика в том, что эта функция вряд ли будет часто вызываться:

```
[[gnu::cold]] void reportError(const char* message) { /*...*/ }
```

При этом не только само определение `reportError` может быть оптимизировано по-другому (например, отдавая предпочтение более компактному размеру, а не скорости выполнения), но и любое использование этой функции, вероятнее всего, будет получать более низкий приоритет при прогнозировании направлений ветвления:

---

[5] Для совместимости с GCC компилятор Clang также поддерживает атрибут `[[gnu::warn_unused_result]]`.

[6] Атрибут `[[nodiscard]]` в стандарте C++17 предназначен для той же цели и является переносимым.

[7] Поскольку атрибут `[[gnu::warn_unused_result]]` не влияет на генерацию кода, он очевидно не является неправильным для клиента при использовании неаннотированного объявления и тем не менее компилирует соответствующее определение в контексте аннотированного или наоборот; но это не всегда справедливо для других атрибутов, и в любом случае наилучшая практическая методика может свидетельствовать в пользу согласованности.

```
void checkBalance(int balance)
{
    if (balance >= 0)   // Более вероятная ветвь.
    {
        // ...
    }
    else                // Менее вероятная ветвь.
    {
        reportError("Negative balance.");
    }
}
```

Поскольку в приведенном выше примере аннотированная функция `reportError(const char*)` располагается в ветви `else` инструкции `if`, компилятор предсказуемо ожидает, что `balance`, вероятнее всего, не будет отрицательным, следовательно, соответствующим образом оптимизирует прогнозируемое ветвление. Следует отметить, что даже если наша подсказка компилятору оказывается неправильной во время выполнения, семантика (смысл) каждой корректно написанной программы остается неизменной.

1.1.2.3. *Объявление явных предположений в коде для улучшения оптимизации*

Хотя присутствие атрибута обычно не воздействует на поведение любой синтаксически корректной программы каким-либо образом, кроме производительности во время выполнения, иногда атрибут передает компилятору информацию, и если она некорректна, то может изменить предполагаемое поведение программы. В качестве примера такой более действенной формы атрибута рассмотрим специфичный для компилятора GCC атрибут `[[gnu::const]]`, также доступный в Clang. Применяемый к функции, этот атрибут сообщает компилятору: предполагается, что эта функция является чистой (pure function), т. е. не имеет побочных эффектов (side effects). Другими словами, функция всегда возвращает одинаковое значение для заданного конкретного набора аргументов, и глобально доступное состояние программы не изменяется этой функцией. Например, функция, выполняющая линейную интерполяцию между двумя значениями, может быть снабжена аннотацией `[[gnu::const]]`:

```
[[gnu::const]]
double linearInterpolation(double start, double end, double factor)
{
    return (start * (1.0 - factor)) + (end * factor);
}
```

В более общем смысле для возвращаемого значения функции с аннотацией `[[gnu::const]]` не допускается зависимость от любого состояния, которое может измениться между ее последовательными вызовами. Например, не разрешается просмотр содержимого памяти, предоставленной функции по адресу. С другой стороны, функциям с похожей, но менее строгой аннотацией `[[gnu::pure]]` разрешается возвращать значения, которые зависят от любого долговременно сохраняющегося состояния. Следовательно, такие функции, как `strlen` или `memcmp`, которые читают, но не изменяют наблюдаемое состояние, могут иметь аннотацию `[[gnu::pure]]`, но не `[[gnu::const]]`.

Показанная ниже функция `vectorLerp` выполняет линейную интерполяцию (сокращенно обозначаемую как LERP) между двумя двумерными векторами. Тело этой функции содержит два вызова функции `linearInterpolation`, определенной в приведенном выше примере, – по одному для каждой компоненты вектора:

```
Vector2D vectorLerp(const Vector2D& start, const Vector2D& end, double factor)
{
    return Vector2D(linearInterpolation(start.x, end.x, factor),
                    linearInterpolation(start.y, end.y, factor));
}
```

Если значения этих двух компонент одинаковы, компилятору разрешается вызвать `linearInterpolation` только один раз, даже если тело этой функции не является видимым в единице трансляции `vectorLerp`:

```
// Псевдокод (гипотетически предполагаемое преобразование компилятора).
Vector2D vectorLerp(const Vector2D& start, const Vector2D& end, double factor)
{
    if (start.x == start.y && end.x == end.y)
    {
        const double cache = linearInterpolation(start.x, end.x, factor);
        return Vector2D(cache, cache);
    }

    return Vector2D(linearInterpolation(start.x, end.x, factor),
                    linearInterpolation(start.y, end.y, factor));
}
```

Если реализация функции, помеченная атрибутом `[[gnu::const]]`, не соблюдает ограничения, накладываемые этим атрибутом, но компилятор не имеет возможности обнаружить эту ошибку, то, вероятнее всего, в итоге эта ошибка проявится во время выполнения – см. подраздел 1.1.3.2.

1.1.2.4. *Использование атрибутов для управления анализом внешней статической компоновки*

Поскольку неизвестные атрибуты игнорируются компилятором, для внешних инструментальных средств статического анализа существует возможность определить собственные специализированные атрибуты, которые могут использоваться для встраивания подробной информации для воздействия или управления этими инструментами без какого-либо влияния на семантику программы. Например, специфичный для Microsoft атрибут `[[gsl::suppress(/* rules */)]]` можно использовать для подавления нежелательных предупреждающих сообщений от инструментальных средств статического анализа, которые проверяют правила Guidelines Support Library[8]. В частности, рассмотрим правило GSL C26481 (Bounds rule 1)[9], которое запрещает любую арифметическую операцию с указателями, предполагая вместо этого, что пользователи полагаются на тип `gsl::span`[10]:

```
void hereticalFunction()
{
    int array[] = {0, 1, 2, 3, 4, 5};

    printElements(array, array + 6);   // Выводит предупреждение C26481.
}
```

Любой блок кода, для которого проверка правила C26481 считается нежелательной, может быть дополнен атрибутом `[[gsl::suppress(bounds.1)]]`:

```
void hereticalFunction()
{
    int array[] = {0, 1, 2, 3, 4, 5};

    [[gsl::suppress(bounds.1)]]            // Подавление предупреждения GSL C26481.
```

[8] Guidelines Support Library (см. microsofta) – библиотека с открытым исходным кодом, разработанная компанией Microsoft, в которой реализованы функции и типы, предлагаемые для использования в «C++ Core Guidelines» (см. stroustrup21).

[9] **microsoftd**.

[10] `gsl::span` – это упрощенный ссылочный тип, который отслеживает непрерывную последовательность или подпоследовательность объектов гомогенного (однообразного) типа. Тип `gsl::span` может использоваться в интерфейсах как альтернатива указателю/размеру или аргументам pair-итераторов, а также в реализациях как альтернатива прямым арифметическим операциям с указателями. С версии C++20 вместо этого типа можно использовать стандартный шаблон `std::span`.

```
    {
        printElements(array, array + 6);   // Тишина!
    }
}
```

1.1.2.5. *Создание новых атрибутов для выражения семантических свойств*

Другие варианты использования атрибутов для статического анализа включают определения свойств, которые невозможно вывести каким-либо иным способом. Рассмотрим функцию `f`, принимающую два указателя `p1` и `p2` и имеющую предусловие (precondition), в соответствии с которым оба указателя обязательно должны ссылаться на один и тот же непрерывный блок памяти. Использование стандартного атрибута для оповещения анализатора о таком условии обладает явным преимуществом, поскольку не требует ничего, кроме соглашения между разработчиком и статическим анализатором, относящегося к пространству имен и имени атрибута. Например, для этой цели мы могли бы выбрать обозначение:

```
// lib.h:

[[home_grown::in_same_block(p1, p2)]]
int f(double* p1, double* p2);
```

Компилятор просто проигнорирует этот неизвестный атрибут. Но поскольку нашему инструментальному средству статического анализа известен смысл атрибута `home_grown::in_same_block`, во время этапа анализа оно сообщит о дефектах, которые в противном случае могли бы привести к неопределенному поведению (undefined behavior) во время выполнения:

```
// client.cpp:
#include <lib.h>

void client()
{
    double a[10], b[10];
    f(a, b);   // Указатели не связаны. Наш статический анализатор сообщает об ошибке.
}
```

1.1.3. Потенциальные опасности

1.1.3.1. *Нераспознаваемые атрибуты обладают поведением, зависящим от реализации*

Стандартные атрибуты работают корректно и являются переносимыми между всеми платформами, но поведение зависимых от конкретного компилятора и создаваемых пользователем атрибутов целиком и полностью определяется реализацией, а нераспознанные атрибуты обычно приводят к выводу предупреждающих сообщений компилятора. Такие предупреждения обычно можно отключить (например, для GCC используется ключ `Wnoattributes`), но в этом случае опечатки даже в стандартных атрибутах останутся незамеченными[11].

1.1.3.2. *Некоторые атрибуты при неправильном использовании могут повлиять на корректность программы*

Многие атрибуты являются безвредными в том смысле, что они могут улучшить диагностику или производительность, но сами по себе не могут стать причиной некорректного поведения программы. Тем не менее неправильное использование некоторых атрибутов может привести к неверным результатам и/или неопределенному поведению.

[11] В идеальном случае каждая сколько-нибудь значимая платформа должна предлагать способ «безмолвного» игнорирования определенного атрибута в каждом конкретном случае.

Например, рассмотрим функцию myRandom, предназначенную для возврата нового случайного числа в диапазоне [0.0, 0.1) при каждом успешном вызове:

```
double myRandom()
{
    static std::random_device randomDevice;
    static std::mt19937 generator(randomDevice());
    static std::uniform_real_distribution<double> distribution(0, 1);

    return distribution(generator);
}
```

Предположим, что мы каким-то образом заметили, что дополнение myRandom атрибутом [[gnu::const]] иногда повышает производительность во время выполнения, и, ничего не подозревая, по простодушию решили использовать его в производственной среде. Это явно некорректное применение атрибута [[gnu::const]], поскольку функция по своей сущности не удовлетворяет требованию получения одинакового результата при вызове с одними и теми же аргументами – в данном случае аргументов вообще нет. Добавление этого атрибута сообщает компилятору, что ему не нужно повторно вызывать эту функцию, и он без каких-либо сомнений может всегда интерпретировать первое возвращаемое значение как константу.

1.1.4. Смотрите также

- "noreturn" (раздел 1.11) представляет стандартный атрибут, используемый для обозначения того факта, что конкретная функция никогда не возвращает управление вызвавшей ее стороне.
- "deprecated" (раздел 1.21) представляет стандартный атрибут, который не рекомендует использовать обозначенную сущность с учетом диагностики компилятора.
- "carries_dependency" (раздел 3.1) представляет стандартный атрибут, используемый для передачи информации о цепочке зависимостей освобождения-потребления компилятору, чтобы избежать ненужных инструкций защиты памяти.

1.1.5. Материал для дополнительного чтения

Более подробную информацию об обычно поддерживаемых атрибутах функций см. в разделе 6.33.1 «Common Function Attributes», **freesoftwarefdn20**.

1.2. Последовательные правые угловые скобки

В контексте списков аргументов шаблона символы >> распознаются как две отдельные закрывающие угловые скобки.

1.2.1. Описание

До версии C++11 пара последовательных направленных вправо угловых скобок в любом месте исходного кода всегда интерпретировалась как битовый оператор сдвига вправо, поэтому требовался промежуточный пробел, чтобы эти скобки воспринимались как элементы закрывающих угловых скобок:

```
// C++03.
std::vector<std::vector<int>> v0;    // Раздражающая ошибка времени компиляции в версии C++03.
std::vector<std::vector<int> > v1;   // OK.
```

Чтобы содействовать правильному использованию показанного выше варианта, было добавлено особое правило, согласно которому при парсинге (синтаксическом анализе) выражения для шаблона с аргументом невложенные, т. е. не размещенные внутри круглых скобок, вхождения символов >, >>, >>> и т. д. должны интерпретироваться как отдельные закрывающие угловые скобки:

```
// C++11.
std::vector<std::vector<int>> v0;                // OK.
std::vector<std::vector<std::vector<int>>> v1;   // OK.
```

1.2.1.1. *Использование знака «больше» или операторов битового сдвига вправо внутри выражений для шаблонов с аргументами*

Для шаблонов, принимающих только типы параметров, проблем не существует. Но если параметр шаблона не является типом, то операторы «больше» или сдвига битов вправо могут оказаться полезными. В маловероятном случае, когда необходим оператор «больше» > или оператор битового сдвига вправо >> внутри выражения для шаблона с аргументом, не определяющим тип, можно сделать это, поместив это выражение в круглые скобки:

```
const int i = 1, j = 2; // Произвольные целочисленные значения (используются ниже).

template <int I> class C { /*...*/ };
             // Класс C, принимающий параметр шаблона I, не являющийся типом, но имеющий тип int.

C<i > j>    a1;   // Ошибка, возникающая всегда.
C<i >> j>   b1;   // Ошибка в версии C++11, OK в версии C++03.
C<(i > j)>  a2;   // OK.
C<(i >> j)> b2;   // OK.
```

В приведенном выше определении `a1` первый символ > интерпретируется как закрывающая угловая скобка, а следующая за ней буква `j` в таком контексте всегда являлась синтаксической ошибкой. В случае `b1` символы >> начиная с версии C++11 при парсинге определяются как пара отдельных элементов в этом контексте, поэтому вторая > теперь считается ошибкой. Но для `a2` и `b2` потенциально распознаваемые операторы размещены внутри круглых скобок, поэтому защищены от сопоставления любой открывающей угловой скобке, находящейся слева от выражения в круглых скобках.

1.2.2. Варианты использования

1.2.2.1. *Устранение раздражающего пробела при формировании типов шаблона*

При использовании вложенных шаблонных типов, например вложенных контейнеров, в C++03 приходится всегда помнить о необходимости вставки пробела между закрывающими угловыми скобками, что не приносит никакой пользы. Еще больше раздражало то, что каждый распространенный компилятор мог безапелляционно сообщить, что мы забыли вставить пробел. С появлением этой новой функциональной возможности (точнее, исправленного недостатка) теперь мы можем записывать закрывающие угловые скобки непрерывно, т. е. точно так же, как скобки и квадратные скобки:

```
// OK в обеих версиях C++03 и C++11.
std::list<std::map<int, std::vector<std::string> > > idToNameMappingList1;

// OK в версии C++11, ошибка времени компиляции в версии C++03.
std::list<std::map<int, std::vector<std::string>>> idToNameMappingList2;
```

1.2.3. Потенциальные опасности

1.2.3.1. *Некоторые программы, использующие версию C++03, могут остановить компиляцию при использовании версии C++11*

Если оператор битового сдвига вправо используется в выражении шаблона, то обновленные правила синтаксического анализа могут привести к ошибке времени компиляции в том месте, где до этого она не возникала:

```
T<1 >> 5> t; // Работает в версии C++03, ошибка времени компиляции в версии C++11.
```

Эту ошибку легко исправить, если заключить выражение сдвига в круглые скобки:

```
T<(1 >> 5)> t; // OK
```

Эта редкая синтаксическая ошибка всегда выявляется во время компиляции, что позволяет избежать скрытых непредвиденных ситуаций во время выполнения.

1.2.3.2. *Смысл программы, использующей версию C++03, теоретически может незаметно измениться в версии C++11*

Одно и то же корректное выражение теоретически может иметь интерпретацию в C++11, отличающуюся от интерпретации при компиляции для C++03, хотя такие случаи встречаются чрезвычайно редко. Рассмотрим случай[12], когда лексема >> является составной частью выражения, включающего шаблоны:

```
S<G< 0 >>::c>::b>::a
// ^~~~~~~
```

В выражении из этого примера `0 >>::c` будет интерпретироваться как битовый оператор сдвига вправо в версии C++03, но не в версии C++11. Можно написать программу, которая компилируется в обеих версиях C++03 и C++11 и выявляет различие в правилах синтаксического анализа:

```
enum Outer { a = 1, b = 2, c = 3 };

template <typename> struct S
{
    enum Inner { a = 100, c = 102 };
};

template <int> struct G
{
    typedef int b;
};

int main()
{
    std::cout << (S<G< 0 >>::c>::b>::a) << '\n';
}
```

Приведенная выше программа выведет числовое значение `100` при компиляции для версии C++03 и значение `0` для версии C++11:

```
// C++03

// (2) создание экземпляра G<0>.
// ╟~~~~~~~~~~~~
// ╟ | ╟ (4) создание экземпляра S<int>
// ~~╟ ↓ ╟~~~~~~~~~~~~~~~↓
   S< G< 0 >>::c > ::b >::a
// ~~╟ ↑ ╟~~~~~~~~~↑
// ╟ | ╟ (3) алиас (псевдоним) типа int
// ╟~~~~~~~
// (1) битовый сдвиг вправо (0 >> 3)

// C++11

//
//
// (2) сравнение (>) Inner::c и Outer::b
```

[12] Пример взят из **gustedt13**.

```
// ↓ ~~~~~~~~~~~~~~~~~~
   S< G< 0 >>::c > ::b >::a
// ↑ ~~~~~~~~~
// (1) создание экземпляра S<G<0>>
//
//
```

Программы, которые синтаксически корректны как в C++03, так и в C++11, но имеют различную семантику, на практике нигде и никогда не появлялись, насколько нам известно, хотя теоретически это возможно.

1.2.4. Материал для дополнительного чтения

Альтернативные проектные решения, которые, как считалось, допускают использование последовательных закрывающих угловых скобок без пробелов, см. в исходном предложении: «Right Angle Brackets», **vandevoorde05**.

1.3. Оператор для извлечения типов выражений

Ключевое слово `decltype` обеспечивает во время компиляции инспекцию объявленного типа (**decl**ared **type**) объекта (класса-сущности – entity) или типа и категорию значения (value category) выражения.

1.3.1. Описание

Результат использования `decltype` зависит от сущности его операнда.

1.3.1.1. *Использование с объектами*

Если операнд представляет собой выражение-идентификатор (id expression), не заключенный в круглые скобки, или выражение доступа к члену класса без круглых скобок, то `decltype` возвращает объявленный тип, т. е. тип объекта, указанный операндом:

```
int i;                  // decltype(i)   -> int
std::string s;          // decltype(s)   -> std::string
int* p;                 // decltype(p)   -> int*
const int& r = *p;      // decltype(r)   -> const int&
struct { char c; } x;   // decltype(x.c) -> char
double f();             // decltype(f)   -> double()
double g(int);          // decltype(g)   -> double(int)
```

1.3.1.2. *Использование с выражениями*

Когда `decltype` используется с любым другим выражением `E` типа `T`, включающим взятое в скобки выражение-идентификатор или выражение доступа к члену (класса), то в результат включают тип самого выражения и соответствующую категорию значения (см. раздел 2.18), как показано в табл. 1.3.1.2.1.

Таблица 1.3.1.2.1. Результаты применения `decltype` к выражениям

Категория значения для `-E`	Результат `decltype(E)`
prvalue	`T`
lvalue	`T&`
xvalue	`T&&`

В общем случае значения категории *prvalue* могут быть переданы в `decltype` несколькими способами, включая числовые литералы, вызовы функций, возвращающие значения, и явно созданные временные объекты:

```
decltype(0) i;      // -> int
int f();
```

```
decltype(f()) j;   // -> int
struct S{};
decltype(S()) k;   // -> S
```

Имя объекта, передаваемого в decltype, как сказано выше, позволяет получить тип этого объекта. Но если имя объекта размещено внутри дополнительной пары круглых скобок, то decltype интерпретирует этот аргумент как выражение и включает в результат категорию значения:

```
int i;
decltype(i)   l = i;   // -> int
decltype((i)) m = i;   // -> int&
```

Аналогично, для всех прочих выражений категории *lvalue* результатом decltype будет ссылка на *lvalue*:

```
int* pi = &i;
decltype(*pi) j = *pi;   // -> int&
decltype(++i) k = ++i;   // -> int&
```

Наконец, категорией значения выражения будет *xvalue*, если оно является приведением типа или функцией, возвращающей ссылку на *rvalue*:

```
int i;
decltype(static_cast<int&&>(i)) j = static_cast<int&&>(i);   // -> int&&
int&& g();
decltype(g()) k = g();                                       // -> int&&
```

Практически так же, как оператор *sizeof* (синтаксический анализ и разрешение которого тоже происходят во время компиляции), выражение-операнд decltype не выполняется:

```
void test1()
{
    int i = 0;
    decltype(i++) j;   // Равнозначно int j;.
    assert(i == 0);    // Выражение i++ не было выполнено.
}
```

Следует отметить, что выбор использования операции постфиксного инкремента имеет весьма большое значение. Операция префиксного инкремента выдает другой тип:

```
void test2()
{
    int i = 0;
    int m = 1;
    decltype(++i) k = m;   // Равнозначно int& k = m;.
    assert(i == 0);        // Выражение ++i не было выполнено.
}
```

1.3.2. Варианты использования

1.3.2.1. *Исключение необходимости использования явных имен типов*

Рассмотрим два логически равнозначных способа объявления вектора итераторов в списке виджетов widgets:

```
std::list<Widget> widgets;
std::vector<std::list<Widget>::iterator> widgetIterators;
    // (1) Для полного типа виджетов требуется повторное определение, и итератору
    // должно быть явно присвоено имя.

std::list<Widget> widgets;
std::vector<decltype(widgets.begin())> widgetIterators;
```

```
    // (2) Ни для std::list, ни для Widget, ни для iterator не требуется
    // явное именование.
```

Обратите внимание: при использовании decltype, если тип C++, представляющий виджет, изменяется (например, с Widget на, скажем, ManagedWidget) или используемый контейнер изменяется (например, с std::list на std::vector), то в объявление widgetIterators вносить изменения не обязательно.

1.3.2.2. *Явное выражение согласованности типов*

В некоторых ситуациях повторение явных имен типов может непреднамеренно привести к скрытым дефектам из-за несовпадения типов во время сопровождения программного обеспечения. Например, рассмотрим класс Packet, предъявляющий const функцию-член, которая возвращает значение типа std::uint8_t, представляющую длину контрольной суммы пакета:

```
class Packet
{
    // ...

public:
    std::uint8_t checksumLength() const;
};
```

Этот беззнаковый 8-битовый тип был выбран для минимизации используемой ширины полосы пропускной способности, так как длина контрольной суммы передается по сети. Далее предположим, что существует цикл, который вычисляет контрольную сумму пакета Packet, используя тот же тип для своей итерационной переменной для соответствия типу, возвращаемому Packet::checksumLength:

```
void f()
{
    Checksum sum;
    Packet data;

    for (std::uint8_t i = 0; i < data.checksumLength(); ++i) // Ненадежно.
    {
        sum.appendByte(data.nthByte(i));
    }
}
```

Теперь предположим, что через некоторое время размер данных, передаваемых в пакете типа Packet, увеличивается до величины, для которой, возможно, диапазон значений std::uint8_t становится недостаточным для представления корректной контрольной суммы. Если тип значения, возвращаемого checksumLength(), изменяется, например, на std::uint16_t без обновления типа строго зафиксированной итерационной переменной i, то цикл может незаметно[13] стать бесконечным[14].

Если бы ключевое слово decltype(packet.checksumLength()) использовалось для выражения типа i, то типы оставались бы согласованными, и, естественно, можно было бы избежать возникающего дефекта:

```
// ...
for (decltype(data.checksumLength()) i = 0; i < data.checksumLength(); ++i)
// ...
```

[13] На момент написания этой книги ни GCC 11.2 (с 2021 г.), ни Clang 12.0.0 (с 2021 г.) не выводили предупреждающее сообщение (при использовании -Wall, -Wextra и -Wpedantic) при сравнении std::uint8_t и std::uint16_t – даже если (1) значение, возвращаемое checksumLength, не умещается в 8-битовом целом числе и (2) тело этой функции является видимым для компилятора. Дополнение checksumLength ключевым словом constexpr приводит к тому, что clang++ выводит предупреждающее сообщение, что явно не является общим универсальным решением.

[14] Выполняется преобразование с повышением типа переменной цикла до unsigned int для операций сравнения, но ее значение возвращается к 0, если перед инкрементированием оно было равно 255.

1.3.2.3. *Создание вспомогательной переменной универсального типа*

Рассмотрим задачу реализации обобщенного (generic) шаблона функции `loggedSum`, которая возвращает сумму двух произвольных объектов `a` и `b` с последующей записью в журнал обоих операндов и полученного значения, например для отладки или мониторинга. Чтобы избежать двукратного вычисления суммы, которая может оказаться дорогостоящей, мы решили создать вспомогательную переменную `result` с областью видимости, ограниченную телом этой функции. Поскольку тип суммы зависит от `a` и `b`, можно воспользоваться `decltype(a + b)` для логического вывода типа и для конечного возвращаемого значения функции (см. раздел 1.16), и для вспомогательной переменной:

```
template <typename A, typename B>
auto loggedSum(const A& a, const B& b)
     -> decltype(a + b)                // (1) Использование типов конечных возвращаемых значений.
{
    decltype(a + b) result = a + b;    // (2) Вспомогательная универсальная переменная.
    LOG_TRACE << a << " + " << b << " = " << result;
    return result;
}
```

Использование `decltype(a + b)` как типа возвращаемого значения существенно отличается от применения автоматического вывода типа по `return` – см. раздел 2.3. Следует отметить, что этот конкретный вариант использования включает значительное повторение выражения `a + b`. См. подраздел 1.3.4.1, где обсуждаются возможные способы устранения такого повторения.

1.3.2.4. *Определение правильности обобщенного универсального выражения*

В контексте разработки библиотеки общего назначения `decltype` можно использовать в сочетании с правилом SFINAE («Substitution Failure Is Not An Error» – «Сбой при подстановке не является ошибкой») для проверки корректности выражения, включающего параметр-шаблон.

Например, рассмотрим задачу написания обобщенного универсального шаблона функции `sortRange`, который при заданном диапазоне `range` либо вызывает функцию-член `sort` аргумента (специально оптимизированную для конкретного передаваемого типа), если она доступна, либо возвращается к более общей функции `std::sort`:

```
template <typename Range>
void sortRange(Range& range)
{
    sortRangeImpl(range, 0);
}
```

Доступная клиенту функция `sortRange` в приведенном выше примере делегирует свое поведение перегружаемой функции `sortRangeImpl` в примере, приведенном ниже, вызывая последнюю с аргументом `range` и устранителем неоднозначности (disambiguator) типа `int`. Тип этого дополнительного параметра, значение которого является произвольным, применяется для назначения приоритета функции-члену `sort` во время компиляции с использованием правил разрешения конфликтов во время перегрузки при наличии неявного стандартного преобразования из `int` в `long`:

```
template <typename Range>
void sortRangeImpl(Range& range, long) // Низкий приоритет: стандартное преобразование.
{
    // Возврат к специальной реализации.
    std::sort(std::begin(range), std::end(range));
}
```

Возврат к перегрузке функции `sortRangeImpl` в приведенном выше фрагменте кода приведет к приему устранителя неоднозначности типа `long`, требующего преобразования из `int`, и просто вызовет стандартную функцию `std::sort`. Более специализирован-

ная перегрузка функции sortRangeImpl в приведенном ниже фрагменте кода принимает устранитель неоднозначности типа int, не требующий преобразования, следовательно, представляет собой более точное соответствие, обеспечивая доступ к специализированной для конкретного диапазона сортировке:

```
template <typename Range>
void sortRangeImpl(Range& range,
                   int,                          // Высокий приоритет: точное соответствие.
                   decltype(range.sort())* = 0)  // Проверка правильности выражения.
{
    // Оптимизированная реализация.
    range.sort();
}
```

Обратите внимание: при представлении decltype(range.sort()) как части объявления функции sortRangeImpl более специализированная перегружаемая версия будет отброшена во время подстановки шаблона, если range.sort() не является допустимым выражением для логически выведенного типа Range[15].

Относительное положение decltype(range.sort()) в сигнатуре функции sortRangeImpl не важно, поскольку это выражение является видимым для компилятора во время подстановки шаблона. В предыдущем примере используется параметр функции, по умолчанию равный 0. В качестве альтернативных вариантов подразумеваются хвостовой возвращаемый тип или принятый по умолчанию аргумент шаблона, которые также являются вполне приемлемыми:

```
#include <utility> // declval
template <typename Range>
auto sortRangeImpl(Range& range, int) -> decltype(range.sort(), void());
    // Оператор запятая используется для принудительного приведения возвращаемого типа к void
    // вне зависимости от типа, возвращаемого range.sort().

template <typename Range, typename = decltype(std::declval<Range&>().sort())>
auto sortRangeImpl(Range& range, int) -> void;
    // std::declval используется для генерации ссылки на Range, которую можно применять
    // в невычисляемом выражении.
```

Объединяя все вышеизложенное вместе, можно видеть, что существуют ровно два возможных результата для исходной, взаимодействующей непосредственно с клиентом функции sortRange, вызываемой с аргументом в виде диапазона типа R:

- если R содержит функцию-член sort, то более специализированный вариант перегрузки sortRangeImpl будет наиболее подходящим, поскольку range-sort() – правильно построенное выражение и наиболее предпочтительное, потому что устранитель неоднозначности 0 типа int не требует преобразований;
- в другом случае более специализированный вариант перегрузки будет отвергнут во время подстановки шаблона, потому что range.sort() не является правильно построенным выражением, и вместо него будет выбран единственный оставшийся более общий вариант перегрузки sortRangeImpl.

1.3.3. Потенциальные опасности

Вероятно, неожиданным является тот факт, что decltype(x) и decltype((x)) иногда будут выдавать различные результаты для одного и того же выражения x:

[15] Методика представления возможно неиспользуемого невычисляемого выражения, например с использованием decltype, в объявлении функции с целью определения правильности выражения перед созданием экземпляра шаблона широко известна как выражение SFINAE, которое является ограниченной формой более общего, классического SFINAE и воздействует исключительно на выражения, видимые в сигнатуре функции, а не на зачастую скрытые вычисления типов на основе шаблонов.

```
int i = 0; // decltype(i) возвращает int.
           // decltype((i)) возвращает int&.
```

В том случае, где операнд без скобок является объектом, имеющим объявленный тип T, а операнд, взятый в скобки, представляет собой выражение, категория значения которого представлена decltype как тот же тип T, результаты будут на удивление одинаковыми:

```
int& ref = i; // decltype(ref) возвращает int&.
              // decltype((ref)) возвращает int&.
```

Заключение этого операнда в скобки гарантирует, что decltype возвращает категорию значения заданного выражения. Эта методика может оказаться полезной в контексте метапрограммирования, особенно в случае распространения категории значения.

1.3.4. Неудобства

1.3.4.1. *Может потребоваться механическое повторение выражений*

Как было отмечено в подразделе 1.3.2.3, применение decltype для захвата значения выражения, которое готовится к использованию, или для возвращаемого значения часто может приводить к повторению одного и того же выражения в нескольких местах: в приведенном выше примере – в трех различных местах.

Другим вариантом решения этой проблемы является захват результата выражения decltype в typedef, псевдонима (алиаса) типа using или принятого по умолчанию параметра шаблона, но при таком подходе возникает другая проблема: его можно использовать только после того, как выражение станет корректным. Принятый по умолчанию параметр шаблона template не может ссылаться на имена параметров, потому что он записан перед ними, а псевдоним типа не может быть введен до того, как потребуется возвращаемый тип. Решение этой проблемы заключается в использовании функции стандартной библиотеки std::declval для создания выражений соответствующего типа без необходимости ссылки на фактические параметры функции по имени:

```
template <typename A, typename B,
          typename Result = decltype(std::declval<const A&>() +
                                     std::declval<const B&>())>
Result loggedSum(const A& a, const B& b)
{
    Result result = a + b;       // Без дублирования выражения decltype.
    LOG_TRACE << a << " + " << b << "=" << result;
    return result;
}
```

Здесь std::declval – функция, которая не может быть выполнена во время выполнения и применима только в невычисляемых контекстах, – создает выражение заданного типа. При совместном использовании с decltype функция std::declval позволяет определить типы результата для выражений без необходимости или даже без возможности создания объектов требуемых типов.

1.3.5. Смотрите также

- «Псевдонимы using» (раздел 1.18) – объясняется, что часто удобно присвоить имя типу, возвращаемому decltype, что можно сделать с помощью определения псевдонима using.
- «Переменные auto» (раздел 2.3) – показано, что для переменных auto имеется аналогичный вывод типа, который отличается от вычисленного с помощью decltype.
- «Ссылки на *rvalue*» (раздел 2.18) – описаны категории значений, которые можно получить для произвольных значений, используя decltype.
- «decltype(auto)» (раздел 3.9) – представлены правила вычисления типа decltype, которые могут оказаться полезными в сочетании с переменной auto.

1.4. Использование = *default* для специальных функций-членов

Ключевое слово `default`, обозначающее объявление специальных функций-членов класса, сообщает компилятору о необходимости попытки сгенерировать такую функцию автоматически.

1.4.1. Описание

Важным аспектом проектирования класса C++ является понимание того, что компилятор будет пытаться сгенерировать конкретные функции-члены класса для создания, копирования, уничтожения, а теперь и перемещения (см. раздел 2.18) объекта, если разработчики сами не реализовали некоторые или все эти функции. Чтобы определить, какие из специальных функций-членов будут продолжать создаваться, а какие будут подавляться при наличии предоставленных пользователем специальных функций-членов, необходимо помнить о многочисленных правилах, используемых компилятором.

1.4.1.1. *Явное определение специальных функций-членов*

Правила, определяющие, что происходит при наличии одной или нескольких предоставленных пользователем специальных функций-членов, сложны по своей сущности и не всегда интуитивно понятны, в действительности некоторые из них устарели. В частности, даже при наличии деструктора, предоставляемого пользователем, копирующий конструктор и оператор присваивания копированием в прошлом генерировались неявно. Полагаться на такое сгенерированное поведение проблематично, потому что маловероятно, что класс, требующий предоставляемого пользователем деструктора, будет правильно функционировать без соответствующих операций копирования, предоставляемых пользователем. Начиная с версии C++11 полагаться на такое сомнительное, неявно сгенерированное поведение не рекомендуется.

Продемонстрируем и кратко опишем несколько общих примеров, а затем рассмотрим широко известную в настоящее время таблицу Говарда Хиннанта (Howard Hinnant) (см. в приложении подраздел 1.4.7.1), чтобы подробнее узнать о том, что происходит внутри.

Пример 1. Предоставление только конструктора по умолчанию

Рассмотрим структуру `struct`, для которой пользователь предоставил конструктор по умолчанию:

```
struct S1
{
    S1();     // Предоставленный пользователем конструктор по умолчанию.
};
```

Предоставленный пользователем конструктор по умолчанию никоим образом не воздействует на прочие специальные функции-члены. Тем не менее предоставление любого конструктора подавляет автоматическую генерацию конструктора по умолчанию. Но мы можем воспользоваться `= default` для восстановления конструктора как тривиальной операции – см. подраздел 1.4.2.1. Следует отметить, что необъявленная функция является несуществующей, т. е. она вообще не будет участвовать в разрешении перегрузки. Напротив, удаленная функция принимает участие в разрешении перегрузки, и если она выбрана, то возникает критическая ошибка компиляции – см. раздел 1.6.

Пример 2. Предоставление только копирующего конструктора

Теперь рассмотрим структуру `struct`, для которой пользователь предоставил копирующий конструктор:

```
struct S2
{
```

```
    S2(const S2&);      // Предоставленный пользователем копирующий конструктор.
};
```

Предоставленный пользователем копирующий конструктор (1) подавляет генерацию конструктора по умолчанию и обеих операций перемещения и (2) разрешает неявную генерацию оператора присваивания копированием и деструктора. Аналогично предоставление только оператора присваивания копированием должно позволить компилятору неявно сгенерировать копирующий конструктор и деструктор, но в этом случае также должен быть сгенерирован конструктор по умолчанию. Обратите внимание: в любом из этих случаев полагаться на неявно сгенерированную компилятором операцию копирования не рекомендуется.

Пример 3. Предоставление только деструктора

Наконец, рассмотрим структуру `struct`, для которой пользователь предоставил деструктор:

```
struct S3
{
    ~S3();      // Предоставленный пользователем деструктор.
};
```

Предоставленный пользователем деструктор подавляет генерацию операций перемещения, но при этом сохраняется возможность генерации операций копирования. И в этом случае также не рекомендуется полагаться на эти неявно сгенерированные компилятором операции копирования.

Пример 4. Предоставление более одной специальной функции-члена

Если явно объявлено более одной специальной функции-члена, то применяется объединение их соответствующих подавлений объявлений и пересечение их соответствующих неявных генераций. Например, если предоставлены только конструктор по умолчанию и деструктор (`S1` + `S3` в примерах 1 и 3), то объявления обеих операций перемещения подавляются, а обе операции копирования генерируются неявно.

1.4.1.2. *Явное определение по умолчанию первого объявления специальной функции-члена*

Использование синтаксиса `= default` для первого объявления специальной функции-члена сообщает компилятору о необходимости автоматической генерации такой функции без ее восприятия как предоставленной пользователем. Сгенерированная компилятором версия специальной функции-члена требуется для вызова соответствующих специальных функций-членов в каждом базовом классе в порядке объявления базовых классов, а затем – каждого члена данных инкапсулированного типа в порядке объявления независимо от любых спецификаторов доступа. Следует отметить, что вызовы деструктора будут выполняться в порядке, прямо противоположном вызовам других специальных функций-членов.

Например, рассмотрим структуру `S4` в приведенном ниже фрагменте кода, в котором мы решили явно указать, что операции копирования должны автоматически генерироваться компилятором. Обратите особое внимание на то, что неявное объявление и генерация каждой из других специальных функций-членов остаются неизменными.

```
struct S4
{
    S4(const S4&) = default;               // Копирующий конструктор.
    S4& operator=(const S4&) = default;    // Оператор копирования с присваиванием.

    // Нет никакого воздействия на другие четыре специальные функции-члены, т. е.
    // неявно генерируется конструктор по умолчанию, деструктор,
    // перемещающий конструктор и оператор перемещения с присваиванием.
};
```

Объявление по умолчанию может записываться с любым спецификатором доступа (т. е. `private`, `protected` или `public`), и доступ к этой генерируемой функции будет управляться соответствующим образом:

```
struct S5
{
private:
    S5(const S5&) = default;                // Закрытый копирующий конструктор.
    S5& operator=(const S5&) = default;     // Закрытый оператор копирования с присваиванием.

protected:
    ~S5() = default;                        // Защищенный деструктор.

public:
    S5() = default;                         // Открытый конструктор по умолчанию.
};
```

В приведенном выше примере операции копирования существуют для использования только функциями-членами и дружественными (`friend`) функциями. Объявление деструктора защищенным (`protected`) или закрытым (`private`) ограничивает возможность создания автоматических переменных определяемого конкретного типа только теми функциями, которые обладают соответствующими правами доступа к этому классу. Объявление конструктора по умолчанию открытым (`public`) необходимо для того, чтобы избежать подавления его объявления другим конструктором, например закрытым копирующим конструктором в приведенном выше фрагменте кода или какой-либо операцией перемещения.

Короче говоря, использование `= default` в первом объявлении означает, что специальная функция-член предназначена для генерации компилятором вне зависимости от любых объявлений, предоставляемых пользователем. В сочетании с `= delete` (см. раздел 1.6) использование `= default` обеспечивает многофункциональное средство управления вариантами генерации специальных функций-членов и/или определения открытости доступа к ним.

1.4.1.3. *Определение по умолчанию реализации предоставленной пользователем специальной функции-члена*

Синтаксис `= default` также можно использовать после первого объявления, но с прямо противоположным смыслом: компилятор будет интерпретировать первое объявление как предоставленную пользователем специальную функцию-член, следовательно, будет подавлять генерацию других специальных функций-членов соответствующим образом:

```
// example.h:

struct S6
{
    S6& operator=(const S6&);      // Предоставленный пользователем оператор копирования
                                   // с присваиванием.

    // Подавляет объявление обеих операций.
    // Неявно генерирует конструктор по умолчанию и копирующий конструктор и деструктор.
};

inline S6& S6::operator=(const S6&) = default;
    // Явно требует от компилятора генерацию реализации по умолчанию для этого
    // оператора копирования с присваиванием. Это требование может привести к критическому
    // отказу, например если структура S6 содержала член данных, который не может
    // копироваться с присваиванием.
```

В качестве альтернативы явно заданная по умолчанию невстроенная реализация этого оператора копирования с присваиванием может появиться в отдельном (*.cpp*) файле – см. подраздел 1.4.2.4.

1.4.2. Варианты использования

1.4.2.1. *Восстановление генерации специальной функции-члена, подавляемой другой функцией-членом*

Включение = default в объявление специальной функции-члена сообщает компилятору о необходимости генерации его собственного определения вне зависимости от наличия любых других предоставленных пользователем специальных функций-членов. В качестве примера рассмотрим класс SecureToken с семантическим значением (value-semantic), который является оберткой для стандартной строки std::string, и целое число с произвольной точностью BigInt, как код маркера (токена), которые вместе соответствуют определенным инвариантам:

```
class SecureToken
{
    std::string d_value;   // Созданное по умолчанию значение – пустая строка.
    BigInt d_code;         // Созданное по умолчанию значение – целочисленный ноль.

public:
    // Все шесть специальных функций-членов неявно созданы по умолчанию.

    void setValue(const char* value);
    const char* value() const;
    BigInt code() const;
};
```

По умолчанию значением защищенного маркера будет пустая строка, а кодом маркера – числовое значение ноль, потому что оба они являются инициализированными по умолчанию (default-initialized) значениями двух членов данных, соответственно d_value и d_code:

```
void f()
{
    SecureToken token;                       // Создан конструктором по умолчанию    (1).
    assert(token.value() == std::string());  // Значение по умолчанию: пустая строка (2).
    assert(token.code() == BigInt());        // Значение по умолчанию: ноль          (3).
}
```

Теперь предположим, что мы получили запрос на добавление конструктора значения (value constructor), который создает и инициализирует SecureToken на основе заданной строки маркера:

```
class SecureToken
{
    std::string d_value;   // Созданное по умолчанию значение – пустая строка.
    BigInt d_code;         // Созданное по умолчанию значение – целочисленный ноль.

public:
    SecureToken(const char* value);   // Новый добавленный конструктор значений.

    // Подавляет только объявление конструктора по умолчанию т. е. компилятор
    // неявно генерирует все остальные пять специальных функций-членов.

    void setValue(const char* value);
    const char* value() const;
    const BigInt& code() const;
};
```

При пробной компиляции функции f теперь должна возникнуть критическая ошибка в первой строке, где компилятор пытается создать маркер с помощью конструктора по умолчанию. Но, используя функциональную возможность = default, мы можем с легкостью вернуть конструктор по умолчанию к работе точно так же, как это делали ранее:

```
class SecureToken
{
    std::string d_value;   // Созданное по умолчанию значение – пустая строка.
    BigInt d_code;         // Созданное по умолчанию значение – целочисленный ноль.

public:
    SecureToken() = default;          // Конструктор по умолчанию снова работает.
    SecureToken(const char* value);   // Новый добавленный конструктор значений.

    // Компилятор неявно генерирует все остальные пять специальных функций-членов.

    void setValue(const char* value);
    const char* value() const;
    const BigInt& code() const;
};
```

1.4.2.2. *Создание API-интерфейсов классов в явной форме без накладных расходов во время выполнения*

На ранних этапах существования C++ стандарты кодирования иногда требовали обязательного явного объявления каждой специальной функции-члена, чтобы ее можно было документировать или даже просто для уверенности в том, что ее не забыли написать:

```
class C1
{
    // ...

public:
    C1();
        // Создает пустой объект.

    C1(const C1& rhs);
        // Создает объект, имеющий то же значение, что и заданный объект rhs.

    ~C1();
        // Уничтожает этот объект.

    C1& operator=(const C1& rhs);
        // Присваивает этому объекту значение заданного объекта rhs.
};
```

Со временем явное указание тех действий, которые сам компилятор мог бы выполнить более надежно, стало очевидно неэффективным использованием времени разработчика и излишней нагрузкой при сопровождении. Более того, даже если определение функции было пустым, его явная реализация часто снижала производительность по сравнению с тривиальным значением по умолчанию. Поэтому стандарты языка начали развиваться в сторону общепринятого комментирования – например, с использованием //! – объявлений функций с пустым телом, а не предоставляющих его явно:

```
class C2
{
    // ...

public:
    //! C2();
        // Создает пустой объект.

    //! C2(const C2& rhs);
        // Создает объект, имеющий то же значение, что и заданный объект rhs.

    //! ~C2();
```

```
        // Уничтожает этот объект.

    //! C2& operator=(const C2& rhs);
        // Присваивает этому объекту значение заданного объекта rhs.
};
```

Но при этом следует отметить, что компилятор не проверяет закомментированный код, а это приводит к повышению вероятности ошибок при копировании-вставке и многих других. При снятии комментариев с кода и явной установке значений по умолчанию в области видимости класса мы восстанавливаем синтаксическую проверку сигнатур функций компилятором без накладных расходов на преобразование того, что должно быть тривиальным (trivial), в равнозначное нетривиальное:

```
class C3
{
    // ...

public:
    C3() = default;
        // Создает пустой объект.

    C3(const C3& rhs) = default;
        // Создает объект, имеющий то же значение, что и заданный объект rhs.

    ~C3() = default;
        // Уничтожает этот объект.

    C3& operator=(const C1& rhs) = default;
        // Присваивает этому объекту значение заданного объекта rhs.
};
```

1.4.2.3. *Сохранение тривиальности типов*

Конкретный тип, который является тривиальным (trival), может оказаться полезным. Тип считается тривиальным, если его конструктор по умолчанию тривиален и является тривиально копируемым (trivially copyable), т. е. у него нет нетривиальных конструкторов копирования или перемещения, нетривиальных операторов присваивания с копированием или перемещением, по крайней мере одного из неудаленных, и имеет тривиальный деструктор. В качестве примера рассмотрим простой тривиальный тип `Metrics` в приведенном ниже фрагменте кода, содержащем конкретные собранные метрики для нашего приложения:

```
struct Metrics
{
    int d_numRequests;   // Число запросов к данному сервису.
    int d_numErrors;     // Число ошибочных ответов.

    // Все специальные функции-члены генерируются неявно.
};
```

Теперь предположим, что мы хотели бы добавить в эту структуру конструктор, чтобы сделать ее использование более удобным:

```
struct Metrics
{
    int d_numRequests;   // Число запросов к данному сервису.
    int d_numErrors;     // Число ошибочных ответов.

    Metrics(int, int);   // Конструктор значений, предоставленный пользователем.

    // Генерация конструктора по умолчанию подавлена.
};
```

Как показано в подразделе приложения 1.4.7.1, наличие предоставленного пользователем конструктора подавляет неявную генерацию конструктора по умолчанию. Может показаться, что замена конструктора по умолчанию на выглядящий равнозначным предоставленный пользователем конструктор обеспечивает предполагаемую работу:

```
struct Metrics
{
    int d_numRequests;   // Число запросов к данному сервису.
    int d_numErrors№;    // Число ошибочных ответов.

    Metrics(int, int);   // Конструктор значений, предоставленный пользователем.
    Metrics() {}         // Конструктор по умолчанию, предоставленный пользователем.

    // Конструктор по умолчанию представлен пользователем:
    // структура Metrics не является тривиальной.
};
```

Но предоставленная пользователем сущность конструктора по умолчанию делает тип `Metrics` нетривиальным, даже если определения идентичны. Напротив, явный запрос на генерацию конструктора по умолчанию с использованием `= default` восстанавливает тривиальность этого типа:

```
struct Metrics
{
    int d_numRequests;   // Число запросов к данному сервису.
    int d_numErrors;     // Число ошибочных ответов.

    Metrics(int, int);     // Конструктор значений, предоставленный пользователем.
    Metrics() = default;   // Тривиальный конструктор по умолчанию с использованием = default.

    // Конструктор по умолчанию объявлен с использованием = default:
    // структура Metrics является тривиальной.
};
```

1.4.2.4. *Физическое отделение интерфейса от реализации*

Иногда, особенно в процессе крупномасштабной разработки, отказ от связывания клиентов во время компиляции с реализациями отдельных методов предоставляет явные преимущества при сопровождении. Указание на то, что при первом объявлении специальной функции-члена, т. е. в области видимости класса, используется `= default`, подразумевает, что внесение любых изменений в эту реализацию неизбежно приведет к необходимости повторной компиляции всех клиентских компонентов:

```
// smallscale.h:

struct SmallScale
{
    SmallScale() = default;   // Конструктор по умолчанию, явно объявленный
                              // с использованием = default.
};
```

Важным вопросом, относящимся к повторной компиляции, здесь является не просто время компиляции как таковое, а связывание во время компиляции[16].

Можно выбрать другой вариант: объявить эту функцию, но преднамеренно не использовать ее по умолчанию в области видимости класса или где-либо в заголовочном *.h*-файле:

```
// largescale.h:

struct LargeScale
{
```

[16] См. **lakos20**, раздел 3.10.5, «Real-World Example of Benefts of Avoiding Compile-Time Coupling», стр. 783–789.

```
    LargeScale();   // Конструктор по умолчанию, предоставленный пользователем.
};
```

Затем можно с помощью `= default` объявить только невстраиваемую реализацию в соответствующем[17] *.cpp*-файле:

```
// largescale.cpp:
#include <largescale.h>

LargeScale::LargeScale() = default;
    // Генерация реализации по умолчанию для этого деструктора по умолчанию.
```

1.4.3. Потенциальные опасности

1.4.3.1. *Специальные функции-члены, использующие = default, не могут восстанавливать тривиальную копируемость*

Классы библиотек часто зависят от того, можно ли копировать тип, с которым они работают, с помощью функции `memcpy` в целях оптимизации. Это может быть вариант реализации, например вектора, который будет выполнять единственный вызов `memcpy` при увеличении своего буфера. Но для правильного определения `memcpy` или `memmove` тип объекта, хранящегося в буфере, обязательно должен быть тривиально копируемым. Можно предположить: эта характеристика означает, что до тех пор, пока конструктор копирования этого типа тривиален, такая оптимизация будет применяться. Объявление операций копирования с использованием `= default` позволило бы достичь этой цели, позволяя типу иметь нетривиальный деструктор или операцию перемещения. Но это не тот случай.

Требования к типу, который считается тривиально копируемым, следовательно, подходящим для использования с `memcpy`, включают тривиальность всех его неудаляемых операций копирования и перемещения, а также его деструктора. Кроме того, авторы библиотек не могут выполнять подробное регулирование, основываясь на том, какие операции над конкретным типом действительно являются тривиальными. Даже если мы обнаружим, что тип является тривиально создаваемым с копированием с типажом (trait) `std::is_trivially_copy_constructible`, и точно знаем, что наш код будет использовать только копирующие конструкторы (а не присваивание с копированием или какие-либо операции перемещения), то все равно не сможем использовать `memcpy`, разве что более строгий ограничивающий типаж `std::is_trivially_copyable` также будет истинным (`true`).

1.4.4. Неудобства

1.4.4.1. *При использовании = default генерация функций не гарантируется*

Использование `= default` не гарантирует, что специальная функция-член типа `T` будет сгенерирована. Например, некопируемая переменная-член или базовый класс `T` воспрепятствует генерации копирующего конструктора `T` даже при использовании `= default`. Такое поведение может наблюдаться при наличии члена данных `std::unique_ptr`[18]:

---

[17] На практике для каждого *cpp*-файла, кроме того что содержит `main`, обычно существует соответствующий заголовочный *h*-файл, и часто наоборот, т. е. пара файлов *.cpp* и *.h* образует компонент. См. lakos20, раздел 1.6 «From .h/.cpp Pairs to Components» и раздел 1.11 «Extracting Actual Dependencies», стр. 209–216 и 256–259 соответственно.

[18] `std::unique_ptr<T>` – только перемещаемый, т. е. перемещаемый, но некопируемый шаблон класса, введенный в стандарте C++11. Он моделирует специфический способ владения динамически размещаемым экземпляром `T`, применяя ссылки на rvalue (см. раздел 2.18) для представления передачи прав владения между экземплярами:

```
int* p = new int(42);
std::unique_ptr<int> up(p);                  // OK: принято владение экземпляром p.
std::unique_ptr<int> upCopy = up;            // Ошибка, копия удаляется.
std::unique_ptr<int> upMove = std::move(up); // OK: передача права владения.
```

```
#include <memory>    // std::unique_ptr
class Connection
{
private:
    class Impl;                       // Вложенный класс реализации.
    std::unique_ptr<Impl> d_impl;     // Некопируемый член данных.

public:
    Connection() = default;
    Connection(const Connection&) = default;
};
```

Несмотря на то что копирующий конструктор объявлен с использованием `= default`, для `Connection` не будет существовать возможность создания копированием, так как `std::unique_ptr` является некопируемым типом. Некоторые компиляторы могут выводить предупреждающее сообщение, относящееся к объявлению `Connection(const Connection&)`, но это не является обязательным требованием, поскольку приведенный выше пример кода грамматически корректен, и ошибка при компиляции должна возникать только при попытке применения конструктора по умолчанию или создания с копированием `Connection`[19].

При необходимости возможным способом обеспечения действительной генерации специальной функции-члена, объявленной с `= default`, является использование `static_assert` (см. раздел 1.15) в сочетании с соответствующим типажом из заголовочного файла `<type_traits>`:

```
class IdCollection
{
    std::vector<int> d_ids;

public:
    IdCollection() = default;
    IdCollection(const IdCollection&) = default;
    // ...
};

static_assert(std::is_default_constructible<IdCollection>::value,
              "IdCollection must be default constructible.");
static_assert(std::is_copy_constructible<IdCollection>::value,
              "IdCollection must be copy constructible.");
// ...
```

Регулярное использование таких методов проверки во время компиляции может способствовать обеспечению того, что тип будет продолжать вести себя так, как ожидается, без дополнительных накладных расходов во время выполнения, даже если члены и базовые типы будут развиваться в процессе непрерывного сопровождения программного обеспечения.

1.4.5. Смотрите также

- «Удаленные функции» (раздел 1.6) – описана сопутствующая `= default` функциональная возможность, которую можно использовать для подавления доступа к неявно сгенерированным специальным функциям-членам.
- «`static_assert`» (раздел 1.15) – описана функциональная возможность, которую можно использовать для проверки во время компиляции того факта, что нежелательные операции копирования и перемещения объявляются как доступные.

[19] Clang 8.0.0 (с 2019 г.) и более поздних версий выводит диагностическую информацию, даже если не заданы флаги предупреждений. MSVC 12.0 (с 2013 г.) выводит диагностическую информацию, если задан флаг `/wall`. На момент написания этой книги GCC 12.1 (с 2021 г.) не выводит предупреждающее сообщение, даже если заданы флаги `-Wall` и `-Wextra`.

- «Ссылки на *rvalue*» (раздел 2.18) – предоставляется основа для операций перемещения, а именно: конструктор с перемещением (move-constructor) и операция присваивания с перемещением для специальных функций-членов, которые также могут быть объявлены с использованием `= default`.

1.4.6. Материал для дополнительного чтения

Более подробную информацию о функциях, объявляемых с использованием `= default`, см. в «Everything You Ever Wanted to Know About Move Semantics» Говарда Хиннанта (Howard Hinnant), **hinnant14** и **hinnant16**.

1.4.7. Приложение

1.4.7.1. *Неявная генерация специальных функций-членов*

Правила, которые использует компилятор для принятия решения о необходимости или отказе от неявной генерации специальной функции-члена, являются абсолютно интуитивными. Говард Хиннант (Howard Hinnant), ведущий разработчик и автор стандарта C++11, внес предложение о семантике перемещения[20] (наряду с другими предложениями), позволяющее получить табличное представление[21] таких правил в ситуации, когда пользователь предоставляет единственную специальную функцию-член, а все прочие оставляет за компилятором. Чтобы понять, как пользоваться табл. 1.4.7.1.1, необходимо выбрать специальную функцию-член в первом столбце, и соответствующая строка покажет, какие функции неявно генерируются компилятором.

Таблица 1.4.7.1.1. Неявная генерация специальных функций-членов

	Конструктор по умолчанию	Деструктор	Копирующий конструктор	Присваивание с копированием	Конструктор с перемещением	Присваивание с перемещением
Нет	По умолчанию	По умолчанию	По умолчанию	По умолчанию	По умолчанию	По умолчанию
Любой конструктор	Не объявлен	По умолчанию	По умолчанию	По умолчанию	По умолчанию	По умолчанию
Конструктор по умолчанию	Объявлен пользователем	По умолчанию	По умолчанию	По умолчанию	По умолчанию	По умолчанию
Деструктор	По умолчанию	Объявлен пользователем	По умолчанию*	По умолчанию*	Не объявлен	Не объявлен
Копирующий конструктор	Не объявлен	По умолчанию	Объявлен пользователем	По умолчанию*	Не объявлен	Не объявлен
Присваивание с копированием	По умолчанию	По умолчанию	По умолчанию*	Объявлен пользователем	Не объявлен	Не объявлен
Конструктор с перемещением	Не объявлен	По умолчанию	Удален	Удален	Объявлен пользователем	Не объявлен
Присваивание с перемещением	По умолчанию	По умолчанию	Удален	Удален	Не объявлен	Объявлен пользователем

* Нерекомендуемое (устаревшее) поведение: компиляторы могут предупреждать об использовании этой неявно сгенерированной функции-члена.

Например, явное объявление оператора присваивания с копированием приведет к тому, что конструктор по умолчанию, деструктор и копирующий конструктор будут созданы по умолчанию, а операции перемещения не будут объявлены. Если пользователь объявил более одной специальной функции-члена, то независимо от того, реализована ли она и каким способом, остальные генерируемые функции-члены находятся на пересечении с соответствующими строками. Например, явное объявление деструктора и конструктора по умолчанию в любом случае приведет к тому, что копирующий конструктор и оператор присваивания с копированием будут созданы по умолчанию, а обе операции перемещения не будут объявлены. Не следует полагать-

[20] **hinnant02**.
[21] **hinnant16**.

ся на сгенерированные компилятором операции копирования, когда деструктор не определен точно, но объявлен по умолчанию; если все сделано правильно, то явное объявление этих операций по умолчанию уточняет их существование и предполагаемое определение.

1.5. Конструкторы, вызывающие другие конструкторы

Использование имени класса в списке инициализации конструктора этого класса позволяет делегировать инициализацию другому конструктору того же класса.

1.5.1. Описание

Делегирующий конструктор (delegating constructor) – это конструктор типа, определенного пользователем (user-defined type – UDT), т. е. `class`, `struct` или `union`, который вызывает другой конструктор, определенный для того же типа, определенного пользователем (далее – ТОП), как часть своей инициализации объекта этого типа. Синтаксис вызова другого конструктора представляет собой указание имени типа как единственного элемента в списке инициализации членов:

```
#include <string>   // std::string

struct S0
{
  int         d_i;
  std::string d_s;

  S0(int i)         : d_i(i) {}          // Неделегирующий конструктор.
  S0()              : S0(0) {}           // OK, делегирование конструктору S0(int).
  S0(const char* s) : S0(0), d_s(s) {}   // Ошибка: делегирование должно выполняться
                                         // для "собственного" типа.
};
```

Несколько делегирующих конструкторов можно объединить в цепочку – каждый вызывает один и только один следующий конструктор, поэтому циклы исключены, – см. подраздел 1.5.3.1. Когда конструктор возвращает цель (target), т. е. вызываемый через делегирование, то вызывается тело делегирующего конструктора:

```
#include <iostream>    // std::cout

struct S1
{
    S1(int, int)       { std::cout << 'a'; }
    S1(int) : S1(0, 0) { std::cout << 'b'; }
    S1() : S1(0)       { std::cout << 'c'; }
};

void f()
{
    S1 s;     // OK, выводится строка "abc" на устройство стандартного вывода stdout.
}
```

Если при выполнении неделегирующего конструктора генерируется исключение, то инициализируемый объект рассматривается только лишь как частично созданный (partially constructed) (т. е. объект пока еще нельзя считать находящимся в корректном состоянии), следовательно, его деструктор не будет вызываться:

```
#include <iostream>    // std::cout

struct S2
{
    S2() { std::cout << "S2() "; throw 0; }
```

```
    ~S2() { std::cout << "~S2() "; }
};

void f() try { S2 s; } catch(int) { }
    // Выводится только "S2() " на устройство стандартного вывода stdout
    // (деструктор объекта S2 никогда не вызывается).
```

Несмотря на то что деструктор частично созданного объекта не будет вызван, деструкторы каждого успешно созданного базового класса и членов данных продолжают вызываться:

```
#include <iostream>   // std::string

using std::cout;
struct A { A() { cout << "A() "; } ~A() { cout << "~A() "; } };
struct B { B() { cout << "B() "; } ~B() { cout << "~B() "; } };

struct C : B
{
    A d_a;

    C() { cout << "C() "; throw 0; }   // Неделегирующий конструктор,
                                       // который генерирует исключение.
    ~C() { cout << "~C() "; }          // Деструктор, который никогда не будет вызван.
};

void f() try { C c; } catch(int) { }
    // Выводится "B() A() C() ~A() ~B()" на устройство стандартного вывода stdout.
```

Обратите внимание: базовый класс `B` и член данных `d_a` типа `A` были созданы полностью, поэтому их соответствующие деструкторы вызываются, даже несмотря на то что деструктор самого класса `C` никогда не выполняется.

Но если генерируется исключение в теле делегирующего конструктора, то инициализируемый объект считается полностью созданным (fully constructed), так как целевой конструктор обязательно должен был вернуть управление делегирующему, следовательно, деструктор созданного объекта непременно вызывается:

```
#include <iostream>    // std::cout

struct S3
{
    S3()           { std::cout << "S3() ";             }
    S3(int) : S3() { std::cout << "S3(int) "; throw 0; }
    ~S3()          { std::cout << "~S3() ";            }
};

void f() try { S3 s(0); } catch(int) { }
    // Выводится "S3() S3(int) ~S3() " на устройство стандартного вывода stdout.
```

1.5.2. Варианты использования

1.5.2.1. *Устранение дублирования кода в нескольких конструкторах*

Многие считают устранение необоснованного дублирования кода наилучшей практической методикой. Когда одна обычная функция-член вызывает другую, это всегда допустимый вариант, но когда один конструктор напрямую вызывает другой конструктор, это не тот случай. Стандартные способы решения этой проблемы включали повторение кода или преобразование кода в закрытую функцию-член, которая вызывалась бы из нескольких конструкторов. Недостатком такого способа является то, что закры-

тая функция-член, не являющаяся конструктором, не может эффективно использовать списки инициализаторов членов для инициализации базовых классов и членов данных. Начиная с C++11 делегирующие конструкторы можно использовать для минимизации дублирования кода, когда одни и те же операции выполняются в нескольких конструкторах, без отказа от эффективной инициализации. Рассмотрим класс `IPV4Host`, представляющий конечный узел сети, которая может быть создана (1) с использованием 32-битового адреса и 16-битового номера порта или (2) с использованием строки адреса IPV4 в формате `XXX.XXX.XXX.XXX:XXXXX`[22]:

```
#include <cstdint>    // std::uint16_t, std::uint32_t
#include <string>     // std::string

class IPV4Host
{
    // ...
private:
    int connect(std::uint32_t address, std::uint16_t port);

public:
    IPV4Host(std::uint32_t address, std::uint16_t port)
    {
        if (!connect(address, port))     // Дублирование кода: НЕУДАЧНОЕ РЕШЕНИЕ.
        {
            throw ConnectionException{address, port};
        }
    }

    IPV4Host(const std::string& ip)
    {
        std::uint32_t address = extractAddress(ip);
        std::uint16_t port = extractPort(ip);

        if (!connect(address, port))      // Дублирование кода: НЕУДАЧНОЕ РЕШЕНИЕ.
        {
            throw ConnectionException{address, port};
        }
    }
};
```

До версии C++11 устранение такого дублирования кода потребовало бы введения отдельной закрытой вспомогательной функции, которую должен вызывать каждый конструктор:

```
// C++03 (устаревшая версия)
#include <cstdint>    // std::uint16_t, std::uint32_t

class IPV4Host
{
    // ...

private:
    int connect(std::uint32_t address, std::uint16_t port);
    void init(std::uint32_t address, std::uint16_t port)     // Вспомогательная функция.
    {
        if (!connect(address, port))     // Факторизованная реализация требуемой логики.
        {
            throw ConnectionException{address, port};
```

[22] Обратите внимание: это первоначальное проектное решение само по себе может быть неоптимальным, поскольку представление адреса IPV4 и номера порта может быть с пользой для дела вынесено в отдельный класс семантического значения, например `IPV4Address`, который сам может быть создан несколькими способами; см. подраздел 1.5.3.2.

```
        }
    }

public:
    IPV4Host(std::uint32_t address, std::uint16_t port)
    {
        init(address, port);     // Вызов факторизованной закрытой вспомогательной функции.
    }

    IPV4Host(const std::string& ip)
    {
        std::uint32_t address = extractAddress(ip);
        std::uint16_t port = extractPort(ip);

        init(address, port);     // Вызов факторизованной закрытой вспомогательной функции.
    }
};
```

После ввода в версии C++11 делегирующих конструкторов принимающий строку конструктор можно переписать для делегирования другому полученных аргументов `address` и `port`, избежав повторения без необходимости использования вспомогательной функции:

```
#include <cstdint>    // std::uint16_t, std::uint32_t
#include <string>     // std::string

class IPV4Host
{
    // ...

private:
    int connect(std::uint32_t address, std::uint16_t port);

public:
    IPV4Host(std::uint32_t address, std::uint16_t port)
    {
        if(!connect(address, port))
        {
            throw ConnectionException{address, port};
        }
    }

    IPV4Host(const std::string& ip) : IPV4Host{extractAddress(ip), extractPort(ip)}
    {
    }
};
```

Использование делегирующих конструкторов приводит к сокращению повторяющихся фрагментов кода и меньшему количеству операций во время выполнения, поскольку члены данных и базовые классы могут быть инициализированы непосредственно через список инициализации членов.

1.5.3. Потенциальные опасности

1.5.3.1. *Циклы делегирования*

Если конструктор выполняет делегирование самому себе прямо или косвенно, то программа становится «плохо (некорректно) согласованной, но диагностика не требуется» (ill formed, no diagnostic required – IFNDR (общепринятая аббревиатура в C++)). Хотя некоторые компиляторы способны при определенных условиях обнаружить циклы делегирования во время компиляции, такая возможность не является ни требуемой,

ни обязательной. Например, даже простейшие циклы делегирования могут не приводить к выводу диагностических сообщений компилятором[23]:

```
struct S     // Объект.
{
    S(int) : S(true) { }    // Делегирующий конструктор.
    S(bool) : S(0) { }      // Делегирующий конструктор.
};
```

1.5.3.2. *Неоптимальная факторизация*

Необходимость в делегировании конструкторов может возникать из-за изначально неоптимальной факторизации, допустим в случае, когда одно и то же значение представляется в разных формах для множества разнообразных механизмов. Например, рассмотрим класс `IPV4Host` в подразделе 1.5.2.1. Наличие двух конструкторов для инициализации хоста может быть уместным, если (1) увеличивается количество способов выражения одного и того же значения или (2) увеличивается число потребителей этого значения, тем не менее мы рекомендовали бы создать отдельный тип семантического значения, например `IPV4Endpoint`, для представления такого значения[24]:

```
#include <cstdint>    // std::uint16_t, std::uint32_t
#include <string>     // std::string

class IPV4Endpoint
{
    std::uint32_t d_address;
    std::uint16_t d_port;

public:
    IPV4Endpoint(std::uint32_t address, std::uint16_t port) : d_address{address}, d_port{port}
    {
    }

    IPV4Endpoint(const std::string& ep) : IPV4Endpoint{extractAddress(ep), extractPort(ep)}
    {
    }
};
```

Обратите внимание: сам класс `IPV4Endpoint` использует делегирующие конструкторы, но как абсолютно закрытые, инкапсулированные детали реализации. После введения `IPV4Endpoint` в кодовую базу класс `IPV4Host` (и аналогичные компоненты, для которых требуется значение `IPV4Endpoint`) можно переопределить, чтобы иметь один конструктор (или другую ранее перегруженную функцию-член), принимающий объект `IPV4Endpoint` как аргумент.

1.5.4. Смотрите также

- «Ссылки с перенаправлением» (раздел 2.10) – представлен превосходный способ перенаправления аргументов из одного конструктора в другой.

---

[23] На момент написания этой книги компилятор GCC 11.2 (с 2021 г.) не обнаруживает показанный здесь цикл делегирования во время компиляции и генерирует бинарный файл, который при запуске будет неизбежно демонстрировать неопределенное поведение (undefined behavior). С другой стороны, компилятор Clang 3.0 (с 2021 г.) и более поздних версий останавливает компиляцию с выводом информативного сообщения об ошибке:

```
error: constructor for S creates a delegation cycle
(ошибка: конструктор для S создает цикл делегирования)
```

[24] Концепцию, определяющую, что каждый компонент в подсистеме идеально выполняет одну четко определенную функцию, иногда называют разделением логических целей или весьма подробной физической факторизацией – см. **dijkstra82** и **lakos20**, раздел 0.4 «Hierarchically Reusable Software», раздел 3.2.7 «Not Just Minimal, Primitive: The Utility struct» и раздел 3.5.9 «Factoring», стр. 20–28, 529–530 и 674–676 соответственно.

- «Шаблоны с переменным количеством аргументов» (раздел 2.21) – описано, как реализовать конструкторы, которые перенаправляют произвольный список аргументов в другие конструкторы.

1.6. Использование = *delete* для любых функций

Ключевое слово `delete`, аннотирующее первое объявление функции, делает любую попытку ее использования или даже доступа к ней некорректной.

1.6.1. Описание

Объявление конкретной функции или перегрузки функции, приводящей к диагностике критической ошибки при вызове, может оказаться полезным, например, для подавления генерации специальной функции-члена или для ограничения типов аргументов, которые может принимать конкретное определение перегрузки. В таких случаях `= delete` с последующей точкой с запятой (;) можно использовать вместо тела любой функции при первом объявлении только для того, чтобы преднамеренно вызвать ошибку времени компиляции, если будет предпринята какая-либо попытка вызвать такую функцию или получить ее адрес.

```
void g(double) { }
void g(int) = delete;

void f()
{
    g(3.14);    // OK, функция f(double) вызывается.
    g(0);       // Ошибка: функция f(int) удалена.
}
```

Обратите внимание: удаленные функции участвуют в разрешении перегрузки (overload resolution), и при выборе такой функции как наилучшего кандидата возникает ошибка времени компиляции.

1.6.2. Варианты использования

1.6.2.1. *Подавление генерации специальных функций-членов*

При создании экземпляра объекта типа, определяемого пользователем, специальные функции-члены, которые не были объявлены явно, часто автоматически генерируются компилятором. На создание отдельных специальных функций-членов может повлиять существование других определяемых пользователем специальных функций-членов или ограничения, налагаемые особыми типами любых членов данных или базовых типов, – см. раздел 1.4. Для некоторых разновидностей типов понятие копирования не имеет смысла, следовательно, разрешение компилятору генерировать операции копирования было бы неуместным. Две специальные функции-члены, управляющие операциями перемещения, введенными в версии C++11, обычно реализуются как эффективная оптимизация операций копирования, следовательно, также противопоказаны. Гораздо реже полезное понятие перемещения существует там, где копирования нет, поэтому можно выбрать создание операций перемещения, в то время как операции копирования явно удаляются – см. раздел 2.18.

Рассмотрим класс `FileHandle`, который использует идиому RAII (resource acquisition is initialization – получение ресурса есть инициализация) для безопасного захвата и освобождения потока ввода/вывода. Поскольку семантика копирования обычно не имеет смысла для ресурсов такого рода, необходимо подавить генерацию копирующего конструктора и оператора присваивания с копированием. До версии C++11 не существовало прямого способа выразить подавление генерации специальных функций-членов в C++. Широко известный рекомендуемый способ решения этой проблемы заключался в объявлении этих двух методов закрытыми (private), при этом они оставались нереа-

лизованными, что обычно приводило к ошибке во время компиляции или во время компоновки при попытке доступа:

```
#include <cstdio>    // FILE
class FileHandle
{
private:
    // ...

    FileHandle(const FileHandle&);                // Не реализована.
    FileHandle& operator=(const FileHandle&);     // Не реализована.

public:
    explicit FileHandle(FILE* filePtr);
    ~FileHandle();

    // ...
};
```

Отсутствие реализации специальной функции-члена, объявленной закрытой, гарантирует, что будет возникать как минимум ошибка во время компоновки в случае непреднамеренной попытки доступа к ней из кода реализации самого этого класса. При использовании синтаксиса `= delete` появляется возможность (1) явно выразить намерение сделать эти специальные функции-члены недоступными, (2) сделать это прямо в области `public` класса и (3) позволить компилятору вывести более понятные диагностические сообщения:

```
class FileHandle
{
private:
    // ...
    // Объявление копирующего конструктора и операции присваивания с копированием
    // теперь расположены в разделе public.

public:
    explicit FileHandle(FILE* filePtr);
    ~FileHandle();

    FileHandle(const FileHandle&) = delete;               // Сделана недоступной.
    FileHandle& operator=(const FileHandle&) = delete;    // Сделана недоступной.

    // ...
};
```

Применение синтаксиса `= delete` в объявлениях закрытых членов приводит к выводу сообщений об ошибках, относящихся к характеристике закрытости, а не к использованию удаленных функций. Необходимо соблюдать осторожность при внесении обоих изменений при преобразовании старого кода к новому синтаксису.

1.6.2.2. *Запрещение конкретного неявного преобразования*

Для определенных функций, особенно для тех, которые принимают данные типа `char` как аргумент, существует большая вероятность случайного некорректного использования. В качестве действительно типичного примера рассмотрим библиотечную функцию языка C `memset`, которую можно использовать для записи символа `'*'` с пятикратным повторением в строке, начиная с заданного адреса памяти `buf`:

```
#include <cstdio>     // puts
#include <cstring>    // memset

void f()
{
    char buf[] = "Hello World!";
```

```
    memset(buf, 5, '*');    // Неопределенное поведение: переполнение буфера.
    puts(buf);              // Ожидаемый вывод: "***** World!".
}
```

К сожалению, непреднамеренное изменение порядка последних двух аргументов на противоположный является часто повторяющейся ошибкой, и язык C не оказывает никакой помощи, для того чтобы избежать этой ошибки. Как показано выше, `memset` записывает непечатаемый символ `'x5'` 42 раза (42 – целочисленное ASCII-значение, соответствующее символу `'*'`), выходя далеко за конечную границу буфера `buf`. В C++ можно устранить этот очевидный промах, используя дополнительную перегрузку с применением `= delete`:

```
namespace my {
void* memset(void* str, int ch, std::size_t n);    // Эквивалент функции стандартной библиотеки.
void* memset(void* str, int n, char) = delete;     // Защита от некорректного использования.
}
```

Теперь появляется возможность сообщать о фатальных пользовательских ошибках во время компиляции:

```
void f2()
{
    char buf[] = "Hello World!";
    my::memset(buf, 5, '*');                // Ошибка: вызов удаленной функции.
    my::memset(buf, '*', (std::size_t)5);   // OK.
}
```

1.6.2.3. *Запрещение всех неявных преобразований*

Приведенная ниже функция-член `ByteStream::send` предназначена для обработки только 8-битовых целых чисел без знака. Объявление удаленной перегружаемой функции, принимающей тип `int`, заставляет вызывающую сторону обеспечить передачу аргумента, который уже имеет требуемый тип:

```
class ByteStream
{
public:
    void send(unsigned char byte) { /*...*/ }
    void send(int) = delete;

    // ...
};

void f()
{
    ByteStream stream;
    stream.send(0);              // Ошибка: функция send(int) удалена.   (1)
    stream.send('a');            // Ошибка: функция send(int) удалена.   (2)
    stream.send(0L);             // Ошибка: неоднозначный аргумент.      (3)
    stream.send(0U);             // Ошибка: неоднозначный аргумент.      (4)
    stream.send(0.0);            // Ошибка: неоднозначный аргумент.      (5)
    stream.send( static_cast<unsigned char>(100));  // OK                (6)
}
```

Вызов функции `send` с аргументом типа `int`, помеченный (1) в приведенном выше коде, или любого целого типа, отличающегося от `unsigned char` и приводимого к `int` (2), будет отображаться исключительно в удаленную перегруженную функцию `sent(int)`. Все прочие целочисленные типы (3) и (4), а также типы с плавающей точкой (5) могут быть преобразованы в оба типа с помощью стандартного преобразования, следовательно, будут неоднозначными. Следует отметить, что неявное преобразование из `unsigned char` в `long` или `unsigned` целое предполагает стандартное преобразование (standard conversion) (а не только расширение по шкале целых типов, приведение типа – integral

promotion), то же самое относится к преобразованию в `double`. Явное приведение к типу `unsigned char` (6) может уже быть задействовано в сервисе, если это необходимо.

1.6.2.4. *Скрытие функции-члена структурного неполиморфного базового класса*

Обычно рекомендуется избегать открытого наследования от конкретных классов, потому что при этом мы не скрываем внутренние функциональные возможности, к которым можно легко получить доступ (потенциально нарушая любые инварианты, которые, возможно, требуется сохранить в производном классе) через присваивание указателю или ссылку на базовый класс без обязательного приведения типа. Хуже того, непреднамеренная передача такого класса в функцию, принимающую базовый класс по значению, приведет к секционированию («расслоению»), что может стать особенно проблематичным, если производный класс содержит данные. Более надежным подходом было бы использование уровней или, по крайней мере, закрытого наследования[25]. Несмотря на передовой практический опыт[26], в краткосрочной перспективе может оказаться эффективным с точки зрения накладных расходов создание скрытого «представления» о конкретном классе для клиентов, заслуживающих доверия. Представьте класс `AudioStream`, предназначенный для воспроизведения звуков и музыки, который – в дополнение к базовым операциям «воспроизведения» и «перемотки» – предъявляет обширный и надежный интерфейс:

```
struct AudioStream
{
    void play();
    void rewind();
    // ...
    // ... (Обширный и надежный интерфейс.)
    // ...
};
```

Предположим, что в срочном порядке мы должны создать аналогичный класс `ForwardAudioStream` для использования с аудиосемплами, которые нельзя перемотать, например поступающими непосредственно из прямой трансляции. Понимая, что можно с легкостью повторно использовать бóльшую часть интерфейса `AudioStream`, мы прагматично решаем создать прототип нового класса, просто применив открытое структурное наследование (structural inheritance), а затем удаляем только единственную ненужную функцию-член `rewind`:

```
struct ForwardAudioStream : AudioStream
{
    void rewind() = delete;     // Делаем только одну эту функцию недоступной.
};

void f()
{
    ForwardAudioStream stream = FMRadio::getStream();
    stream.play();       // Все верно.
    stream.rewind();     // Ошибка: функция rewind() удалена.
}
```

Если потребность в типе `ForwardAudioStream` сохраняется, то мы всегда можем рассмотреть возможность более тщательной повторной реализации его в дальнейшем[27]. Как было отмечено в начале данного раздела, защиту, обеспечиваемую этим примером, легко обойти:

[25] Для получения дополнительной информации об улучшении при масштабировании сложных проектных решений см. **lakos20**, раздел 3.5.10.5 «Realizing Multicomponent Wrappers» и раздел 3.7.3 «Improving Purely Compositional Designs», стр. 687–703 и 726–727 соответственно.

[26] См. **meyers92** «Item 38: Never defne an inherited default parameter value», стр. 132–135.

[27] **lakos20**, разделы 3.5.10.5 «Realizing Multicomponent Wrappers» и 3.7.3 «Improving Purely Compositional Designs», стр. 687–703 и 726–727 соответственно.

```
void g(const ForwardAudioStream &stream)
{
  AudioStream fullStream = stream;
  fullStream.play();    // OK
  fullStream.rewind();  // Этот код успешно компилируется, но что происходит во время выполнения?
}
```

Скрытие невиртуальных (non-virtual) функций начинают обеспечивать только после полного понимания того, что делает такой необычный прием безопасным, в частности см. раздел 3.2.

1.6.3. Неудобства

1.6.3.1. *Удаление функции фактически объявляет ее*

Не должно вызывать удивления следующее обстоятельство: когда после имени свободной функции следует = delete, мы фактически объявляем ее. Например, рассмотрим пару перегружаемых функций f, объявленных как принимающие аргументы типа char и int соответственно:

```
int f(char);          // (1) Допустимое объявление функции f, принимающей тип char.
int f(int) = delete;  // (2) Недопустимое объявление функции f, принимающей тип int.

int x = f('a');    // OK, точное соответствие для (1) f(char), которая является допустимой.
int y = f(123);    // Ошибка: точное соответствие для (2) f(int), которая является удаленной.
```

Обе приведенные выше функции обязательно должны быть объявлены, чтобы каждая из них могла принять участие в разрешении перегрузки, и только после того, как будет выбран недопустимый вариант перегрузки, о нем будет выведено сообщение как об ошибке времени компиляции.

Но когда дело доходит до удаления некоторых специальных функций-членов класса (или шаблона класса), использование элементов, которые, возможно, выглядят как совсем незначительный дополнительный самодокументируемый код, может иметь практически незаметные, непредсказуемые последствия, как показано ниже. Начнем с рассмотрения пустой структуры S0:

```
struct S0 { };    // Конструктор по умолчанию объявляется неявно.

S0 x0;            // OK, вызывается неявно сгенерированный конструктор по умолчанию.
```

Поскольку S0 не определяет конструкторы, деструкторы и операторы присваивания, компилятор сгенерирует (т. е. объявит и определит) для S0 все шесть специальных функций-членов, доступных в соответствии со стандартом C++11, – см. раздел 1.4.

Далее предположим, что мы создали вторую структуру S1, которая отличается от S0 только тем, что объявляет конструктор с передачей в него значения (конструктор с параметром), принимающий тип int:

```
struct S1      // Неявное объявление конструктора по умолчанию подавляется.
{
    S1(int);   // Явное объявление конструктора с параметром.
};

S1 y1(5);   // OK, вызывается явно объявленный конструктор с параметром.
S1 x1;      // Ошибка: нет объявления для конструктора по умолчанию S1::S1().
```

При явном объявлении конструктора с параметром (да и вообще любого другого конструктора) мы автоматически подавляем неявное объявление конструктора по умолчанию для структуры S1. Если подавление конструктора по умолчанию не является нашим намерением, то всегда можно восстановить его с помощью явного объявления, сопровождаемого = default (см. раздел 1.4).

Теперь предположим, что нашим намерением становится подавление генерации конструктора по умолчанию, и чтобы сделать его очевидным, мы решаем явно объявить и удалить его:

```
struct S2   // Неявное объявление конструктора по умолчанию подавляется.
{
    S2() = delete;   // Явное объявление недоступного конструктора по умолчанию.
    S2(int);         // Явное объявление конструктора с параметром.
};

S2 y2(5);   // OK, вызывается явно объявленный конструктор с параметром.
S2 x2;      // Ошибка: использование удаленной функции S2::S2().
```

После объявления и последующего удаления конструктора по умолчанию все должно выглядеть следующим образом: (1) мы сделали наши намерения явными и (2) улучшили диагностику для клиентских компонентов за счет единственной дополнительной строки самодокументируемого кода. Ах, если бы весь C++ был таким простым.

Удаление конкретных специальных функций-членов, т. е. конструктора по умолчанию, конструктора с перемещением или оператора присваивания с перемещением, не требующих неявного объявления, может иметь неочевидное последствие, которое отрицательно и воздействует на скрытые свойства класса во время компиляции. Одно из таких скрытых свойств связано с интерпретацией компилятором объекта такого класса как литерального типа, т. е. типа, значение которого можно использовать как часть константного выражения. Это же свойство принадлежности к литеральному типу определяет, может ли произвольный тип передаваться по значению в интерфейсе функции `constexpr` – см. раздел 2.5.

В качестве простого примера, демонстрирующего тонкое различие во время компиляции между `S1` и `S2`, рассмотрим следующий практически полезный шаблон для «тестовой» функции разработчика, которая будет компилироваться, если и только если передаваемый в нее по значению параметр `x` имеет литеральный тип:

```
constexpr int test(S0 x) { return 0; }    // OK,     S0  – литеральный тип.
constexpr int test(S1 x) { return 0; }    // Ошибка: S1 нелитеральный тип.
constexpr int test(S2 x) { return 0; }    // OK,     S2  – литеральный тип.
```

Чтобы компилятор интерпретировал данный тип класса как литеральный, такой класс должен, помимо всего прочего, иметь как минимум один конструктор (кроме копирующего конструктора или конструктора перемещения), объявленный как `constexpr`.

В случае с пустым классом `S0` неявно генерируемый конструктор по умолчанию является тривиальным (trivial), поэтому также неявно объявляется как `constexpr`. В классе `S1` явно объявленный не как `constexpr` конструктор с параметром подавляет объявление своего единственного `constexpr` конструктора, т. е. конструктора по умолчанию, следовательно, `S1` не интерпретируется как литеральный тип.

Наконец, явно объявляя и удаляя конструктор по умолчанию в `S2`, мы тем не менее объявляем его. Более того, объявление, вызванное его удалением, является точно таким же, как если бы оно было сгенерировано неявно (или объявлено явно, а затем установлено по умолчанию с помощью `= default`); следовательно, `S2`, в отличие от `S1`, является литеральным типом. Поди разберись во всем этом!

1.6.4. Смотрите также

- «Использование `= default` для специальных функций-членов» (раздел 1.4) – описание сопутствующей функциональной возможности, которая позволяет объявлять по умолчанию в противоположность удалению специальные функции-члены.
- «Ссылки на *rvalue*» (раздел 2.18) – объясняет, как эта функциональная возможность приводит к двум вариантам перемещения специальных функций-членов, которые также могут быть объявлены удаленными.