

# Эффективный и современный C++

42 РЕКОМЕНДАЦИИ ПО ИСПОЛЬЗОВАНИЮ C++11 И C++14

Скотт Мейерс

# Эффективный и современный C++

# Effective Modern C++

*Scott Meyers*

Beijing ♦ Cambridge ♦ Farnham ♦ Köln ♦ Sebastopol ♦ Tokyo ♦ O'Reilly

O'REILLY®

# Эффективный и современный C++

42 рекомендации по использованию C++11 и C++14

Скотт Мейерс



Москва • Санкт-Петербург • Киев  
2016

ББК 32.973.26-018.2.75

М45

УДК 681.3.07

Издательский дом “Вильямс”

Зав. редакцией С.Н. Трибу

Перевод с английского и редакция канд. техн. наук И.В. Красикова

По общим вопросам обращайтесь в Издательский дом “Вильямс” по адресу:  
info@williamspublishing.com, <http://www.williamspublishing.com>

**Мейерс, Скотт.**

**М45 Эффективный и современный C++: 42 рекомендации по использованию C++11 и C++14.** : Пер. с англ. – М. : ООО “И.Д. Вильямс”, 2016. – 304 с. : ил. – Парал. тит. англ.

ISBN 978-5-8459-2000-3 (рус.)

**БК 32.973.26-018.2.75**

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства O'Reilly & Associates.

Authorized Russian translation of the English edition of *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14* (ISBN 978-1-49-190399-5) © 2015 Scott Meyers.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the Publisher.

*Научно-популярное издание*  
**Скотт Мейерс**

## **Эффективный и современный C++: 42 рекомендации по использованию C++11 и C++14**

Литературный редактор *Л.Н. Красножон*

Верстка *О.В. Мишутина*

Художественный редактор *В.Г. Павлютин*

Корректор *Л.А. Гордиенко*

Подписано в печать 26.10.2015. Формат 70x100/16

Гарнитура Times

Усл. печ. л. 24,51 Уч.-изд. л. 18,3

Тираж 500 экз. Заказ № 6310

Отпечатано способом ролевой струйной печати

в АО “Первая Образцовая типография”

Филиал “Чеховский Печатный Двор”

142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1

ООО “И. Д. Вильямс”, 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-2000-3 (рус.)

ISBN 978-1-49-190399-5 (англ.)

© 2016 Издательский дом “Вильямс”

© 2015 Scott Meyers. All rights reserved.

# Оглавление

<b>Введение</b>	<b>15</b>
<b>Глава 1. Вывод типов</b>	<b>23</b>
<b>Глава 2. Объявление auto</b>	<b>49</b>
<b>Глава 3. Переход к современному C++</b>	<b>61</b>
<b>Глава 4. Интеллектуальные указатели</b>	<b>125</b>
<b>Глава 5. Rvalue-ссылки, семантика перемещений и прямая передача</b>	<b>165</b>
<b>Глава 6. Лямбда-выражения</b>	<b>221</b>
<b>Глава 7. Параллельные вычисления</b>	<b>245</b>
<b>Глава 8. Тонкости</b>	<b>281</b>
<b>Предметный указатель</b>	<b>301</b>

# Содержание

<b>Об авторе</b>	<b>11</b>
<b>Введение</b>	<b>15</b>
Терминология и соглашения	16
Замечания и предложения	20
От редакции	20
Ждем ваших отзывов!	21
<b>Глава 1. Вывод типов</b>	<b>23</b>
1.1. Вывод типа шаблона	23
1.2. Вывод типа auto	31
1.3. Знакомство с decltype	36
1.4. Как просмотреть выведенные типы	42
<b>Глава 2. Объявление auto</b>	<b>49</b>
2.1. Предпочитайте auto явному объявлению типа	49
2.2. Если auto выводит нежелательный тип, используйте явно типизированный инициализатор	54
<b>Глава 3. Переход к современному C++</b>	<b>61</b>
3.1. Различие между {} и () при создании объектов	61
3.2. Предпочитайте nullptr значениям 0 и NULL	69
3.3. Предпочитайте объявление псевдонимов применению typedef	73
3.4. Предпочитайте перечисления с областью видимости перечислениям без таковой	78
3.5. Предпочитайте удаленные функции закрытым неопределенным	84
3.6. Объявляйте перекрывающие функции как override	88
3.7. Предпочитайте итераторы const_iterator итераторам iterator	95
3.8. Если функции не генерируют исключений, объявляйте их как noexcept	98
3.9. Используйте, где это возможно, constexpr	105
3.10. Делайте константные функции-члены безопасными в смысле потоков	111
3.11. Генерация специальных функций-членов	116

## **Глава 4. Интеллектуальные указатели**

4.1. Используйте <code>std::unique_ptr</code> для управления ресурсами путем исключительного владения	126
4.2. Используйте <code>std::shared_ptr</code> для управления ресурсами путем совместного владения	133
4.3. Используйте <code>std::weak_ptr</code> для <code>std::shared_ptr</code> -подобных указателей, которые могут быть висячими	142
4.4. Предпочитайте использование <code>std::make_unique</code> и <code>std::make_shared</code> непосредственному использованию оператора <code>new</code>	146
4.5. При использовании идиомы указателя на реализацию определяйте специальные функции-члены в файле реализации	155

## **Глава 5. Rvalue-ссылки, семантика перемещений и прямая передача**

5.1. Азы <code>std::move</code> и <code>std::forward</code>	166
5.2. Отличие универсальных ссылок от rvalue-ссылок	171
5.3. Используйте <code>std::move</code> для rvalue-ссылок, а <code>std::forward</code> — для универсальных ссылок	176
5.4. Избегайте перегрузок для универсальных ссылок	184
5.5. Знакомство с альтернативами перегрузки для универсальных ссылок	190
Отказ от перегрузки	190
Передача <code>const T&amp;</code>	190
Передача по значению	190
Диспетчеризация дескрипторов	191
Ограничения шаблонов, получающих универсальные ссылки	194
Компромиссы	200
5.6. Свертывание ссылок	202
5.7. Считайте, что перемещающие операции отсутствуют, дороги или не используются	208
5.8. Познакомьтесь с случаями некорректной работы прямой передачи	211
Инициализаторы в фигурных скобках	213
0 и NULL в качестве нулевых указателей	214
Целочисленные члены-данные <code>static const</code> и <code>constexpr</code> без определений	214
Имена перегруженных функций и имена шаблонов	216
Битовые поля	217
Резюме	219

## **Глава 6. Лямбда-выражения**

6.1. Избегайте режимов захвата по умолчанию	222
6.2. Используйте инициализирующий захват для перемещения объектов в замыкания	229

6.3. Используйте параметры decltype для auto&& для передачи с помощью std::forward	234
6.4. Предпочитайте лямбда-выражения применению std::bind	237
<b>Глава 7. Параллельные вычисления</b>	<b>245</b>
7.1. Предпочитайте программирование на основе задач программированию на основе потоков	245
7.2. Если важна асинхронность, указывайте std::launch::async	249
7.3. Делайте std::thread неподключаемым на всех путях выполнения	254
7.4. Помните о разном поведении деструкторов дескрипторов потоков	260
7.5. Применяйте фьючерсы void для одноразовых сообщений о событиях	265
7.6. Используйте std::atomic для параллельности, volatile — для особой памяти	272
<b>Глава 8. Тонкости</b>	<b>281</b>
8.1. Рассмотрите передачу по значению для копируемых параметров, которые легко перемещаются и всегда копируются	281
8.2. Рассмотрите применение размещения вместо вставки	291
<b>Предметный указатель</b>	<b>301</b>

## **Отзывы о книге “Эффективный и современный C++”**

Вас интересует C++? Современный C++ (т.е. C++11/C++14) — это гораздо больше чем простое внесение косметических изменений в старый стандарт. Учитывая новые возможности языка, это скорее его переосмысление. Вам нужна помошь в его освоении? Тогда перед вами именно та книга, которая вам нужна. Что касается C++, то Скотт Мейерс был и остается синонимом точности, качества и удовольствия от чтения.

— Герхард Крейцер (*Gerhard Kreuzer*)  
Инженер-исследователь в Siemens AG

Трудно получить достаточный опыт и стать экспертом. Не менее трудно стать настоящим учителем, способным просто и ясно донести сложный материал до ученика. Если вы читаете эту книгу, то вы знаете человека, который объединяет оба эти качества. Книга Эффективный и современный C++ написана непревзойденным техническим писателем, который умеет излагать сложные взаимосвязанные темы ясно и понятно, блестящим литературным стилем. При этом вряд ли вам удастся найти в книге хотя бы одну техническую ошибку.

— Андрей Александреску (*Andrei Alexandrescu*)  
Доктор философии, исследователь, автор книги Современное проектирование на C++

Когда человек с более чем двадцатилетним опытом работы с C++ берется рассказать, как получить максимальную отдачу от современного C++ (рассказывая как о лучших подходах, так и о возможных ловушках, которых следует избегать), я настоятельно рекомендую внимательно прочесть его книгу! Я, определенно, узнал из нее много нового!

— Невин Либер (*Nevin Liber*)  
Старший программист в DRW Trading Group

Бьярне Страуструп — создатель C++ — сказал: “C++11 выглядит как новый язык программирования”. Книга Эффективный и современный C++ заставляет нас разделить это впечатление, поясняя, как использовать новые возможности и идиомы C++11 и C++14 в повседневной практике. Это еще одна талантливая книга Скотта Мейерса.

— Кассио Нери (*Cassio Neri*)  
Аналитик в Lloyds Banking Group

Скотт умеет добраться до самой сути любой технической проблемы. Книги серии Эффективный C++ способствовали улучшению стиля кодирования предыдущего поколения программистов C++; новая книга делает то же самое с программистами на современном C++.

— Роджер Орр (*Roger Orr*)  
OR/2 Limited, член Комитета ISO по стандартизации C++

*Эффективный и современный C++* — отличный инструмент для повышения вашего уровня как программиста на современном C++. Книга не только учит тому, как, когда и где эффективно использовать современный C++, но и почему делать это именно так. Вне всякого сомнения, эта книга Скотта Мейерса даст программистам гораздо лучшее понимание языка.

— Bart Vandewoestijn (*Bart Vandewoestijn*)  
Инженер, исследователь и просто энтузиаст C++

Я люблю C++, он десятилетиями был моей рабочей лошадкой. А с новыми копытами эта лошадка стала еще сильнее и привлекательнее, чем я мог ранее себе представить. Но при больших изменениях всегда встает вопрос “Когда и как пользоваться всем этим богатством?” Как всегда, книга Скотта Мейерса компетентно и исчерпывающе отвечает на поставленный вопрос.

— Damien Watkins (*Damien Watkins*)  
Руководитель группы программной инженерии в CSIRO

Отличное чтение для перехода к современному C++ — новинки языка C++11/14 описаны наряду с C++98, разделение содержимого книги на разделы позволяет легко найти интересующую тему, а в конце каждого раздела приведены итоговые рекомендации. Книга интересна и полезна для программистов на C++ всех уровней.

— Рейчел Чэнг (*Rachel Cheng*)  
F5 Networks

Если вы переходите с C++98/03 на C++11/14, вам нужна точная практическая информация, которую вам предоставляет Скотт Мейерс в книге *Эффективный и современный C++*. Если вы уже пишете код на C++11, то, вероятно, сталкивались с проблемами при использовании новых возможностей, которые легко решаются с помощью книги Скотта. В любом случае можно уверенно утверждать, что время, затраченное на чтение этой книги, не пропадет впустую.

— Роб Стюарт (*Rob Stewart*)  
Член Boost Steering Committee ([boost.org](http://boost.org))

# Об авторе

**Скотт Мейерс** — один из ведущих мировых экспертов по C++, широко востребованный как инструктор, консультант и докладчик на разных конференциях. Более чем 20 лет книги Скотта Мейерса серии *Эффективный C++* являются критерием уровня книг по программированию на C++. Скотт Мейерс имеет степень доктора философии (Ph.D.) в области компьютерных наук в Университете Брауна (Brown University). Его сайт находится по адресу [aristeia.com](http://aristeia.com).

## Об изображении на обложке

На обложке книги *Эффективный и современный C++* изображен розовошапочный пестрый голубь (*Ptilinopus regina*). Еще одно имя этого вида голубя — *Свенсонов пестрый голубь* (*Swainson's fruit dove*). Он отличается ярким оперением: серые голова и грудь, оранжевый живот, беловатое горло, желто-оранжевый цвет радужки и серо-зеленые ноги.

Голубь распространен в равнинных лесах восточной Австралии, муссонных лесах северной части Австралии, а также на малых Зондских островах и Молуккских островах Индонезии. Рацион голубя состоит из различных фруктов наподобие инжира (который он поедает целиком), пальм и лоз. Еще одним источником пищи голубя является камфорный лавр, большое вечнозеленое дерево. Они питаются парами, небольшими стайками или поодиночке в тропических лесах, обычно утром или поздно вечером. Для питья они используют воду из листьев или росу, но не воду с земли.

Пестрый голубь в Новом Южном Уэльсе считается видом, находящимся под угрозой исчезновения из-за изменения среды обитания — лесозаготовок, очистки и улучшения земель, а также вырубки камфорного лавра без адекватной альтернативы.

Многие из животных, представленных на обложках O'Reilly, находятся под угрозой исчезновения. Все они имеют очень важное значение для нашего мира. Чтобы узнать больше о том, как вы можете помочь им, посетите сайт [animals.oreilly.com](http://animals.oreilly.com).

Изображение для обложки взято из *Иллюстрированной истории природы* Вуда (Wood), из тома, посвященного птицам.

Дарле,  
выдающемуся черному лабрадору-ретриверу

# Благодарности

Я начал исследовать то, что тогда было известно как C++0x (зарождающийся C++11), в 2009 году. Я опубликовал множество вопросов в группе новостей Usenet comp.std.c++ и благодарен членам этого сообщества (в особенности Дэниелу Крюглеру (Daniel Krugler)) за их очень полезные сообщения. В последние годы с вопросами по C++11 и C++14 я обращался к Stack Overflow, и я многим обязан этому сообществу за его помощь в понимании тонкостей современного C++.

В 2010 году я подготовил материалы для учебного курса по C++0x (в конечном итоге опубликованные как *Overview of the New C++*, Artima Publishing, 2010). На эти материалы, как и на мои знания, большое влияние оказала техническая проверка, выполненная Стивеном Т. Лававеем (Stephan T. Lavavej), Бернгардом Меркль (Bernhard Merkle), Стенли Фризеном (Stanley Friesen), Леором Зорманом (Leor Zolman), Хендриком Шобером (Hendrik Schober) и Энтони Вильямсом (Anthony Williams). Без их помощи я не смог бы довести мою книгу до конца. Кстати, ее название было предложено несколькими читателями в ответ на мое сообщение в блоге от 18 февраля 2014 года (“Помогите мне назвать мою книгу”), и Андрей Александреску (Andrei Alexandrescu) (автор книги *Modern C++ Design*<sup>1</sup>, Addison-Wesley, 2001) был достаточно великодушен, чтобы не счесть это название незаконным вторжением на его территорию.

Я не могу указать источники всей информации в этой книге, но некоторые из них непосредственно повлияли на мою книгу. Применение в разделе 1.4 неопределенного шаблона для получения информации о типе от компилятора было предложено Стивеном Т. Лававеем, а Мэтт П. Дзюбински (Matt P. Dziubinski) обратил мое внимание на Boost.TypeIndex. В разделе 2.1 пример `unsigned std::vector<int>::size_type` взят из статьи Андрея Карпова (Andrey Karpov) от 28 февраля 2010 года “*In what way can C++0x standard help you eliminate 64-bit errors*”. Пример `std::pair<std::string, int>/std::pair<const std::string, int>` в том же разделе книги взят из сообщения “*STL11: Magic & Secrets*” Стивена Т. Лававея на *Going Native 2012*. Раздел 2.2 появился благодаря статье Герба Саттера (Herb Sutter) “*Got W*

<sup>1</sup> Русский перевод — Александреску Андрей. *Современное проектирование на C++*. — М.: Издательский дом “Вильямс”, 2002.

#94 Solution: AAA Style (Almost Always Auto)" от 12 августа 2013 года, а раздел 3.3 — благодаря сообщению в блоге Мартинго Фернандеса (Martinho Fernandes) от 27 мая 2012 года — "Handling dependent names". Пример в разделе 3.6 демонстрирует перегрузку квалификаторов ссылок, основанную на ответе Кейси (Casey) на вопрос "What's a use case for overloading member functions on reference qualifiers?" в сообществе Stack Overflow от 14 января 2014 года. В разделе 3.9 описание расширенной поддержки `constexpr`-функций в C++14 включает информацию, которую я получил от Рейна Халберсма (Rein Halbersma). Раздел 3.10 основан на презентации Герба Саттера на конференции C++ and Beyond 2012 под названием "You don't know `const` и `mutable`". Совет в разделе 4.1, гласящий, что фабричная функция должна возвращать `std::unique_ptr`, основан на статье Герба Саттера "GotW# 90 Solution: Factories" от 13 мая 2013 года. `fastLoadWidget` в разделе 4.2 получен из презентации Герба Саттера "My Favorite C++ 10-Liner" на конференции Going Native 2013. В моем описании `std::unique_ptr` и неполных типов в разделе 4.5 использованы статья Герба Саттера от 27 ноября 2011 года "GotW #100: Compilation Firewalls", а также ответ Говарда Хиннанта (Howard Hinnant) от 22 мая 2011 года на вопрос в Stack Overflow "Is `std::unique_ptr<T>` required to know the full definition of `T`?" Дополнительный пример `Matrix` в разделе 5.3 основан на письме Дэвида Абрахамса (David Abrahams). Комментарий Джо Аргонна (Joe Argonne) от 8 декабря 2012 года к материалу из блога "Another alternative to lambda move capture" от 30 ноября 2013 года стал источником для описанного в разделе 6.2 подхода к имитации инициализации на основе `std::bind` в C++11. Пояснения в разделе 7.3 проблемы с неявным отключением в деструкторе `std::thread` взяты из статьи Ганса Бехма (Hans-J. Boehm) "N2802: A plea to reconsider detach-on-destruction for thread objects" от 4 декабря 2008 года. Раздел 8.1 появился благодаря обсуждению материала в блоге Дэвида Абрахамса "Want speed? Pass by value" от 15 августа 2009 года. Идея о том, что типы, предназначенные только для перемещения, должны рассматриваться отдельно, взята у Мэттью Фьюранте (Matthew Fioravante), в то время как анализ копирования на основе присваивания взят из комментариев Говарда Хиннанта (Howard Hinnant). В разделе 8.2 Стивен Т. Лававей и Говард Хиннант помогли мне понять вопросы, связанные с относительной производительностью функций размещения и вставки, а Майкл Винтерберг (Michael Winterberg) привлек мое внимание к тому, как размещение может приводить к утечке ресурсов. (Майкл, в свою очередь, называет своим источником презентацию "C++ Seasoning" Шона Парента (Sean Parent) на конференции Going Native 2013. Майкл также указал, что функции размещения используют непосредственную инициализацию, в то время как функции вставки используют копирующую инициализацию.)

Проверка черновиков технической книги является длительной и критичной, но совершенно необходимой работой, и мне повезло, что так много людей были готовы за нее взяться. Черновики этой книги были официально просмотрены такими специалистами, как Кассио Нери (Cassio Neri), Нейт Кёль (Nate Kohl), Герхард Крейцер (Gerhard Kreuzer), Леон Золман (Leor Zolman), Барт Вандевоистин (Bart Vandewoestyne), Стивен Т. Лававей (Stephan T. Lavavej), Невин Либер (Nevin ":-" Liber), Речел Чэнг (Rachel Cheng), Роб Стюарт (Rob Stewart), Боб Стигалл (Bob Steagall), Дамьян Уоткинс (Damien Watkins), Брэдли Нидхам (Bradley E. Needham), Рейнер Гримм (Rainer Grimm), Фредрик Винклер (Fredrik Winkler), Джонатан Уокели (Jonathan Wakely), Герб Саттер (Herb Sutter), Андрей

Александреску (Andrei Alexandrescu), Эрик Ниблер (Eric Niebler), Томас Беккер (Thomas Becker), Роджер Опп (Roger Opp), Энтони Вильямс (Anthony Williams), Майкл Винтерберг (Michael Winterberg), Бенджамин Хахли (Benjamin Huchley), Том Кирби-Грин (Tom Kirby-Green), Алексей Никитин (Alexey A. Nikitin), Вильям Дилтрай (William Dealtry), Хуберт Мэттьюс (Hubert Matthews) и Томаш Каминьски (Tomasz Kamiński). Я также получил отзывы ряда читателей с помощью сервисов O'Reilly's Early Release EBooks и Safari Books Online's Rough Cuts, посредством комментариев в моем блоге (*The View from Aristeia*) и электронной почты. Я благодарен каждому, кто высказал свои замечания. Эта книга получилась гораздо лучше, чем она была бы без этой помощи. В особенности я признателен Стивену Т. Лававею и Робу Стюарту, чьи чрезвычайно подробные и всеохватывающие замечания заставили меня забеспокоиться: кто из нас потратил больше сил и времени на эту книгу — я или они? Моя особая благодарность — Леору Золману (Leor Zolman), который не только просмотрел рукопись, но и дважды проверил все приведенные в ней примеры кода.

Черновики цифровых версий книги были подготовлены Герхардом Крейцером (Gerhard Kreuzer), Эмиром Вильямсом (Emry Williams) и Брэдли Нидхэмом (Bradley E. Needham).

Мое решение ограничить длину строки кода 64 символами (максимум для правильного отображения на печати, а также на различных цифровых устройствах при разной ориентации и конфигурации шрифтов) было основано на данных, предоставленных Майклом Махером (Michael Maher).

С момента первой публикации я исправил ряд ошибок и внес некоторые усовершенствования, предложенные такими читателями, как Костас Влахавас (Kostas Vlahavas), Да-ниэль Алонсо Алеман (Daniel Alonso Alemany), Такатоши Кондо (Takatoshi Kondo), Бартек Сургот (Bartek Szurgot), Тайлер Брок (Tyler Brock), Джай Ципник (Jay Zipnick), Барри Ревзин (Barry Revzin), Роберт Маккейб (Robert McCabe), Оливер Брунс (Oliver Bruns), Фабрис Ферино (Fabrice Ferino), Дэнэз Джонитис (Dainis Jonitis), Петр Валашек (Petr Valasek) и Барт Вандевойстин (Bart Vandewoestyne). Большое спасибо всем им за помощь в повышении точности и ясности изложенного материала.

Эшли Морган Вильямс (Ashley Morgan Williams) готовила отличные обеды у себя в Lake Oswego Pizzicato. Им (и ей) нет равных.

И более двадцати лет моя жена, Нэнси Л. Урбано (Nancy L. Urbano), как обычно во время моей работы над новой книгой, терпит мою раздражительность и оказывает мне всесмерную поддержку. В ходе написания книги постоянным напоминанием о том, что за пределами клавиатуры есть другая жизнь, служила мне наша собака Дарла.

# Введение

Если вы — опытный программист на языке программирования C++, как, например, я, то, наверное, первое, о чем вы подумали в связи с C++11, — “Да, да, вот и он — тот же C++, только немного улучшенный”. Но познакомившись с ним поближе, вы, скорее всего, были удивлены количеством изменений. Объявления `auto`, циклы `for` для диапазонов, лямбда-выражения и `value`-ссылки изменили лицо C++, — и это не говоря о новых возможностях параллельности. Произошли и идиоматические изменения. `0` и `typedef` уступили место `nullptr` и объявлениям псевдонимов. Перечисления получили области видимости. Интеллектуальные указатели стали предпочтительнее встроенных; перемещение объектов обычно предпочтительнее их копирования.

Даже без упоминания C++14 в C++11 есть что поизучать.

Что еще более важно, нужно очень многое изучить, чтобы использовать новые возможности эффективно. Если вам нужна базовая информация о “современных” возможностях C++, то ее можно найти в избытке. Но если вы ищете руководство о том, как использовать эти возможности для создания правильного, эффективного, сопровождаемого и переносимого программного обеспечения, поиск становится более сложным. Вот здесь вам и пригодится данная книга. Она посвящена не описанию возможностей C++11 и C++14, а их эффективному применению.

Информация в книге разбита на отдельные разделы, посвященные тем или иным рекомендациям. Вы хотите разобраться в разных видах вывода типов? Или хотите узнать, когда следует (а когда нет) использовать объявление `auto`? Вас интересует, почему функция-член, объявленная как `const`, должна быть безопасна с точки зрения потоков, как реализовать идиому `Rimpl` с использованием `std::unique_ptr`, почему следует избегать режима захвата по умолчанию в лямбда-выражениях или в чем различие между `std::atomic` и `volatile`? Ответы на эти вопросы вы найдете в книге. Более того, эти ответы не зависят от платформы и соответствуют стандарту. Это книга о *переносимом C++*.

Разделы книги представляют собой рекомендации, а не жесткие правила, поскольку рекомендации имеют исключения. Наиболее важной частью каждого раздела является не предлагаемая в нем рекомендация, а ее обоснование. Прочитав раздел, вы сможете сами определить, оправдывают ли обстоятельства вашего конкретного проекта отход от данной рекомендации. Истинная цель книги не в том, чтобы рассказать вам, как надо поступать или как поступать не надо, а в том, чтобы обеспечить вас более глубоким пониманием, как та или иная концепция работает в C++11 и C++14.

# Терминология и соглашения

Чтобы мы правильно понимали друг друга, важно согласовать используемую терминологию, начиная, как ни странно это звучит, с термина “C++”. Есть четыре официальные версии C++, и каждая именуется с использованием года принятия соответствующего стандарта ISO: C++98, C++03, C++11 и C++14. C++98 и C++03 отличаются один от другого только техническими деталями, так что в этой книге обе версии я называю как C++98. Говоря о C++11, я подразумеваю и C++11, и C++14, поскольку C++14 является надмножеством C++11. Когда я пишу “C++14”, я имею в виду конкретно C++14. А если я просто упоминаю C++, я делаю утверждение, которое относится ко всем версиям языка.

Использованный термин	Подразумеваемая версия
C++	Все
C++98	C++98 и C++03
C++11	C++11 и C++14
C++14	C++14

В результате я мог бы сказать, что в C++ придается большое значение эффективности (справедливо для всех версий), в C++98 отсутствует поддержка параллелизма (справедливо только для C++98 и C++03), C++11 поддерживает лямбда-выражения (справедливо для C++11 и C++14) и C++14 предлагает обобщенный вывод возвращаемого типа функции (справедливо только для C++14).

Наиболее важной особенностью C++11, вероятно, является семантика перемещения, а основой семантики перемещения является отличие *rvalue*-выражений от *lvalue*-выражений. Поэтому *rvalue* указывают объекты, которые могут быть перемещены, в то время как *lvalue* в общем случае перемещены быть не могут. Концептуально (хотя и не всегда на практике), *rvalue* соответствуют временным объектам, возвращаемым из функций, в то время как *lvalue* соответствуют объектам, на которые вы можете ссылаться по имени, следуя указателю или *lvalue*-ссылке.

Полезной эвристикой для выяснения, является ли выражение *lvalue*, является ответ на вопрос, можно ли получить его адрес. Если можно, то обычно это *lvalue*. Если нет, это обычно *rvalue*. Приятной особенностью этой эвристики является то, что она помогает помнить, что тип выражения не зависит от того, является ли оно *lvalue* или *rvalue*. Иначе говоря, для данного типа T можно иметь как *lvalue* типа T, так и *rvalue* типа T. Особенно важно помнить это, когда мы имеем дело с параметром *rvalue* ссылочного типа, поскольку сам по себе параметр является *lvalue*:

```
class Widget {  
public:  
    Widget(Widget&& rhs); // rhs является lvalue, хотя  
                           // и имеет ссылочный тип rvalue  
};
```

Здесь совершенно корректным является взятие адреса `rhs` в перемещающем конструкторе `Widget`, так что `rhs` представляет собой `lvalue`, несмотря на то что его тип — ссылка `rvalue`. (По сходным причинам все параметры являются `lvalue`.)

Этот фрагмент кода демонстрирует несколько соглашений, которым я обычно следую.

- Имя класса — `Widget`. Я использую слово `Widget`, когда хочу сослаться на произвольный пользовательский тип. Если только мне не надо показать конкретные детали класса, я использую имя `Widget`, не объявляя его.
- Я использую имя параметра `rhs` (“right-hand side”, правая сторона). Это предпочтительное мною имя параметра для *операций перемещения* (например, перемещающего конструктора и оператора перемещающего присваивания) и *операций копирования* (например, копирующего конструктора и оператора копирующего присваивания). Я также использую его в качестве правого параметра бинарных операторов:

```
Matrix operator+(const Matrix& lhs, const Matrix& rhs);
```

Я надеюсь, для вас не станет сюрпризом, что `lhs` означает “left-hand side” (левая сторона).

- Я использую специальное форматирование для частей кода или частей комментариев, чтобы привлечь к ним ваше внимание. В перемещающем конструкторе `Widget` выше я подчеркнул объявление `rhs` и часть комментария, указывающего, что `rhs` представляет собой `lvalue`. Выделенный код сам по себе не является ни плохим, ни хорошим. Это просто код, на который вы должны обратить внимание.
- Я использую “...”, чтобы указать “здесь находится прочий код”. Такое “узкое” троеточие отличается от широкого “...”, используемого в исходных текстах шаблонов с переменным количеством параметров в C++11. Это кажется запутанным, но на самом деле это не так. Вот пример.

```
template<typename... Ts> // Эти троеточки
void processVals(const Ts&... params) // в исходном
{ // тексте C++
    // Это троеточие означает какой-то код
}
```

Объявление `processVals` показывает, что я использую ключевое слово `typename` при объявлении параметров типов в шаблонах, но это просто мое личное предпочтение; вместо него можно использовать ключевое слово `class`. В тех случаях, когда я показываю код, взятый из стандарта C++, я объявляю параметры типа с использованием ключевого слова `class`, поскольку так делает стандарт.

Когда объект инициализирован другим объектом того же типа, новый объект является *копией инициализирующего* объекта, даже если копия создается с помощью перемещающего конструктора. К сожалению, в C++ нет никакой терминологии, которая позволяла бы различать объекты, созданные с помощью копирующих и перемещающих конструкторов:

```

void someFunc(Widget w); // Параметр w функции someFunc
                        // передается по значению

Widget wid;           // wid – объект класса Widget

someFunc(wid);        // В этом вызове someFunc w
                        // является копией wid, созданной
                        // копирующим конструктором

someFunc(std::move(wid)); // В этом вызове SomeFunc w
                        // является копией wid, созданной
                        // перемещающим конструктором

```

Копии `rvalue` в общем случае конструируются перемещением, в то время как копии `lvalue` обычно конструируются копированием. Следствием является то, что если вы знаете только то, что объект является копией другого объекта, то невозможно сказать, насколько дорогостоящим является создание копии. В приведенном выше коде, например, нет возможности сказать, насколько дорогостоящим является создание параметра `w`, без знания того, какое значение передано функции `someFunc` — `rvalue` или `lvalue`. (Вы также должны знать стоимости перемещения и копирования `Widget`.)

В вызове функции выражения, переданные в источнике вызова, являются *аргументами* функции. Эти аргументы используются для инициализации *параметров* функции. В первом вызове `someFunc`, показанном выше, аргументом является `wid`. Во втором вызове аргументом является `std::move(wid)`. В обоих вызовах параметром является `w`. Разница между аргументами и параметрами важна, поскольку параметры являются `lvalue`, но аргументы, которыми они инициализируются, могут быть как `rvalue`, так и `lvalue`. Это особенно актуально во время *прямой передачи*, при которой аргумент, переданный функции, передается другой функции так, что при этом сохраняется его “правосторонность” или “левосторонность”. (Прямая передача подробно рассматривается в разделе 5.8.)

Хорошо спроектированные функции *безопасны с точки зрения исключений*, что означает, что они обеспечивают как минимум базовую *гарантию*, т.е. гарантируют, что, даже если будет сгенерировано исключение, инварианты программы останутся нетронутыми (т.е. не будут повреждены структуры данных) и не будет никаких утечек ресурсов. Функции, обеспечивающие *строгую гарантию*, гарантируют, что, даже если будет сгенерировано исключение, состояние программы останется тем же, что и до вызова функции.

Говоря о *функциональном объекте*, я обычно имею в виду объект типа, поддерживающего функцию-член `operator()`. Другими словами, это объект, действующий, как функция. Иногда я использую термин в несколько более общем смысле для обозначения чего угодно, что может быть вызвано с использованием синтаксиса вызова функции, не являющейся членом (т.е. `function Name(arguments)`). Это более широкое определение охватывает не только объекты, поддерживающие `operator()`, но и функции и указатели на функции в стиле С. (Более узкое определение происходит из C++98, более широкое — из C++11.) Дальнейшее обобщение путем добавления указателей на функции-члены дает то, что известно как *вызываемый объект* (*callable object*). Вообще говоря,

можно игнорировать эти тонкие отличия и просто рассматривать функциональные и вызываемые объекты как сущности в C++, которые могут быть вызваны с помощью некоторой разновидности синтаксиса вызова функции.

Функциональные объекты, создаваемые с помощью лямбда-выражений, известны как *замыкания* (closures). Различать лямбда-выражения и замыкания, ими создаваемые, приходится редко, так что я зачастую говорю о них обоих как о лямбдах (*lambda*). Точно так же я редко различаю *шаблоны функций* (function templates) (т.е. шаблоны, которые генерируют функции) и *шаблонные функции* (template functions) (т.е. функции, сгенерированные из шаблонов функций). То же самое относится к *шаблонам классов* и *шаблонным классам*.

Многие сущности в C++ могут быть как объявлены, так и определены. *Объявления* вводят имена и типы, не детализируя информацию о них, такую как их местоположение в памяти или реализация:

```
extern int x;                      // Объявление объекта

class Widget;                      // Объявление класса

bool func(const Widget& w);        // Объявление функции

enum class Color;                  // Объявление перечисления
                                    // с областью видимости
                                    // (см. раздел 3.4)
```

*Определение* предоставляет информацию о расположении в памяти и деталях реализации:

```
int x;                            // Определение объекта

class Widget {                     // Определение класса

};

bool func(const Widget& w)
{ return w.size() < 10; }          // Определение функции

enum class Color
{ Yellow, Red, Blue };           // Определение перечисления
```

Определение можно квалифицировать и как *объявление*, так что, если только то, что нечто представляет собой определение, не является действительно важным, я предпочитаю использовать термин “*объявление*”.

Сигнатуру функции я определяю как часть ее *объявления*, определяющую типы параметров и возвращаемый тип. Имена функции и параметров значения не имеют. В приведенном выше примере сигнатура функции `func` представляет собой `bool(const Widget&)`. Исключаются элементы *объявления* функции, отличные от типов ее параметров и возвращаемого типа (например, `noexcept` или `constexpr`, если таковые имеются). (Модификаторы `noexcept` и `constexpr` описаны в разделах 3.8

и 3.9.) Официальное определение термина “сигнатура” несколько отличается от моего, но в данной книге мое определение оказывается более полезным. (Официальное определение иногда опускает возвращаемый тип.)

Новый стандарт C++ в общем случае сохраняет корректность кода, написанного для более старого стандарта, но иногда Комитет по стандартизации не рекомендует применять те или иные возможности. Такие возможности находятся в “камере смертников” стандартизации и могут быть убраны из новых версий стандарта. Компиляторы могут предупреждать об использовании программистом таких устаревших возможностей (но могут и не делать этого), но в любом случае их следует избегать. Они могут не только привести в будущем к головной боли при переносе, но и в общем случае они ниже по качеству, чем возможности, заменившие их. Например, `std::auto_ptr` не рекомендуется к применению в C++11, поскольку `std::unique_ptr` выполняет ту же работу, но лучше.

Иногда стандарт гласит, что результатом операции является *неопределенное поведение* (*undefined behavior*). Это означает, что поведение времени выполнения непредсказуемо, и от такой непредсказуемости, само собой разумеется, следует держаться подальше. Примеры действий с неопределенным поведением включают использование квадратных скобок (`[]`) для индексации за границами `std::vector`, разыменование неинициализированного итератора или гонку данных (т.е. когда два или более потоков, как минимум один из которых выполняет запись, одновременно обращаются к одному и тому же месту в памяти).

Я называю встроенный указатель, такой как возвращаемый оператором `new`, *обычным указателем* (*raw pointer*). Противоположностью обычному указателю является *интеллектуальный указатель* (*smart pointer*). Интеллектуальные указатели обычно перегружают операторы разыменования указателей (`operator->` и `operator*`), хотя в разделе 4.3 поясняется, что интеллектуальный указатель `std::weak_ptr` является исключением.

## Замечания и предложения

Я сделал все возможное, чтобы книга содержала только ясную, точную, полезную информацию, но наверняка есть способы сделать ее еще лучшей. Если вы найдете в книге ошибки любого рода (технические, разъяснительные, грамматические, типографские и т.д.) или если у вас есть предложения о том, как можно улучшить книгу, пожалуйста, напишите мне по адресу `emc++@aristeia.com`. В новых изданиях книги ваши замечания и предложения обязательно будут учтены.

Список исправлений обнаруженных ошибок можно найти по адресу <http://www.aristeia.com/BookErrata/emc++-errata.html>.

## От редакции

Редакция выражает признательность профессору университета Иннополис Е. Зуеву за обсуждения и советы при работе над переводом данной книги.

# **Ждем ваших отзывов!**

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: info@williamspublishing.com

WWW: <http://www.williamspublishing.com>

Наши почтовые адреса:

в России: 127055, Москва, ул. Лесная, д. 43, стр. 1

в Украине: 03150, Киев, а/я 152



# Вывод типов

В C++98 имеется единственный набор правил вывода типов — для шаблонов функций. C++11 немного изменяет этот набор правил и добавляет два новых — для `auto` и для `decltype`. C++14 расширяет контексты использования ключевых слов `auto` и `decltype`. Все более широкое применение вывода типов освобождает вас от необходимости правильной записи очевидных или излишних типов. Он делает программы на C++ более легко адаптируемыми, поскольку изменение типа в одной точке исходного текста автоматически распространяется с помощью вывода типов на другие точки. Однако он может сделать код более сложным для восприятия, так как выводимые компилятором типы могут не быть настолько очевидными, как вам бы хотелось.

Без ясного понимания того, как работает вывод типов, эффективное программирование на современном C++ невозможно. Просто есть слишком много контекстов, в которых имеет место вывод типа: в вызовах шаблонов функций, в большинстве ситуаций, в которых встречается ключевое слово `auto`, в выражениях `decltype` и, начиная с C++14, там, где применяется загадочная конструкция `decltype(auto)`.

Эта глава содержит информацию о выводе типов, которая требуется каждому разработчику на языке программирования C++. Здесь поясняется, как работает вывод типа шаблона, как строится `auto` и как проходит свой путь `decltype`. Здесь даже объясняется, как заставить компилятор сделать видимыми результаты своего вывода типа, чтобы убедиться, что компилятор выводит именно тот тип, который вы хотели.

## 1.1. Вывод типа шаблона

Когда пользователи сложной системы не знают, как она работает, но их устраивает то, что она делает, это говорит об удачном проектировании системы. Если мерить такой мерой, то вывод типа шаблона в C++ является огромным успехом. Миллионы программистов передают аргументы шаблонным функциям с вполне удовлетворительными результатами несмотря на то, что многие из этих программистов не способны на большее, чем очень приближенное и расплывчатое описание того, как же были выведены эти типы.

Если вы относитесь к числу этих программистов, у меня для вас две новости — хорошая и плохая. Хорошая новость заключается в том, что вывод типов для шаблонов является основой для одной из наиболее привлекательных возможностей современного C++: `auto`. Если вас устраивало, как C++98 выводит типы для шаблонов, вас устроит и то, как

C++11 выводит типы для `auto`. Плохая новость заключается в том, что когда правила вывода типа шаблона применяются в контексте `auto`, они оказываются немного менее интуитивными, чем в приложении к шаблонам. По этой причине важно действительно понимать аспекты вывода типов шаблонов, на которых построен вывод типов для `auto`. Этот раздел содержит информацию, которую вы должны знать.

Если вы готовы посмотреть сквозь пальцы на применение небольшого количества псевдокода, то можно рассматривать шаблон функции как имеющий следующий вид:

```
template<typename T>
void f(ParamType param);
```

Вызов может выглядеть следующим образом:

```
f(expr); // Вызов f с некоторым выражением
```

В процессе компиляции компилятор использует `expr` для вывода двух типов: типа `T` и типа `ParamType`. Эти типы зачастую различны, поскольку `ParamType` часто содержит “украшения”, например `const` или квалификаторы ссылки. Например, если шаблон объявлен как

```
template<typename T>
void f(const T& param); // ParamType – const T&
```

и мы осуществляляем вызов

```
int x = 0;
f(x); // Вызов f с параметром int
```

то `T` выводится как `int`, а `ParamType` — как `const int&`.

Вполне естественно ожидать, что тип, выведенный для `T`, тот же, что и тип переданного функции аргумента, т.е. что `T` — это тип выражения `expr`. В приведенном выше примере это так: `x` — значение типа `int` и `T` выводится как `int`. Но вывод не всегда работает таким образом. Тип, выведенный для `T`, зависит не только от типа `expr`, но и от вида `ParamType`. Существует три случая.

- `ParamType` представляет собой указатель или ссылку, но не универсальную ссылку. (Универсальные ссылки рассматриваются в разделе 5.2. Пока что все, что вам надо знать, — что они существуют и не являются ни ссылками `lvalue`, ни ссылками `rvalue`.)
- `ParamType` является универсальной ссылкой.
- `ParamType` не является ни указателем, ни ссылкой.

Следовательно, нам надо рассмотреть три сценария вывода. Каждый из них основан на нашем общем виде шаблонов и их вызова:

```
template<typename T>
void f(ParamType param);
f(expr); // Вывод T и ParamType из expr
```

## Случай 1. *ParamType* является указателем или ссылкой, но не универсальной ссылкой

Простейшая ситуация — когда *ParamType* является ссылочным типом или типом указателя, но не универсальной ссылкой. В этом случае вывод типа работает следующим образом.

1. Если типом *expr* является ссылка, ссылочная часть игнорируется.
2. Затем выполняется сопоставление типа *expr* с *ParamType* для определения *T*.

Например, если у нас имеются шаблон

```
template<typename T>
void f(T& param); // param представляет собой ссылку
```

и объявления переменных

```
int x = 27;           // x имеет тип int
const int cx = x;    // cx имеет тип const int
const int& rx = x;  // rx является ссылкой на x как на const int
```

то выводимые типы для *param* и *T* в различных выводах будут следующими:

```
f(x); // T - int, тип param - int&
f(cx); // T - const int, тип param - const int&
f(rx); // T - const int, тип param - const int&
```

Во втором и третьем вызовах обратите внимание, что, поскольку *cx* и *rx* объявлены как константные значения, *T* выводится как *const int*, тем самым приводя к типу параметра *const int&*. Это важно для вызывающего кода. Передавая константный объект параметру-ссылке, он ожидает, что объект останется неизменным, т.е. что параметр будет представлять собой ссылку на *const*. Вот почему передача константного объекта в шаблон, получающий параметр *T&*, безопасна: константность объекта становится частью выведенного для *T* типа.

В третьем примере обратите внимание, что несмотря на то, что типом *rx* является ссылка, тип *T* выводится как не ссылочный. Вот почему при выводе типа игнорируется “ссылочность” *rx*.

Все эти примеры показывают ссылочные параметры, являющиеся *lvalue*, но вывод типа точно так же работает и для ссылочных параметров *rvalue*. Конечно, *rvalue*-аргументы могут передаваться только ссылочным параметрам, являющимся *rvalue*, но это ограничение никак не влияет на вывод типов.

Если мы изменим тип параметра *f* с *T&* на *const T&*, произойдут небольшие изменения, но ничего удивительного не случится. Константность *cx* и *rx* продолжает соблюдаться, но поскольку теперь мы считаем, что *param* является ссылкой на *const*, *const* как часть выводимого типа *T* не требуется:

```
template<typename T>
void f(const T& param); // param является ссылкой на const
```

```
int x = 27;           // Как и ранее
const int cx = x;    // Как и ранее
const int& rx = x;  // Как и ранее

f(x);                // T - int, тип param - const int&
f(cx);               // T - int, тип param - const int&
f(rx);               // T - int, тип param - const int&
```

Как и ранее, “ссылочность” rx при выводе типа игнорируется.

Если бы param был указателем (или указателем на const), а не ссылкой, все бы работало, по сути, точно так же:

```
template<typename T>
void f(T* param); // Теперь param является указателем

int x = 27;          // Как и ранее
const int *px = &x; // px - указатель на x, как на const int

f(&x);             // T - int,      тип param - int*
f(px);              // T - const int, тип param - const int*
```

Сейчас вы можете обнаружить, что давно усердно зеваете, потому что все это очень скучно, правила вывода типов в C++ работают так естественно для ссылок и указателей, что все просто очевидно! Это именно то, что вы хотите от системы вывода типов.

## Случай 2. ParamType является универсальной ссылкой

Все становится менее очевидным в случае шаблонов, принимающих параметры, являющиеся универсальными ссылками. Такие параметры объявляются как ссылки rvalue (т.е. в шаблоне функции, принимающем параметр типа T, объявленным типом универсальной ссылки является T&&), но ведут себя иначе при передаче аргументов, являющихся lvalue. Полностью вопрос рассматривается в разделе 5.2, здесь приводится его сокращенная версия.

- Если expr представляет собой lvalue, как T, так и ParamType выводятся как lvalue-ссылки. Это вдвое необычно. Во-первых, это единственная ситуация в выводе типа шаблона, когда T выводится как ссылка. Во-вторых, хотя ParamType объявлен с использованием синтаксиса rvalue-ссылки, его выводимым типом является lvalue-ссылка.
- Если expr представляет собой rvalue, применяются “обычные” правила (из случая 1).

Примеры

```
template<typename T>
void f(T&& param); // param является универсальной ссылкой

int x = 27;          // Как и ранее
const int cx = x;   // Как и ранее
```

```

const int& rx = x; // Как и ранее

f(x);           // x - lvalue, так что T - int&,
                // тип param также является int&
f(cx);          // cx - lvalue, так что T - const int&,
                // тип param также является const int&
f(rx);          // rx - lvalue, так что T - const int&,
                // тип param также является const int&
f(27);          // 27 - rvalue, так что T - int,
                // следовательно, тип param - int&&

```

В разделе 5.2 поясняется, почему эти примеры работают именно так, а не иначе. Ключевым моментом является то, что правила вывода типов для параметров, являющихся универсальными ссылками, отличаются от таковых для параметров, являющихся lvalue- или rvalue-ссылками. В частности, когда используются универсальные ссылки, вывод типов различает аргументы, являющиеся lvalue, и аргументы, являющиеся rvalue. Этого никогда не происходит для неуниверсальных ссылок.

### **Случай 3. ParamType не является ни указателем, ни ссылкой**

Когда ParamType не является ни указателем, ни ссылкой, мы имеем дело с передачей по значению:

```

template<typename T>
void f(T param);      // param передается по значению

```

Это означает, что param будет копией переданного функции — совершенно новым объектом. Тот факт, что param будет совершенно новым объектом, приводит к правилам, которые регулируют вывод T из expr.

1. Как и ранее, если типом expr является ссылка, ссылочная часть игнорируется.
2. Если после отбрасывания ссылочной части expr является const, это также игнорируется. Игнорируется и модификатор volatile (объекты volatile являются редкостью и в общем случае используются только при реализации драйверов устройств; детальную информацию на эту тему вы найдете в разделе 7.6.)

Таким образом, получаем следующее:

```

int x = 27;           // Как и ранее
const int cx = x;     // Как и ранее
const int& rx = x;    // Как и ранее
f(x);                 // Типами и T, и param являются int
f(cx);                // Типами и T, и param вновь являются int
f(rx);                // Типами и T, и param опять являются int

```

Обратите внимание, что даже несмотря на то, что cx и rx представляют константные значения, param не является const. Это имеет смысл. param представляет собой объект, который полностью независим от cx и rx, — это копия cx или rx. Тот факт, что cx и rx

не могут быть модифицированы, ничего не говорит о том, может ли быть модифицирован `param`. Вот почему константность `expr` (как и `volatile`, если таковой модификатор присутствует) игнорируется при выводе типа `param`: то, что `expr` не может быть модифицировано, не означает, что таковой должна быть и его копия.

Важно понимать, что `const` (и `volatile`) игнорируются только параметрами, передаваемыми по значению. Как мы уже видели, для параметров, которые являются ссылками или указателями на `const`, константность `expr` при выводе типа сохраняется. Но рассмотрим случай, когда `expr` представляет собой `const`-указатель на константный объект, а передача осуществляется по значению:

```
template<typename T>
void f(T param);           // param передается по значению

const char* const ptr =   // ptr – константный указатель на
    "Fun with pointers"; // константный объект

f(ptr);                  // Передача arg типа const char* const
```

Здесь `const` справа от звездочки объявляет `ptr` константным: `ptr` не может ни указывать на другое место в памяти, ни быть обнуленным. (`const` слева от звездочки гласит, что `ptr` указывает на то, что (строка символов) является `const`, а следовательно, не может быть изменено.) Когда `ptr` передается в функцию `f`, биты, составляющие указатель, копируются в `param`. Как таковой *сам* указатель (`ptr`) будет передан по значению. В соответствии с правилом вывода типа при передаче параметров по значению константность `ptr` будет проигнорирована, а выведенным для `param` типом будет `const char*`, т.е. изменяемый указатель на константную строку символов. Константность того, на что указывает `ptr`, в процессе вывода типа сохраняется, но константность самого `ptr` игнорируется при создании нового указателя `param`.

## Аргументы-массивы

Мы рассмотрели большую часть материала, посвященного выводу типов шаблонов, но есть еще один угол, в который стоит заглянуть. Это отличие типов массивов от типов указателей, несмотря на то что зачастую они выглядят взаимозаменяемыми. Основной вклад в эту иллюзию вносит то, что во множестве контекстов массив преобразуется в указатель на его первый элемент. Это преобразование позволяет компилироваться коду наподобие следующего:

```
const char name[] = "Briggs"; // Тип name – const char[13]
const char * ptrToName = name; // Массив становится указателем
```

Здесь указатель `ptrToName` типа `const char*` инициализируется переменной `name`, которая имеет тип `const char[13]`. Эти типы (`const char*` и `const char[13]`) не являются одним и тем же типом, но благодаря правилу преобразования массива в указатель приведенный выше код компилируется.

Но что будет, если передать массив шаблону, принимающему параметр по значению?

```
template<typename T>
void f(T param); // Шаблон, получающий параметр по значению

f(name); // Какой тип T и param будет выведен?
```

Начнем с наблюдения, что не существует такой вещи, как параметр функции, являющийся массивом. Да, да — приведенный далее синтаксис корректен:

```
void myFunc(int param[]);
```

Однако объявление массива рассматривается как объявление указателя, а это означает, что функция myFunc может быть эквивалентно объявлена как

```
void myFunc(int* param); // Та же функция, что и ранее
```

Эта эквивалентность параметров, представляющих собой массив и указатель, образно говоря, представляет собой немного листвы от корней C на дереве C++ и способствует возникновению иллюзии, что типы массивов и указателей представляют собой одно и то же.

Поскольку объявление параметра-массива рассматривается так, как если бы это было объявление параметра-указателя, тип массива, передаваемого в шаблонную функцию по значению, выводится как тип указателя. Это означает, что в вызове шаблонной функции f ее параметр типа T выводится как const char\*:

```
f(name); // name — массив, но T — const char*
```

А вот теперь начинаются настоящие хитрости. Хотя функции не могут объявлять параметры как истинные массивы, они могут объявлять параметры, являющиеся ссылками на массивы! Так что если мы изменим шаблон f так, чтобы он получал свой аргумент по ссылке,

```
template<typename T>
void f(T& param); // Шаблон с передачей параметра по ссылке
```

и передадим ему массив

```
f(name); // Передача массива функции f
```

то тип, выведенный для T, будет в действительности типом массива! Этот тип включает размер массива, так что в нашем примере T выводится как const char[13], а типом параметра f (ссылки на этот массив) является const char (&)[13]. Да, выглядит этот синтаксис как наркотический бред, но знание его прибавит вам веса в глазах понимающих людей.

Интересно, что возможность объявлять ссылки на массивы позволяет создать шаблон, который выводит количество элементов, содержащихся в массиве:

```
// Возвращает размер массива как константу времени компиляции.
// Параметр не имеет имени, поскольку, кроме количества
// содержащихся в нем элементов, нас ничто не интересует.
template<typename T, std::size_t N>
constexpr std::size_t arraySize(T (&)[N]) noexcept
```

```
{  
    return N;  
}
```

Как поясняется в разделе 3.9, объявление этой функции как `constexpr` делает ее результат доступным во время компиляции. Это позволяет объявить, например, массив с таким же количеством элементов, как и у второго массива, размер которого вычисляется из инициализатора в фигурных скобках:

```
// keyVals содержит 7 элементов:  
int keyVals[] = { 1, 3, 7, 9, 11, 22, 35 };  
  
int mappedVals[arraySize(keyVals)]; // mappedVals – тоже
```

Конечно, как разработчик на современном C++ вы, естественно, предпочтете `std::array` встроенному массиву:

```
// Размер mappedVals равен 7  
std::array<int, arraySize(keyVals)> mappedVals;
```

Что касается объявления `arraySize` как `constexpr`, то это помогает компилятору генерировать лучший код. Детальнее этот вопрос рассматривается в разделе 3.8.

## Аргументы-функции

Массивы — не единственные сущности в C++, которые могут превращаться в указатели. Типы функций могут превращаться в указатели на функции, и все, что мы говорили о выводе типов для массивов, применимо к выводу типов для функций и их преобразованию в указатели на функции. В результате получаем следующее:

```
void someFunc(int, double); // someFunc – функция;  
                           // ее тип – void(int, double)  
  
template<typename T>  
void f1(T param);          // В f1 param передается по значению  
  
template<typename T>  
void f2(T& param);         // В f2 param передается по ссылке  
  
f1(someFunc);              // param выводится как указатель на  
                           // функцию; тип – void(*)(int,double)  
  
f2(someFunc);              // param выводится как ссылка на  
                           // функцию; тип – void(&)(int,double)
```

Это редко приводит к каким-то различиям на практике, но если вы знаете о преобразовании массивов в указатели, то разберетесь и в преобразовании функций в указатели.

Итак, у нас есть правила для вывода типов шаблонов, связанные с `auto`. В начале я заметил, что они достаточно просты, и по большей части так оно и есть. Немного усложняет жизнь отдельное рассмотрение согласованных `lvalue` при выводе типов

для универсальных ссылок, да еще несколько “мутят воду” правила преобразования в указатели для массивов и функций. Иногда так и хочется, разозлившись, схватить компилятор и вытрясти из него — “А скажи-ка, любезный, какой же тип ты выводишь?” Когда это произойдет, обратитесь к разделу 1.4, поскольку он посвящен тому, как уговорить компилятор это сделать.

### Следует запомнить

- В процессе вывода типа шаблона аргументы, являющиеся ссылками, рассматриваются как ссылками не являющиеся, т.е. их “ссылочность” игнорируется.
- При выводе типов для параметров, являющихся универсальными ссылками, lvalue-аргументы рассматриваются специальным образом.
- При выводе типов для параметров, передаваемых по значению, аргументы, объявленные как `const` и/или `volatile`, рассматриваются как не являющиеся ни `const`, ни `volatile`.
- В процессе вывода типа шаблона аргументы, являющиеся именами массивов или функций, преобразуются в указатели, если только они не использованы для инициализации ссылок.

## 1.2. Вывод типа `auto`

Если вы прочли раздел 1.1 о выводе типов шаблонов, вы знаете почти все, что следует знать о выводе типа `auto`, поскольку за одним любопытным исключением вывод типа `auto` представляет собой вывод типа шаблона. Но как это может быть? Вывод типа шаблона работает с шаблонами, функциями и параметрами, а `auto` не имеет дела ни с одной из этих сущностей.

Да, это так, но это не имеет значения. Существует прямая взаимосвязь между выводом типа шаблона и выводом типа `auto`. Существует буквальное алгоритмическое преобразование одного в другой.

В разделе 1.1 вывод типа шаблона пояснялся с использованием обобщенного шаблона функции

```
template<typename T>
void f(ParamType param);
```

и обобщенного вызова

```
f(expr); // Вызов f с некоторым выражением
```

При вызове `f` компиляторы используют `expr` для вывода типов `T` и `ParamType`.

Когда переменная объявлена с использованием ключевого слова `auto`, оно играет роль `T` в шаблоне, а спецификатор типа переменной действует как `ParamType`. Это проще показать, чем описать, так что рассмотрим следующий пример:

```
auto x = 27;
```

Здесь спецификатором типа для `x` является `auto` само по себе. С другой стороны, в объявлении

```
const auto cx = x;
```

спецификатором типа является `const auto`. А в объявлении

```
const auto& rx = x;
```

спецификатором типа является `const auto&`. Для вывода типов для `x`, `cx` и `rx` в приведенных примерах компилятор действует так, как если бы для каждого объявления имелся шаблон, а также вызов этого шаблона с соответствующим инициализирующим выражением:

```
template<typename T> // Концептуальный шаблон для
void func_for_x(T param); // вывода типа x

func_for_x(27); // Концептуальный вызов: выве-
// денный тип param является
// типом x

template<typename T> // Концептуальный шаблон для
void func_for_cx(const T param); // вывода типа cx

func_for_cx(x); // Концептуальный вызов: выве-
// денный тип param является
// типом cx

template<typename T> // Концептуальный шаблон для
void func_for_rx(const T& param); // вывода типа rx

func_for_rx(x); // Концептуальный вызов: выве-
// денный тип param является
// типом rx
```

Как я уже говорил, вывод типов для `auto` представляет собой (с одним исключением, которое мы вскоре рассмотрим) то же самое, что и вывод типов для шаблонов.

В разделе 1.1 вывод типов шаблонов был разделен на три случая, основанных на характеристиках `ParamType`, спецификаторе типа `param` в обобщенном шаблоне функции. В объявлении переменной с использованием `auto` спецификатор типа занимает место `ParamType`, так что у нас опять имеются три случая.

- Случай 1. Спецификатор типа представляет собой ссылку или указатель, но не универсальную ссылку.
- Случай 2. Спецификатор типа представляет собой универсальную ссылку.
- Случай 3. Спецификатор типа не является ни ссылкой, ни указателем.

Мы уже встречались со случаями 1 и 3:

```
auto      x = 27; // Случай 3 (x не указатель и не ссылка)
const auto cx = x; // Случай 3 (cx не указатель и не ссылка)
const auto& rx = x; // Случай 1 (rx – неуниверсальная ссылка)
```

Случай 2 работает, как и ожидалось:

```
auto&& uref1 = x; // x – int и lvalue, так что тип uref1 – int&
auto&& uref2 = cx; // cx – const int и lvalue, так что тип
                  // uref2 – const int&
auto&& uref3 = 27; // 27 – int и rvalue, так что тип
                  // uref3 – int&&
```

Раздел 1.1 завершился обсуждением того, как имена массивов и функций превращаются в указатели для спецификаторов типа, не являющихся ссылками. То же самое происходит и при выводе типа `auto`:

```
const char name[] =           // Тип name – const char[13]
    "R. N. Briggs";
auto arr1 = name;            // Тип arr1 – const char*
auto& arr2 = name;           // Тип arr2 – const char (&) [13]

void someFunc(int, double); // someFunc – функция, ее тип
                           // void(int, double)
auto func1 = someFunc;       // Тип func1 – void (*)(int, double)
auto& func2 = someFunc;       // Тип func2 – void (&)(int, double)
```

Как можно видеть, вывод типа `auto` работает подобно выводу типа шаблона. По сути это две стороны одной медали.

Они отличаются только в одном. Начнем с наблюдения, что если вы хотите объявить `int` с начальным значением 27, C++98 предоставляет вам две синтаксические возможности:

```
int x1 = 27;
int x2(27);
```

C++11, поддерживая старые варианты инициализации, добавляет собственные:

```
int x3 = { 27 };
int x4{ 27 };
```

Таким образом, у нас есть четыре разных синтаксиса, но результат один: переменная типа `int` со значением 27.

Но, как поясняется в разделе 2.1, объявление переменных с использованием ключевого слова `auto` вместо фиксированных типов обладает определенными преимуществами, поэтому в приведенных выше объявлениях имеет смысл заменить `int` на `auto`. Простая замена текста приводит к следующему коду:

```
auto x1 = 27;
auto x2(27);
auto x3 = { 27 };
auto x4{ 27 };
```

Все эти объявления компилируются, но их смысл оказывается не тем же, что и у объявлений, которые они заменяют. Первые две инструкции в действительности объявляют переменную типа `int` со значением 27. Вторые две, однако, определяют переменную типа `std::initializer_list<int>`, содержащую единственный элемент со значением 27!

```
auto x1 = 27;      // Тип int, значение – 27
auto x2(27);      // То же самое
auto x3 = { 27 }; // std::initializer_list<int>, значение {27}
auto x4{ 27 };    // То же самое
```

Это объясняется специальным правилом вывода типа для `auto`. Когда инициализатор для переменной, объявленной как `auto`, заключен в фигурные скобки, выведенный тип — `std::initializer_list`. Если такой тип не может быть выведен (например, из-за того, что значения в фигурных скобках относятся к разным типам), код будет отвергнут:

```
auto x5 = { 1, 2, 3.0 }; // Ошибка! Невозможно вывести Т
                        // для std::initializer_list<T>
```

Как указано в комментарии, в этом случае вывод типа будет неудачным, но важно понимать, что на самом деле здесь имеют место два вывода типа. Один из них вытекает из применения ключевого слова `auto`: тип `x5` должен быть выведен. Поскольку инициализатор `x5` находится в фигурных скобках, тип `x5` должен быть выведен как `std::initializer_list`. Но `std::initializer_list` — это шаблон. Конкретизация представляет собой создание `std::initializer_list<T>` с некоторым типом `T`, а это означает, что тип `T` также должен быть выведен. Такой вывод относится ко второй разновидности вывода типов — выводу типа шаблона. В данном примере этот второй вывод неудачен, поскольку значения в фигурных скобках не относятся к одному и тому же типу.

Рассмотрение инициализаторов в фигурных скобках является единственным отличием вывода типа `auto` от вывода типа шаблона. Когда объявленная с использованием ключевого слова `auto` переменная инициализируется с помощью инициализатора в фигурных скобках, выведенный тип представляет собой конкретизацию `std::initializer_list`. Но если тот же инициализатор передается шаблону, вывод типа оказывается неудачным, и код отвергается:

```
auto x = { 11, 23, 9 }; // Тип x – std::initializer_list<int>

template<typename T>      // Объявление шаблона с параметром
void f(T param);          // эквивалентно объявлению x

f({ 11, 23, 9 });        // Ошибка вывода типа для T
```

Однако, если вы укажете в шаблоне, что `param` представляет собой `std::initializer_list<T>` для некоторого неизвестного `T`, вывод типа шаблона сможет определить, чем является `T`:

```
template<typename T>
void f(std::initializer_list<T> initList);
```

```
f({ 11, 23, 9 }); // Вывод int в качестве типа T, а тип
                  // initList - std::initializer_list<int>
```

Таким образом, единственное реальное различие между выводом типа `auto` и выводом типа шаблона заключается в том, что `auto` *предполагает*, что инициализатор в фигурных скобках представляет собой `std::initializer_list`, в то время как вывод типа шаблона этого не делает.

Вы можете удивиться, почему вывод типа `auto` имеет специальное правило для инициализаторов в фигурных скобках, в то время как вывод типа шаблона такого правила не имеет. Но я и сам удивлен. Увы, я не в состоянии найти убедительное объяснение. Но “закон есть закон”, и это означает, что вы должны помнить, что если вы объявляете переменную с использованием ключевого слова `auto` и инициализируете ее с помощью инициализатора в фигурных скобках, то выводимым типом всегда будет `std::initializer_list`. Особенно важно иметь это в виду, если вы приверженец философии унифицированной инициализации — заключения инициализирующих значений в фигурные скобки как само собой разумеющегося стиля. Классической ошибкой в C++11 является случайное объявление переменной `std::initializer_list` там, где вы намеревались объявить нечто иное. Эта ловушка является одной из причин, по которым некоторые разработчики используют фигурные скобки в инициализаторах только тогда, когда обязаны это делать. (Когда именно вы обязаны так поступать, мы рассмотрим в разделе 3.1.)

Что касается C++11, то на этом история заканчивается, но для C++14 это еще не конец. C++14 допускает применение `auto` для указания того, что возвращаемый тип функции должен быть выведен (см. раздел 1.3), а кроме того, лямбда-выражения C++14 могут использовать `auto` в объявлениях параметров. Однако такое применение `auto` использует вывод типа шаблона, а не вывод типа `auto`. Таким образом, функция с возвращаемым типом `auto`, которая возвращает инициализатор в фигурных скобках, компилироваться не будет:

```
auto createInitList()
{
    return { 1, 2, 3 }; // Ошибка: невозможно вывести
                      // тип для { 1, 2, 3 }
```

То же самое справедливо и тогда, когда `auto` используется в спецификации типа параметра в лямбда-выражении C++14:

```
std::vector<int> v;

auto resetV =
    [&v] (const auto& newValue) { v = newValue; }; // C++14

resetV({ 1, 2, 3 }); // Ошибка: невозможно вывести
                      // тип для { 1, 2, 3 }
```

### Следует запомнить

- Вывод типа `auto` обычно такой же, как и вывод типа шаблона, но вывод типа `auto`, в отличие от вывода типа шаблона, предполагает, что инициализатор в фигурных скобках представляет `std::initializer_list`.
- `auto` в возвращаемом типе функции или параметре лямбда-выражения влечет применение вывода типа шаблона, а не вывода типа `auto`.

## 1.3. Знакомство с `decltype`

`decltype` — создание странное. Для данного имени или выражения `decltype` сообщает вам тип этого имени или выражения. Обычно то, что сообщает `decltype`, — это именно то, что вы предсказываете. Однако иногда он дает результаты, которые заставляют вас чесать в затылке и обращаться к справочникам или сайтам.

Мы начнем с типичных случаев, в которых нет никаких подводных камней. В отличие от того, что происходит в процессе вывода типов для шаблонов и `auto` (см. разделы 1.1 и 1.2), `decltype` обычно попугайничает, возвращая точный тип имени или выражения, которое вы передаете ему:

```
const int i = 0;           // decltype(i) – const int

bool f(const Widget& w); // decltype(w) – const Widget&
                        // decltype(f) – bool(const Widget&)

struct Point {
    int x, y;           // decltype(Point::x) – int
};                      // decltype(Point::y) – int

Widget w;               // decltype(w) – Widget

if (f(w)) ...           // decltype(f(w)) – bool

template<typename T>   // Упрощенная версия std::vector
class vector {
public:
    ...
    T& operator[](std::size_t index);

};

vector<int> v;          // decltype(v) – vector<int>

...
if (v[0] == 0) ...        // decltype(v[0]) – int&
```

Видите? Никаких сюрпризов.

Пожалуй, основное применение decltype в C++11 — объявление шаблонов функций, в которых возвращаемый тип функции зависит от типов ее параметров. Предположим, например, что мы хотим написать функцию, получающую контейнер, который поддерживает индексацию с помощью квадратных скобок (т.е. с использованием “[ ]”) с индексом, а затем аутентифицирует пользователя перед тем как вернуть результат операции индексации. Возвращаемый тип функции должен быть тем же, что и тип, возвращаемый операцией индексации.

operator[] для контейнера объектов типа T обычно возвращает T&. Например, это так в случае std::deque и почти всегда — в случае std::vector. Однако для std::vector<bool> оператор operator[] не возвращает bool&. Вместо этого он возвращает новый объект. Все “почему” и “как” данной ситуации рассматриваются в разделе 2.2, но главное здесь то, что возвращаемый оператором operator[] контейнера тип зависит от самого контейнера.

decltype упрощает выражение этой зависимости. Вот пример, показывающий применение decltype для вычисления возвращаемого типа. Этот шаблон требует уточнения, но пока что мы его отложим.

```
template<typename Container, typename Index> // Работает, но
auto authAndAccess(Container& c, Index i)      // требует
    -> decltype(c[i])                         // уточнения
{
    authenticateUser();
    return c[i];
}
```

Использование auto перед именем функции не имеет ничего общего с выводом типа. На самом деле оно указывает, что использован синтаксис C++11 — завершающий возвращаемый тип (trailing return type), т.е. что возвращаемый тип функции будет объявлен после списка параметров (после “->”). Завершающий возвращаемый тип обладает тем преимуществом, что в спецификации возвращаемого типа могут использоваться параметры функции. В authAndAccess, например, мы указываем возвращаемый тип с использованием c и i. Если бы возвращаемый тип, как обычно, предшествовал имени функции, c и i были бы в нем недоступны, поскольку в этот момент они еще не были объявлены.

При таком объявлении authAndAccess возвращает тот тип, который возвращает operator[] при применении к переданному контейнеру, в точности как мы и хотели.

C++11 разрешает вывод возвращаемых типов лямбда-выражений из одной инструкции, а C++14 расширяет эту возможность на все лямбда-выражения и все функции, включая состоящие из множества инструкций. В случае authAndAccess это означает, что в C++14 мы можем опустить завершающий возвращаемый тип, оставляя только одно ведущее ключевое слово auto. При таком объявлении auto означает, что имеет место вывод типа. В частности, это означает, что компиляторы будут выводить возвращаемый тип функции из ее реализации:

```
template<typename Container, typename Index> // C++14;
auto authAndAccess(Container& c, Index i)      // Не совсем
```

```
{ // корректно
    authenticateUser();
    return c[i]; // Возвращаемый тип выводится из c[i]
}
```

В разделе 1.2 поясняется, что для функций с auto-спецификацией возвращаемого типа компиляторы применяют вывод типа шаблона. В данном случае это оказывается проблематичным. Как уже говорилось, operator[] для большинства контейнеров с объектами типа T возвращает T&, но в разделе 1.1 поясняется, что в процессе вывода типа шаблона “ссылочность” инициализирующего выражения игнорируется. Рассмотрим, что это означает для следующего клиентского кода:

```
std::deque<int> d;
...
authAndAccess(d, 5) = 10; // Аутентифицирует пользователя, воз-
                           // вращает d[5], затем присваивает ему
                           // значение 10. Код не компилируется!
```

Здесь d[5] возвращает int&, но вывод возвращаемого типа auto для authAndAccess отбрасывает ссылку, тем самым давая возвращаемый тип int. Этот int, будучи возвращаемым значением функции, является rvalue, так что приведенный выше код пытается присвоить этому rvalue типа int значение 10. Это запрещено в C++, так что данный код не компилируется.

Чтобы заставить authAndAccess работать так, как мы хотим, нам надо использовать для ее возвращаемого типа вывод типа decltype, т.е. указать, что authAndAccess должна возвращать в точности тот же тип, что и выражение c[i]. Защитники C++, предвидя необходимость использования в некоторых случаях правил вывода типа decltype, сделали это возможным в C++14 с помощью спецификатора decltype(auto). То, что изначально может показаться противоречием (decltype и auto?), в действительности имеет смысл: auto указывает, что тип должен быть выведен, а decltype говорит о том, что в процессе вывода следует использовать правила decltype. Итак, можно записать authAndAccess следующим образом:

```
template<typename Container, typename Index> // C++14; работает,
    decltype(auto) // но все еще
    authAndAccess(Container& c, Index i) // требует
    { // уточнения
        authenticateUser();
        return c[i];
    }
```

Теперь authAndAccess действительно возвращает то же, что и c[i]. В частности, в распространенном случае, когда c[i] возвращает T&, authAndAccess также возвращает T&, и в том редком случае, когда c[i] возвращает объект, authAndAccess также возвращает объект.

Использование decltype(auto) не ограничивается возвращаемыми типами функций. Это также может быть удобно для объявления переменных, когда вы хотите применять правила вывода типа decltype к инициализирующему выражению:

```

Widget w;
const Widget& cw = w;

auto myWidget1 = cw;           // Вывод типа auto:
                                // тип myWidget1 - Widget

decltype(auto) myWidget2 = cw; // Вывод типа decltype:
                                // тип myWidget2 - const Widget&

```

Я знаю, что вас беспокоят два момента. Один из них — упомянутое выше, но пока не описанное уточнение authAndAccess. Давайте, наконец-то, разберемся в этом вопросе.

Еще раз посмотрим на версию authAndAccess в C++14:

```

template<typename Container, typename Index>
decltype(auto) authAndAccess(Container& c, Index i);

```

Контейнер передается как lvalue-ссылка на неконстантный объект, поскольку возвращаемая ссылка на элемент контейнера позволяет клиенту модифицировать этот контейнер. Но это означает, что этой функции невозможно передавать контейнеры, являющиеся rvalue. rvalue невозможно связать с lvalue-ссылками (если только они не являются lvalue-ссылками на константные объекты, что в данном случае очевидным образом не выполняется).

Надо сказать, что передача контейнера, являющегося rvalue, в authAndAccess является крайним случаем. Такой rvalue-контейнер, будучи временным объектом, обычно уничтожается в конце инструкции, содержащей вызов authAndAccess, а это означает, что ссылка на элемент в таком контейнере (то, что должна вернуть функция authAndAccess) окажется “висячей” в конце создавшей ее инструкции. Тем не менее передача временного объекта функции authAndAccess может иметь смысл. Например, клиент может просто хотеть сделать копию элемента во временном контейнере:

```

std::deque<std::string> makeStringDeque(); // Фабричная функция

// Делаем копию пятого элемента deque, возвращаемого
// функцией makeStringDeque
auto s = authAndAccess(makeStringDeque(), 5);

```

Поддержка такого использования означает, что мы должны пересмотреть объявление функции authAndAccess, которая должна принимать как lvalue, так и rvalue. Можно использовать перегрузку (одна функция объявлена с параметром, представляющим собой lvalue-ссылку, а вторая — с параметром, представляющим собой rvalue-ссылку), но тогда нам придется поддерживать две функции. Избежать этого можно, если у нас будет функция authAndAccess, использующая ссылочный параметр, который может быть связан как с lvalue, так и с rvalue, и в разделе 5.2 поясняется, что это именно то, что делают универсальные ссылки. Таким образом, authAndAccess может быть объявлена следующим образом:

```

template<typename Container, typename Index> // Теперь с -
decltype(auto) authAndAccess(Container&& c, // универсальная
                           Index i);          // ссылка

```

В этом шаблоне мы не знаем, с каким типом контейнера работаем, и точно так же не знаем тип используемых им индексных объектов. Использование передачи по значению для объектов неизвестного типа обычно сопровождается риском снижения производительности из-за ненужного копирования, проблемами со срезкой объектов (см. раздел 8.1) и насмешками коллег. Но в случае индексов контейнеров, следя примеру стандартной библиотеки для значений индексов (например, в `operator[]` для `std::string`, `std::vector` и `std::deque`) это решение представляется разумным, так что мы будем придерживаться для них передачи по значению.

Однако нам нужно обновить реализацию шаблона для приведения его в соответствие с предостережениями из раздела 5.3 о применении `std::forward` к универсальным ссылкам:

```
template<typename Container, typename Index> // Окончательная
    decltype(auto)                                // версия для
        authAndAccess(Container&& c, Index i)      // C++14
{
    authenticateUser();
    return std::forward<Container>(c)[i];
}
```

Этот код должен делать все, что мы хотели, но он требует компилятора C++14. Если у вас нет такого, вам следует использовать версию шаблона для C++11. Она такая же, как и ее аналог C++14, за исключением того, что вы должны самостоятельно указатьозвращаемый тип:

```
template<typename Container, typename Index> // Окончательная
auto                                         // версия для
authAndAccess(Container&& c, Index i)       // C++11
-> decltype(std::forward<Container>(c)[i])
{
    authenticateUser();
    return std::forward<Container>(c)[i];
}
```

Вторым беспокоящим моментом является мое замечание в начале этого раздела о том, что `decltype` *почти* всегда дает тип, который вы ожидаете, т.е. что он *редко* преподносит сюрпризы. По правде говоря, вряд ли вы столкнетесь с этими исключениями из правила, если только вы не занимаетесь круглосуточно написанием библиотек.

Чтобы *полностью* понимать поведение `decltype`, вы должны познакомиться с некоторыми особыми случаями. Большинство из них слишком невразумительны, чтобы быть размещенными в этой книге, но один из них приводит к лучшему пониманию `decltype` и его применения.

Применение `decltype` к имени дает объявленный тип для этого имени. Имена представляют собой *lvalue*-выражения, но это не влияет на поведение `decltype`. Однако для *lvalue*-выражений, более сложных, чем имена, `decltype` гарантирует, что возвращаемый тип всегда будет *lvalue*-ссылкой. Иначе говоря, если *lvalue*-выражение, отличное от имени, имеет тип `T`, то `decltype` сообщает об этом типе как об `T&`. Это редко на что-то

влияет, поскольку тип большинства lvalue-выражений в обязательном порядке включает квалификатор lvalue-ссылки. Например, функции, возвращающие lvalue, всегда возвращают lvalue-ссылки.

Однако у этого поведения есть следствия, о которых необходимо знать. В коде

```
int x = 0;
```

x является именем переменной, так что decltype(x) представляет собой int. Однако “заворачивание” имени x в скобки — “(x)” — дает выражение, более сложное, чем имя. Будучи именем, x представляет собой lvalue, и C++ также определяет выражение (x) как lvalue. Следовательно, decltype((x)) представляет собой int&. Добавление скобок вокруг имени может изменить тип, возвращаемый для него decltype!

В C++11 это просто любопытный факт, но в сочетании с поддержкой в C++14 decltype(auto) это означает, что, казалось бы, тривиальные изменения в способе записи инструкции return могут повлиять на выводимый тип функции:

```
decltype(auto) f1()
{
    int x = 0;
    ...
    return x;    // decltype(x) представляет собой int,
}                // так что f1 возвращает int

decltype(auto) f2()
{
    int x = 0;
    ...
    return (x); // decltype((x)) представляет собой int&,
}                // так что f2 возвращает int&
```

Обратите внимание, что f2 не только имеет возвращаемый тип, отличный от f1, но и возвращает ссылку на локальную переменную! Этот код ведет вас к неопределенному поведению, что вряд ли является вашей целью.

Основной урок состоит в том, чтобы при использовании decltype(auto) уделять деталям самое пристальное внимание. Кажущиеся совершенно незначительными детали в выражении, для которого выводится тип, могут существенно повлиять на тип, возвращаемый decltype(auto). Чтобы гарантировать, что выводимый тип — именно тот, который вы ожидаете, используйте методы, описанные в разделе 1.4.

В то же время не забывайте и о более широкой перспективе. Конечно, decltype (как автономный, так и в сочетании с auto) при выводе типов иногда может привести к сюрпризам, но это не нормальная ситуация. Как правило, decltype возвращает тот тип, который вы ожидаете. Это особенно верно, когда decltype применяется к именам, потому что в этом случае decltype делает именно то, что скрывается в его названии: сообщает объявленный тип (*declared type*) имени.

### Следует запомнить

- decltype почти всегда дает тип переменной или выражения без каких-либо изменений.
- Для lvalue-выражений типа T, отличных от имени, decltype всегда дает тип T&.
- C++14 поддерживает конструкцию decltype(auto), которая, подобно auto, выводит тип из его инициализатора, но выполняет вывод типа с использованием правил decltype.

## 1.4. Как просмотреть выведенные типы

Выбор инструментов для просмотра результатов вывода типа зависит от фазы процесса разработки программного обеспечения, на которой вы хотите получить эту информацию. Мы рассмотрим три возможности: получение информации о выводе типа при редактировании кода, во время компиляции и во время выполнения.

### Редакторы IDE

Редакторы исходных текстов в IDE часто показывают типы программных сущностей (например, переменных, параметров, функций и т.п.), когда вы, например, помещаете указатель мыши над ними. Например, пусть у вас есть код

```
const int theAnswer = 42;
auto x = theAnswer;
auto y = &theAnswer;
```

Редактор, скорее всего, покажет, что выведенный тип x представляет собой `int`, а выведенный тип y — `const int*`.

Чтобы это сработало, ваш код должен быть в более-менее компилируемом состоянии, поскольку такого рода информация поставляется среди разработки компилятором C++ (или как минимум его клиентской частью), работающим в IDE. Если компилятор не в состоянии получить достаточно информации о вашем коде, чтобы выполнить вывод типа, вы не сможете увидеть выведенные типы.

Для простых типов наподобие `int` информация из IDE в общем случае вполне точна. Однако, как вы вскоре увидите, когда приходится иметь дело с более сложными типами, информация, выводимая IDE, может оказаться не особенно полезной.

### Диагностика компилятора

Эффективный способ заставить компилятор показать выведенный тип — использовать данный тип так, чтобы это привело к проблемам компиляции. Сообщение об ошибке практически обязательно будет содержать тип, который к ней привел.

Предположим, например, что мы хотели бы узнать типы, выведенные для x и y из предыдущего примера. Сначала мы объявляем шаблон класса, но не определяем его. Чего-то такого вполне хватит:

```
template<typename T> // Только объявление TD;  
class TD;
```

Попытки инстанцировать этот шаблон приведут к сообщению об ошибке, поскольку инстанцируемый шаблон отсутствует. Чтобы увидеть типы *x* и *y*, просто попробуйте инстанцировать TD с их типами:

```
TD<decltype(x)> xType; // Сообщение об ошибке будет  
TD<decltype(y)> yType; // содержать типы x и y
```

Я использую имена переменных вида *variableNameType*, чтобы проще найти интересующую меня информацию в сообщении об ошибке. Мой компилятор для приведенного выше кода сообщает, в частности, следующее (я выделил интересующую меня информацию о типах):

```
error: aggregate 'TD<int> xType' has incomplete type and  
cannot be defined  
error: aggregate 'TD<const int *> yType' has incomplete type  
and cannot be defined
```

Другой компилятор выдает ту же информацию, но в несколько ином виде:

```
error: 'xType' uses undefined class 'TD<int>'  
error: 'yType' uses undefined class 'TD<const int *>'
```

Если не учитывать разницу в оформлении, все протестированные мною компиляторы при использовании этого метода генерировали сообщения об ошибках с интересующей меня информацией о типах.

## Вывод времени выполнения

Подход с использованием функции вывода для отображения сведений о типе может быть использован только во время выполнения программы, зато он предоставляет полный контроль над форматированием вывода. Вопрос в том, чтобы создать подходящее для вывода текстовое представление информации. “Без проблем, — скажете вы. — Нам на помощь придут typeid и std::type\_info::name”. В наших поисках информации о выведенных для *x* и *y* типах можно написать следующий код:

```
std::cout << typeid(x).name() << '\n'; // Выведенные типы  
std::cout << typeid(y).name() << '\n'; // для x и y
```

Этот подход основан на том факте, что вызов typeid для такого объекта, как *x* или *y*, дает объект std::type\_info, а он имеет функцию-член name, которая дает С-строку (т.е. const char\*), представляющую имя типа.

Не гарантируется, что вызов std::type\_info::name вернет что-то разумное, но его реализации изо всех сил пытаются быть полезными. Уровень этой полезности варьируется от компилятора к компилятору. Компиляторы GNU и Clang, например, сообщают, что тип *x* — это “i”, а тип *y* — “PKi”. Эти результаты имеют смысл, если вы будете знать, что “i” у данных компиляторов означает “int”, а “PK” — “указатель на константу”.

(Оба компилятора поддерживают инструмент `c++filt`, который расшифровывает эти имена.) Компилятор Microsoft генерирует менее зашифрованный вывод: “`int`” для `x` и “`int const*`” для `y`.

Поскольку это корректные результаты для типов `x` и `y`, вы можете подумать, что задача получения информации о типах решена, но не делайте скоропалительных выводов. Рассмотрим более сложный пример:

```
template<typename T> // Шаблонная функция,  
void f(const T& param); // вызываемая далее  
  
std::vector<Widget> createVec(); // Фабричная функция  
  
const auto vw = createVec(); // Инициализация vw возвратом  
// фабричной функции  
  
if (!vw.empty()) {  
    f(&vw[0]); // Вызов f  
  
}
```

Этот код, включающий пользовательский тип (`Widget`), контейнер STL (`std::vector`) и переменную `auto` (`vw`), является более представительным и интересным примером. Было бы неплохо узнать, какие типы выводятся для параметра типа шаблона `T` и для параметра `param` функции `f`.

Воспользоваться `typeid` в этой задаче достаточно просто. Надо всего лишь добавить немного кода в функцию `f` для вывода интересующих нас типов:

```
template<typename T>  
void f(const T& param)  
{  
    using std::cout;  
  
    // Вывод в поток cout типа T:  
    cout << "T = " << typeid(T).name() << '\n';  
  
    // Вывод в поток cout типа param:  
    cout << "param = " << typeid(param).name() << '\n';  
  
}
```

Выполнимые файлы, полученные с помощью компиляторов GNU и Clang, дают следующий результат:

```
T      = PK6Widget  
param = PK6Widget
```

Мы уже знаем, что в этих компиляторах `PK` означает указатель на константу, так что вся загадка — в цифре 6. Это просто количество символов в следующем за ней имени класса

(Widget). Таким образом, данные компиляторы сообщают нам, что и T, и param имеют один и тот же тип — const Widget\*.

Компилятор Microsoft согласен:

```
T      = class Widget const *
param = class Widget const *
```

Три независимых компилятора дают одну и ту же информацию, что свидетельствует о том, что эта информация является точной. Но давайте посмотрим более внимательно. В шаблоне f объявленным типом param является тип const T&. В таком случае не кажется ли вам странным, что и T, и param имеют один и тот же тип? Если тип T, например, представляет собой int, то типом param должен быть const int& — совершенно другой тип.

К сожалению, результат std::type\_info::name недостоверен. Например, в данном случае тип, который все три компилятора приписывают param, является неверным. Кроме того, он по сути обязан быть неверным, так как спецификация std::type\_info::name разрешает, чтобы тип рассматривался как если бы он был передан в шаблонную функцию по значению. Как поясняется в разделе 1.1, это означает, что если тип является ссылкой, его "ссылочность" игнорируется, а если тип после удаления ссылочности оказывается const (или volatile), то соответствующие модификаторы также игнорируются. Вот почему информация о типе param — который на самом деле представляет собой const Widget\*const& — выводится как const Widget\*. Сначала удаляется ссылочность, а затем у получившегося указателя удаляется константность.

Не менее печально, что информация о типе, выводимая редакторами IDE, также недостоверна — или как минимум недостоверно полезна. Для этого же примера мой редактор IDE сообщает о типе T как (я не придумываю!):

```
const
std::_Simple_types<std::_Wrap_alloc<std::_Vec_base_types<Widget,
std::allocator<Widget> >::_Alloc>::value_type>::value_type *
```

Тот же редактор IDE показывает, что тип param следующий:

```
const std::_Simple_types<...>::value_type *const &
```

Это выглядит менее страшно, чем тип T, но троеточие в середине типа сбивает с толку, пока вы не поймете, что это редактор IDE попытался сказать "Я опускаю все, что является частью типа T". Ваша среда разработки, быть может, работает лучше моей — если вы достаточно везучий.

Если вы склонны полагаться на библиотеки больше, чем на удачу, то будете рады узнать, что там, где std::type\_info::name и IDE могут ошибаться, библиотека Boost TypeIndex (часто именуемая как Boost.TypeIndex) приведет к успеху. Эта библиотека не является частью стандарта C++, но точно так же частью стандарта не являются ни IDE, ни шаблоны наподобие рассмотренного выше TD. Кроме того, тот факт, что библиотеки Boost (доступные по адресу boost.org) являются кроссплатформенными, с открытым исходным кодом и с лицензией, разработанной так, чтобы быть приемлемой даже

для самых параноидальных юристов, означает, что код с применением библиотек Boost переносим практически так же хорошо, как и код, основанный на стандартной библиотеке.

Вот как наша функция `f` может выдать точную информацию о типах с использованием Boost.TypeIndex:

```
#include <boost/type_index.hpp>

template<typename T>
void f(const T& param)
{
    using std::cout;
    using boost::typeindex::type_id_with_cvr;

    // Вывод информации о T
    cout << "T = "
        << type_id_with_cvr<T>().pretty_name()
        << '\n';

    // Вывод информации о типе param
    cout << "param = "
        << type_id_with_cvr<decltype(param)>().pretty_name()
        << '\n';
}

}
```

Как это работает? Шаблон функции `boost::typeindex::type_id_with_cvr` получает аргумент типа (тип, о котором мы хотим получить информацию) и не удаляет `const`, `volatile` или квалификатор ссылки (о чем и говорит “`with_cvr`” в имени шаблона). Результатом является объект `boost::typeindex::type_index`, функция-член `pretty_name` которого дает `std::string` с удобочитаемым представлением типа.

При такой реализации `f` обратимся вновь к вызову, который давал нам неверную информацию о типе `param` при использовании `typeid`:

```
std::vector<Widget> createVec(); // Фабричная функция

const auto vw = createVec();      // Инициализация vw с помощью
                                 // фабричной функции

if (!vw.empty()) {
    f(&vw[0]);                  // Вызов f
}

}
```

После компиляции с помощью компиляторов GNU и Clang Boost.TypeIndex дает следующий (точный) результат:

```
T      = Widget const*
param = Widget const* const&
```

Применение компилятора Microsoft дает по сути то же самое:

```
T      = class Widget const *
param = class Widget const * const &
```

Такое единобразие — это хорошо, но важно помнить, что редакторы IDE, сообщения об ошибках компилятора и библиотеки наподобие Boost.TypeIndex являются всего лишь инструментами, которые можно использовать для выяснения того, какие типы выводит ваш компилятор. Это может быть полезно, но не может заменить понимания информации о выводе типов, приведенной в разделах 1.1–1.3.

### Следует запомнить

- Выводимые типы часто можно просмотреть с помощью редакторов IDE, сообщений об ошибках компиляции и с использованием библиотеки Boost.TypeIndex.
- Результаты, которые выдают некоторые инструменты, могут оказаться как неточными, так и бесполезными, так что понимание правил вывода типов в C++ является совершенно необходимым.



# Объявление auto

Концептуально объявление `auto` настолько простое, насколько может быть, но все же сложнее, чем выглядит. Его применение экономит исходный текст, вводимый программистом, но при этом предупреждает появление вопросов корректности и производительности, над которыми вынужден мучиться программист при ручном объявлении типов. Кроме того, некоторые выводы типов `auto`, хотя и послушно соблюдают предписанные алгоритмы, дают результаты, некорректные с точки зрения программиста. Когда такое происходит, важно знать, как привести `auto` к верному ответу, поскольку возврат к указанию типов вручную — альтернатива, которой чаще всего лучше избегать.

В этой короткой главе описаны основы работы с `auto`.

## 2.1. Предпочитайте `auto` явному объявлению типа

Легко и радостно написать

```
int x;
```

Стоп! #@\$. Я забыл инициализировать `x`, так что эта переменная имеет неопределенное значение. Может быть. Но она может быть инициализирована и нулем — в зависимости от контекста. Жуть!

Ну, ладно. Давайте лучше порадуемся объявлению локальной переменной, инициализированной разыменованием итератора:

```
template<typename It> // Некий алгоритм, работающий с
void dwim(It b, It e) // элементами из диапазона от b до e
{
    while (b != e) {
        typename std::iterator_traits<It>::value_type
        currValue = *b;
    }
}
```

Жуть. `typename std::iterator_traits<It>::value_type` — просто чтобы записать тип значения, на которое указывает итератор? Нет, я такой радости не переживу... #@\$. Или я это уже говорил?..

Ладно, третья попытка. Попробую объявить локальную переменную, тип которой такой же, как у лямбда-выражения. Но его тип известен только компилятору. #@\$(Это становится привычкой...)

Да что же это такое — никакого удовольствия от программирования на C++!

Так не должно быть. И не будет! Мы дождались C++11, в котором все эти проблемы решены с помощью ключевого слова `auto`. Тип переменных, объявленных как `auto`, выводится из их инициализатора, так что они обязаны быть инициализированными. Это значит — прощай проблема неинициализированных переменных:

```
int x1;      // Потенциально неинициализированная переменная
auto x2;     // Ошибка! Требуется инициализатор
auto x3 = 0; // Все отлично, переменная x корректно определена
```

Нет проблем и с объявлением локальной переменной, значением которой является разыменование итератора:

```
template<typename It> // Все, как и ранее
void dwim(It b, It e)
{
    while (b != e) {
        auto currValue = *b;
        ...
    }
}
```

А поскольку `auto` использует вывод типов (см. раздел 1.2), он может представлять типы, известные только компиляторам:

```
auto derefUPLess =           // Функция сравнения
[](const std::unique_ptr<Widget>& p1, // объектов Widget, на
   const std::unique_ptr<Widget>& p2) // которые указывают
{ return *p1 < *p2; };           // std::unique_ptr
```

Просто круто! В C++14 все еще круче, потому что параметры лямбда-выражений также могут включать `auto`:

```
auto derefLess =           // Функция сравнения в C++14,
[](const auto& p1,         // для значений, на которые
   const auto& p2)         // указывает что угодно
{ return *p1 < *p2; };    // указателеобразное
```

Несмотря на всю крутость вы, вероятно, думаете, что можно обойтись и без `auto` для объявления переменной, которая хранит лямбда-выражение, поскольку мы можем использовать объект `std::function`. Это так, можем, но, возможно, это не то, что вы на самом деле подразумеваете. А может быть, вы сейчас думаете “А что это такое — объект `std::function`?“ Давайте разбираться.

`std::function` — шаблон стандартной библиотеки C++11, который обобщает идею указателя на функцию. В то время как указатели на функции могут указывать только

на функции, объект `std::function` может ссылаться на любой вызываемый объект, т.е. на все, что может быть вызвано как функция. Так же как при создании указателя на функцию вы должны указать тип функции, на которую указываете (т.е. сигнатуру функции, на которую хотите указать), вы должны указать тип функции, на которую будет ссылаться создаваемый объект `std::function`. Это делается с помощью параметра шаблона `std::function`. Например, для объявления объекта `std::function` с именем `func`, который может ссылаться на любой вызываемый объект, действующий так, как если бы его сигнатура была

```
bool(const std::unique_ptr<Widget>&, // Сигнатура C++11 для
      const std::unique_ptr<Widget>&) // функции сравнения
                                         // std::unique_ptr<Widget>
```

следует написать следующее:

```
std::function<bool(const std::unique_ptr<Widget>&,
                    const std::unique_ptr<Widget>&)> func;
```

Поскольку лямбда-выражения дают вызываемые объекты, замыкания могут храниться в объектах `std::function`. Это означает, что можно объявить C++11-версию `derefUPLess` без применения `auto` следующим образом:

```
std::function<bool(const std::unique_ptr<Widget>&,
                    const std::unique_ptr<Widget>&)>
derefUPLess = [](const std::unique_ptr<Widget>& p1,
                  const std::unique_ptr<Widget>& p2)
{ return *p1 < *p2; };
```

Важно понимать, что, даже если оставить в стороне синтаксическую многословность и необходимость повторения типов параметров, использование `std::function` — не то же самое, что использование `auto`. Переменная, объявленная с использованием `auto` и хранящая замыкание, имеет тот же тип, что и замыкание, и как таковая использует только то количество памяти, которое требуется замыканию. Тип переменной, объявленной как `std::function` и хранящей замыкание, представляет собой конкретизацию шаблона `std::function`, которая имеет фиксированный размер для каждой заданной сигнатуры. Этот размер может быть не адекватным для замыкания, которое требуется хранить, и в этом случае конструктор `std::function` будет выделять для хранения замыкания динамическую память. В результате объект `std::function` использует больше памяти, чем объект, объявленный с помощью `auto`. Кроме того, из-за деталей реализации это ограничивает возможности встраивания и приводит к косвенным вызовам функции, так что вызовы замыкания через объект `std::function` обычно выполняются медленнее, чем вызовы посредством объекта, объявленного как `auto`. Другими словами, подход с использованием `std::function` в общем случае более громоздкий, требующий больше памяти и более медленный, чем подход с помощью `auto`, и к тому же может приводить к генерации исключений, связанных с нехваткой памяти. Ну и, как вы уже видели в примерах выше, написать “`auto`” — гораздо проще, чем указывать тип для инстанцирования `std::function`. В соревновании между `auto`

и `std::function` для хранения замыкания побеждает `auto`. (Подобные аргументы можно привести и в пользу предпочтения `auto` перед `std::function` для хранения результатов вызовов `std::bind`, но все равно в разделе 6.4 я делаю все, чтобы убедить вас использовать вместо `std::bind` лямбда-выражения...)

Преимущества `auto` выходят за рамки избегания неинициализированных переменных, длинных объявлений переменных и возможности непосредственного хранения замыкания. Кроме того, имеется возможность избежать того, что я называю проблемой “сокращений типа” (type shortcuts). Вот кое-что, что вы, вероятно, уже видели, а возможно, даже писали:

```
std::vector<int> v;  
...  
unsigned sz = v.size();
```

Официальный возвращаемый тип `v.size()` — `std::vector<int>::size_type`, но об этом знает не так уж много разработчиков. `std::vector<int>::size_type` определен как беззнаковый целочисленный тип, так что огромное количество программистов считают, что `unsigned` вполне достаточно, и пишут исходные тексты, подобные показанному выше. Это может иметь некоторые интересные последствия. В 32-разрядной Windows, например, и `unsigned`, и `std::vector<int>::size_type` имеют один и тот же размер, но в 64-разрядной Windows `unsigned` содержит 32 бита, а `std::vector<int>::size_type` — 64 бита. Это означает, что код, который работал в 32-разрядной Windows, может вести себя некорректно в 64-разрядной Windows. И кому хочется тратить время на подобные вопросы при переносе приложения с 32-разрядной операционной системы на 64-разрядную?

Применение `auto` гарантирует, что вам не придется этим заниматься:

```
auto sz = v.size(); // Тип sz — std::vector<int>::size_type
```

Все еще не уверены в разумности применения `auto`? Тогда рассмотрите следующий код.

```
std::unordered_map<std::string, int> m;  
...  
for (const std::pair<std::string, int>& p : m)  
{  
    ... // Что-то делаем с p  
}
```

Выглядит вполне разумно... но есть одна проблема. Вы ее не видите?

Чтобы разобраться, что здесь не так, надо вспомнить, что часть `std::unordered_map`, содержащая ключ, является константной, так что тип `std::pair` в хеш-таблице (которой является `std::unordered_map`) *вовсе не* `std::pair<std::string, int>`, а `std::pair<const std::string, int>`. Но переменная `p` в приведенном выше цикле объявлена иначе. В результате компилятор будет искать способ преобразовать объекты `std::pair<const std::string, int>` (хранившиеся в хеш-таблице) в объекты `std::pair<std::string, int>` (объявленный тип `p`). Этот способ — создание временного объекта типа, требуемого `p`, чтобы скопировать в него каждый объект из `m` с последующим

связыванием ссылки `p` с этим временным объектом. В конце каждой итерации цикла временный объект уничтожается. Если этот цикл написан вами, вы, вероятно, будете удивлены его поведением, поскольку почти наверняка планировали просто связывать ссылку `p` с каждым элементом в `m`.

Такое непреднамеренное несоответствие легко лечится с помощью `auto`:

```
for (const auto& p : m)
{
    ... // Как и ранее
}
```

Это не просто более эффективно — это еще и менее многословно. Кроме того, этот код имеет очень привлекательную особенность — если вы возьмете адрес `p`, то можете быть уверены, что получите указатель на элемент в `m`. В коде, не использующем `auto`, вы получите указатель на временный объект — объект, который будет уничтожен в конце итерации цикла.

Два последних примера — запись `unsigned` там, где вы должны были написать `std::vector<int>::size_type`, и запись `std::pair<std::string, int>` там, где вы должны были написать `std::pair<const std::string, int>`, — демонстрируют, как явное указание типов может привести к неявному их преобразованию, которое вы не хотели и не ждали. Если вы используете в качестве типа целевой переменной `auto`, вам не надо беспокоиться о несоответствиях между типом объявленной переменной и типом инициализирующего ее выражения.

Таким образом, имеется несколько причин для предпочтительного применения `auto` по сравнению с явным объявлением типа. Но `auto` не является совершенным. Тип для каждой переменной, объявленной как `auto`, выводится из инициализирующего ее выражения, а некоторые инициализирующие выражения имеют типы, которые не предполагались и нежелательны. Условия, при которых возникают такие ситуации, и что при этом можно сделать, рассматриваются в разделах 1.2 и 2.2, поэтому здесь я не буду их рассматривать. Вместо этого я уделю внимание другому вопросу, который может вас волновать при использовании `auto` вместо традиционного объявления типа, — удобочитаемость полученного исходного текста.

Для начала сделайте глубокий вдох и расслабьтесь. Применение `auto` — возможность, а не требование. Если, в соответствии с вашими профессиональными представлениями, ваш код будет понятнее или легче сопровождаемым или лучше в каком-то ином отношении при использовании явных объявлений типов, вы можете продолжать их использовать. Но имейте в виду, что C++ — не первый язык, принявший на вооружение то, что в мире языков программирования известно как *вывод типов* (*type inference*). Другие процедурные статически типизированные языки программирования (например, C#, D, Scala, Visual Basic) обладают более или менее эквивалентными возможностями, не говоря уже о множестве статически типизированных функциональных языков (например, ML, Haskell, OCaml, F# и др.). В частности, это объясняется успехом динамически типизированных языков программирования, таких как Perl, Python и Ruby, в которых явная типизация переменных — большая редкость. Сообщество разработчиков программного

обеспечения имеет обширный опыт работы с выводом типов, и он продемонстрировал, что в такой технологии нет ничего мешающего созданию и поддержке крупных приложений промышленного уровня.

Некоторых разработчиков беспокоит тот факт, что применение `auto` исключает возможность определения типа при беглом взгляде на исходный текст. Однако возможности IDE показывать типы объектов часто устраниют эту проблему (даже если принять во внимание обсуждавшиеся в разделе 1.4 вопросы, связанные с выводом типов в IDE), а во многих случаях абстрактный взгляд на тип объекта столь же полезен, как и точный тип. Зачастую достаточно, например, знать, что объект является контейнером, счетчиком или интеллектуальным указателем, не зная при этом точно, каким именно контейнером, счетчиком или указателем. При правильном подборе имен переменных такая абстрактная информация о типе почти всегда оказывается под рукой.

Суть дела заключается в том, что явно указываемые типы зачастую мало что дают, кроме того что открывают возможности для ошибок — в плане как корректности, так и производительности программ. Кроме того, типы `auto` автоматически изменяются при изменении типов инициализирующих их выражений, а это означает облегчение выполнения рефакторинга при использовании `auto`. Например, если функция объявлена как возвращающая `int`, но позже вы решите, что `long` вас больше устраивает, вызывающий код автоматически обновится при следующей компиляции (если результат вызова функции хранится в переменной, объявленной как `auto`). Если результат хранится в переменной, объявленной как `int`, вы должны найти все точки вызова функции и внести необходимые изменения.

#### Следует запомнить

- Переменные, объявленные как `auto`, должны быть инициализированы; в общем случае они невосприимчивы к несоответствиям типов, которые могут привести к проблемам переносимости или эффективности; могут облегчить процесс рефакторинга; и обычно требуют куда меньшего количества ударов по клавишам, чем переменные с явно указанными типами.
- Переменные, объявленные как `auto`, могут быть подвержены неприятностям, описанным в разделах 1.2 и 2.2.

## 2.2. Если `auto` выводит нежелательный тип, используйте явно типизированный инициализатор

В разделе 2.1 поясняется, что применение `auto` для объявления переменных предоставляет ряд технических преимуществ по сравнению с явным указанием типов, но иногда вывод типа `auto` идет налево там, где вы хотите направо. Предположим, например, что у меня есть функция, которая получает `Widget` и возвращает `std::vector<bool>`, где каждый `bool` указывает, обладает ли `Widget` определенным свойством:

```
std::vector<bool> features(const Widget& w);
```

Предположим далее, что пятый бит указывает наличие высокого приоритета у Widget. Мы можем написать следующий код.

```
Widget w;  
...  
bool highPriority = features(w)[5]; // Имеет ли w высокий  
// приоритет?  
...  
processWidget(w, highPriority); // Обработка w в соответ-  
// ствии с приоритетом
```

В этом коде нет ничего неверного. Он корректно работает. Но если мы внесем кажущееся безобидным изменение и заменим явный тип highPriority типом auto

```
auto highPriority = features(w)[5]; // Имеет ли w высокий  
// приоритет?
```

то ситуация изменится. Код будет продолжать компилироваться, но его поведение больше не будет предсказуемым:

```
processWidget(w, highPriority); // Неопределенное поведение!
```

Как указано в комментарии, вызов processWidget теперь имеет неопределенное поведение. Но почему? Ответ, скорее всего, вас удивит. В коде, использующем auto, тип highPriority больше не является bool. Хотя концептуально std::vector<bool> хранит значения bool, operator[] у std::vector<bool> не возвращает ссылку на элемент контейнера (то, что std::vector::operator[] возвращает для всех типов за исключением bool). Вместо этого возвращается объект типа std::vector<bool>::reference (класса, вложенного в std::vector<bool>).

Тип std::vector<bool>::reference существует потому, что std::vector<bool> определен как хранящий значения bool в упакованном виде, по одному биту на каждое значение. Это создает проблему для оператора operator[] класса std::vector<bool>, поскольку operator[] класса std::vector<T> должен возвращать T&, но C++ запрещает ссылаться на отдельные биты. Будучи не в состоянии вернуть bool&, operator[] класса std::vector<bool> возвращает объект, который *действует подобно* bool&. Для успешной работы объекты std::vector<bool>::reference должны быть применимы по сути во всех контекстах, где применим bool&. Среди прочих возможностей std::vector<bool>::reference обладает неявным преобразованием в bool. (Не в bool&, а именно в bool. Пояснение всего набора методов, используемых std::vector<bool>::reference для эмуляции поведения bool&, завело бы нас слишком далеко, так что я просто замечу, что это неявное преобразование является только одним из камней в существенно большей мозаике.)

С учетом этой информации посмотрим еще раз на следующую часть исходного кода:

```
bool highPriority = features(w)[5]; // Явное объявление типа  
// highPriority
```

Здесь `features` возвращает объект `std::vector<bool>`, для которого вызывается `operator[]`. Этот оператор возвращает объект типа `std::vector<bool>::reference`, который затем неявно преобразуется в значение типа `bool`, необходимое для инициализации `highPriority`. Таким образом, `highPriority` в конечном итоге получает значение пятого бита из `std::vector<bool>`, возвращенного функцией `features`, так, как и предполагалось.

Но что же произойдет, если переменная `highPriority` будет объявлена как `auto`?

```
auto highPriority = features(w)[5]; // Вывод типа highPriority
```

Функция `features`, как и ранее, возвращает объект типа `std::vector<bool>`, и, как и ранее, выполняется его `operator[]`. Оператор возвращает объект типа `std::vector<bool>::reference`, но дальше привычный ход событий изменяется, так как `auto` приводит к выводу типа переменной `highPriority`. Теперь переменная `highPriority` не получает значение пятого бита `std::vector<bool>`, возвращенного вызовом `features`.

Полученное ею значение зависит от того, как реализован тип `std::vector<bool>::reference`. Одна из реализаций таких объектов состоит в том, чтобы содержать указатель на машинное слово с интересующим нас битом и смещение этого бита в слове. Рассмотрим, что это означает для инициализации `highPriority`, в предположении, что имеет место именно такая реализация `std::vector<bool>::reference`.

Вызов `features` возвращает временный объект `std::vector<bool>`. Этот объект не имеет имени, но для упрощения нашего рассмотрения я буду называть его `temp`. Для `temp` вызывается `operator[]`, в результате чего возвращается объект `std::vector<bool>::reference`, содержащий указатель на слово в структуре данных, хранящей интересующий нас бит (эта структура находится под управлением `temp`), плюс смещение в слове, соответствующее пятому биту. Переменная `highPriority` представляет собой копию этого объекта `std::vector<bool>::reference`, так что `highPriority` тоже содержит указатель на слово в `temp` плюс смещение, соответствующее пятому биту. В конце инструкции объект `temp` уничтожается, так как это объект временный. В результате переменная `highPriority` содержит висячий указатель, что и дает неопределенное поведение при вызове `processWidget`:

```
processWidget(w, highPriority); // Неопределенное поведение!
                                // highPriority содержит
                                // висячий указатель!
```

Класс `std::vector<bool>::reference` является примером *прокси-класса* (*proxy class*), т.е. класса, цель которого — эмуляция и дополнение поведения некоторого другого типа. Прокси-классы применяются для множества разных целей. Например, `std::vector<bool>::reference` нужен для того, чтобы создать иллюзию, что `operator[]` класса `std::vector<bool>` возвращает ссылку на бит, а интеллектуальные указатели стандартной библиотеки (см. главу 4, “Интеллектуальные указатели”) являются прокси-классами, которые добавляют к обычным указателям управление ресурсами. Полезность прокси-классов — давно установленный и не вызывающий сомнения факт. Фактически

шаблон проектирования “Прокси” — один из наиболее давних членов пантеона шаблонов проектирования программного обеспечения.

Одни прокси-классы спроектированы так, чтобы быть очевидными для клиентов. Это, например, такие классы, как `std::shared_ptr` и `std::unique_ptr`. Другие прокси-классы спроектированы для более-менее невидимой работы. Примером такого “невидимого” прокси-класса является `std::vector<bool>::reference`, как и его собрат `std::bitset::reference` из класса `std::bitset`.

В этом же лагере находятся и некоторые классы библиотек C++, применяющих технологию, известную как *шаблоны выражений* (expression templates). Такие библиотеки изначально разрабатывались для повышения эффективности кода для числовых вычислений. Например, для заданного класса `Matrix` и объектов `m1`, `m2`, `m3` и `m4` класса `Matrix`, выражение

```
Matrix sum = m1 + m2 + m3 + m4;
```

может быть вычислено более эффективно, если `operator+` для объектов `Matrix` возвращает не сам результат, а его прокси-класс. Иначе говоря, `operator+` для двух объектов `Matrix` должен возвращать объект прокси-класса, такого как `Sum<Matrix, Matrix>`, а не объект `Matrix`. Как и в случае с `std::vector<bool>::reference` и `bool`, должно иметься неявное преобразование из прокси-класса в `Matrix`, которое позволит инициализировать `sum` прокси-объектом, полученным из выражения справа от знака “`=`”. (Тип этого объекта будет традиционно кодировать все выражение инициализации, т.е. быть чем-то наподобие `Sum<Sum<Sum<Matrix, Matrix>, Matrix>`. Определенно, это тип, от которого следует защитить клиентов.)

В качестве общего правила “невидимые” прокси-классы не умеют хорошо работать вместе с `auto`. Для объектов таких классов зачастую не предусматривается существование более длительное, чем одна инструкция, так что создание переменных таких типов, как правило, нарушает фундаментальные предположения проекта библиотеки. Это справедливо для `std::vector<bool>::reference`, и мы видели, как нарушение предположений ведет к неопределенному поведению.

Следовательно, надо избегать кода следующего вида:

```
auto someVar = выражение с типом "невидимого" прокси-класса;
```

Но как распознать, когда используется прокси-объект? Программное обеспечение, использующее невидимый прокси, вряд ли станет его рекламировать. Ведь эти прокси-объекты должны быть *невидимыми*, по крайней мере концептуально! И если вы обнаружите их, то действительно ли следует отказываться от `auto` и массы преимуществ, продемонстрированных для него в разделе 2.1?

Давайте сначала зададимся вопросом, как найти прокси. Хотя “невидимые” прокси-классы спроектированы таким образом, чтобы при повседневном применении “летать вне досягаемости радара программиста”, использующие их библиотеки часто документируют такое применение. Чем лучше вы знакомы с основными проектными решениями

используемых вами библиотек, тем менее вероятно, что вы пропустите такой прокси не замеченный.

Там, где документация слишком краткая, на помощь могут прийти заголовочные файлы. Возможность скрытия прокси-объектов в исходном коде достаточно редка. Обычно прокси-объекты возвращаются из функций, которые вызываются клиентами, так что сигнатуры этих функций отражают существование прокси-объектов. Например, вот как выглядит `std::vector<bool>::operator[]`:

```
namespace std {    // Из стандарта C++
    template <class Allocator>
    class vector<bool, Allocator> {
        public:
            ...
            class reference { ... };

            reference operator[](size_type n);

        };
    }
}
```

В предположении, что вы знаете, что `operator[]` у `std::vector<T>` обычно возвращает `T&`, необычный возвращаемый тип у `operator[]` в данном случае должен навести вас на мысль о применении здесь прокси-класса. Уделяя повышенное внимание используемым интерфейсам, часто можно выявить наличие прокси-классов.

На практике многие разработчики обнаруживают применение прокси-классов только тогда, когда пытаются отследить источник таинственных проблем при компиляции или отладить никак не проходящий тесты модуль. Независимо от того, как вы его обнаружили, после того как выясняется, что `auto` определен как выведенный тип прокси-класса вместо “проксифицируемого” типа, решение не требует отказа от `auto`. Само по себе ключевое слово `auto` проблемой не является. Проблема в том, что `auto` выводит не тот тип, который вам нужен. Решение заключается в том, чтобы обеспечить вывод другого типа. Способ достижения этого заключается в том, что я называю *идиомой явной типизации инициализатора*.

Идиома явной типизации инициализатора включает объявление переменной с использованием `auto`, но с приведением инициализирующего выражения к тому типу, который должен вывести `auto`. Например, вот как можно использовать эту идиому, чтобы заставить `highPriority` стать переменной типа `bool`:

```
auto highPriority = static_cast<bool>(features(w)[5]);
```

Здесь `features(w)[5]` продолжает, как и ранее, возвращать объект типа `std::vector<bool>::reference`, но приведение изменяет тип выражения на `bool`, который `auto` затем выводит в качестве типа переменной `highPriority`. Во время выполнения программы объект `std::vector<bool>::reference`, который возвращается вызовом `std::vector<bool>::operator[]`, преобразуется в значение `bool` и в качестве

части преобразования выполняется разыменование все еще корректного указателя на `std::vector<bool>`, возвращенного вызовом `features`. Это позволяет избежать неопределенного поведения, с которым мы сталкивались ранее. Затем к битам, на которые указывает указатель, применяется индексация с индексом 5 и полученное значение типа `bool` используется для инициализации переменной `highPriority`.

В примере с `Matrix` идиома явно типизированного инициализатора выглядит следующим образом:

```
auto sum = static_cast<Matrix>(m1 + m2 + m3 + m4);
```

Применение идиомы не ограничивается инициализаторами, производимыми прокси-классами. Она может быть полезной для того, чтобы подчеркнуть, что вы сознательно создаете переменную типа, отличного от типа, генерируемого инициализирующим выражением. Предположим, например, что у вас есть функция для вычисления некоторого значения отклонения:

```
double calcEpsilon(); // Возвращает значение отклонения
```

Очевидно, что `calcEpsilon` возвращает значение `double`, но предположим, что вы знаете, что для вашего приложения точности `float` вполне достаточно и для вас существенна разница в размерах между `float` и `double`. Вы можете объявить переменную типа `float` для хранения результата функции `calcEpsilon`

```
float ep = calcEpsilon(); // Неявное преобразование  
                          // double -> float
```

но это вряд ли выражает мысль “я намеренно уменьшаю точность значения, возвращенного функцией”. Зато это делает идиома явной типизации инициализатора:

```
auto ep = static_cast<float>(calcEpsilon());
```

Аналогичные рассуждения применяются, если у вас есть выражение с плавающей точкой, которое вы преднамеренно сохраняете как целочисленное значение. Предположим, что вам надо вычислить индекс элемента в контейнере с итераторами произвольного доступа (например, `std::vector`, `std::deque` или `std::array`) и вы получаете значение типа `double` между 0.0 и 1.0, указывающее, насколько далеко от начала контейнера расположен этот элемент (0.5 указывает на середину контейнера). Далее, предположим, что вы уверены в том, что полученный индекс можно разместить в `int`. Если ваш контейнер — `c`, а значение с плавающей точкой — `d`, индекс можно вычислить следующим образом:

```
int index = d * (c.size() - 1);
```

Но здесь скрыт тот факт, что вы преднамеренно преобразуете `double` справа от знака “`=`” в `int`. Идиома явно типизированного инициализатора делает этот факт очевидным:

```
auto index = static_cast<int>(d * (c.size() - 1));
```

### **Следует запомнить**

- “Невидимые” прокси-типы могут привести `auto` к выводу неверного типа инициализирующего выражения.
- Идиома явно типизированного инициализатора заставляет `auto` выводить тот тип, который нужен вам.

# Переход к современному C++

Когда дело доходит до умных терминов, C++11 и C++14 есть чем похвастаться. `auto`, интеллектуальные указатели, семантика перемещения, лямбда-выражения, параллелизм — каждая возможность настолько важна, что я посвящаю ей отдельную главу. Очень важно освоить все эти возможности, но большой путь к эффективному программированию на современном C++ требует множества маленьких шажков. Каждый шаг отвечает на конкретные вопросы, возникающие во время путешествия от C++98 к современному C++. Когда следует использовать фигурные скобки вместо круглых для создания объектов? Почему объявление псевдонимов лучше, чем применение `typedef`? Чем `constexpr` отличается от `const`? Как связаны константные функции-члены и безопасность с точки зрения потоков? Этот список можно продолжать и продолжать. В этой главе постепенно, один за другим, даются ответы на эти вопросы.

## 3.1. Различие между {} и () при создании объектов

В зависимости от вашей точки зрения выбор синтаксиса для инициализации объектов в C++11 либо очень богатый, либо запутанный и беспорядочный. Как правило, инициализирующие значения указываются с помощью круглых скобок, знака равенства или фигурных скобок:

```
int x(0);    // Инициализатор в круглых скобках
int y = 0;   // Инициализатор после "="
int z{ 0 }; // Инициализатор в фигурных скобках
```

Во многих случаях можно использовать знак равенства и фигурные скобки одновременно:

```
int z = { 0 }; // Инициализатор использует "=" и фигурные скобки
```

В оставшейся части данного раздела я в основном буду игнорировать синтаксис, в котором одновременно используются знак равенства и фигурные скобки, поскольку C++ обычно трактует его так же, как и версию только с фигурными скобками.

Сторонники “полного беспорядка” указывают на то, что применение знака равенства для инициализации часто сбивает с толку новичков в C++, которые считают, что имеют дело с присваиванием, хотя на самом деле это не так. Для встроенных типов наподобие

`int` эта разница носит чисто академический характер, но в случае пользовательских типов очень важно отличать инициализацию от присваивания, поскольку при этом вызываются различные функции:

```
Widget w1;      // Вызов конструктора по умолчанию
Widget w2 = w1; // Не присваивание, а копирующий конструктор
w1 = w2;        // Присваивание; вызов оператора operator=()
```

Даже при наличии нескольких синтаксисов инициализации существовали определенные ситуации, когда в C++98 не было возможности выразить желаемую инициализацию. Например, было невозможно прямо указать, что контейнер STL должен быть создан содержащим определенный набор значений (например, 1, 3 и 5).

Для устранения путаницы из-за нескольких синтаксисов инициализации и решения проблемы охвата всех сценариев инициализации C++11 вводит *унифицированную инициализацию* (*uniform initialization*): единый синтаксис инициализации, который может, как минимум концептуально, использоваться везде и выражать все. Он основан на фигурных скобках, и по этой причине я лично предпочитаю термин “*фигурная инициализация*” (*braced initialization*). Унифицированная инициализация — это идея. Фигурная инициализация — это синтаксическая конструкция.

Фигурная инициализация позволяет выразить то, что было невозможно выразить ранее. С помощью фигурных скобок легко указать начальное содержимое контейнера:

```
std::vector<int> v{ 1, 3, 5 }; // v изначально содержит 1, 3, 5
```

Фигурные скобки могут также использоваться для указания значений инициализации по умолчанию для нестатических членов-данных. Эта возможность — новая в C++11 — может использоваться с синтаксисом “`=`”, но не с круглыми скобками:

```
class Widget {
...
private:
    int x{ 0 }; // OK, значение x по умолчанию равно 0
    int y = 0;   // Тоже OK
    int z(0);   // Ошибка!
};
```

С другой стороны, некопируемые объекты (например, `std::atomic` — см. раздел 7.6) могут быть инициализированы с помощью фигурных или круглых скобок, но не с помощью знака равенства:

```
std::atomic<int> ai1{ 0 }; // OK
std::atomic<int> ai2(0);  // OK
std::atomic<int> ai3 = 0;  // Ошибка!
```

Легко понять, почему фигурная инициализация названа “*унифицированной*”. Из трех способов обозначения выражений инициализации только фигурные скобки могут использоваться везде.

Новая возможность фигурной инициализации заключается в том, что она запрещает неявные сужающие преобразования среди встроенных типов. Если значение выражения в фигурном инициализаторе не может быть гарантированно выражено типом инициализируемого объекта, код не компилируется:

```
double x, y, z;  
...  
int sum1{ x + y + z }; // Ошибка! Сумма double может  
// не выражаться с помощью int
```

Инициализация с использованием круглых скобок и знака равенства не выполняет проверку сужающего преобразования, поскольку это может привести к неработоспособности большого количества старого кода:

```
int sum2(x+y+z); // OK (значение выражения усекается до int)  
int sum3 = x+y+z; //
```

Обращает на себя внимание еще одна особенность фигурной инициализации — она не подвержена *наиболее неприятному анализу* в C++. Побочным эффектом правила C++, согласно которому все, что в ходе синтаксического анализа может рассматриваться как объявление, должно рассматриваться как таковое, является так называемый наиболее неприятный анализ, который чаще всего досаждает разработчикам, когда они хотят создать объект по умолчанию, а в результате получают объявление функции. Корень проблемы кроется в том, что если вы хотите вызвать конструктор с аргументом, вы делаете это примерно следующим образом:

```
Widget w1(10); // Вызов конструктора Widget с аргументом 10
```

Но если вы пытаетесь вызвать конструктор Widget без аргументов с помощью аналогичного синтаксиса, то фактически объявляете функцию вместо объекта:

```
Widget w2(); // Синтаксический анализ рассматривает это как  
// объявление функции w2, возвращающей Widget!
```

Функции не могут быть объявлены с использованием фигурных скобок для списка параметров, так что конструирование объекта по умолчанию с применением фигурных скобок такой проблемы не вызовет:

```
Widget w3{}; // Вызов конструктора Widget без аргументов
```

Таким образом, в пользу фигурной инициализации имеется много “за”. Это синтаксис, который может использоваться в самых разнообразных контекстах, предотвращающий неявные сужающие преобразования и не подверженный неприятностям с синтаксическим анализом C++. Тройное “за”! Так почему бы не озаглавить раздел просто “Используйте синтаксис фигурной инициализации”?

Основной недостаток фигурной инициализации — временами сопровождающее ее удивительное поведение. Такое поведение вырастает из необыкновенно запутанных взаимоотношений между фигурной инициализацией, `std::initializer_list` и разрешением перегрузки конструкторов. Их взаимодействие может привести к коду, который, как

кажется, должен делать что-то одно, а в результате делает что-то совсем другое. Например, в разделе 1.2 поясняется, что когда переменная, объявленная как auto, имеет инициализатор в фигурных скобках, то выводимым типом является std::initializer\_list, несмотря на то что другой способ объявления переменной с тем же инициализатором даст более ожидаемый тип. В результате чем больше вам нравится auto, тем меньше энтузиазма вы должны проявить по отношению к фигурной инициализации.

В вызовах конструктора круглые и фигурные скобки имеют один и тот же смысл, пока в конструкторах не принимают участие параметры std::initializer\_list:

```
class Widget {  
public:  
    Widget(int i, bool b); // Конструкторы не имеют параметров  
    Widget(int i, double d); // std::initializer_list  
  
};  
  
Widget w1{10, true}; // Вызов первого конструктора  
Widget w2{10, true}; // Вызов первого конструктора  
Widget w3(10, 5.0); // Вызов второго конструктора  
Widget w4{10, 5.0}; // Вызов второго конструктора
```

Если же один или несколько конструкторов объявляют параметр типа std::initializer\_list, вызовы, использующие синтаксис фигурной инициализации, строго предпочитают перегрузки, принимающие std::initializer\_list. Строго. Если у компилятора есть любой способ истолковать вызов с фигурным инициализатором как конструктор, принимающий std::initializer\_list, он использует именно это толкование. Если класс Widget выше дополнить конструктором, принимающим, например, std::initializer\_list<long double>

```
class Widget {  
public:  
    Widget(int i, bool b); // Как и ранее  
    Widget(int i, double d); // Как и ранее  
  
    Widget(std::initializer_list<long double> il); // Добавлен  
  
};
```

то w2 и w4 будут созданы с помощью нового конструктора, несмотря на то что тип элементов std::initializer\_list (в данном случае — long double) хуже соответствует обоим аргументам по сравнению с конструкторами, не принимающими std::initializer\_list! Смотрите сами:

```
Widget w1(10, true); // Использует круглые скобки и, как и  
                      // ранее, вызывает первый конструктор  
Widget w2{10, true}; // Использует фигурные скобки, но теперь
```

```

    // вызывает третий конструктор
    // (10 и true преобразуются в long double)
Widget w3(10, 5.0); // Использует круглые скобки и, как и
                    // ранее, вызывает второй конструктор
Widget w4{10, 5.0}; // Использует фигурные скобки, но теперь
                    // вызывает третий конструктор
                    // (10 и 5.0 преобразуются в long double)

```

Даже то, что в обычной ситуации представляет собой копирующий или перемещающий конструктор, может быть перехвачено конструктором с `std::initializer_list`:

```

class Widget {
public:
    Widget(int i, bool b);                                // Как ранее
    Widget(int i, double d);                             // Как ранее
    Widget(std::initializer_list<long double> il); // Как ранее

    operator float() const;                // Преобразование во float

};

Widget w5(w4);           // Использует круглые скобки, вызывает
                        // копирующий конструктор
Widget w6{w4};          // Использует фигурные скобки, вызов
                        // конструктора с std::initializer_list
                        // (w4 преобразуется во float, а float
                        // преобразуется в long double)
Widget w7(std::move(w4)); // Использует круглые скобки, вызывает
                        // перемещающий конструктор
Widget w8{std::move(w4)}; // Использует фигурные скобки, вызов
                        // конструктора с std::initializer_list
                        // (все, как для w6)

```

Определение компилятором соответствия фигурных инициализаторов конструкто-  
рам с `std::initializer_list` настолько строгое, что доминирует даже тогда, когда кон-  
структор с `std::initializer_list` с наилучшим соответствием не может быть вызван,  
например:

```

class Widget {
public:
    Widget(int i, bool b); // Как ранее
    Widget(int i, double d); // Как ранее
    Widget(std::initializer_list<bool> il); // Теперь тип
                                              // элемента - bool
    // Нет функций неявного преобразования

Widget w{10, 5.0}; // Ошибка! Требуется сужающее преобразование

```

Здесь компилятор игнорирует первые два конструктора (второй из которых в точности соответствует обоим типам аргументов) и пытается вызвать конструктор, получающий аргумент типа `std::initializer_list<bool>`. Вызов этого конструктора требует преобразования значений `int` (10) и `double` (5.0) в `bool`. Оба эти преобразования являются сужающими (`bool` не может в точности представить ни первое, ни второе значения), а так как сужающие преобразования запрещены в фигурных инициализаторах, вызов является некорректным, и код отвергается.

И только если нет никакой возможности преобразовать типы аргументов в фигурном инициализаторе в типы в `std::initializer_list`, компилятор возвращает к нормальному разрешению перегрузки. Например, если мы заменим конструктор `c std::initializer_list<bool>` конструктором, принимающим `std::initializer_list<std::string>`, то кандидатами на вызов вновь станут конструкторы, не принимающие `std::initializer_list` (поскольку нет никакого способа преобразовать `int` и `bool` в `std::string`):

```
class Widget {  
public:  
    Widget(int i, bool b); // Как ранее  
    Widget(int i, double d); // Как ранее  
  
    // Теперь тип элементов std::initializer_list – std::string:  
    Widget(std::initializer_list<std::string> il);  
  
    // Нет функций неявного преобразования  
};  
  
Widget w1(10, true); // Круглые скобки, первый конструктор  
Widget w2{10, true}; // Фигурные скобки, первый конструктор  
Widget w3(10, 5.0); // Круглые скобки, второй конструктор  
Widget w4{10, 5.0}; // Фигурные скобки, второй конструктор
```

Это приводит нас к завершению изучения фигурных инициализаторов и перегрузки конструкторов, но есть еще один интересный предельный случай, который хотелось бы рассмотреть. Предположим, что вы используете пустые фигурные скобки для создания объекта, который поддерживает конструктор по умолчанию и конструктор `c std::initializer_list`. Что при этом будут означать пустые фигурные скобки? Если они означают “без аргументов”, будет вызван конструктор по умолчанию, но если они означают “пустой `std::initializer_list`”, то будет вызван конструктор `c std::initializer_list` без элементов.

Правило заключается в том, что будет вызван конструктор по умолчанию. Пустые фигурные скобки означают отсутствие аргументов, а не пустой `std::initializer_list`:

```
class Widget {  
public:  
    // Конструктор по умолчанию:  
    Widget();
```

```
// Конструктор с std::initializer_list
Widget(std::initializer_list<int> il);

    // Нет функций неявного преобразования
};

Widget w1;    // Вызов конструктора по умолчанию
Widget w2{}; // Вызов конструктора по умолчанию
Widget w3(); // Трактуется как объявление функции!
```

Если вы хотите вызвать конструктор с пустым std::initializer\_list, то это можно сделать, передавая пустые фигурные скобки в качестве аргумента конструктора в круглых или фигурных скобках, окружающих передаваемые вами:

```
Widget w4({}); // Вызов конструктора с пустым
                // std::initializer_list
Widget w5{}; // То же самое
```

Сейчас, когда кажущиеся магическими правила фигурной инициализации, std::initializer\_list и перегрузки конструкторов переполняют ваш мозг, вы можете удивиться, какое большое количество информации влияет на повседневное программирование. На самом деле даже больше, чем вы думаете, потому что одним из классов, на которые все это оказывает непосредственное влияние, является std::vector. Класс std::vector имеет конструктор без std::initializer\_list, который позволяет вам указать начальный размер контейнера и значение, присваиваемое каждому из его элементов; но при этом имеется также конструктор, принимающий std::initializer\_list и позволяющий указать начальные значения контейнера. Если вы создаете std::vector числового типа (например, std::vector<int>) и передаете ему два аргумента, то при использовании круглых и фигурных скобок вы получите совершенно разные результаты:

```
std::vector<int> v1(10, 20); // Используется конструктор без
                            // std::initializer_list: создает
                            // std::vector с 10 элементами;
                            // значение каждого равно 20
std::vector<int> v2{10, 20}; // Используется конструктор с
                            // std::initializer_list: создает
                            // std::vector с 2 элементами со
                            // значениями 10 и 20
```

Но давайте сделаем шаг назад от std::vector, а также от деталей применения круглых скобок, фигурных скобок и правил перегрузки конструкторов. Имеются два основных вывода из этого обсуждения. Во-первых, как автор класса вы должны быть осведомлены о том, что если ваш набор перегружаемых конструкторов включает один или несколько конструкторов, использующих std::initializer\_list, то клиентский код с фигурной инициализацией может рассматривать только перегрузки с std::initializer\_list. В результате лучше проектировать конструкторы так, чтобы перегрузка не зависела от того, используете вы круглые или фигурные скобки. Другими словами, вынесите уроки

из того, что сейчас рассматривается как ошибка дизайна класса `std::vector`, и проектируйте свои классы так, чтобы избегать подобных ошибок.

Следствием этого является то, что если у вас есть класс без конструктора `std::initializer_list` и вы добавляете таковой, то клиентский код, использующий фигурную инициализацию, может обнаружить, что вызовы, разрешавшиеся с использованием конструкторов без `std::initializer_list`, теперь разрешаются в новые функции. Конечно, такое может случиться в любой момент при добавлении новой функции ко множеству перегруженных функций: вызов, который разрешался в одну из старых функций, теперь может приводить к вызову новой. Разница в данном случае в том, что перегрузки с `std::initializer_list` не только конкурируют с другими перегрузками, но практически полностью перекрывают для них возможность быть рассмотренными в качестве потенциальных кандидатов. Поэтому такое добавление должно выполняться только после тщательного обдумывания.

Второй урок заключается в том, что в качестве клиента класса вы должны тщательно выбирать между круглыми и фигурными скобками при создании объектов. Большинство разработчиков в конечном итоге выбирают один вид скобок как применяемый по умолчанию, а другой — только при необходимости. Применение по умолчанию фигурных скобок привлекает их непревзойденным диапазоном применимости, запретом применения сужающих преобразований и их иммунитетом к особенностям синтаксического анализа. Такие люди понимают, что в некоторых случаях (например, при создании вектора `std::vector` с заданными размером и начальным значением элемента) необходимо использовать круглые скобки. С другой стороны, немало программистов используют в качестве выбора по умолчанию круглые скобки. Они привлекательны своей согласованностью с синтаксическими традициями C++98, тем, что позволяют избегать проблем с выводом `auto` как `std::initializer_list`, и уверенностью, что вызовы при создании объектов не приведут к случайным вызовам конструкторов с `std::initializer_list`. Эти программисты признают, что иногда следует использовать именно фигурные скобки (например, при создании контейнера с определенными значениями). Нет определенного превалирующего мнения о том, какой подход лучше, поэтому могу посоветовать только выбрать один из них и постоянно ему следовать.

Если вы автор шаблона, противостояние в применении круглых и фигурных скобок может быть особенно неприятным, потому что в общем случае невозможно сказать, какие скобки должны использоваться. Предположим, например, что вы хотите создать объект произвольного типа с произвольным количеством аргументов. Использование шаблонов с переменным количеством параметров позволяет сделать это концептуально достаточно просто:

```
template<typename T,           // Тип создаваемого объекта
         typename... Ts> // Типы используемых аргументов
void doSomeWork(Ts&... params)
{
    Создание локального объекта T из params...
}
```

Есть два способа превратить строку псевдокода в реальный код (см. в разделе 5.3 информацию о `std::forward`):

```
T localObject(std::forward<Ts>(params)...); // Круглые скобки  
T localObject{std::forward<Ts>(params)...}; // Фигурные скобки
```

Рассмотрим следующий вызывающий код:

```
std::vector<int> v;  
...  
doSomeWork<std::vector<int>>(10, 20);
```

Если `doSomeWork` использует при создании объекта `localObject` круглые скобки, в результате будет получен `std::vector` с 10 элементами. Если же `doSomeWork` использует фигурные скобки, то результатом будет `std::vector` с двумя элементами. Какой из этих вариантов корректен? Автор `doSomeWork` не может этого знать. Это может знать только вызывающий код.

Это именно та проблема, которая встает перед функциями стандартной библиотеки `std::make_unique` и `std::make_shared` (см. раздел 4.4). Эти функции решают проблему, используя круглые скобки и документируя это решение как части своих интерфейсов<sup>1</sup>.

### Следует запомнить

- Фигурная инициализация является наиболее широко используемым синтаксисом инициализации, предотвращающим сужающие преобразования и нечувствительным к особенностям синтаксического анализа C++.
- В процессе разрешения перегрузки конструкторов фигурные инициализаторы соответствуют параметрам `std::initializer_list`, если это возможно, даже если другие конструкторы обеспечивают лучшее соответствие.
- Примером, в котором выбор между круглыми и фигурными скобками приводит к значительно отличающимся результатам, является создание `std::vector<числовой_тип>` с двумя аргументами.
- Выбор между круглыми и фигурными скобками для создания объектов внутри шаблонов может быть очень сложным.

## 3.2. Предпочитайте `nullptr` значениям 0 и NULL

Дело вот в чем: литерал 0 представляет собой `int`, а не указатель. Если C++ встретит 0 в контексте, где может использоваться только указатель, он интерпретирует 0 как нулевой указатель, но это — запасной выход. Фундаментальная стратегия C++ состоит в том, что 0 — это значение типа `int`, а не указатель.

С практической точки зрения то же самое относится и к NULL. В случае NULL имеется некоторая неопределенность в деталях, поскольку реализациям позволено придавать

<sup>1</sup> Возможен и более гибкий дизайн, который позволяет вызывающим функциям определить, должны ли использоваться круглые или фигурные скобки в функциях, генерируемых из шаблонов. Подробности можно найти в записи от 5 июня 2013 года в *Andrzej's C++ blog*, "Intuitive interface — Part I".

NULL целочисленный тип, отличный от `int` (например, `long`). Это не является распространенной практикой, но в действительности не имеет значения, поскольку вопрос не в точном типе NULL, а в том, что ни 0, ни NULL не имеют тип указателя.

В C++98 основным следствием этого факта было то, что перегрузка с использованием типов указателей и целочисленных типов могла вести к сюрпризам. Передача 0 или NULL таким перегрузкам никогда не приводила к вызову функции с указателем:

```
void f(int);      // Три перегрузки функции f
void f(bool);
void f(void*);

f(0);            // Вызов f(int), не f(void*)
f(NULL);         // Может не компилироваться, но обычно
                  // вызывает f(int) и никогда – f(void*)
```

Неопределенность в отношении поведения `f(NULL)` является отражением свободы, предоставленной реализациям в отношении типа NULL. Если NULL определен, например, как 0L (т.е. 0 как значение типа `long`), то вызов является неоднозначным, поскольку преобразования `long` в `int`, `long` в `bool` и 0L в `void*` рассматриваются как одинаково подходящие. Интересно, что этот вызов является противоречием между *видимым* смыслом исходного текста (“вызываем `f` с нулевым указателем `NULL`”) и *фактическим* смыслом (“вызываем `f` с некоторой разновидностью целых чисел — не указателем”). Это противоречащее интуиции поведение приводит к рекомендации программистам на C++98 избегать перегрузки типов указателей и целочисленных типов. Эта рекомендация остается в силе и в C++11, поскольку, несмотря на рекомендации данного раздела, некоторые разработчики, определенно, продолжат применять 0 и NULL, несмотря на то что `nullptr` является лучшим выбором.

Преимущество `nullptr` заключается в том, что это значение не является значением целочисленного типа. Честно говоря, он не имеет и типа указателя, но его можно рассматривать как указатель *любого* типа. Фактическим типом `nullptr` является `std::nullptr_t`, но, а тип `std::nullptr_t` циклически определяется как тип значения `nullptr...`. Тип `std::nullptr_t` неявно преобразуется во все типы обычных указателей, и именно это делает `nullptr` действующим как указатель всех типов.

Вызов перегруженной функции `f` с `nullptr` приводит к вызову перегрузки `void*` (т.е. перегрузки с указателем), поскольку `nullptr` нельзя рассматривать как что-то целочисленное:

```
f(nullptr);    // Вызов f(void*)
```

Использование `nullptr` вместо 0 или NULL, таким образом, позволяет избежать сюрпризов перегрузки, но это не единственное его преимущество. Оно позволяет также повысить ясность кода, в особенности при применении `auto`-переменных. Предположим, например, что у нас есть следующий исходный текст:

```
auto result = findRecord( /* Аргументы */ );
if (result == 0) {
}
```

Если вы случайно не знаете (или не можете быстро найти), какой тип возвращает `findRecord`, может быть неясно, имеет ли `result` тип указателя или целочисленный тип. В конце концов, значение 0 (с которым сравнивается `result`) может быть в обоих случаях. С другой стороны, если вы увидите код

```
auto result = findRecord( /* Аргументы */ );
if (result == nullptr) {  
}
```

то здесь нет никакой неоднозначности: `result` должен иметь тип указателя.

Особенно ярко сияет `nullptr`, когда на сцене появляются шаблоны. Предположим, что у вас есть несколько функций, которые должны вызываться только при блокировке соответствующего мьютекса. Каждая функция получает указатель определенного вида:

```
int f1(std::shared_ptr<Widget> spw);      // Вызывается только при
double f2(std::unique_ptr<Widget> upw); // блокировке соответ-
bool f3(Widget* pw);                      // ствующего мьютекса
```

Вызывающий код с передачей нулевых указателей может выглядеть следующим образом:

```
std::mutex f1m, f2m, f3m; // Мьютексы для f1, f2 и f3  
  
using MuxGuard =           // C++11 typedef; см. раздел 3.3
    std::lock_guard<std::mutex>;  
  
{  
    MuxGuard g(f1m);          // Блокировка мьютекса для f1  
    auto result = f1(0);       // Передача 0 функции f1  
}                                // Разблокирование мьютекса  
...  
{  
    MuxGuard g(f2m);          // Блокировка мьютекса для f2  
    auto result = f2(nullptr); // Передача NULL функции f2  
}                                // Разблокирование мьютекса  
...  
{  
    MuxGuard g(f3m);          // Блокировка мьютекса для f3  
    auto result = f3(nullptr); // Передача nullptr функции f3  
}                                // Разблокирование мьютекса
```

То, что в первых двух вызовах не был передан `nullptr`, грустно; тем не менее код работает, а это чего-то да стоит. Однако повторяющиеся действия еще более грустны. Они просто беспокоят. Во избежание дублирования такого вида и предназначаются шаблоны, так что давайте превратим эти действия в шаблон.

```

template<typename FuncType,
         typename MuxType,
         typename PtrType>
auto lockAndCall(FuncType func,
                 MuxType& mutex,
                 PtrType ptr) -> decltype(func(ptr))
{
    using MuxGuard = std::lock_guard<MuxType>;
    MuxGuard g(mutex);
    return func(ptr);
}

```

Если возвращаемый тип этой функции (`auto...->decltype(func(ptr))`) заставляет вас чесать затылок, обратитесь к разделу 1.3, в котором объясняется происходящее. Там вы узнаете, что в C++14 возвращаемый тип можно свести к простому `decltype(auto)`:

```

template<typename FuncType,
         typename MuxType,
         typename PtrType>
decltype(auto) lockAndCall(FuncType func, // C++14
                           MuxType& mutex,
                           PtrType ptr)
{
    using MuxGuard = std::lock_guard<MuxType>;
    MuxGuard g(mutex);
    return func(ptr);
}

```

Для данного шаблона `lockAndCall` (любой из версий), вызывающий код может иметь следующий вид:

```

auto result1 = lockAndCall(f1, f1m, 0);           // Ошибка!
auto result2 = lockAndCall(f2, f2m, NULL);        // Ошибка!
auto result3 = lockAndCall(f3, f3m, nullptr); // OK

```

Такой код можно написать, но, как показывают комментарии, в двух случаях из трех этот код компилироваться не будет. В первом вызове проблема в том, что когда 0 передается в `lockAndCall`, происходит вывод соответствующего типа шаблона. Типом 0 является, был и всегда будет `int`, как и тип параметра `ptr` в инстанцировании данного вызова `lockAndCall`. К сожалению, это означает, что в вызов `func` в `lockAndCall` передается `int`, а этот тип несовместим с параметром `std::shared_ptr<Widget>`, ожидаемым функцией `f1`. Значение 0, переданное в вызове `lockAndCall`, призвано представлять нулевой указатель, но на самом деле передается заурядный `int`. Попытка передать этот `int` функции `f1` как `std::shared_ptr<Widget>` представляет собой ошибку типа. Вызов `lockAndCall` с 0

оказывается неудачным, поскольку в шаблоне функции, которая требует аргумент типа `std::shared_ptr<Widget>`, передается значение `int`.

Анализ вызова с переданным `NULL` по сути такой же. Когда в функцию `lockAndCall` передается `NULL`, для параметра `ptr` выводится целочисленный тип, и происходит ошибка, когда целочисленный тип передается функции `f2`, которая ожидает аргумент типа `std::unique_ptr<Widget>`.

В противоположность первым двум вызовам вызов с `nullptr` никакими неприятностями не отличается. Когда функции `lockAndCall` передается `nullptr`, выведенным типом `ptr` является `std::nullptr_t`. При передаче `ptr` в функцию `f3` выполняется неявное преобразование `std::nullptr_t` в `Widget*`, поскольку `std::nullptr_t` неявно преобразуется во все типы указателей.

Тот факт, что вывод типа шаблона приводит к “неверным” типам для `0` и `NULL` (т.е. к их истинным типам, а не к представлению с их использованием нулевых указателей), является наиболее убедительной причиной для использования `nullptr` вместо `0` или `NULL`, когда вы хотите представить нулевой указатель. При применении `nullptr` шаблоны не представляют собой никаких особых проблем. Вместе с тем фактом, что `nullptr` не приводят к неприятностям при разрешении перегрузки, которым подвержены `0` и `NULL`, все это приводит к однозначному выводу — если вам нужен нулевой указатель, используйте `nullptr`, но не `0` и не `NULL`.

#### Следует запомнить

- Предпочитайте применение `nullptr` использованию `0` или `NULL`.
- Избегайте перегрузок с использованием целочисленных типов и типов указателей.

### 3.3. Предпочитайте объявление псевдонимов применению `typedef`

Я уверен, что мы можем сойтись на том, что применение контейнеров STL — хорошая идея, и я надеюсь, что раздел 4.1 убедит вас, что хорошей идеей является применение `std::unique_ptr`, но думаю, что ни один из вас не увлечется многократным написанием типов наподобие `std::unique_ptr<std::unordered_map<std::string, std::string>>`. Одна мысль о таких типах лично у меня вызывает все симптомы синдрома запястного канала<sup>2</sup>.

Избежать такой медицинской трагедии несложно, достаточно использовать `typedef`:

```
typedef  
std::unique_ptr<std::unordered_map<std::string, std::string>>  
UPtrMapSS;
```

<sup>2</sup> Синдром запястного канала — неврологическое заболевание, проявляющееся длительной болью и онемением пальцев рук. Широко распространено представление, что длительная ежедневная работа на компьютере, требующая постоянного использования клавиатуры, является фактором риска развития синдрома запястного канала. — Примеч. пер.

Но `typedef` слишком уж какой-то девяносто восьмой... Конечно, он работает и в C++11, но стандарт C++11 предлагает еще и *объявление псевдонима* (*alias declaration*):

```
using UPtrMapSS =
    std::unique_ptr<std::unordered_map<std::string, std::string>>;
```

С учетом того, что `typedef` и *объявление псевдонима* делают в точности одно и то же, разумно задаться вопросом "А есть ли какое-то техническое основание для того, чтобы предпочесть один способ другому?"

Да, есть, но перед тем как я его укажу, замечу, что многие программисты считают *объявление псевдонима* более простым для восприятия при работе с типами, включающими указатели на функции:

```
// FP является синонимом для указателя на функцию, принимающую
// int и const std::string& и ничего не возвращающую
typedef void (*FP)(int, const std::string&);

// То же самое, но как объявление псевдонима
using FP = void (*)(int, const std::string&);
```

Конечно, ни одна из разновидностей не оказывается существенно проще другой, а ряд программистов тратит немало времени для того, чтобы верно записать синонимы для типов указателей на функции, так что пока что убедительных причин для предпочтения *объявления псевдонима* пока что нет.

Однако убедительная причина все же существует, и называется она — шаблоны. В частности, *объявления псевдонимов* могут быть шаблонизированы (и в этом случае они называются *шаблонами псевдонимов*), в то время как `typedef` — нет. Это дает программистам на C++11 простой механизм для выражения того, что в C++98 можно было выразить только хакерскими способами, с помощью `typedef`, вложенных в шаблонные `struct`. Рассмотрим, например, определение синонима для связанного списка, который использует пользовательский распределитель памяти `MyAlloc`. В случае шаблонов псевдонимов это просто, как семечки щелкать:

```
// MyAllocList<T> является синонимом для std::list<T, MyAlloc<T>>:
template<typename T>
using MyAllocList = std::list<T, MyAlloc<T>>;
```

MyAllocList<Widget> lw; // Клиентский код

В случае `typedef` эти семечки приходится сначала долго растить:

```
// MyAllocList<T>::type — синоним для std::list<T, MyAlloc<T>>:
template<typename T>
struct MyAllocList {
    typedef std::list<T, MyAlloc<T>> type;
};

MyAllocList<Widget>::type lw; // Клиентский код
```

Все еще хуже. Если вы хотите использовать `typedef` в шаблоне для создания связанных списков, хранящего объекты типа, указанного параметром шаблона, имя, указанное в `typedef`, следует предварять ключевым словом `typename`:

```
template<typename T>
class Widget { // Widget<T> содержит
private: // MyAllocList<T>,
    typename MyAllocList<T>::type list; // как член-данные
};
```

Здесь `MyAllocList<T>::type` ссылается на тип, который зависит от параметра типа шаблона (`T`). Тем самым `MyAllocList<T>::type` является **зависимым типом** (*dependent type*), а одно из многих милых правил C++ требует, чтобы имена зависимых типов предварялись **ключевым словом `typename`**.

Если `MyAllocList` определен как шаблон псевдонима, это требование использования **ключевого слова `typename`** убирается (как и громоздкий суффикс “`::type`”):

```
template<typename T>
using MyAllocList = std::list<T, MyAlloc<T>>; // Как и ранее

template<typename T>
class Widget {
private:
    MyAllocList<T> list; // Ни typename,
                         // ни ::type
};
```

Для вас `MyAllocList<T>` (т.е. использование шаблона псевдонима) может выглядеть как **зависимый от параметра шаблона `T`**, как и `MyAllocList<T>::type` (т.е. как и использование вложенного `typedef`), но вы не компилятор. Когда компилятор обрабатывает шаблон `Widget` и встречает использование `MyAllocList<T>` (т.е. использование шаблона псевдонима), он знает, что `MyAllocList<T>` является именем типа, поскольку `MyAllocList` является шаблоном псевдонима: он **обязан** быть именем типа. Тем самым `MyAllocList<T>` оказывается **независимым типом**, и спецификатор `typename` не является ни требуемым, ни разрешенным.

С другой стороны, когда компилятор видит `MyAllocList<T>::type` (т.е. использование вложенных `typedef`) в шаблоне `Widget`, он не может знать наверняка, что эта конструкция именует тип, поскольку это может быть специализация `MyAllocList`, с которой он еще не встречался и в которой `MyAllocList<T>::type` ссылается на нечто, отличное от типа. Это звучит глупо, но не вините компиляторы за то, что они рассматривают такую возможность. В конце концов, это люди пишут такой код.

Например, некая заблудшая душа вполне в состоянии написать следующее:

```
class Wine { ... };

template<> // Специализация MyAllocList в
```

```

class MyAllocList<Wine> { // которой T представляет собой Wine
private:
    enum class WineType // См. в разделе 3.4 информацию об
    { White, Red, Rose }; // "enum class"

    WineType type; // В этом классе type представляет
                    // собой данные-член!
};

```

Как видите, `MyAllocList<Wine>::type` не является типом. Если `Widget` инстанцирован с `Wine`, `MyAllocList<T>::type` в шаблоне `Widget` представляет собой данные-член, а не тип. Ссылается ли `MyAllocList<T>::type` на тип в шаблоне `Widget`, зависит от того, чем является `T`, а потому компиляторы требуют, чтобы вы точно указывали, что это тип, предваряя его ключевым словом `typename`.

Если вы занимаетесь метапрограммированием с использованием шаблонов (template metaprogramming — TMP), то вы, скорее всего, сталкивались с необходимостью получать параметры типов шаблонов и создавать из них новые типы. Например, для некоторого заданного типа `T` вы можете захотеть удалить квалификатор `const` или квалификатор ссылки, содержащийся в `T`, например преобразовать `const std::string&` в `std::string`. Вы можете также захотеть добавить `const` к типу или преобразовать его в `lvalue`-ссылку, например, превращая `Widget` в `const Widget` или в `Widget&`. (Если вы еще не занимались TMP, это плохо, потому что, если вы действительно хотите быть эффективным программистом на C++, вы должны быть знакомы как минимум с основами этого аспекта C++. Вы можете увидеть примеры TMP в действии, включая различные преобразования типов, о которых я упоминал, в разделах 5.1 и 5.5.)

C++11 дает вам инструменты для такого рода преобразований в виде *свойств типов* (type traits), набора шаблонов в заголовочном файле `<type_traits>`. В нем вы найдете десятки свойств типов; не все из них выполняют преобразования типов, но те, которые это делают, предлагают предсказуемый интерфейс. Для заданного типа `T`, к которому вы хотели бы применить преобразование, результирующий тип имеет вид `std::преобразование<T>::type`, например:

```

std::remove_const<T>::type // Дает T из const T
std::remove_reference<T>::type // Дает T из T& и T&&
std::add_lvalue_reference<T>::type // Дает T& из T

```

Комментарии просто резюмируют, что делают эти преобразования, так что не принимайте их слишком буквально. Перед тем как использовать их в своем проекте, я настоятельно советую ознакомиться с их точной спецификацией.

В любом случае я не стремлюсь обеспечить вас учебником по свойствам типов. Вместо этого я прошу вас обратить внимание на то, что каждое преобразование завершается `::type`. Если вы применяете их к параметру типа в шаблоне (что практически всегда является их применением в реальном коде), то вы также должны предварять каждое их применение ключевым словом `typename`. Причина обоих этих синтаксических требований заключается в том, что свойства типов в C++11 реализованы как вложенные `typedef`

внутри шаблонных структур `struct`. Да, это так — они реализованы с помощью технологии, о которой я говорю, что она уступает шаблонам псевдонимов!

Тому есть исторические причины, но здесь мы их опустим (честное слово, это слишком скучно), поскольку Комитет по стандартизации с опозданием признал, что шаблоны псевдонимов оказываются лучшим способом реализации, и соответствующие шаблоны включены в C++14 для всех преобразований типов C++11. Псевдонимы имеют общий вид: для каждого преобразования C++11 `std::преобразование<T>::type` имеется соответствующий шаблон псевдонима C++14 с именем `std::преобразование_t`. Вот примеры, поясняющие, что я имею в виду:

```
std::remove_const<T>::type           // C++11: const T -> T
std::remove_const_t<T>                // Эквивалент в C++14
std::remove_reference<T>::type        // C++11: T&/T&& -> T
std::remove_reference_t<T>            // Эквивалент в C++14
std::add_lvalue_reference<T>::type    // C++11: T -> T&
std::add_lvalue_reference_t<T>         // Эквивалент в C++14
```

Конструкции C++11 остаются в силе в C++14, но я не знаю, зачем вам может захотеться их использовать. Даже если у вас нет компилятора C++14, написание таких шаблонов псевдонимов самостоятельно — детская игра. Требуются только языковые возможности C++11, и даже ребенок сможет написать такие шаблоны. Если у вас есть доступ к электронной копии стандарта C++14, то все становится еще проще — вы можете просто скопировать необходимый код оттуда и вставить в свою программу. Вот вам для начала:

```
template <class T>
using remove_const_t = typename remove_const<T>::type;

template <class T>
using remove_reference_t = typename remove_reference<T>::type;

template <class T>
using add_lvalue_reference_t =
typename add_lvalue_reference<T>::type;
```

Судите сами — что может быть проще?

### Следует запомнить

- В отличие от объявлений псевдонимов, `typedef` не поддерживает шаблонизацию.
- Шаблоны псевдонимов не требуют суффикса “`::type`”, а в шаблонах — префикса `typename`, часто требуемого при обращении к `typedef`.
- C++14 предлагает шаблоны псевдонимов для всех преобразований свойств типов C++11.

### 3.4. Предпочтайте перечисления с областью видимости перечислением без таковой

В качестве общего правила объявление имени в фигурных скобках ограничивает видимость этого имени областью видимости, определяемой этими скобками. Но не так обстоит дело с перечислениями в C++98. Имена в таких перечислениях принадлежат области видимости, содержащей `enum`, а это означает, что ничто иное в этой области видимости не должно иметь такое же имя:

```
enum Color {black, white, red}; // black, white, red находятся
                                // в той же области видимости,
                                // что и Color
auto white = false;           // Ошибка! Имя white уже объяв-
                                // лено в этой области видимости
```

Тот факт, что эти имена перечисления “вытекают” в область видимости, содержащую определение их `enum`, приводит к официальному термину для данной разновидности перечислений: *без области видимости* (*unscoped*). Их новый аналог в C++11, *перечисления с областью видимости* (*scoped enum*), не допускает такой утечки имен:

```
enum class Color
{ black, white, red }; // black, white, red принадлежат
                      // области видимости Color
auto white = false;   // OK, других white нет

Color c = white;      // Ошибка! Нет имени перечисления
                      // "white" в этой области видимости

Color c = Color::white; // OK

auto c = Color::white; // OK (и соответствует совету
                      // из раздела 2.1)
```

Поскольку `enum` с областью видимости объявляются с помощью ключевого слова `class`, о них иногда говорят как о *классах перечислений*.

Снижение загрязнения пространства имен, обеспечиваемое применением перечислений с областью видимости, само по себе является достаточной причиной для предпочтения таких перечислений их аналогам без областей видимости. Однако перечисления с областью видимости имеют и второе убедительное преимущество: они существенно строже типизированы. Значения в перечислениях без областей видимости неявно преобразуются в целочисленные типы (а оттуда — в типы с плавающей точкой). Поэтому вполне законными оказываются такие семантические карикатуры:

```
enum Color {black, white, red}; // Перечисление без
                                // области видимости
std::vector<std::size_t>        // Функция, возвращающая
primeFactors(std::size_t x);    // простые делители x
```

```

Color c = red;
...
if (c < 14.5) { // Сравнение Color и double (!)
    auto factors = // Вычисление простых делителей
        primeFactors(c); // значения Color (!)

}

```

Добавление простого ключевого слова `class` после `enum` преобразует перечисление без области видимости в перечисление с областью видимости, и это — совсем другая история. Не имеется никаких неявных преобразований элементов перечисления с областью видимости в любой другой тип:

```

enum class Color // Перечисление с областью видимости
{black, white, red};

Color c = Color::red; // Как и ранее, но с квалификатором
... // области видимости
if (c < 14.5) { // Ошибка! Нельзя сравнивать
    auto factors = // Ошибка! Нельзя передавать Color в
        primeFactors(c); // функцию, ожидающую std::size_t

}

```

Если вы хотите честно выполнить преобразование из `Color` в другой тип, сделайте то же, что вы всегда делаете для осуществления своих грязных желаний, — воспользуйтесь явным приведением типа:

```

if (static_cast<double>(c) < 14.5) { // Странный, но
    // корректный код
    auto factors = // Сомнительно, но компилируется
        primeFactors(static_cast<std::size_t>(c));
}

```

Может показаться, что перечисления с областями видимости имеют и третье преимущество перед перечислениями без областей видимости, поскольку могут быть предварительно объявлены, их имена могут быть объявлены без указания перечислителей:

```

enum Color; // Ошибка!
enum class Color; // OK

```

Это заблуждение. В C++11 перечисления без областей видимости также могут быть объявлены предварительно, но только с помощью небольшой дополнительной работы, которая вытекает из того факта, что каждое перечисление в C++ имеет целочисленный базовый тип (*underlying type*), который определяется компилятором. Для перечисления без области видимости наподобие `Color`

```
enum Color { black, white, red };
```

компилятор может выбрать в качестве базового типа `char`, так как он должен представить всего лишь три значения. Однако некоторые перечисления имеют куда больший диапазон значений, например:

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              indeterminate = 0xFFFFFFFF
};
```

Здесь должны быть представлены значения в диапазоне от 0 до 0xFFFFFFFF. За исключением необычных машин (где `char` состоит как минимум из 32 битов), компилятор выберет для предоставления значений `Status` целочисленный тип, больший, чем `char`.

Для эффективного использования памяти компиляторы часто выбирают наименьший базовый тип, которого достаточно для представления значений перечислителей. В некоторых случаях, когда компиляторы выполняют оптимизацию по скорости, а не по размеру, они могут выбрать не наименьший допустимый тип, но при этом они, определенно, захотят иметь возможность оптимизации размера. Для этого C++98 поддерживает только определения `enum` (в которых перечислены все значения); объявления `enum` не разрешены. Это позволяет компиляторам выбирать базовый тип для каждого `enum` до его использования.

Но невозможность предварительного объявления `enum` имеет свои недостатки. Наиболее важным из них, вероятно, является увеличение зависимостей при компиляции. Вновь обратимся к перечислению `Status`:

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              indeterminate = 0xFFFFFFFF
};
```

Это разновидность перечисления, которая, скорее всего, будет использоваться во всей системе, а следовательно, включенная в заголовочный файл, от которой зависит каждая из частей системы. Если добавить новое значение состояния

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              audited = 500,
              indeterminate = 0xFFFFFFFF
};
```

то, вероятно, придется перекомпилировать всю систему полностью, даже если только одна подсистема — возможно, одна-единственная функция! — использует это новое

значение. Это одна из тех вещей, которые программисты просто ненавидят. И это та вещь, которую исключает возможность предварительного объявления `enum` в C++11. Например, вот совершенно корректное объявление `enum` с областью видимости, и функции, которая получает его в качестве параметра:

```
enum class Status; // Предварительное объявление
void continueProcessing(Status s); // и его использование
```

Заголовочный файл, содержащий эти объявления, не требует перекомпиляции при просмотре определения `Status`. Кроме того, если изменено перечисление `Status` (например, добавлено значение `audited`), но поведение `continueProcessing` не изменилось (например, потому что `continueProcessing` не использует значение `audited`), то не требуется и перекомпиляция реализации `continueProcessing`.

Но если компилятор должен знать размер `enum` до использования, то как могут перечисления C++11 быть предварительно объявлены, в то время как перечисления C++98 этого не могут? Ответ прост: базовый тип перечислений с областью видимости всегда известен, а для перечислений без областей видимости вы можете его указать.

По умолчанию базовым типом для `enum` с областью видимости является `int`:

```
enum class Status; // Базовый тип – int
```

Если вас не устраивает значение по умолчанию, вы можете его перекрыть:

```
enum class Status: std::uint32_t; // Базовый тип для Status –
// std::uint32_t (из <cstdint>)
```

В любом случае компиляторы знают размер перечислителей в перечислении с областью видимости.

Чтобы указать базовый тип для перечисления без области видимости, вы делаете то же, что и для перечисления с областью видимости, и полученный результат может быть предварительно объявлен:

```
enum Color: std::uint8_t; // Предварительное объявление
// перечисления без области видимости;
// базовый тип – std::uint8_t
```

Спецификация базового типа может быть указана и в определении `enum`:

```
enum class Status: std::uint32_t { good = 0,
                                    failed = 1,
                                    incomplete = 100,
                                    corrupt = 200,
                                    audited = 500,
                                    indeterminate = 0xFFFFFFFF
};
```

С учетом того факта, что `enum` с областью видимости устраниет загрязнение пространства имен и невосприимчиво к бессмысленным преобразованиям типов, вас может удивить тот факт, что имеется как минимум одна ситуация, в которой могут быть полезны

перечисления без области видимости, а именно — при обращении к полям в кортежах C++11 `std::tuple`. Предположим, например, что у нас есть кортеж, содержащий имя, адрес электронной почты и значение репутации пользователя на сайте социальной сети:

```
using UserInfo = std::tuple<std::string, std::string, std::size_t>; // Псевдоним типа; см. раздел 3.3
```

Хотя комментарии указывают, что представляет собой каждое поле кортежа, это, вероятно, не слишком полезно, когда вы сталкиваетесь с кодом наподобие следующего в отдельном исходном файле:

```
UserInfo uInfo; // Объект с типом кортежа  
...  
auto val = std::get<1>(uInfo); // Получение значения поля 1
```

Как программисту вам приходится отслеживать множество вещей. Вы действительно должны помнить, что поле 1 соответствует адресу электронной почты пользователя? Я думаю, нет. Использование `enum` без области видимости для сопоставления имен полей с их номерами позволяет избежать необходимости перегружать память:

```
enum UserInfoFields { uiName, uiEmail, uiReputation };  
  
UserInfo uInfo; // Как и ранее  
...  
auto val = std::get<uiEmail>(uInfo); // Значение адреса
```

Все было бы гораздо сложнее без явного преобразования значений из `UserInfoFields` в тип `std::size_t`, который является типом, требующимся для `std::get`.

Соответствующий код с применением перечисления с областью видимости существенно многословнее:

```
enum class UserInfoFields { uiName, uiEmail, uiReputation };  
  
UserInfo uInfo; // Как и ранее  
...  
auto val = std::get<static_cast<std::size_t>(UserInfoFields::uiEmail)>(uInfo);
```

Эта многословность может быть сокращена с помощью функции, которая принимает перечислитель и возвращает соответствующее значение типа `std::size_t`, но это немногого сложнее. `std::get` является шаблоном, так что предоставляемое значение является аргументом шаблона ( обратите внимание на применение не круглых, а угловых скобок), так что функция, преобразующая перечислитель в значение `std::size_t`, должна давать результат во время компиляции. Как поясняется в разделе 3.9, это означает, что нам нужна функция, являющаяся `constexpr`.

Фактически это должен быть `constexpr`-шаблон функции, поскольку он должен работать с любыми перечислениями. И если мы собираемся делать такое обобщение, то должны обобщить также и возвращаемый тип. Вместо того чтобы возвращать `std::size_t`, мы должны возвращать базовый тип перечисления. Он доступен с помощью свойства типа `std::underlying_type` (о свойствах типов рассказывается в разделе 3.3). Наконец мы объявим его как `noexcept` (см. раздел 3.8), поскольку мы знаем, что он никогда не генерирует исключений. В результате мы получим шаблон функции `toUType`, который получает произвольный перечислитель и может возвращать значение как константу времени компиляции:

```
template<typename E>
constexpr typename std::underlying_type<E>::type
    toUType(E enumerator) noexcept
{
    return
        static_cast<typename
            std::underlying_type<E>::type>(enumerator);
}
```

В C++14 `toUType` можно упростить заменой `typename std::underlying_type<E>::type` более изящным `std::underlying_type_t` (см. раздел 3.3):

```
template<typename E> // C++14
constexpr std::underlying_type_t<E>
    toUType(E enumerator) noexcept
{
    return static_cast<std::underlying_type_t<E>>(enumerator);
}
```

Еще более изящный возвращаемый тип `auto` (см. раздел 1.3) также корректен в C++14:

```
template<typename E> // C++14
constexpr auto
    toUType(E enumerator) noexcept
{
    return static_cast<std::underlying_type_t<E>>(enumerator);
}
```

Независимо от того, как он написан, шаблон `toUType` позволяет нам обратиться к полю кортежа следующим образом:

```
auto val = std::get<toUType(UserInfoFields::uiEmail)>(uInfo);
```

Это все же больше, чем достаточно написать при использовании перечисления без области видимости, но зато позволяет избежать загрязнения пространства имён и непреднамеренных преобразований перечислителей. Во многих случаях вы можете решить, что набор нескольких дополнительных символов является разумной ценой за возможность избежать ловушек перечислений, появление которых восходит ко времени, когда вершиной достижений в цифровых телекоммуникациях был модем со скоростью 2400 бод.

### Следует запомнить

- Перечисления в стиле C++98 в настоящее время известны как перечисления без областей видимости.
- Перечислители перечислений с областями видимости видимы только внутри перечислений. Они преобразуются в другие типы только с помощью явных приведений.
- Как перечисления с областями видимости, так и без таковых поддерживают указание базового типа. Базовым типом по умолчанию для перечисления с областью видимости является `int`. Перечисление без области видимости базового типа по умолчанию не имеет.
- Перечисления с областями видимости могут быть предварительно объявлены. Перечисления без областей видимости могут быть предварительно объявлены, только если их объявление указывает базовый тип.

## 3.5. Предпочтайте удаленные функции закрытым неопределенным

Если вы предоставляете код другим разработчикам и хотите предотвратить вызовами некоторой функции, обычно вы просто ее не объявляете. Нет объявления функции — нечего и вызывать. Но иногда C++ объявляет функции вместо вас, и если вы хотите предотвратить вызов таких функций клиентами вашего кода, придется постараться.

Эта ситуация возникает только для “специальных функций-членов”, т.е. функций-членов, которые при необходимости C++ генерирует автоматически. В разделе 3.11 эти функции рассматриваются более подробно, а пока что мы будем беспокоиться только о копирующем конструкторе и копирующем операторе присваивания. Эта глава во многом посвящена распространенным практикам C++98, для которых есть более эффективная замена в C++11, а в C++98, когда вы хотите подавить применение функции-члена, это почти всегда копирующий конструктор, оператор присваивания или они оба.

Подход C++98 для предотвращения применения этих функций состоит в объявлении их как `private` без предоставления определений. Например, вблизи с основанием иерархии потоков ввода-вывода в стандартной библиотеке C++ находится шаблонный класс `basic_ios`. Все классы потоков наследуют (возможно, косвенно) этот класс. Копирование потоков ввода-вывода нежелательно, поскольку не совсем очевидно, что же должна делать такая операция. Объект `istream`, например, представляет поток входных значений, одни из которых могут уже быть считаны, а другие могут потенциально быть считаны позже. Если копировать такой поток, то должно ли это повлечь копирование всех считанных значений, а также значений, которые будут считаны в будущем? Простейший способ разобраться в таких вопросах — объявить их несуществующими. Именно это делает запрет на копирование потоков.

Чтобы сделать классы потоков некопируемыми, `basic_ios` в C++98 объявлен следующим образом (включая комментарии):

```
template <class charT, class traits = char_traits<charT> >
class basic_ios : public ios_base {
public:
    ...
private:
    basic_ios(const basic_ios&); // Не определен
    basic_ios& operator=(const basic_ios&); // Не определен
};
```

Объявление этих функций как `private` предотвращает их вызов клиентами. Умышленное отсутствие их определений означает, что если код, все еще имеющий к ним доступ (т.е. функции-члены или друзья класса), ими воспользуется, то компоновка (редактирование связей) будет неудачной из-за отсутствия определений функций.

В C++11 имеется лучший способ достичь по сути того же самого: воспользоваться конструкцией “`= delete`”, чтобы пометить копирующий конструктор и копирующее присваивание как *удаленные функции*. Вот та же часть `basic_ios` в C++11:

```
template <class charT, class traits = char_traits<charT> >
class basic_ios : public ios_base {
public:
    ...
    basic_ios(const basic_ios& ) = delete;
    basic_ios& operator=(const basic_ios&) = delete;
};
```

Отличие удаления этих функций от их объявления как `private` может показаться больше вопросом вкуса, чем чем-то иным, но на самом деле в это заложено больше, чем вы думаете. Удаленные функции не могут использоваться никоим образом, так что даже код функции-члена или функций, объявленных как `friend`, не будет компилироваться, если попытается копировать объекты `basic_ios`. Это существенное улучшение по сравнению с поведением C++98, где такое некорректное применение функций не диагностируется до компоновки.

По соглашению удаленные функции объявляются как `public`, а не `private`. Тому есть причина. Когда код клиента пытается использовать функцию-член, C++ проверяет доступность до проверки состояния удаленности. Когда клиентский код пытается использовать функцию, объявленную как `private`, некоторые компиляторы жалуются на то, что это закрытая функция, несмотря на то что доступность функции никак не влияет на возможность ее использования. Стоит принять это во внимание при пересмотре старого кода и замене не определенных функций-членов, объявленных как `private`, удаленными, поскольку объявление удаленных функций как `public` в общем случае приводит к более корректным сообщениям об ошибках.

Важным преимуществом удаленных функций является то, что удаленной может быть любая функция, в то время как быть `private` могут только функции-члены. Предположим, например, что у нас есть функция, не являющаяся членом, которая принимает целочисленное значение и сообщает, является ли оно “счастливым числом”:

```
bool isLucky(int number);
```

То, что C++ является наследником C, означает, что почти любой тип, который можно рассматривать как отчасти целочисленный, будет неявно преобразовываться в `int`, но некоторые компилируемые вызовы могут не иметь смысла:

```
if (isLucky('a')) ... // Является ли 'a' счастливым числом?  
if (isLucky(true)) ... // Является ли true счастливым числом?  
if (isLucky(3.5)) ... // Следует ли выполнить усечение до 3  
                      // перед проверкой на "счастливость"?
```

Если счастливые числа действительно должны быть только целыми числами, хотелось бы предотвратить такие вызовы, как показано выше.

Один из способов достичь этого — создание удаленных перегрузок для типов, которые мы хотим отфильтровывать:

```
bool isLucky(int number);      // Исходная функция  
bool isLucky(char)    = delete; // Отвергаем символы  
bool isLucky(bool)   = delete; // Отвергаем булевые значения  
bool isLucky(double) = delete; // Отвергаем double и float
```

(Комментарий у перегрузки с `double`, который гласит, что отвергаются как `double`, так и `float`, может вас удивить, но ваше удивление исчезнет, как только вы вспомните, что при выборе между преобразованием `float` в `int` и `float` в `double` C++ предпочитает преобразование в `double`. Вызов `isLucky` со значением типа `float` приведет к вызову перегрузки с типом `double`, а не `int`. Вернее, постараётся привести. Тот факт, что данная перегрузка является удаленной, не позволит скомпилировать такой вызов.)

Хотя удаленные функции использовать нельзя, они являются частью вашей программы. Как таковые они принимаются во внимание при разрешении перегрузки. Вот почему при указанных выше объявлениях нежелательные вызовы `isLucky` будут отклонены:

```
if (isLucky('a')) ... // Ошибка! Вызов удаленной функции  
if (isLucky(true)) ... // Ошибка!  
if (isLucky(3.5f)) ... // Ошибка!
```

Еще один трюк, который могут выполнять удаленные функции (а функции-члены, объявленные как `private` — нет), заключается в предотвращении использования инстанцирований шаблонов, которые должны быть запрещены. Предположим, например, что нам нужен шаблон, который работает со встроенными указателями (несмотря на совет из главы 4, “Интеллектуальные указатели”, предпочитать интеллектуальные указатели `встроенным`):

```
template<typename T>  
void processPointer(T* ptr);
```

В мире указателей есть два особых случая. Один из них — указатели `void*`, поскольку их нельзя разыменовывать, увеличивать или уменьшать и т.д. Второй — указатели `char*`, поскольку они обычно представляют указатели на C-строки, а не на отдельные символы. Эти особые случаи часто требуют особой обработки; будем считать, что в случае шаблона `processPointer` эта особая обработка заключается в том, чтобы отвергнуть вызовы

с такими типами (т.е. должно быть невозможно вызвать processPointer с указателями типа void\* или char\*).

Это легко сделать. Достаточно удалить эти инстанцирования:

```
template<>
void processPointer<void>(void* ) = delete;
```

```
template<>
void processPointer<char>(char* ) = delete;
```

Теперь, если вызов processPointer с указателями void\* или char\* является некорректным, вероятно, таковым же является и вызов с указателями const void\* или const char\*, так что эти инстанцирования обычно также следует удалить:

```
template<>
void processPointer<const void>(const void* ) = delete;
```

```
template<>
void processPointer<const char>(const char* ) = delete;
```

И если вы действительно хотите быть последовательными, то вы также удалите перегрузки const volatile void\* и const volatile char\*, а затем приступите к работе над перегрузками для указателей на другие стандартные типы символов: wchar\_t, char16\_t и char32\_t.

Интересно, что если у вас есть шаблон функции внутри класса и вы попытаетесь отключить некоторые инстанцирования, объявляя их private (в духе классического соглашения C++98), то у вас ничего не получится, потому что невозможно дать специализации шаблона функции-члена другой уровень доступа, отличный от доступа в главном шаблоне. Например, если processPointer представляет собой шаблон функции-члена в Widget и вы хотите отключить вызовы для указателей void\*, то вот как будет выглядеть (не компилируемый) подход C++98:

```
class Widget {
public:
    ...
    template<typename T>
    void processPointer(T* ptr)
    { ... }

private:
    template<> // Ошибка!
    void processPointer<void>(void* );
};
```

Проблема заключается в том, что специализации шаблона должны быть написаны в области видимости пространства имен, а не области видимости класса. Эта проблема не возникает для удаленных функций, поскольку они не нуждаются в другом уровне доступа. Они могут быть удалены за пределами класса (а следовательно, в области видимости пространства имен):

```
class Widget {  
public:  
    ...  
    template<typename T>  
    void processPointer(T* ptr)  
    { ... }  
  
};  
  
template<> // Все еще public, но удаленная  
void Widget::processPointer<void>(void*) = delete;
```

Истина заключается в том, что практика C++98 объявления функций `private` без их определения в действительности была попыткой добиться того, для чего на самом деле созданы удаленные функции C++11. Будучи всего лишь имитацией, подход C++98 работает не так хорошо, как вещь, которую он имитирует. Он не работает вне классов, не всегда работает внутри классов, а когда работает, то может не работать до компоновки. Так что лучше придерживаться удаленных функций.

#### Следует запомнить

- Предпочитайте удаленные функции закрытым функциям без определений.
- Удаленной может быть любая функция, включая функции, не являющиеся членами, и инстанцирования шаблонов.

## 3.6. Объявляйте перекрывающие функции как `override`

Мир объектно-ориентированного программирования C++ вращается вокруг классов, наследования и виртуальных функций. Среди наиболее фундаментальных идей этого мира — та, что реализации виртуальных функций в производных классах *перекрывают* (`override`) реализации их коллег в базовых классах. Понимание того, насколько легко все может пойти наперекосяк при перекрытии функций, попросту обескураживает. Полное ощущение, что эта часть языка создавалась как иллюстрация к законам Мерфи.

Очень часто термин “перекрытие” путают с термином “перегрузка”, хотя они совершенно не связаны друг с другом. Поэтому позвольте мне пояснить, что перекрытие виртуальной функции — это то, что делает возможным вызов функции производного класса через интерфейс базового класса:

```
class Base {  
public:  
    virtual void doWork();           // Виртуальная функция  
                                    // базового класса  
};  
  
class Derived: public Base {
```

```

public:
    virtual void doWork();           // Перекрывает Base::doWork
                                    // Ключевое слово virtual
};

std::unique_ptr<Base> upb =      // Указатель на базовый класс
    std::make_unique<Derived>(); // указывает на объект
                                // производного класса;
                                // см. std::make_unique
                                // в разделе 4.4

upb->doWork();                // Вызов doWork через указатель
                                // на базовый класс; вызывается
                                // функция производного класса

```

Для осуществления перекрытия требуется выполнение нескольких условий.

- Функция базового класса должна быть виртуальной.
- Имена функций в базовом и производном классах должны быть одинаковыми (за исключением деструктора).
- Типы параметров функций в базовом и производном классах должны быть одинаковыми.
- Константность функций в базовом и производном классах должна совпадать.
- Возвращаемые типы и спецификации исключений функций в базовом и производном классах должны быть совместимыми.

К этим ограничениям, являющимся частью C++98, C++11 добавляет еще одно.

- *Ссылочные квалификаторы* функций должны быть идентичными. Ссылочные квалификаторы функции-члена являются одной из менее известных возможностей C++11, так что не удивляйтесь, если вы до сих пор ничего о них не слышали. Они позволяют ограничить использование функции-члена только объектами *lvalue* или только объектами *rvalue*. Для использования этих квалификаторов функции-члены не обязаны быть виртуальными.

```

class Widget {
public:
    ...
    void doWork() &;   // Эта версия doWork применима, только
                      // если *this представляет собой lvalue
    void doWork() &&; // Эта версия doWork применима, только
                      // если *this представляет собой rvalue
};

...
Widget makeWidget(); // Фабричная функция (возвращает rvalue)

```

```

Widget w;           // Обычный объект (lvalue)
...
w.doWork();        // Вызов Widget::doWork для lvalue
                   // (т.е. Widget::doWork &)
makeWidget().doWork(); // Вызов Widget::doWork для rvalue
                   // (т.е. Widget::doWork &&)

```

Позже я расскажу побольше о функциях-членах со ссылочными квалификаторами, а пока что просто заметим, что если виртуальная функция в базовом классе имеет ссылочный квалифиликатор, то производный класс, перекрывающий эту функцию, должен иметь тот же ссылочный квалифиликатор. Если это не так, объявленные функции все еще остаются в производном классе, но они ничего не перекрывают в базовом классе.

Все эти требования к перекрытию означают, что маленькие ошибки могут привести к большим последствиям. Код, содержащий ошибки перекрытия, обычно корректен, но делает совсем те то, что хотел программист. Поэтому в данном случае нельзя полагаться на уведомления компиляторов о том, что вы что-то делаете неверно. Например, приведенный далее код является абсолютно законным и, на первый взгляд, выглядит разумным, но в нем нет перекрытия виртуальной функции — нет ни одной функции производного класса, связанной с функцией базового класса. Сможете ли вы самостоятельно определить, в чем заключается проблема в каждом конкретном случае (т.е. почему каждая функция производного класса не переопределяет функцию базового класса с тем же именем)?

```

class Base {
public:
    virtual void mf1() const;
    virtual void mf2(int x);
    virtual void mf3() &;
    void mf4() const;
};

class Derived: public Base {
public:
    virtual void mf1();
    virtual void mf2(unsigned int x);
    virtual void mf3() &&;
    void mf4() const;
};

```

Нужна помощь?

- **mf1 объявлена как const в классе Base, но в классе Derived этого модификатора нет.**
- **mf2 получает аргумент типа int в классе Base, но в классе Derived она получает аргумент типа unsigned int.**

- `mf3` определена с квалификатором `lvalue` в классе `Base` и с квалификатором `rvalue` в классе `Derived`.
- `mf4` не объявлена в классе `Base` как `virtual`.

Вы можете подумать “На практике все это вызовет предупреждения компилятора, так что мне не о чем беспокоиться”. Может быть, это и так. А может быть, и не так. В двух из проверенных мною компиляторах код был принят без единой жалобы — в режиме, когда все предупреждения были включены. (Другие компиляторы выдавали предупреждения о некоторых из проблем, но не обо всех одновременно.)

Поскольку очень важно правильно объявить производный класс перекрывающим, но при этом очень легко ошибиться, C++11 дает вам возможность явно указать, что функция производного класса предназначена для того, чтобы перекрывать функцию из базового класса: ее можно объявить как `override`. Применяя это ключевое слово к приведенному выше примеру, мы получим следующий производный класс:

```
class Derived: public Base {  
public:  
    virtual void mf1() override;  
    virtual void mf2(unsigned int x) override;  
    virtual void mf3() & override;  
    virtual void mf4() const override;  
};
```

Этот код компилироваться не будет, поскольку теперь компиляторы знают о том, что эти функции предназначены для перекрытия функций из базового класса, а потому могут определить наличие описанных нами проблем. Это именно то, что нам надо, и потому мы должны объявлять все свои перекрывающие функции как `override`.

Код с использованием `override`, который будет успешно скомпилирован, выглядит следующим образом (в предположении, что нашей целью является перекрытие функциями в классе `Derived` всех виртуальных функций в классе `Base`):

```
class Base {  
public:  
    virtual void mf1() const;  
    virtual void mf2(int x);  
    virtual void mf3() &;  
    virtual void mf4() const;  
};  
  
class Derived: public Base {  
public:  
    virtual void mf1() const override;  
    virtual void mf2(int x) override;  
    virtual void mf3() & override;  
    void mf4() const override; // Слово virtual не мешает, но  
                           // и не является обязательным  
};
```

Обратите внимание, что в этом примере часть работы состояла в том, чтобы объявить `mf4` в классе `Base` как виртуальную функцию. Большинство ошибок, связанных с перекрытием, совершаются в производном классе, но можно сделать такую ошибку и в базовом классе.

Стратегия использования ключевого слова `override` во всех перекрытиях производного класса способна на большее, чем просто позволить компиляторам сообщать, когда функции, которые должны быть перекрытыми, ничего не перекрывают. Они также могут помочь вам оценить последствия предполагаемого изменения сигнатуры виртуальной функции в базовом классе. Если производные классы везде используют `override`, вы можете просто изменить сигнатуру и перекомпилировать систему. Вы увидите, какие повреждения нанесли своей системе (т.е. сколько классов перестали компилироваться), и после этого сможете принять решение, стоит ли изменение сигнатуры таких хлопот. Без `override` вы должны были бы надеяться на наличие достаточно всеобъемлющих тестов, поскольку, как мы видели, виртуальные функции, которые предназначены перекрывать функции базового класса, но не делают этого, не приводят ни к какой диагностике со стороны компилятора.

В C++ всегда имелись ключевые слова, но C++11 вводит два контекстных *ключевых слов* (*contextual keywords*) — `override` и `final`<sup>1</sup>. Эти ключевые слова являются зарезервированными, но только в некоторых контекстах. В случае `override` оно имеет зарезервированное значение только тогда, когда находится в конце объявления функции-члена. Это значит, что если у вас есть старый код, который уже использовал имя `override`, его не надо изменять при переходе к C++11:

```
class Warning {           // Потенциально старый класс из C++98
public:
    ...
    void override(); // Корректно как в C++98, так и в C++11
                      // (с тем же смыслом)
};
```

Это все, что следует сказать об `override`, но это не все, что следует сказать о ссылочных квалифициаторах функций-членов. Я обещал, что поговорю о них позже, и вот сейчас как раз и настало это “позже”.

Если мы хотим написать функцию, которая принимает только аргументы, являющиеся `lvalue`, мы объявляем параметр, который представляет собой неконстантную `lvalue`-ссылку:

```
void doSomething(Widget& w); // Принимает только lvalue Widget
```

Если же мы хотим написать функцию, которая принимает только аргументы, являющиеся `rvalue`, мы объявляем параметр, который представляет собой `rvalue`-ссылку:

```
void doSomething(Widget&& w); // Принимает только rvalue Widget
```

<sup>1</sup> Применение ключевого слова `final` к виртуальной функции препятствует перекрытию этой функции в производном классе. Ключевое слово `final` также может быть применено к классу; в этом случае класс становится неприменимым в качестве базового.

Ссылочные квалификаторы функции-члена позволяют проводить такое же различие для объектов, функции-члены которых вызываются, т.е. `*this`. Это точный аналог модификатора `const` в конце объявления функции-члена, который указывает, что объект, для которого вызывается данная функция-член (т.е. `*this`), является `const`.

Необходимость в функциях-членах со ссылочными квалификаторами нужна не так уж часто, но может и возникнуть. Предположим, например, что наш класс `Widget` имеет член-данные `std::vector`, и мы предлагаем функцию доступа, которая обеспечивает клиентам к нему непосредственный доступ:

```
class Widget {  
public:  
    using DataType = std::vector<double>; // См. информацию о  
                                         // using в разделе 3.3  
  
    DataType& data() { return values; }  
    ...  
private:  
    DataType values;  
};
```

Вряд ли это наиболее инкапсулированный дизайн, который видел свет, но оставим этот вопрос в стороне и рассмотрим, что происходит в следующем клиентском коде:

```
Widget w;  
...  
auto vals1 = w.data(); // Копирует w.values в vals1
```

Возвращаемый тип `Widget::data` представляет собой lvalue-ссылку (чтобы быть точным — `std::vector<double>&`), а поскольку lvalue-ссылки представляют собой lvalue, мы инициализируем `vals1` из lvalue. Таким образом, `vals1` создается копирующим конструктором из `w.values`, как и утверждает комментарий.

Теперь предположим, что у нас имеется фабричная функция, которая создает `Widget`:

```
Widget makeWidget();
```

и мы хотим инициализировать переменную с помощью `std::vector` в `Widget`, возвращенном из `makeWidget`:

```
auto vals2 = makeWidget().data(); // Копирование значений в  
                                // Widget в vals2
```

И вновь `Widget::data` возвращает lvalue-ссылку, и вновь lvalue-ссылка представляет собой lvalue, так что наш новый объект (`vals2`) опять является копией, построенной из `values` в объекте `Widget`. Однако в этот раз `Widget` представляет собой временный объект, возвращенный из `makeWidget` (т.е. представляет собой rvalue), так что копирование в него `std::vector` представляет собой напрасную трату времени. Предпочтительнее выполнить перемещение, но, поскольку `data` возвращается как lvalue-ссылка, правила C++ требуют, чтобы компиляторы генерировали код для копирования. (Имеется некоторый маневр

для оптимизации на основе правила “как если бы”<sup>4</sup>, но было бы глупо полагаться та то, что ваш компилятор найдет способ им воспользоваться.)

Нам необходим способ указать, что, когда `data` вызывается для `Widget`, являющегося `rvalue`, результат также будет представлять собой `rvalue`. Использование ссылочных квалификаторов для перегрузки `data` для `Widget`, являющихся `lvalue` и `rvalue`, делает это возможным:

```
class Widget {  
public:  
    using DataType = std::vector<double>;  
  
    DataType& data() &           // Для lvalue Widget,  
    { return values; }          // возвращает lvalue  
    DataType&& data() &&        // Для rvalue Widget,  
    { return std::move(values); } // возвращает rvalue  
    ...  
private:  
    DataType values;  
};
```

Обратите внимание на разные возвращаемые типы перегрузок `data`. Перегрузка для `lvalue`-ссылки возвращает `lvalue`-ссылку (т.е. `lvalue`), а перегрузка для `rvalue`-ссылки возвращает `rvalue`-ссылку (которая, как возвращаемый тип функции, является `rvalue`). Это означает, что клиентский код ведет теперь себя так, как мы и хотели:

```
auto vals1 = w.data();           // Вызывает lvalue-перегрузку  
                                // Widget::data, vals1  
                                // создается копированием  
auto vals2 = makeWidget().data(); // Вызывает rvalue-перегрузку  
                                // Widget::data, vals2  
                                // создается перемещением
```

Это, конечно, хорошо, но не позволяйте теплому сиянию этого хэппи-энда отвлечь вас от истинной цели этого раздела. Эта цель в том, чтобы убедить вас, что всякий раз, когда вы объявляете в производном классе функцию, предназначенную для перекрытия виртуальной функции базового класса, вы не забывали делать это с использованием ключевого слова `override`.

Кстати, если функция-член использует ссылочный квалификатор, все перегрузки этой функции также должны использовать его. Это связано с тем, что перегрузки без этих квалификаторов могут вызываться как для объектов `lvalue`, так и для объектов `rvalue`. Такие перегрузки будут конкурировать с перегрузками, имеющими ссылочные квалифиликаторы, так что все вызовы функции будут неоднозначными.

<sup>4</sup> “As if rule” — правило, согласно которому разрешены любые преобразования кода, не изменяющие наблюдаемое поведение программы. — Примеч. ред.

### Следует запомнить

- Объявляйте перекрывающие функции как `override`.
- Ссыластные квалификаторы функции-члена позволяют по-разному рассматривать `lvalue`- и `rvalue`-объекты (`*this`).

## 3.7. Предпочтайте итераторы `const_iterator` итераторам `iterator`

Итераторы `const_iterator` представляют собой STL-эквивалент указателя на `const`. Они указывают на значения, которые не могут быть изменены. Стандартная практика применения `const` там, где это только возможно, требует применения `const_iterator` везде, где нужен итератор, но не требуется изменять то, на что этот итератор указывает.

Это верно как для C++98, так и для C++11, но в C++98 поддержка `const_iterator` носит половинчатый характер. Такие итераторы не так легко создавать, а если у вас уже имеется такой итератор, его использование весьма ограничено. Предположим, например, что вы хотите выполнить в `std::vector<int>` поиск первого встречающегося значения 1983 (год, когда название языка программирования “C с классами” сменилось на C++), а затем вставить в это место значение 1998 (год принятия первого ISO-стандарты C++). Если в векторе нет значения 1983, вставка выполняется в конец вектора. При использовании итераторов `iterator` в C++98 сделать описанное просто:

```
std::vector<int> values;

std::vector<int>::iterator it =
    std::find(values.begin(), values.end(), 1983);
values.insert(it, 1998);
```

Но `iterator` — в данном случае не совсем верный выбор, поскольку этот код не изменяет объект, на который указывает `iterator`. Переделка кода для использования `const_iterator` должна быть тривиальной задачей... но не в C++98. Вот один из подходов, концептуально надежный, но все еще не совсем корректный:

```
typedef std::vector<int>::iterator IterT;
std::vector<int>::const_iterator ConstIterT;

std::vector<int> values;

ConstIterT ci =
    std::find(static_cast<ConstIterT>(values.begin()),
              static_cast<ConstIterT>(values.end()),
              1983);
```

```
values.insert(static_cast<IterT>(ci), // Может не
1998); // компилироваться!
```

Конструкции `typedef` применять, конечно, необязательно, но они облегчают написание приведений. (Если вы удивляетесь, почему я использую `typedef`, а не следую собственному совету из раздела 3.3 и не применяю объявлений псевдонимов, то я напомню, что в этом примере демонстрируется код C++98, а объявления псевдонимов — новая возможность в C++11.)

Приведения в вызове `std::find` присутствуют потому, что `values` является неконстантным контейнером, а в C++98 нет простого способа получить константный итератор из неконстантного контейнера. Приведения не являются строго необходимыми, так как можно получить `const_iterator` другими способами (например, вы можете связать `values` с переменной, являющейся ссылкой на константный объект, а затем использовать ее в своем коде вместо `values`), но к какому бы способу вы ни прибегли, процесс получения `const_iterator`, указывающего на элементы неконстантного контейнера, включает определенное количество “кривизны”.

После того как вы получаете `const_iterator`, ситуация ничуть не улучшается, поскольку в C++98 местоположения для вставки (и удаления) могут указывать только неконстантные итераторы `iterator`. Итераторы `const_iterator` для этого неприменимы. Вот почему в приведенном выше коде я выполняю приведение `const_iterator` (который я с таким трудом получил из `std::find`) в `iterator`: передача `const_iterator` в функцию `insert` не будет компилироваться.

Честно говоря, показанный мной код может не скомпилироваться, поскольку не существует переносимого преобразования `const_iterator` в `iterator`, даже с помощью `static_cast`. Может не сработать даже семантическая кувалда `reinterpret_cast`. (Это не ограничение C++98. Это справедливо и для C++11. `const_iterator` просто не преобразуется в `iterator` — неважно, насколько простым кажется такое преобразование.) Есть несколько переносимых способов генерировать `iterator`, который указывает на то же, на что и `const_iterator`, но они не очевидны, не универсальны и не стоят того, чтобы рассматривать их в данной книге. Кроме того, я надеюсь, что теперь понятна моя позиция: `const_iterator` причиняли так много неприятностей в C++98, что редко стоили того, чтобы о них беспокоиться и их использовать. По большому счету программисты всегда использовали `const` не где это только возможно, а только там, где это *практично*, а в C++98 `const_iterator` особо практическим не является.

Все изменилось с появлением C++11. Теперь `const_iterator` легко получить и легко использовать. Функции-члены контейнера `cbegin` и `end` возвращают `const_iterator` даже для неконстантных контейнеров, а функции-члены STL, которые применяют итераторы для указания позиций (например, `insert` и `erase`), в действительности используют итераторы `const_iterator`. Переделка кода C++98, который использовал `iterator`, в код с `const_iterator` в C++11 воистину тривиальна:

```
std::vector<int> values; // Как и ранее
...
auto it =
    std::find(values.cbegin(), // Используем cbegin
```

```
values.cend(), 1983); // и cend  
values.insert(it, 1998);
```

Теперь код, использующий `const_iterator`, стал действительно практическим!

Почти единственной ситуацией, в которой поддержка C++11 для `const_iterator` оказывается недостаточной, является ситуация, когда вы хотите написать максимально обобщенный библиотечный код. Такой код принимает во внимание то, что некоторые контейнеры и контейнерообразные структуры данных предоставляют функции `begin` и `end` (а также `cbegin`, `cend`, `rbegin` и т.д.) как функции, *не являющиеся* членами. Это происходит, например, в случае встроенных массивов или при использовании некоторых библиотек сторонних производителей с интерфейсами, состоящими только из свободных функций. Максимально обобщенный код использует функции, *не являющиеся* членами, а не предполагает наличие функций-членов.

Например, мы можем обобщить код, с которым только что работали, в шаблон `findAndInsert` следующим образом:

```
template<typename C, typename V>  
void findAndInsert(C& container,           // В container находится  
                    const V& targetVal, // первое значение  
                    const V& insertVal) // targetVal, затем  
{                                         // вставляет insertVal  
    using std::cbegin;  
    using std::cend;  
  
    auto it = std::find(cbegin(container), // Не член cbegin  
                        cend(container), // Не член cend  
                        targetVal);  
  
    container.insert(it, insertVal);  
}
```

Этот код прекрасно работает в C++14, но, к сожалению, не в C++11. Из-за недосмотра в процессе стандартизации в C++11 добавлены *не являющиеся* членами функции `begin` и `end`, но не были добавлены `cbegin`, `cend`, `rbegin`, `rend`, `crbegin` и `crend`. C++14 исправляет это упущение.

Если вы используете C++11, хотите написать максимально обобщенный код и ни одна из используемых вами библиотек не предоставляет отсутствующие шаблоны для `cbegin` и подобных функций, *не являющихся* шаблонами, можете легко написать собственные реализации. Например, вот как выглядит реализация функции `cbegin`, *не являющейся* членом:

```
template <class C>  
auto cbegin(const C& container)->decltype(std::begin(container))  
{  
    return std::begin(container); // См. пояснения ниже  
}
```

Вы удивлены тем, что эта функция `cbegin` не вызывает функцию-член `begin`, *не так ли?* Я тоже был удивлен. Но давайте подумаем логически. Этот шаблон `cbegin` принимает

аргументы любого типа, представляющего контейнерообразную структуру данных C, и обращается к этому аргументу через ссылку на константный объект `container`. Если C — обычный тип контейнера (например, `std::vector<int>`), то `container` будет ссылкой на const-версию этого контейнера (например, `const std::vector<int>&`). Вызов функции `begin`, не являющейся членом (и предоставленной C++11), для константного контейнера дает константный итератор `const_iterator`, и именно этот итератор и возвращает наш шаблон. Преимущества такой реализации в том, что она работает даже для контейнеров, которые предоставляют функцию-член `begin` (которую функция C++11 `begin`, не являющаяся членом, вызывает для контейнеров), но не имеют функции-члена `cbegin`. Таким образом, вы можете использовать эту свободную функцию `cbegin` с контейнерами, поддерживающими единственную функцию-член `begin`.

Этот шаблон работает и в случае, когда C представляет собой встроенный массив. При этом `container` становится ссылкой на константный массив. C++11 предоставляет для массивов специализированную версию функции `begin`, не являющейся членом, которая возвращает указатель на первый элемент массива. Элементы константного массива являются константами, так что указатель, который возвращает свободная функция `begin`, возвращает для константного массива указатель на константу, который фактически и является `const_iterator` для массива. (Для понимания, как специализировать шаблон для встроенных массивов, обратитесь к обсуждению в разделе 1.1 вывода типов в шаблонах, которые получают в качестве параметров ссылки на массивы.)

Но вернемся к основам. Суть этого раздела заключается в том, чтобы призывать вас использовать `const_iterator` везде, где возможно. Фундаментальный мотив для этого — применение `const` всегда, когда это имеет смысл — существовал и до C++11, но в C++98 этот принцип при работе с итераторами был непрактичным. В C++11 это сугубо практический совет, а в C++14 к тому же “доведены до ума” некоторые мелочи, остававшиеся незавершенными в C++11.

#### Следует запомнить

- Предпочитайте использовать `const_iterator` вместо `iterator` там, где это можно.
- В максимально обобщенном коде предпочтительно использовать версии функций `begin`, `end`, `rbegin` и прочих, не являющиеся членами.

## 3.8. Если функции не генерируют исключений, объявляйте их как noexcept

В C++98 спецификации исключений были довольно темпераментными созданиями. От вас требовалось собрать информацию обо всех типах исключений, которые могла генерировать функция, так что при изменении реализации функции могла потребовать изменения и спецификация исключений. Изменение спецификации исключения могло нарушить клиентский код, так как вызывающий код мог зависеть от исходной спецификации исключений. Компиляторы обычно не предлагали никакой помощи в поддержании согласованности между реализациями функций, спецификациями исключений

и клиентским кодом. Большинство программистов в конечном итоге сочло, что спецификации исключений в C++98 не стоят затрачиваемых на них усилий и ими лучше не пользоваться вовсе. Во время работы над C++11 было достигнуто согласие, что действительно важная информация о поведении функций в смысле исключений — это информация, может ли вообще такое исключение быть сгенерировано. Либо функция может генерировать исключение, либо гарантируется, что это невозможно. Именно эта дилемма лежит в основе спецификаций исключений C++11, которые, по сути, заменили спецификации исключений C++98. (Спецификации исключений в стиле C++98 остаются корректными, но не рекомендуются к употреблению.) В C++11 безусловный модификатор `noexcept` применяется к функциям, которые гарантированно не могут генерировать исключения.

Должна ли функция быть объявлена таким образом — вопрос проектирования интерфейса. Поведение генерирующих исключения функций представляет большой интерес для клиентов. Вызывающий код может запросить статус `noexcept` функции, и результат этого запроса может повлиять на безопасность в смысле исключений или эффективность вызывающего кода. Таким образом, является ли функция объявлена как `noexcept`, представляет собой столь же важную часть информации, как и является ли функция-член объявленной как `const`. Отсутствие объявления функции как `noexcept`, когда вы точно знаете, что она не в состоянии генерировать исключения, — не более чем просто плохая спецификация интерфейса.

Но есть и дополнительный стимул для применения `noexcept` к функциям, которые не генерируют исключений: это позволяет компиляторам генерировать лучший объектный код. Чтобы понять, почему это так, рассмотрим разницу между способами, которыми C++98 и C++11 сообщают о том, что функция не может генерировать исключения. Пусть имеется функция `f`, которая обещает вызывающему коду, что тот никогда не получит исключения. Вот как выглядят два способа это выразить:

```
int f(int x) throw(); // f не генерирует исключений: C++98  
int f(int x) noexcept; // f не генерирует исключений: C++11
```

Если во время выполнения некоторое исключение покинет `f`, тем самым будет нарушена спецификация исключений `f`. При спецификации исключений C++98 стек вызовов сворачивается<sup>5</sup> до вызывающего `f` кода, и после некоторых действий, не имеющих значения для данного рассмотрения, выполнение программы прекращается. При спецификации исключений C++11 поведение времени выполнения несколько иное: стек только, возможно, сворачивается перед завершением выполнения программы.

<sup>5</sup> Обычно в русскоязычной литературе для термина *stack unwinding* используется перевод “разворачивание стека”. Однако это не совсем верный перевод. Вот что в переписке с редактором книги пишет по этому поводу профессор университета Иннополис Е. Зуев: “Самый частый вариант перевода — «раскрутка стека» — не просто заменяет существо дела, но просто-таки противоположен ему. При срабатывании исключения начинается процесс поиска в стеке секции (“кадра стека”, *stack frame*), для которой задан *перехват* случившегося исключения. В тексте исходной программы такой секции соответствует *try*-блок с *catch*-обработчиком, в котором задано имя случившегося исключения. И в процессе этого поиска все секции стека, для которых такой *перехват* не задан, из стека *удаляются*. Говорят еще, что производится поиск секции по всей динамической цепочке вызовов. Тем самым стек в целом *сокращается, сворачивается*. Таким образом, самый адекватный вариант перевода — *сворачивание стека*”. Поэтому принято решение переводить *stack unwinding* как *сворачивание стека*. — Примеч. ред.

Разница между сворачиванием стека и возможным сворачиванием оказывает на удивление большое влияние на генерацию кода. В случае функции, объявленной как noexcept, оптимизаторам не надо ни поддерживать стек в сворачиваемом состоянии, ни гарантировать, что объекты в такой функции будут уничтожены в порядке, обратном созданию, если вдруг такую функцию покинет исключение. Функции со спецификацией throw() не имеют такой гибкости оптимизации, как и функции без спецификаций вообще. Ситуацию можно резюмировать следующим образом:

```
RetType function(params) noexcept; // Наиболее оптимизируема  
RetType function(params) throw(); // Менее оптимизируема  
RetType function(params); // Менее оптимизируема
```

Этого одного достаточно для того, чтобы объявлять функции, о которых точно известно, что они не генерируют исключений, как noexcept.

Для некоторых функций все оказывается еще более интересным. Выдающимся примером являются операции перемещения. Предположим, что у вас имеется код C++98, использующий std::vector<Widget>. Объекты типа Widget время от времени добавляются в std::vector с помощью функции push\_back:

```
std::vector<Widget> vw;  
...  
Widget w;  
... // Работа с w  
vw.push_back(w); // Добавление w к vw
```

Предположим, что этот код отлично работает, и нет никакой необходимости изменять его для C++11. Однако вы хотите воспользоваться тем фактом, что семантика перемещения C++11 может улучшить производительность старого кода при участии типов, допускающих перемещающие операции. Вы уверены, что класс Widget такие операции имеет — либо потому, что вы написали их самостоятельно, либо потому, что вы убедились в осуществлении условий для их автоматической генерации (см. раздел 3.11).

Когда в std::vector добавляется новый элемент, может оказаться, что в std::vector для него не хватает места, т.е. что размер std::vector равен его емкости. Когда такое случается, std::vector выделяет новый, больший по размеру блок памяти для хранения своих элементов и переносит элементы из старого блока в новый. В C++98 перенос осуществляется с помощью копирования каждого элемента из старой памяти в новую с последующим удалением объекта в старой памяти. Этот подход позволяет push\_back обеспечить строгую гарантию безопасности исключений: если исключение будет сгенерировано в процессе копирования элементов, то состояние std::vector останется неизменным, поскольку ни один элемент в старом блоке памяти не будет удален, пока все элементы не будут успешно скопированы в новое место в памяти.

В C++11 естественной оптимизацией была бы замена копирования элементов std::vector перемещениями. К сожалению, это ведет к риску нарушения строгой гарантии push\_back. Если n элементов перемещены из старого блока памяти в новый и при перемещении n+1-го элемента генерируется исключение, операция push\_back не может

быть завершена. Но при этом исходный `std::vector` находится в измененном состоянии: `n` его элементов уже перемещены. Восстановление их исходных состояний может оказаться невозможным, поскольку попытка перемещения каждого объекта обратно в исходное местоположение в памяти также может привести к генерации исключений.

Это серьезная проблема, поскольку поведение старого кода может зависеть от строгой гарантии безопасности функции `push_back`. Следовательно, реализации C++11 не могут молча заменить операции копирования внутри `push_back` операциями перемещения, если только точно не известно, что операции перемещения не генерируют исключений. В таком случае замена копирований перемещениями должна быть безопасной, и единственным побочным эффектом этой замены будет повышение быстродействия.

Функция `std::vector::push_back` использует преимущество этой стратегии (“перемести, если можешь, но копирай, если должен”), и это не единственная функция стандартной библиотеки, поступающая таким образом. Другие функции, обеспечивающие строгую гарантию безопасности исключений в C++98 (например, `std::vector::reserve`, `std::deque::insert` и др.), ведут себя точно так же. Все эти функции заменяют вызовы копирующих операций в C++98 вызовами перемещающих операций в C++11, только если известно, что перемещающие операции не генерируют исключений. Но как функция может узнать, генерирует ли исключения операция перемещения? Ответ очевиден: она должна проверить, объявлена ли операция как `noexcept`<sup>6</sup>.

Функции `swap` являются еще одним случаем, когда модификатор `noexcept` особенно желателен. Функция `swap` является ключевым компонентом множества реализаций алгоритмов STL и обычно использует операторы копирующего присваивания. Ее широкое применение делает оптимизацию на основе `noexcept` особенно важной. Интересно, что то, является ли функция `swap` в стандартной библиотеке `noexcept`, иногда зависит от того, являются ли таковыми функции `swap`, определенные пользователями. Например, объявления `swap` в стандартной библиотеке для массивов и `std::pair` имеют следующий вид:

```
template <class T, size_t N>
void swap(T (&a)[N],                                     // См. ниже
          T (&b)[N]) noexcept(noexcept(swap(*a, *b)));
```

```
template <class T1, class T2>
struct pair {
    ...
    void swap(pair& p) noexcept(noexcept(swap(first, p.first)) &&
                                noexcept(swap(second, p.second)));
    ...
};
```

<sup>6</sup> Проверка обычно выполняется окольным путем. Функции наподобие `std::vector::push_back` вызывают шаблон `std::move_if_noexcept`, вариацию шаблона `std::move`, который условно выполняет приведение к `rvalue` (см. раздел 5.1), в зависимости от того, объявлен ли перемещающий конструктор как `noexcept`. В свою очередь, `std::move_if_noexcept` консультируется с `std::is_nothrow_move_constructible`, а значение этого свойства типа (см. раздел 3.3) устанавливается компиляторами в зависимости от того, объявлен ли перемещающий конструктор как `noexcept` (или `throw()`).

Эти функции являются условно поэксперт: являются ли они поэксперт-функциями, зависит от того, являются ли таковыми инструкции в конструкции поэксперт. Для двух заданных массивов Widget, например, их обмен с помощью функции swap будет считаться поэксперт только в том случае, если таковыми будут операции обмена отдельных элементов; т.е. если функция swap для Widget объявлена как поэксперт. Таким образом, автор функции swap класса Widget определяет, будет ли операция обмена массивов Widget рассматриваться как поэксперт. Это, в свою очередь, определяет, будут ли таковыми другие операции обмена, такие как swap для массива массивов Widget. Аналогично, является ли операция swap двух объектов std::pair, содержащих Widget, не генерирующей исключений, зависит от того, является ли таковой операция swap для Widget. Тот факт, что обмен высокоуровневых структур данных в общем случае может быть поэксперт, только если таковым является обмен их более низкоуровневых составляющих, должен мотивировать вас объявлять функции swap как поэксперт везде, где только это возможно.

Я надеюсь, вы достаточно заинтересовались возможностями оптимизации, которые предоставляет поэксперт. Увы, должен умерить ваш энтузиазм. Оптимизация важна, но корректность важнее. В начале этого раздела я отмечал, что поэксперт является частью интерфейса функции, так что вы должны объявлять функцию как поэксперт, только если вы готовы обеспечивать это свойство в течение длительного срока. Если вы объявили функцию как поэксперт, а позже измениете ваше решение, то ваши перспективы окажутся невеселыми. Удаление поэксперт из объявления функции (т.е. изменение ее интерфейса) ведет к риску нарушения клиентского кода. Можно также изменить реализацию так, что генерация исключений будет возможна, не меняя при этом (теперь уже некорректную) спецификацию исключений. В этом случае, если исключение попытается покинуть вашу функцию, программа завершит работу. Вы можете также подчиниться существующей реализации, забыв о своем желании ее изменить. Ни один из перечисленных вариантов привлекательным не выглядит.

Дело в том, что большинство функций *нейтральны по отношению к исключениям*. Такие функции сами по себе исключений не генерируют, но это могут делать функции, вызываемые ими. Когда такое происходит, нейтральная функция позволяет сгенерированному исключению пройти дальше, к обработчику, находящемуся выше по цепочке вызовов. Нейтральные по отношению к исключениям функции никогда не являются поэксперт, поскольку их могут покидать такие “проходящие” исключения. Поэтому большинство функций совершенно корректно не используют спецификацию поэксперт.

Некоторые функции, однако, имеют естественные реализации, которые не генерируют никаких исключений, а еще некоторое их количество (в основном операции перемещения и обмена), будучи поэксперт, могут дать столь значительный выигрыш, что их стоит реализовывать как поэксперт, насколько это возможно<sup>7</sup>. Когда вы можете уверенно

<sup>7</sup> В спецификациях интерфейса для операций перемещения в контейнерах стандартной библиотеки поэксперт отсутствует. Однако разработчикам разрешено усиливать спецификации интерфейсов функций стандартной библиотеки, и на практике как минимум для некоторых контейнеров операции перемещения объявляются как поэксперт. Эта практика является примером следования совету из данного раздела. Обнаружив, что можно написать операции перемещения так, что исключения гарантированно не будут генерироваться, разработчики часто объявляют такие операции как поэксперт, несмотря на то что стандарт языка от них этого не требует.

сказать, что за пределы функции не должно выйти ни одно исключение, вы, определенно, должны объявить ее как поехсерт.

Пожалуйста, обратите внимание на мои слова о том, что некоторые функции имеют естественную реализацию поехсерт. Изменение реализации функции только для того, чтобы объявить ее как поехсерт, — это хвост, виляющий собакой. Это телега перед лошадью. Это неумение увидеть лес за деревьями. Впрочем, довольно метафор. Если простая реализация функции может приводить к исключениям (например, путем вызова функции, которая может сгенерировать исключение), то попытка скрыть это исключение от вызывающего кода (например, перехватывая все исключения и заменяя их кодами состояния или специальными возвращаемыми значениями) усложнит не только реализацию вашей функции, но обычно и код в точке вызова. Например, вызывающая функция, как может оказаться, должна проверять код состояния или наличие специальных возвращаемых значений. Стоимость времени выполнения таких усложнений (например, дополнительных ветвлений, вызовов функций, которые будут влиять на кеши команд, и т.п.) легко может превысить ускорение, которое вы надеялись получить благодаря поехсерт; кроме того, такой исходный код труднее понимать и поддерживать. Ничего хорошего из этого не получается.

Для некоторых функций отсутствие исключений настолько важно, что они являются поехсерт по умолчанию. В C++98 считается плохим стилем разрешать генерировать исключения функциям освобождения памяти (т.е. операторам `operator delete` и `operator delete[]`) и деструкторам, а в C++11 это правило стиля стало правилом языка. По умолчанию все функции освобождения памяти и все деструкторы — как пользовательские, так и генерируемые компиляторами — неявно являются поехсерт. Поэтому необходимости явно объявлять их таковыми нет. (Это ничему не повредит, это просто необычно.) Единственная ситуация, когда деструктор не является неявно объявлением как поехсерт, — это когда член-данные класса (включая унаследованные члены и содержащиеся внутри других членов-данных) имеет тип, который явно указывает, что его деструктор может генерировать исключения (например, объявленный как `poehscept(false)`). Такие деструкторы являются редкостью. Их нет в стандартной библиотеке, и если деструктор объекта, используемого стандартной библиотекой (например, как элемент контейнера или переданный алгоритму аргумент), генерирует исключение, поведение программы является неопределенным.

Стоит отметить, что некоторые проектировщики библиотечных интерфейсов различают функции с широкими контрактами от функций с узкими контрактами. Функция с широким контрактом не имеет предусловий. Такая функция может быть вызвана независимо от состояния программы и не накладывает никаких ограничений на аргументы, передаваемые ей вызывающим кодом<sup>\*</sup>. Функции с широким контрактом никогда не демонстрируют неопределенного поведения.

\* “Независимо от состояния программы” и “без ограничений” не узаконивает программы, поведение которых уже является неопределенным. Например, `std::vector::size` имеет широкий контракт, но это не означает, что данная функция должна разумно себя вести при применении к произвольному блоку памяти, приведенному к типу `std::vector`. Результат приведения не определен, поэтому нет никаких гарантий, касающихся поведения программы, содержащей такое приведение.

Функции без широких контрактов имеют узкие контракты. Если предусловия для таких функций нарушены, их результаты являются неопределенными.

Если вы пишете функцию с широким контрактом и знаете, что ее не покинут никакие исключения, легко следовать совету из данного раздела и объявить ее как `noexcept`. Для функций с узкими контрактами ситуация сложнее. Предположим, например, что вы пишете функцию `f`, принимающую параметр `std::string`, и пусть естественная реализация `f` никогда не генерирует исключений. Это предполагает, что функция `f` должна быть объявлена как `noexcept`.

Предположим теперь, что `f` имеет предусловие: длина ее строкового параметра `std::string` не должна превышать 32 символа. Если `f` вызывается со строкой `std::string`, длина которой больше 32 символов, поведение будет неопределенным, поскольку нарушение предусловия *по определению* приводит к неопределенному поведению. Функция `f` не обязана проверять это предусловие, так как функции могут считать свои предусловия выполненными. (За обеспечение выполнения таких предположений отвечает вызывающий код.) Тогда даже при наличии предусловия объявление `f` как `noexcept` выглядит целесообразным:

```
void f(const std::string& s) noexcept; // Предусловие:  
// s.length() <= 32
```

Но предположим, что разработчик `f` решил проверять нарушения предусловия. Такая проверка не обязательна, но она и не запрещена; более того, проверка предусловия может быть полезной, например, в процессе системного тестирования. Отладка сгенерированного исключения в общем случае проще, чем попытки отследить причину неопределенного поведения. Но как следует сообщать о нарушении предусловий, чтобы проверки (или клиентский обработчик ошибок) могли его обнаружить? Простейший подход — генерация исключения “предусловие нарушено”, но если `f` объявлена как `noexcept`, это может быть невозможным; генерация исключения приведет к завершению программы. По этой причине разработчики библиотек, различающие широкие и узкие контракты, в общем случае резервируют `noexcept` для функций с широкими контрактами.

В заключение позвольте мне остановиться на моих ранних наблюдениях, что обычно компиляторы не предлагают помочь в выявлении несоответствий между реализациями функций и их спецификациями исключений. Рассмотрим следующий совершенно корректный код:

```
void setup(); // Функции, определенные в другом месте  
void cleanup();  
  
void doWork() noexcept  
{  
    setup(); // Настройка для выполнения некоторой работы  
    ... // Выполнение работы  
    cleanup(); // Очистка после выполнения работы  
}
```

Здесь функция `doWork` объявлена как `noexcept`, несмотря на то, что она вызывает функции `setup` и `cleanup` без такого модификатора. Это кажется противоречием, но ведь может быть так, что функции `setup` и `cleanup` документированы как не генерирующие исключений, пусть при этом они и не объявлены как `noexcept`. Для такого их объявления могут иметься некоторые веские причины. Например, они могут быть частью библиотеки, написанной на языке программирования С. (Даже у функций стандартной библиотеки С, перемещенных в пространство имен `std`, отсутствуют спецификации исключений; например, `std::strlen` не объявлена как `noexcept`.) Они также могут быть частью стандартной библиотеки C++98, в которой решено не использовать спецификации исключений C++98 и которая еще не пересмотрена на предмет использования возможностей C++11.

Поскольку для функций, объявленных как `noexcept`, имеются законные основания полагаться на код, не имеющий гарантии `noexcept`, С++ допускает существование такого кода, и компиляторы в общем случае не выдают при этом никаких предупреждений о нем.

### Следует запомнить

- `noexcept` является частью интерфейса функции, а это означает, что вызывающий код может зависеть от наличия данного модификатора.
- Функции, объявленные как `noexcept`, предоставляют большие возможности оптимизации, чем функции без такой спецификации.
- Спецификация `noexcept` имеет особое значение для операций перемещения, обмена, функций освобождения памяти и деструкторов.
- Большинство функций нейтральны по отношению к исключениям, а не являются `noexcept`-функциями.

## 3.9. Используйте, где это возможно, `constexpr`

Если бы присуждалась награда за наиболее запутанную новую возможность в С++11, без сомнений, ее бы получило новое ключевое слово `constexpr`. При применении к объектам это, по сути, усиленная разновидность `const`, но при применении к функциям оно имеет совсем другой смысл. Имеет смысл разобраться в этой путанице, потому что, когда ключевое слово `constexpr` соответствует тому, что вы, хотите выразить, вы определенно, захотите его использовать.

Концептуально ключевое слово `constexpr` указывает значение, которое не просто является константой, но и известно во время компиляции. Эта концепция — лишь часть всей истории, поскольку при применении `constexpr` к функциям появляется больше нюансов, чем можно предположить. Чтобы не забегать вперед и не портить сюрприз, пока что я только скажу, что вы не можете ни считать, что результат `constexpr`-функции представляет константу, ни считать, что эти значения известны во время компиляции. Пожалуй, наиболее интригующим является то, что это *возможности* данного ключевого слова. Это *хорошо*, что `constexpr`-функция не обязана давать константный результат или результат, известный во время компиляции!

Но давайте начнем с объектов `constexpr`. Такие объекты являются, по сути, константными (`const`) и на самом деле обладают значениями, известными во время компиляции. (Технически их значения определяются во время *трансляции*, а трансляция состоит не только из компиляции, но и из компоновки. Однако, если вы не пишете компиляторы или редакторы связей для C++, это не имеет для вас значения, так что можете беспечно программировать так, как будто эти значения объектов `constexpr` определяются во время компиляции.)

Значения, известные во время компиляции, являются привилегированными. Они, например, могут быть размещены в памяти, предназначеннной только для чтения, и это может представлять особое значение для разработчиков встроенных систем. Широко применяется то, что целочисленные значения, являющиеся константами и известные во время компиляции, могут использоваться в контекстах, где C++ требует *целочисленное константное выражение*. Такие контексты включают спецификации размеров массивов, целочисленные аргументы шаблонов (включая длину объектов `std::array`), значения перечислителей, спецификаторы выравнивания и прочее. Если вы хотите использовать для этих вещей переменные, вы, определенно, захотите объявить их как `constexpr`, поскольку тогда компиляторы будут точно знать, что имеют дело со значением времени компиляции:

```
int sz;                                // Неконстантная переменная
...
constexpr auto arraySize1 = sz;           // Ошибка! Значение sz
                                         // неизвестно при компиляции
std::array<int, sz> data1;               // Ошибка! Та же проблема
constexpr auto arraySize2 = 10;            // OK, 10 представляет собой
                                         // константу времени компиляции
std::array<int,
          arraySize2> data2;             // OK, arraySize2 представляет
                                         // собой constexpr
```

Обратите внимание, что `const` не предоставляет таких же гарантий, что и `constexpr`, поскольку объекты `const` не обязаны инициализироваться значениями, известными во время компиляции:

```
int sz;                                // Как и ранее
...
const auto arraySize = sz;               // OK, arraySize является
                                         // константной копией sz
std::array<int,
          arraySize> data;             // Ошибка! Значение arraySize
                                         // при компиляции неизвестно
```

Проще говоря, все объекты, являющиеся `constexpr`, являются `const`, но не все объекты, являющиеся `const`, являются `constexpr`. Если вы хотите, чтобы компиляторы гарантировали, что переменная имеет значение, которое можно использовать в требующих константы времени компиляции контекстах, то следует использовать `constexpr`, а не `const`.

Сценарии применения объектов `constexpr` становятся более интересными, когда в дело вступают функции `constexpr`. Такие функции производят константы времени

компиляции, когда они вызываются с константами времени компиляции. Если они вызываются со значениями, неизвестными до времени выполнения, они производят значения времени выполнения. Это может выглядеть так, как будто вы не знаете, что они будут делать, но так думать — неверно. Вот как выглядит правильный взгляд на эти моменты:

- Функции, объявленные как `constexpr`, могут использоваться в контекстах, требующих константы времени компиляции. Если значения передаваемых вами аргументов в `constexpr`-функцию в таком контексте известны во время компиляции, результат функции будет вычислен в процессе компиляции. Если любое из значений аргументов неизвестно во время компиляции, ваш код будет отвергнут.
- Когда `constexpr`-функция вызывается с одним или несколькими значениями, неизвестными во время компиляции, она действует так же, как и обычная функция, выполняя вычисления во время выполнения. Это означает, что вам не нужны две функции для выполнения одних и тех же операций, одной — для констант времени компиляции, другой — для всех прочих значений. Функция, объявленная как `constexpr`, выполняет их все.

Предположим, что нам нужна структура данных для хранения результатов эксперимента, который может быть проведен при разных условиях. Например, уровень освещения в ходе эксперимента может быть высоким, низким или освещение может быть отключено вовсе; может быть разная температура, и т.д. Если всего имеется  $n$  условий, влияющих на проведение эксперимента, и у каждого по три возможных состояния, то общее количество комбинаций составит  $3^n$ . Хранение результатов экспериментов для всех комбинаций условий требует структуры данных с достаточным количеством памяти для хранения  $3^n$  значений. В предположении, что каждый результат представляет собой `int` и что  $n$  известно (или может быть вычислено) во время компиляции, подходящим выбором структуры данных может быть `std::array`. Однако нам требуется способ вычисления  $3^n$  во время компиляции. Стандартная библиотека C++ предоставляет функцию `std::pow`, обеспечивающую интересующую нас математическую функциональность, но с точки зрения наших целей имеются две проблемы. Во-первых, `std::pow` работает с типами с плавающей точкой, а нам нужен целочисленный результат. Во-вторых, `std::pow` не является `constexpr` (т.е. не гарантирует возврат времени компиляции при переданных ей значениях времени компиляции), так что мы не можем использовать ее для указания размера `std::array`.

К счастью, мы можем написать функцию `pow`, которая нам нужна. Как это сделать, я покажу чуть позже, но сначала давайте взглянем, каким образом эта функция может быть объявлена и использована:

```
constexpr // pow является constexpr
int pow(int base, int exp) noexcept // Не генерирует исключений
{
    // Ее реализация - ниже
}
```

```
constexpr auto numConds = 5;           // Количество условий
                                         // std::array<int, pow(3, numConds)>
                                         // results содержит
                                         // 3^numConds элементов
```

Вспомним, что `constexpr` перед `pow` не говорит о том, что `pow` возвращает константное значение; оно говорит, что если `base` и `exp` являются константами времени компиляции, то результат `pow` может быть использован как константа времени компиляции. Если `base` и/или `exp` не являются константами времени компиляции, то результат `pow` будет вычисляться во время выполнения. Это означает, что `pow` может быть вызвана не только для вычисления во время компиляции таких вещей, как размер `std::array`, но и в контексте времени выполнения, как здесь:

```
auto base = readFromDB("base");      // Эти значения получаются
auto exp = readFromDB("exponent");   // во время компиляции
auto baseToExp = pow(base, exp);     // Вызов функции pow
                                         // во время выполнения
```

Поскольку функции `constexpr` должны быть способны возвращать результаты во время компиляции при вызове со значениями времени компиляции, на их реализации накладываются ограничения. Эти ограничения различны в C++11 и C++14.

В C++11 функции `constexpr` могут содержать не более одной выполнимой инструкции — `return`. Это выглядит более ограничивающим, чем является на самом деле, поскольку для повышения выразительности `constexpr`-функций можно использовать две хитрости. Во-первых, можно применять условный оператор “`? :`” вместо инструкции `if-else`, а во-вторых, вместо циклов можно использовать рекурсию. Таким образом, функция `pow` может быть реализована следующим образом:

```
constexpr int pow(int base, int exp) noexcept
{
    return (exp == 0 ? 1 : base * pow(base, exp - 1));
}
```

Этот код работает, но только очень непритязательный функциональный программист сможет назвать его красивым. В C++14 ограничения на `constexpr`-функции существенно слабее, так что становится возможной следующая реализация:

```
constexpr int pow(int base, int exp) noexcept // C++14
{
    auto result = 1;
    for (int i = 0; i < exp; ++i) result *= base;
    return result;
}
```

Функции `constexpr` ограничены приемом и возвратом только *литеральных типов* (*literal types*), которые, по сути, означают типы, могущие иметь значения, определяемые во время компиляции. В C++11 к ним относятся все встроенные типы за исключением

`void`, но литеральными могут быть и пользовательские типы, поскольку конструкторы и прочие функции-члены также могут являться `constexpr`:

```
class Point {  
public:  
    constexpr Point(double xVal = 0, double yVal = 0) noexcept  
        : x(xVal), y(yVal)  
    {}  
  
    constexpr double xValue() const noexcept { return x; }  
    constexpr double yValue() const noexcept { return y; }  
  
    void setX(double newX) noexcept { x = newX; }  
    void setY(double newY) noexcept { y = newY; }  
private:  
    double x, y;  
};
```

Здесь конструктор `Point` может быть объявлен как `constexpr`, поскольку, если переданные ему аргументы известны во время компиляции, значения членов-данных созданного `Point` также могут быть известны во время компиляции. А значит, инициализированный таким образом объект `Point` может быть `constexpr`:

```
constexpr Point p1(9.4, 27.7); // OK, во время компиляции  
                                // работает constexpr конструктор  
constexpr Point p2(28.8, 5.3); // То же самое
```

Аналогично функции доступа `xValue` и `yValue` могут быть `constexpr`, поскольку если они вызываются для объекта `Point` со значением, известным во время компиляции (например, объект `constexpr Point`), значения членов-данных `x` и `y` могут быть известны во время компиляции. Это делает возможным написать `constexpr`-функции, которые вызывают функции доступа `Point` и инициализируют `constexpr`-объекты результатами вызовов этих функций:

```
constexpr  
Point midpoint(const Point& p1, const Point& p2) noexcept  
{  
    return {(p1.xValue() + p2.xValue()) / 2, // Вызов constexpr  
            (p1.yValue() + p2.yValue()) / 2}; // функции-члена  
}  
  
constexpr auto mid = midpoint(p1, p2); // Инициализация  
                                         // constexpr объекта результатом constexpr-функции
```

Это очень интересно. Это означает, что объект `mid` может быть создан в памяти, предназначенной только для чтения, несмотря на то что его инициализация включает вызовы конструкторов, функций доступа и функций, не являющейся членом! Это означает, что вы можете использовать выражение наподобие `mid.xValue() * 10` в аргументе шаблона

или в выражении, определяющем значение перечислителя<sup>9</sup>! Это означает, что традиционно довольно строгая граница между работой во время компиляции и работой во время выполнения начинает размываться, и некоторые вычисления, традиционно являющиеся вычислениями времени выполнения, могут перейти на стадию компиляции. Чем больший код участвует в таком переходе, тем быстрее будет работать ваша программа. (Однако компилироваться она может существенно дольше.)

В C++11 два ограничения предотвращают объявление функций-членов `Point setX` и `setY` как `constexpr`. Во-первых, они модифицируют объект, с которым работают, а в C++11 функции-члены `constexpr` неявно являются `const`. Во-вторых, они имеют возвращаемый тип `void`, а `void` не является литеральным типом в C++11. Оба эти ограничения сняты в C++14, так что в C++14 даже функции установки полей `Point` могут быть объявлены как `constexpr`:

```
class Point {  
public:  
  
    constexpr void setX(double newX) noexcept // C++14  
    { x = newX; }  
  
    constexpr void setY(double newY) noexcept // C++14  
    { y = newY; }  
};
```

Это делает возможным написание функций наподобие следующей:

```
// Возвращает отражение точки p  
// относительно начала координат (C++14)  
constexpr Point reflection(const Point& p) noexcept  
{  
    Point result; // Неконстантный объект Point  
    result.setX(-p.xValue()); // Установка его полей x и y  
    result.setY(-p.yValue());  
    return result; // Возврат копии  
}
```

Соответствующий клиентский код имеет вид:

```
constexpr Point p1(9.4, 27.7); // Как и выше  
constexpr Point p2(28.8, 5.3);  
constexpr auto mid = midpoint(p1, p2);  
constexpr auto reflectedMid = // reflectedMid представляет
```

<sup>9</sup> Поскольку `Point::xValue` возвращает `double`, типом `mid.xValue() * 10` также является `double`. Типы с плавающей точкой не могут использоваться для инстанцирования шаблонов или для указания значений перечислений, но они могут быть использованы как части больших выражений, дающих интегральные типы. Например, для инстанцирования шаблона или для указания значения перечислителя может использоваться выражение `static_cast<int>(mid.xValue() * 10)`.

```
reflection(mid);           // собой (-19.1 -16.5) и  
                           // известно во время компиляции
```

Совет из этого раздела заключается в том, чтобы использовать `constexpr` везде, где это только возможно, и теперь, надеюсь, вам понятно, почему: и объекты `constexpr`, и `constexpr`-функции могут применяться в более широком диапазоне контекстов, чем объекты и функции, не являющиеся `constexpr`. Применяя `constexpr`, где это возможно, вы максимизируете диапазон ситуаций, в которых ваши объекты и функции могут быть использованы.

Важно отметить, что `constexpr` является частью интерфейса объекта или функции. `constexpr` провозглашает: “Меня можно использовать в контексте, где для C++ требуется константное выражение”. Если вы объявляете объект или функцию как `constexpr`, клиенты могут использовать их в указанных контекстах. Если позже вы решите, что такое использование `constexpr` было ошибкой, и удалите его, то это может привести к тому, что большое количество клиентского кода перестанет компилироваться. (Простое действие, заключающееся в добавлении в функцию отладочного вывода для отладки или настройки производительности может привести к таким проблемам, поскольку инструкции ввода-вывода в общем случае в `constexpr`-функциях недопустимы.) Часть “где это возможно” совета является вашей добной волей на приданье долгосрочного характера данному ограничению на объекты и функции, к которым вы его применяете.

#### Следует запомнить

- Объекты `constexpr` являются константными и инициализируются объектами, значения которых известны во время компиляции.
- Функции `constexpr` могут производить результаты времени компиляции при вызове с аргументами, значения которых известны во время компиляции.
- Объекты и функции `constexpr` могут использоваться в более широком диапазоне контекстов по сравнению с объектами и функциями, не являющимися `constexpr`.
- `constexpr` является частью интерфейса объектов и функций.

## 3.10. Делайте константные функции-члены безопасными в смысле потоков

Если мы работаем в области математики, нам может пригодиться класс, представляющий полиномы. В этом классе было бы неплохо иметь функцию для вычисления корней полинома, т.е. значений, при которых значение полинома равно нулю. Такая функция не должна модифицировать полином, так что ее естественно объявить как `const`:

```
class Polynomial {  
public:  
    using RootsType =           // Структура данных, хранящая  
        std::vector<double>; // значения, где полином равен нулю
```

```
... // (см. "using" в разделе 3.3)
```

```
RootsType roots() const;
```

```
};
```

Вычисление корней — трудная дорогостоящая операция, так что мы не хотим их вычислять до того, как они реально потребуются. Но если они нам требуются, то, определенно, требуются не один раз. Поэтому мы будем кешировать корни полиномов, если нам приходится их вычислять, и реализуем `roots` так, чтобы функция возвращала кешированное значение. Вот как выглядит такой подход:

```
class Polynomial {  
public:  
    using RootsType = std::vector<double>;  
  
    RootsType roots() const  
    {  
        if (!rootsAreValid) { // Если кеш некорректен,  
            // вычисляем корни и сохраняем  
            // их в in rootVals  
        rootsAreValid = true;  
    }  
    return rootVals;  
}  
private:  
    mutable bool rootsAreValid{ false }; // См. инициализаторы  
    mutable RootsType rootVals{}; // в разделе 3.1  
};
```

Концептуально `roots` не изменяет объект `Polynomial`, с которым работает, но в качестве части кеширующих действий может потребоваться изменение `rootVals` и `rootsAreValid`. Это классический случай использования `mutable`, и именно поэтому эти данные-члены объявлены с данным модификатором.

Представим теперь, что два потока одновременно вызывают `roots` для объекта `Polynomial`:

```
Polynomial p;  
...  
/*---- Поток 1 ----- */ /*---- Поток 2 ----- */  
auto rootsOfP = p.roots(); auto valsGivingZero = p.roots();
```

Этот клиентский код совершенно разумен. Функция `roots` является константной функцией-членом, и это означает, что она представляет операцию чтения. Выполнение операций чтения несколькими потоками одновременно без синхронизации вполне безопасно. Как минимум предполагается, что это так. В данном случае это не так, поскольку в функции `roots` один или оба эти потока могут попытаться изменить члены-данные `rootsAreValid` и `rootVals`. Это означает, что данный код может одновременно читать

и записывать одни и те же ячейки памяти без синхронизации, а это — определение гонки данных. Такой код имеет неопределенное поведение.

Проблема заключается в том, что функция `roots` объявлена как `const`, но не является безопасной с точки зрения потоков. Объявление `const` является корректным как в C++11, так и в C++98 (вычисление корней полинома не изменяет сам полином), так что коррекция нужна для повышения безопасности потоков.

Простейший способ решения проблемы обычно один: применение `mutex`:

```
class Polynomial {  
public:  
    using RootsType = std::vector<double>;  
    RootsType roots() const  
    {  
        std::lock_guard<std::mutex> g(m); // Блокировка мьютекса  
        if (!rootsAreValid) {           // Если кеш некорректен  
            ...                         // Вычисление корней  
            rootsAreValid = true;  
        }  
        return rootVals;               // Разблокирование  
    }  
private:  
    mutable std::mutex m;  
    mutable bool rootsAreValid{ false };  
    mutable RootsType rootVals{};  
};
```

Мьютекс `std::mutex m` объявлен как `mutable`, поскольку его блокировка и разблокирование являются неконстантными функциями, а в противном случае в константной функции-члене `roots` мьютекс `m` рассматривается как константный объект.

Следует отметить, что поскольку `std::mutex` не может быть ни скопирован, ни перемещен, побочным эффектом добавления `m` к `Polynomial` является то, что `Polynomial` теряет возможность копирования и перемещения.

В некоторых ситуациях мьютекс является излишеством. Например, если все, что вы делаете, — это подсчитываете, сколько раз вызывается функция-член, то часто более дешевым средством является счетчик `std::atomic` (т.е. счетчик, для которого гарантируется атомарность операций — см. раздел 7.6). (Действительно ли это более дешевое средство, зависит от аппаратного обеспечения и реализации мьютексов в вашей стандартной библиотеке.) Вот как можно использовать `std::atomic` для подсчета вызовов:

```
class Point {                                // Двумерная точка  
public:  
    ...  
    double distanceFromOrigin() // См. описание noexcept  
        const noexcept          // в разделе 3.8  
    {  
        ++callCount;           // Атомарный инкремент
```

```

        return std::hypot(x, y); // std::hypot – новинка C++11
    }
private:
    mutable std::atomic<unsigned> callCount{ 0 };
    double x, y;
};

```

Как и std::mutex, std::atomic невозможно копировать и неперемещать, так что наличие callCount в Point означает, что Point также невозможно и перемещать.

Поскольку операции над переменными std::atomic зачастую менее дорогостоящи, чем захват и освобождение мьютекса, вы можете соблазниться использовать std::atomic больше, чем следует. Например, в классе, кеширующем дорогостоящее для вычисления значение int, вы можете попытаться использовать вместо мьютекса пару переменных std::atomic:

```

class Widget {
public:
    ...
    int magicValue() const
    {
        if (cacheValid) return cachedValue;
        else {
            auto val1 = expensiveComputation1();
            auto val2 = expensiveComputation2();
            cachedValue = val1 + val2; // Часть 1
            cacheValid = true;      // Часть 2
            return cachedValue;
        }
    }
private:
    mutable std::atomic<bool> cacheValid{ false };
    mutable std::atomic<int> cachedValue;
};

```

Этот способ работает, но иногда выполняет существенно большую работу, чем требуется. Рассмотрим такой сценарий.

- Поток вызывает Widget::magicValue, видит, что cacheValid равно false, выполняет два дорогостоящих вычисления и присваивает их сумму переменной cachedValue.
- В этот момент второй поток вызывает Widget::magicValue, также видит, что значение cacheValid равно false, а потому выполняет те же дорогостоящие вычисления, что и только что завершивший их первый поток. (Этот “второй поток” на самом деле может быть *несколькими* другими потоками.)

Чтобы справиться с этой проблемой, можно пересмотреть порядок присваиваний значений переменным cachedValue и cacheValid, но вы вскоре поймете, что (1) вычислять

vall и val2 перед тем, как cacheValid устанавливается равным true, по-прежнему могут несколько потоков, тем самым провалив цель нашего упражнения, и (2) на самом деле все может быть еще хуже:

```
class Widget {
public:
    ...
    int magicValue() const
    {
        if (cacheValid) return cachedValue;
        else {
            auto vall = expensiveComputation1();
            auto val2 = expensiveComputation2();
            cacheValid = true; // Часть 1
            return cachedValue = vall + val2; // Часть 2
        }
    }
};
```

Представим, что значение cacheValid равно false. Тогда возможно следующее.

- Один поток вызывает Widget::magicValue и выполняет код до точки, где переменная cacheValid устанавливается равной true.
- В этот момент второй поток вызывает Widget::magicValue и проверяет значение cacheValid. Увидев, что оно равно true, поток возвращает cachedValue, несмотря на то, что первый поток еще не выполнил присваивание этой переменной. Таким образом, возвращенное значение оказывается некорректным.

Это неплохой урок. Для единственной переменной или ячейки памяти, требующей синхронизации, применение std::atomic является адекватным решением, но как только у вас имеется две и более переменных или ячеек памяти, которыми надо оперировать как единым целым, вы должны использовать мьютекс. Для Widget::magicValue это выглядит следующим образом:

```
class Widget {
public:
    ...
    int magicValue() const
    {
        std::lock_guard<std::mutex> guard(m); // Блокировка m
        if (cacheValid) return cachedValue;
        else {
            auto vall = expensiveComputation1();
            auto val2 = expensiveComputation2();
            cachedValue = vall + val2;
            cacheValid = true;
        }
    }
};
```

```

        return cachedValue;
    }
}

...
private:
    mutable std::mutex m;
    mutable int cachedValue; // Не атомарное
    mutable bool cacheValid{ false }; // Не атомарное
};


```

Сейчас данный раздел основывается на предположении, что несколько потоков могут одновременно выполнять константную функцию-член объекта. Если вы пишете константную функцию-член там, где это не так — т.е. там, где вы можете *гарантировать*, что эта функция-член объекта никогда не будет выполняться более чем одним потоком, — безопасность с точки зрения потоков является несущественной. Например, совершенно неважно, являются ли безопасными с точки зрения потоков функции-члены классов, разработанные исключительно для однопоточного применения. В таких случаях вы можете избежать расходов, связанных с мьютексами и `std::atomic`, а также побочного эффекта, заключающегося в том, что содержащие их классы становятся некопируемыми и неперемещаемыми. Однако такие сценарии, в которых нет потоков, становятся все более редкими и, вероятно, дальше будут становиться только все более редкими. Безопаснее считать, что константные функции-члены будут участвовать в параллельных вычислениях, и именно поэтому следует гарантировать безопасность таких функций с точки зрения потоков.

#### Следует запомнить

- Делайте константные функции-члены безопасными с точки зрения потоков, если только вы не можете быть уверены, что они *гарантированно* не будут использоваться в контексте параллельных вычислений.
- Использование переменных `std::atomic` может обеспечить более высокую по сравнению с мьютексами производительность, но они годятся только для работы с единственной переменной или ячейкой памяти.

## 3.11. Генерация специальных функций-членов

В официальной терминологии C++ *специальные функции-члены* — это те функции-члены, которые C++ готов генерировать сам. С ++ 98 включает четыре такие функции: конструктор по умолчанию, деструктор, копирующий конструктор и оператор копирующего присваивания. Эти функции создаются, только если они необходимы, т.е. если некоторый код использует их без их явного объявления в классе. Конструктор по умолчанию генерируется только в том случае, если в классе не объявлен ни один конструктор. (Это предотвращает компиляторы от создания конструктора по умолчанию для класса, для которого вы указали, что конструктору требуются аргументы. Сгенерированные

специальные функции-члены неявно являются открытыми и встраиваемыми; они также являются не виртуальными, если только таковой функцией не является деструктор производного класса, унаследованного от базового класса с виртуальным деструктором. В этом случае генерируемый компилятором деструктор производного класса также является виртуальным.

Ну, конечно же, вы отлично все это знаете. Да, да, древняя история: Месопотамия, династия Цинь, Владимир Красное Солнышко, FORTRAN, C++98... Но времена изменились, а с ними изменились и правила генерации специальных функций-членов в C++. Важно быть в курсе новых правил, потому что мало вещей имеют такое же важное значение для эффективного программирования на C++, как знание о том, когда компиляторы молча добавляют функции-члены в ваши классы.

Что касается C++11, то в клуб специальных функций-членов приняты два новых игрока: перемещающий конструктор и оператор перемещающего присваивания. Их сигнатуры имеют следующий вид:

```
class Widget {
public:
    ...
    Widget(Widget&& rhs);           // Перемещающий конструктор
    Widget& operator=(Widget&& rhs); // Оператор перемещающего
                                       // присваивания
};
```

Правила, регулирующие их создание и поведение, аналогичны правилам для их копирующих двойников. Перемещающие операции генерируются только если они необходимы, и если они генерируются, то выполняют “почленное перемещение” нестатических членов-данных класса. Это означает, что перемещающий конструктор создает каждый нестатический член-данные класса из соответствующего члена его параметра `rhs` с помощью перемещения, а оператор перемещающего присваивания выполняет перемещающее присваивание каждого нестатического члена-данных из переданного ему параметра. Перемещающий конструктор также выполняет перемещающее конструирование частей базового класса (если такие имеются), а оператор перемещающего присваивания выполняет соответственно перемещающее присваивание частей базового класса.

Когда я говорю о перемещающей операции над членом-данными или базовым классом, нет никакой гарантии, что перемещение в действительности имеет место. “Почленные перемещения” в действительности представляют собой *запросы* на почленное перемещение, поскольку типы, которые *не могут* быть *перемещены* (т.е. не обладают поддержкой операций перемещения; например, таковыми являются большинство старых классов C++98), будут “перемещены” с помощью операций копирования. Сердцем каждого почленного “перемещения” является применение `std::move` к объекту, из которого выполняется перемещение, а результат используется в процессе разрешения перегрузки функций для выяснения, должно ли выполняться перемещение или копирование. Этот процесс детально описывается в разделе 5.1. В этом разделе просто помните, что почленное перемещение состоит из операций перемещения для тех членов-данных и базовых

классов, которые поддерживают перемещающие операции, и из операций копирования для тех, которые перемещающие операции не поддерживают.

Как и в случае с копирующими операциями, перемещающие операции не генерируются, если вы сами их не объявляете. Однако точные условия, при которых они генерируются, несколько отличаются от условий для копирующих операций.

Две копирующие операции независимы одна от другой: объявление одной не препятствует компилятору генерировать другую. Так что если вы объявляете копирующий конструктор, но не копирующий оператор присваивания, а затем пишете код, которому требуется копирующее присваивание, то компиляторы будут генерировать оператор копирующего присваивания вместо вас. Аналогично, если вы объявили оператор копирующего присваивания, но не копирующий конструктор, а вашему коду нужен копирующий конструктор, то последний будет сгенерирован компилятором вместо вас. Это правило работало в C++98 и остается справедливым в C++11.

Две перемещающие операции не являются независимыми. Если вы объявите одну из них, это не позволит компиляторам сгенерировать вторую. Это объясняется тем, что если вы объявляете, скажем, перемещающий конструктор для вашего класса, то вы указываете, что есть что-то, что при перемещающем конструировании должно быть реализовано иначе, чем почленное перемещение по умолчанию, генерируемое компиляторами. Но если это что-то неверно при почленном перемещающем конструировании, то, вероятно, оно будет неверно и при почленном перемещающем присваивании. Поэтому объявление перемещающего конструктора предохраняет от генерации перемещающего оператора присваивания, а объявление перемещающего оператора присваивания предохраняет от генерации перемещающего конструктора.

Кроме того, перемещающие операции не будут генерироваться для любого класса, у которого явно объявлены копирующие операции. Объяснение этому заключается в том, что объявление копирующих операций (конструктора и присваивания) указывает, что обычный подход к копированию объекта (почленное копирование) не годится для этого класса, и компиляторы делают заключение, что если для класса не подходит почленное копирование, то, вероятно, почленное перемещение для него тоже не подойдет.

Этот вывод справедлив и в обратном направлении. Объявление в классе перемещающей операции (конструктора или присваивания) приводит к тому, что компиляторы не генерируют копирующие операции. (Копирующие операции отключаются с помощью их удаления; см. раздел 3.5). В конце концов, если почленное перемещение не является корректным способом перемещения, то нет причин ожидать, что почленное копирование окажется корректным способом копирования. Это выглядит как возможное нарушение работоспособности кода C++98, поскольку условия, при которых разрешена генерация операций копирования, являются более ограничивающими в C++11, чем в C++98, но на самом деле это не так. Код C++98 не может иметь перемещающие операции, поскольку в C++98 нет такого понятия, как “перемещение” объектов. Единственный способ для старого класса иметь пользовательские перемещающие операции — это если они будут добавлены в коде C++11. Но классы, которые изменены таким образом,

чтобы использовать преимущества семантики перемещений, обязаны играть по правилам C++11, касающимся генерации специальных функций-членов.

Вероятно, вы слышали о рекомендации о *большой тройке*. Она утверждает, что если вы объявили хотя бы одну из трех операций — копирующий конструктор, копирующий оператор присваивания или деструктор, — то вы должны объявить все три операции. Это правило вытекает из наблюдения, что необходимость изменения смысла копирующей операции почти всегда подразумевает, что (1) какое бы управление ресурсами ни выполнялось в одной копирующей операции, вероятно, такое же управление ресурсами потребуется и во второй копирующей операции; и (2) деструктор класса также должен участвовать в управлении ресурсами (обычно — освобождать их). Классическим управляемым ресурсом является память, и именно поэтому все классы стандартной библиотеки, управляющие памятью (например, контейнеры STL, управляющие динамической памятью), объявляют всю “большую тройку”: обе копирующие операции и деструктор.

Следствием правила большой тройки является то, что наличие пользовательского деструктора указывает на вероятную неприменимость простого почленного копирования для копирующих операций класса. Это, в свою очередь, предполагает, что если класс объявляет деструктор, то копирующие операции, по всей вероятности, не должны генерироваться автоматически, так как они будут выполнять неверные действия. Во времена принятия C++98 важность этих рассуждений не была оценена должным образом, так что в C++98 наличие пользовательского деструктора не влияло на генерацию компиляторами копирующих операций. Это правило перешло и в C++11, но только потому, что ограничение условий, при которых могут автоматически генерироваться копирующие операции, сделает некомпилируемым слишком большое количество старого кода.

Однако рассуждения, лежащие в основе “правила большой тройки”, остаются в силе, и это, в сочетании с наблюдением, что объявление копирующей операции исключает неявную генерацию перемещающих операций, обосновывает тот факт, что C++11 не генерирует операции перемещения для класса с пользовательским деструктором.

Таким образом, перемещающие операции генерируются (при необходимости) для классов, только если выполняются три следующие условия:

- в классе не объявлены никакие копирующие операции;
- в классе не объявлены никакие перемещающие операции;
- в классе не объявлен деструктор.

В некоторый момент аналогичные правила могут быть распространены на копирующие операции, поскольку C++11 выступает против автоматической генерации копирующих операций для классов, объявляющих копирующую операцию или деструктор. Это означает, что если у вас есть код, зависящий от генерации копирующих операций в классе с объявленным деструктором или одной из копирующих операций, то вы должны рассмотреть обновление таких классов для устранения указанной зависимости. Если поведение генерированных компилятором функций вас устраивает (т.е. почленное копирование нестатических членов-данных — именно то, что вам надо), то пересмотр кода

будет очень простым делом, поскольку в C++11 сказать это явно позволяет простая конструкция “`= default`”:

```
class Widget {  
public:  
    ...  
    ~Widget(); // Пользовательский деструктор  
  
    ...  
    Widget(const Widget&); // Поведение копирующего  
    = default; // конструктора по умолчанию  
  
    Widget& operator=(const Widget&); // Поведение копирующего  
    = default; // присваивания по умолчанию  
    // правильное  
  
};
```

Этот подход часто полезен в полиморфных базовых классах, т.е. в классах, определяющих интерфейсы, посредством которых происходит управление производными классами. Обычно полиморфные базовые классы имеют виртуальные деструкторы, поскольку, если это не так, некоторые операции (например, использование `delete` или `typeid` с объектом производного класса через указатель или ссылку на базовый класс) дают неопределенный или вводящий в заблуждение результат. Если только класс не наследует деструктор, являющийся виртуальным, единственный способ сделать деструктор виртуальным — явно объявить его таковым. Зачастую реализация по умолчанию является корректной, и использовать конструкции “`= default`” — хороший способ выразить это. Однако пользовательский деструктор подавляет генерацию перемещающих операций, так что, если требуется поддержка перемещаемости, “`= default`” зачастую находит второе применение. Объявление перемещающих операций отключает копирующие операции, так что, если копируемость также желательна, это делает еще один круг “`= default`”:

```
class Base {  
public:  
    virtual ~Base() = default; // Делает деструктор виртуальным  
  
    Base(Base&&) = default; // Поддержка перемещения  
    Base& operator=(Base&&) = default;  
  
    Base(const Base&); // Поддержка копирования  
    Base& operator=(const Base&); = default;  
  
};
```

Фактически, даже если у вас есть класс, в котором компиляторы могут генерировать копирующие и перемещающие операции и в котором генерируемые функции ведут себя так, как надо, вы можете выбрать стратегию их объявления и применения конструкции

`= default`" в качестве определений. Это требует большего количества работы, но делает ваши намерения более ясными, и это может помочь обойти некоторые довольно трудно выявляемые ошибки. Предположим, например, что у нас есть класс, представляющий таблицу строк, т.е. структуру данных, которая обеспечивает быстрый поиск строкового значения по его целочисленному идентификатору:

```
class StringTable {  
public:  
    StringTable() {}  
    // Функции вставки, удаления, поиска и т.п., но нет  
    // функциональности копирования/перемещения/деструкции  
private:  
    std::map<int, std::string> values;  
};
```

Предположим, что в классе не объявлены ни копирующие операции, ни перемещающие операции, ни деструктор, так что компиляторы автоматически генерируют эти функции при их использовании. Это очень удобно.

Но предположим, что немного позже решено записывать в журнал конструирование по умолчанию и деструкцию таких объектов. Добавление соответствующей функциональности выполняется очень просто:

```
class StringTable {  
public:  
    StringTable()  
    { makeLogEntry("Создание StringTable"); }      // Добавлено  
    ~StringTable()  
    { makeLogEntry("Уничтожение StringTable"); } // Добавлено  
    ...                                         // Прочие функции,  
private:                                         // как и раньше  
    std::map<int, std::string> values; // Как и раньше  
};
```

Выглядит разумно, но объявление деструктора потенциально имеет важное побочное действие: оно предотвращает генерацию перемещающих операций. Создание класса на эти операции не влияет. Таким образом, код, вероятно, будет без проблем компилироваться, выполняться и проходить функциональное тестирование.

Сюда включается и тестирование функциональности перемещения, поскольку, хотя даже этот класс и не является перемещаемым, запрос на перемещение будет компилироваться и работать. Такой запрос, как отмечалось ранее в данном разделе, приводит к выполнению копирования. Это означает, что код, "перемещающий" объекты `StringTable`, в действительности их копирует, т.е. копирует лежащие в их основе объекты `std::map<int, std::string>`. А копирование `std::map<int, std::string>`, скорее всего, окажется на порядки медленнее перемещения. Простое добавление деструктора к классу может тем самым внести значительные проблемы, связанные

с производительностью! Если бы копирующие и перемещающие операции были явно определены как “`= default`”, такая проблема не могла бы возникнуть.

Теперь, благополучно пережив мою бесконечную болтовню о правилах, управляющих операциями копирования и перемещения в C++11, вы можете озадачиться: когда же я, наконец, обращаю свое внимание на две другие специальные функции — конструктор по умолчанию и деструктор? Да прямо сейчас, в этом предложении, и только в нем, потому что для этих функций-членов почти ничего не изменилось: правила в C++11 практически те же, что и в C++98.

Резюмируем правила C++11, управляющие специальными функциями-членами.

- **Конструктор по умолчанию.** Правила те же, что и в C++98. Генерируется, только если класс не содержит пользовательских конструкторов.
- **Деструктор.** Правила по сути те же, что и в C++98; единственное отличие заключается в том, что деструктор по умолчанию является поэксперт (см. раздел 3.8). Как и в C++98, деструктор является виртуальным, только если виртуальным является деструктор базового класса.
- **Копирующий конструктор.** То же поведение времени выполнения, что и в C++98: почленное копирующее конструирование нестатических данных-членов. Генерируется, только если класс не содержит пользовательского копирующего конструктора. Удаляется, если класс объявляет перемещающую операцию. Генерация этой функции в классе с пользовательским оператором копирующего присваивания или деструктором является устаревшей и может быть отменена в будущем.
- **Оператор копирующего присваивания.** То же поведение времени выполнения, что и в C++98: почленное копирующее присваивание нестатических данных-членов. Генерируется, только если класс не содержит пользовательского копирующего оператора присваивания. Удаляется, если класс объявляет перемещающую операцию. Генерация этой функции в классе с пользовательским копирующим конструктором или деструктором является устаревшей и может быть отменена в будущем.
- **Перемещающий конструктор и перемещающий оператор присваивания.** Каждая из этих функций выполняет почленное перемещение нестатических членов-данных. Генерируется, только если класс не содержит пользовательских копирующих операций, пользовательских перемещающих операций или пользовательский деструктор.

Обратите внимание, что в приведенных правилах ничего не говорится о том, что наличие шаблона функции-члена препятствует компиляторам генерировать специальные функции-члены. Это означает, что, если `Widget` имеет следующий вид:

```
class Widget {  
    ...  
    template<typename T> Widget(const T& rhs); // Создание Widget  
                                                // из чего угодно
```

```
template<typename T> // Присваивание Widget
Widget& operator=(const T& rhs); // чего угодно
};
```

компиляторы по-прежнему будут генерировать копирующие и перемещающие операции для `Widget` (в предположении выполнения обычных условий, регулирующих их генерацию), несмотря на то что эти шаблоны могут инстанцироваться как копирующий конструктор и копирующий оператор присваивания (это произойдет в случае, когда `T` представляет собой `Widget`). По всей вероятности, это выглядит как крайний случай, едва стоящий ознакомления с ним, однако я упоминаю его не случайно. В разделе 5.4 вы увидите, что все это может иметь важные последствия.

### Следует запомнить

- Специальные функции-члены — это те функции-члены, которые компиляторы могут генерировать самостоятельно: конструктор по умолчанию, деструктор, копирующие и перемещающие операции.
- Перемещающие операции генерируются только для классов, в которых нет явно объявленных перемещающих операций, копирующих операций и деструктора.
- Копирующий конструктор генерируется только для классов, в которых нет явно объявленного копирующего конструктора, и удаляется, если объявляется перемещающая операция. Копирующий оператор присваивания генерируется только для классов, в которых нет явно объявленного копирующего оператора присваивания, и удаляется, если объявляется перемещающая операция. Генерация копирующих операций в классах с явно объявленным деструктором является устаревшей и может быть отменена в будущем.
- Шаблоны функций-членов не подавляют генерацию специальных функций-членов.



# Интеллектуальные указатели

Поэты и композиторы пишут о любви. Иногда — о подсчетах. Иногда — и о том, и о другом одновременно. Стоит только вспомнить Элизабет Барретт Браунинг (Elizabeth Barrett Browning) с ее “Как я люблю тебя? Позволь мне счесть...” (How do I love thee? Let me count the ways.) и Пола Саймона (Paul Simon) с “Наверняка есть 50 способов уйти от любимого человека” (“There must be 50 ways to leave your lover”). Давайте и мы посчитаем причины, по которым так тяжело любить обычные встроенные указатели.

1. Их объявление не дает информации о том, указывают ли они на один объект или на массив.
2. Их объявление ничего не говорит о том, должны ли вы уничтожить то, на что он указывает, когда завершите работу, т.е. владеет ли указатель тем, на что указывает.
3. Если вы определили, что должны уничтожить то, на что указывает указатель, нет никакого способа указать, как это сделать. Должны ли вы использовать `delete` или имеется иной механизм деструкции (например, специальная функция уничтожения, которой следует передать этот указатель)?
4. Если вам удалось выяснить, что требуется использовать оператор `delete`, то причина 1 означает, что нет никакого способа узнать, следует ли использовать оператор для удаления одного объекта (`delete`) или для удаления массива (`delete []`). Если вы используете оператор неверного вида, результат будет неопределенным.
5. Если вы определили, что указатель владеет тем, на что указывает, и выяснили, каким образом уничтожить то, на что он указывает, оказывается очень трудно обеспечить уничтожение *ровно один раз* на каждом пути вашего кода (включая те, которые возникают благодаря исключениям). Пропущенный путь ведет к утечке ресурсов, а выполнение уничтожения более одного раза — к неопределенному поведению.
6. Обычно нет способа выяснить, не является ли указатель висячим, т.е. не указывает ли он на память, которая больше не хранит объект, на который должен указывать указатель. Висячие указатели образуются, когда объекты уничтожаются, в то время как указатели по-прежнему указывают на них.

Встроенные указатели являются острым инструментом, но десятилетия опыта показывают, что достаточно малейшей небрежности или невнимательности — и об этот инструмент можно очень сильно порезаться.

Одним из средств решения указанных проблем являются *интеллектуальные указатели*. Интеллектуальные указатели представляют собой оболочки вокруг встроенных указателей, которые действуют так же, как и встроенные указатели, но позволяют избежать многих связанных с последними ловушек. Поэтому вы должны предпочтовать встроенным указателям интеллектуальные. Интеллектуальные указатели могут делать почти все то, что и встроенные указатели, но предоставляют куда меньше возможностей для ошибок.

В C++11 имеются четыре интеллектуальных указателя: `std::auto_ptr`, `std::unique_ptr`, `std::shared_ptr` и `std::weak_ptr`. Все они разработаны для того, чтобы помочь управлять временем жизни динамически выделяемых объектов, т.е. избежать утечек ресурсов, гарантируя, что такие объекты уничтожаются соответствующим образом в нужный момент (включая генерацию исключений).

Интеллектуальный указатель `std::auto_ptr` является устаревшим указателем, доставшимся в наследство от C++98. Попытка его стандартизации привела к тому, что в C++11 он превратился в `std::unique_ptr`. Правильное выполнение некоторых работ требовало семантики перемещения, которой не было в C++98. В качестве обходного пути был придуман интеллектуальный указатель `std::auto_ptr`, который превращал операцию копирования в перемещение. Это приводило к удивительному коду (копирование `std::auto_ptr` превращало его в нулевой указатель!) и к разочаровывающим ограничениям при использовании (например, было нельзя хранить `std::auto_ptr` в контейнерах).

Интеллектуальный указатель `std::unique_ptr` делает все то же, что и `std::auto_ptr`, плюс еще кое-что. Он делает это максимально эффективно и безо всяких искажений понятия копирования объекта. Он во всех отношениях лучше `std::auto_ptr`. Единственный случай обоснованного применения `std::auto_ptr` — необходимость компиляции кода компилятором C++98. Если у вас нет такого ограничения, вы должны заменять `std::auto_ptr` указателем `std::unique_ptr`.

API интеллектуальных указателей на удивление разнообразны. Практически единственной общей для всех них функциональностью является наличие конструктора по умолчанию. Поскольку найти описание этих API несложно, я сосредоточусь на информации, которой часто недостает в описаниях API, например заслуживающие внимания способы применения, анализ стоимости времени выполнения и т.п. Освоение такой информации может оказаться существенным для разницы между простым использованием интеллектуальных указателей и их эффективным использованием.

## 4.1. Используйте `std::unique_ptr` для управления ресурсами путем исключительного владения

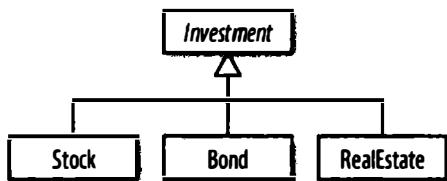
Когда вы обращаетесь к интеллектуальным указателям, обычно ближе других под рукой должен находиться `std::unique_ptr`. Разумно предположить, что по умолчанию `std::unique_ptr` имеет тот же размер, что и обычный указатель, и для большинства

операций (включая разыменования) выполняются точно такие же команды. Это означает, что такие указатели можно использовать даже в ситуациях, когда важны расход памяти и процессорного времени. Если встроенные указатели для вас достаточно малы и быстры, то почти наверняка такими же будут для вас и указатели `std::unique_ptr`.

Интеллектуальные указатели `std::unique_ptr` воплощают в себе семантику исключительного владения. Ненулевой `std::unique_ptr` всегда владеет тем, на что указывает. Перемещение `std::unique_ptr` передает владение от исходного указателя целевому. (Исходный указатель при этом становится нулевым.) Копирование `std::unique_ptr` не разрешается, так как если вы можете копировать `std::unique_ptr`, то у вас будут два `std::unique_ptr`, указывающих на один и тот же ресурс, и каждый из них будет считать, что именно он владеет этим ресурсом (а значит, должен его уничтожить). Таким образом, `std::unique_ptr` является только *перемещаемым типом*. При деструкции ненулевой `std::unique_ptr` освобождает ресурс, которым владеет. По умолчанию освобождение ресурса выполняется с помощью оператора `delete`, примененного ко встроенному указателю в `std::unique_ptr`.

Обычное применение `std::unique_ptr` — возвращаемый тип фабричных функций для объектов иерархии. Предположим, что у нас имеется иерархия типов инвестиций (например, акции, облигации, недвижимость и т.п.) с базовым классом `Investment`.

```
class Investment { ... };
class Stock:
    public Investment { ... };
class Bond:
    public Investment { ... };
class RealEstate:
    public Investment { ... };
```



Фабричная функция для такой иерархии обычно выделяет объект в динамической памяти и возвращает указатель на него, так что за удаление объекта по завершении работы с ним отвечает вызывающая функция. Это в точности соответствует интеллектуальному указателю `std::unique_ptr`, поскольку вызывающий код получает ответственность за ресурс, возвращенный фабрикой (т.е. исключительное владение ресурсом), и `std::unique_ptr` автоматически удаляет то, на что указывает, при уничтожении указателя `std::unique_ptr`. Фабричная функция для иерархии `Investment` может быть объявлена следующим образом:

```
template<typename... Ts>          // Возвращает std::unique_ptr
std::unique_ptr<Investment>      // на объект, созданный из
makeInvestment(Ts&... params);   // данных аргументов
```

Вызывающий код может использовать возвращаемый `std::unique_ptr` как в одной области видимости,

```
{
...
auto pInvestment =           // pInvestment имеет тип
```

```
makeInvestment(arguments); // std::unique_ptr<Investment>
}
} // Уничтожение *pInvestment
```

так и в сценарии передачи владения, таком, как когда std::unique\_ptr, возвращенный фабрикой, перемещается в контейнер, элемент контейнера впоследствии перемещается в член-данные объекта, а этот объект позже уничтожается. Когда это происходит, член-данные std::unique\_ptr объекта также уничтожаются, что приводит к освобождению ресурса, полученного от фабрики. Если цепочка владения прерывается из-за исключения или иного нетипичного потока управления (например, раннего возврата из функции или из-за break в цикле), для std::unique\_ptr, владеющего ресурсом, в конечном итоге вызывается деструктор<sup>1</sup>, и тем самым освобождается захваченный ресурс.

По умолчанию это освобождение выполняется посредством оператора delete, но в процессе конструирования объект std::unique\_ptr можно настроить для использования пользовательских удалителей (custom deleters): произвольных функций (или функциональных объектов, включая получающиеся из лямбда-выражений), вызываемых для освобождения ресурсов. Если объект, созданный с помощью makeInvestment, не должен быть удален непосредственно с помощью delete, а сначала должна быть внесена запись в журнал, makeInvestment можно реализовать следующим образом (пояснения приведены после кода, так что не беспокойтесь, если вы увидите что-то не совсем очевидное для вас).

```
auto delInvmt = [](Investment* pInvestment) // Пользовательский
{
    // удалитель.
    makeLogEntry(pInvestment); // (Лямбда-выражение)
    delete pInvestment;
};

template<typename... Ts> // Исправленный
std::unique_ptr<Investment, // возвращаемый тип
               decltype(delInvmt)>
makeInvestment(Ts&... params)
{
    std::unique_ptr<Investment,
                   decltype(delInvmt)> // Возвращаемый
        pInv(nullptr, delInvmt); // указатель
    if ( /* Должен быть создан объект Stock */ )
    {
        pInv.reset(new Stock(std::forward<Ts>(params)...));
    }
    else if ( /* Должен быть создан объект Bond */ )
    {
```

<sup>1</sup> Из этого правила есть несколько исключений. Большинство из них обусловлено аварийным завершением программы. Если исключение выходит за пределы основной функции потока (например, main в случае начального потока программы) или если нарушена спецификация noexcept (см. раздел 3.8), локальные объекты могут не быть уничтожены; они, определенно, не уничтожаются, когда вызывается функция std::abort или функция выхода (т.е. std::\_Exit, std::exit или std::quick\_exit).

```

        pInv.reset(new Bond(std::forward<Ts>(params) ...));
    }
    else if ( /* Должен быть создан объект RealEstate */
    {
        pInv.reset(new RealEstate(std::forward<Ts>(params) ...));
    }
    return pInv;
}

```

Сейчас я поясню вам, как это работает, но сначала рассмотрим, как все это выглядит с точки зрения вызывающей функции. Предположим, что вы сохраняете результат вызова `makeInvestment` в переменной, объявленной как `auto`, и тем самым остаетесь в блаженном неведении о том, что используемый вами ресурс требует специальной обработки в процессе удаления. Вы просто купаетесь в этом блаженстве, поскольку использование `std::unique_ptr` означает, что вам не надо рассматривать самостоятельно вопросы освобождения ресурса, тем более не требуется беспокоиться о том, чтобы это уничтожение выполнялось в точности один раз на каждом пути в программе. Обо всем этом `std::unique_ptr` заботится автоматически. С точки зрения клиента интерфейс `makeInvestment` — просто конфетка.

Реализация очень красивая, если вы понимаете следующее.

- `delInvmt` представляет собой пользовательский удалитель для объекта, возвращаемого функцией `makeInvestment`. Все функции пользовательских удалителей принимают обычный указатель на удаляемый объект и затем выполняют все необходимые действия по его удалению. В нашем случае действие заключается в вызове `makeLogEntry` и последующем применении `delete`. Применение лямбда-выражения для создания `delInvmt` удобно, но, как вы вскоре увидите, оно также гораздо эффективнее написания обычной функции.
- Когда используется пользовательский удалитель, его тип должен быть указан в качестве второго аргумента типа `std::unique_ptr`. В нашем случае это тип `delInvmt`, и именно поэтому возвращаемым типом `makeInvestment` является `std::unique_ptr<Investment, decltype(delInvmt)>`. (О том, что такое `decltype`, рассказывается в разделе 1.3.)
- Основная стратегия `makeInvestment` состоит в создании нулевого указателя `std::unique_ptr`, после чего он делается указывающим на объект соответствующего типа и возвращается из функции. Для связи пользовательского удалителя `delInvmt` с `pInv` мы передаем его в качестве второго аргумента конструктора.
- Попытка присвоить обычный указатель (например, возвращенный оператором `new`) указателю `std::unique_ptr` компилироваться не будет, поскольку она будет содержать неявное преобразование обычного указателя в интеллектуальный. Такие неявные преобразования могут быть проблематичными, так что интеллектуальные указатели C++11 их запрещают. Вот почему для того, чтобы `pInv` взял на себя владение объектом, созданным с помощью оператора `new`, применяется вызов `reset`.

- С каждым использованием new мы применяем std::forward для прямой передачи аргументов, переданных в makeInvestment (см. раздел 5.3). Это делает всю информацию, предоставляемую вызывающей функцией, доступной конструкторам создаваемых объектов.
- Пользовательский удалитель получает параметр типа Investment\*. Независимо от фактического типа объекта, создаваемого в функции makeInvestment (т.е. Stock, Bond или RealEstate), он в конечном итоге будет удален с помощью оператора delete в лямбда-выражении как объект Investment\*. Это означает, что мы удаляем производный класс через указатель на базовый класс. Чтобы это сработало, базовый класс Investment должен иметь виртуальный деструктор:

```
class Investment {
public:
    ...
    virtual ~Investment(); // Важная часть дизайна
};
```

В C++14 существование вывода возвращаемого типа функции (раздел 1.3) означает, что makeInvestment можно реализовать проще и несколько более инкапсулированно:

```
template<typename... Ts>
auto makeInvestment(Ts&... params) // C++14
{
    auto delInvmt = [](Investment* pInvestment)
    {                               // Теперь размещается в
        makeLogEntry(pInvestment); // пределах makeInvestment
        delete pInvestment;
    };

    std::unique_ptr<Investment, decltype(delInvmt)>
    pInv(nullptr, delInvmt);      // Как и ранее
    if (...)                      // Как и ранее
    {
        pInv.reset(new Stock(std::forward<Ts>(params)...));
    }
    else if (...)                 // Как и ранее
    {
        pInv.reset(new Bond(std::forward<Ts>(params)...));
    }
    else if (...)                 // Как и ранее
    {
        pInv.reset(new RealEstate(std::forward<Ts>(params)...));
    }
    return pInv;                  // Как и ранее
}
```

Ранее я отмечал, что при использовании удалителя по умолчанию (т.е. `delete`) можно разумно предположить, что объекты `std::unique_ptr` имеют тот же размер, что и обычные указатели. Когда в игру вступают пользовательские указатели, это предположение перестает быть верным. Удалители являются указателями на функции, которые в общем случае приводят к увеличению размера `std::unique_ptr` на слово или два. Для удалителей, являющихся функциональными объектами, изменение размера зависит от того, какое состояние хранится в функциональном объекте. Функциональные объекты без состояний (например, получающиеся из лямбда-выражений без захватов) не приводят к увеличению размеров, а это означает что когда пользовательский удалитель может быть реализован как функция или как лямбда-выражение, то реализация в виде лямбда-выражения предпочтительнее:

```
auto delInvmt1 = [](Investment* pInvestment)
{                                              // Пользовательский удалитель
    makeLogEntry(pInvestment); // как лямбда-выражение
    delete pInvestment;       // без состояния
};

template<typename... Ts>           // Возвращаемый тип имеет размер
std::unique_ptr<Investment,      // Investment*
    decltype(delInvmt1)>
makeInvestment(Ts&... args);

void delInvmt2(Investment* pInvestment) // Пользовательский
{                                         // удалитель как
    makeLogEntry(pInvestment);          // функция
    delete pInvestment;
}

template<typename... Ts>           // Возвращаемый тип имеет размер
std::unique_ptr<Investment,      // Investment* плюс как минимум
    void (*)(Investment*)> // размер указателя на функцию!
makeInvestment(Ts&... params);
```

Удалители в виде функциональных объектов с большим размером состояния могут привести к значительным размерам объектов `std::unique_ptr`. Если вы обнаружите, что пользовательский удалитель делает ваш интеллектуальный указатель `std::unique_ptr` неприемлемо большим, вам, вероятно, стоит изменить свой дизайн.

Фабричные функции — не единственный распространенный способ использования `std::unique_ptr`. Эти интеллектуальные указатели еще более популярны в качестве механизма реализации идиомы `Rimpl` (указателя на реализацию). Соответствующий код не сложен, но его рассмотрение отложено до раздела 4.5, посвященного данной теме.

Интеллектуальный указатель `std::unique_ptr` имеет две разновидности: одну — для индивидуальных объектов (`std::unique_ptr<T>`), а другую — для массивов (`std::unique_ptr<T[]>`). В результате никогда не возникает неясность в том, на какую сущность указывает `std::unique_ptr`. API `std::unique_ptr` разработан так, чтобы соответствовать используемой разновидности. Например, в случае указателя для одного объекта отсутствует оператор индексирования (`operator[]`), в то время как в случае указателя для массива отсутствуют операторы разыменования (`operator*` и `operator->`).

Существование `std::unique_ptr` для массивов должно представлять только интеллектуальный интерес, поскольку `std::array`, `std::vector` и `std::string` почти всегда оказываются лучшим выбором, чем встроенные массивы. Я могу привести только одну ситуацию, когда `std::unique_ptr<T[]>` имеет смысл — при использовании C-образного API, который возвращает встроенный указатель на массив в динамической памяти, которым вы будете владеть.

Интеллектуальный указатель `std::unique_ptr` представляет собой способ выражения исключительного владения на C++11, но одной из наиболее привлекательных возможностей является то, что его можно легко и эффективно преобразовать в `std::shared_ptr`:

```
std::shared_ptr<Investment> sp = // Конвертация std::unique_ptr
makeInvestment( arguments ); // в std::shared_ptr
```

Это ключевой момент, благодаря которому `std::unique_ptr` настолько хорошо подходит для возвращаемого типа фабричных функций. Фабричные функции не могут знать, будет ли вызывающая функция использовать семантику исключительного владения возвращенным объектом или он будет использоваться совместно (т.е. `std::shared_ptr`). Возвращая `std::unique_ptr`, фабрики предоставляют вызывающим функциям наиболее эффективный интеллектуальный указатель, но не мешают им заменить этот указатель более гибким. (Информация об интеллектуальном указателе `std::shared_ptr` приведена в разделе 4.2.)

### Следует запомнить

- `std::unique_ptr` представляет собой маленький, быстрый, предназначенный только для перемещения интеллектуальный указатель для управления ресурсами с семантикой исключительного владения.
- По умолчанию освобождение ресурсов выполняется с помощью оператора `delete`, но могут применяться и пользовательские удалители. Удалители без состояний и указатели на функции в качестве удалителей увеличивают размеры объектов `std::unique_ptr`.
- Интеллектуальные указатели `std::unique_ptr` легко преобразуются в интеллектуальные указатели `std::shared_ptr`.

## 4.2. Используйте `std::shared_ptr` для управления ресурсами путем совместного владения

Программисты на языках программирования со сборкой мусора показывают пальцами на программистов на C++ и смеются над ними, потому что те озабочены предотвращением утечек ресурсов. “Какой примитив! — издеваются они. — Вы что, застряли в 1960-х годах и в Lisp? Управлять временем жизни ресурсов должны машины, а не люди”. Разработчики на C++ не остаются в долгу: “Вы имеете в виду, что единственным ресурсом является память, а время освобождения ресурса должно быть принципиально неопределенным? Спасибо, мы предпочитаем обобщенность и предсказуемость деструкторов!” Впрочем, в нашей браваде есть немного бахвальства. Сборка мусора в действительности достаточно удобна, а управление временем жизни вручную действительно может показаться похожим на создание микросхем памяти из шкуры медведя с помощью каменных ножей. Почему же не получается взять лучшее из двух миров — систему, которая работает автоматически (как сборка мусора) и к тому же применима ко всем ресурсам и имеет предсказуемое время выполнения (подобно деструкторам)?

Интеллектуальный указатель `std::shared_ptr` представляет собой способ, которым C++11 объединяет эти два мира. Объект, доступ к которому осуществляется через указатели `std::shared_ptr`, имеет время жизни, управление которым осуществляется этими указателями посредством *совместного владения*. Никакой конкретный указатель `std::shared_ptr` не владеет данным объектом. Вместо этого все указатели `std::shared_ptr`, указывающие на него, сотрудничают для обеспечения гарантии, что его уничтожение произойдет в точке, где он станет более ненужным. Когда последний указатель `std::shared_ptr`, указывающий на объект, прекратит на него указывать (например, из-за того, что этот `std::shared_ptr` будет уничтожен или перенаправлен на другой объект), этот `std::shared_ptr` уничтожит объект, на который он указывал. Как и в случае сборки мусора, клиентам не надо самим беспокоиться об управлении временем жизни объектов, на которые они указывают, но, как и при работе с деструкторами, время уничтожения объекта оказывается строго определенным.

Указатель `std::shared_ptr` может сообщить, является ли он последним указателем, указывающим на ресурс, с помощью счетчика ссылок, значения, связанного с ресурсом и отслеживающего, какое количество указателей `std::shared_ptr` указывает на него. Конструкторы `std::shared_ptr` увеличивают этот счетчик (обычно увеличивают — см. ниже), деструкторы `std::shared_ptr` уменьшают его, а операторы копирующего присваивания делают и то, и другое. (Если `sp1` и `sp2` являются указателями `std::shared_ptr`, указывающими на разные объекты, присваивание “`sp1 = sp2;`” модифицирует `sp1` так, что он указывает на объект, на который указывает `sp2`. Конечным результатом присваивания является то, что счетчик ссылок для объекта, на который изначально указывал `sp1`, уменьшается, а значение счетчика для объекта, на который указывает `sp2`, увеличивается.) Если `std::shared_ptr` после выполнения декремента видит нулевой счетчик ссылок, это означает, что на ресурс не указывает больше ни один `std::shared_ptr`, так что наш интеллектуальный указатель освобождает этот ресурс.

Наличие счетчиков ссылок влияет на производительность.

- **Размер std::shared\_ptr в два раза больше размера обычного указателя**, поскольку данный интеллектуальный указатель содержит обычный указатель на ресурс и другой обычный указатель на счетчик ссылок<sup>2</sup>.
- **Память для счетчика ссылок должна выделяться динамически**. Концептуально счетчик ссылок связан с объектом, на который он указывает, однако сам указываемый объект об этом счетчике ничего не знает. В нем нет места для хранения счетчика ссылок. (Приятным следствием этого является то, что интеллектуальный указатель std::shared\_ptr может работать с объектами любого типа (в том числе встроенных типов).) В разделе 4.4 поясняется, что можно избежать стоимости динамического выделения при создании указателя std::shared\_ptr с помощью вызова std::make\_shared, однако имеются ситуации, когда функция std::make\_shared неприменима. В любом случае счетчик ссылок хранится в динамически выделенной памяти.
- **Инкремент и декремент счетчика ссылок должны быть атомарными**, поскольку могут присутствовать одновременное чтение и запись в разных потоках. Например, std::shared\_ptr, указывающий на ресурс в одном потоке, может выполнять свой деструктор (тем самым уменьшая количество ссылок на указываемый им ресурс), в то время как в другом потоке указатель std::shared\_ptr на тот же объект может быть скопирован (а следовательно, увеличивает тот же счетчик ссылок). Атомарные операции обычно медленнее неатомарных, так что несмотря на то, что обычно счетчики ссылок имеют размер в одно слово, следует рассматривать их чтение и запись как относительно дорогостоящие операции.

Возбудил ли я ваше любопытство, когда написал, что конструкторы std::shared\_ptr “обычно” увеличивают счетчик ссылок для указываемого объекта? При создании std::shared\_ptr, указывающего на объект, всегда добавляется еще один интеллектуальный указатель std::shared\_ptr, так почему же счетчик ссылок может увеличиваться не всегда?

Из-за перемещающего конструирования — вот почему. Перемещающее конструирование указателя std::shared\_ptr из другого std::shared\_ptr делает исходный указатель нулевым, а это означает, что старый std::shared\_ptr перестает указывать на ресурс в тот же момент, в который новый std::shared\_ptr начинает это делать. В результате изменение значения счетчика ссылок не требуется. Таким образом, перемещение std::shared\_ptr оказывается быстрее копирования: копирование требует увеличения счетчика ссылок, а перемещение — нет. Это справедливо как для присваивания, так и для конструирования, так что перемещающее конструирование быстрее копирующего конструирования, а перемещающее присваивание быстрее копирующего присваивания.

<sup>2</sup> Стандарт не требует использования именно такой реализации, но все известные мне реализации стандартной библиотеки поступают именно так.

Подобно `std::unique_ptr` (см. раздел 4.1), `std::shared_ptr` в качестве механизма удаления ресурса по умолчанию использует `delete`, но поддерживает и пользовательские удалители. Однако дизайн этой поддержки отличается от дизайна для `std::unique_ptr`. Для `std::unique_ptr` тип удалителя является частью типа интеллектуального указателя. Для `std::shared_ptr` это не так:

```
auto loggingDel = [](Widget *pw) // Пользовательский удалитель
{
    // (как в разделе 4.1)
    makeLogEntry(pw);
    delete pw;
};

std::unique_ptr< // Тип удалителя является
Widget, decltype(loggingDel) // частью типа указателя
> upw(new Widget, loggingDel);

std::shared_ptr<Widget> // Тип удалителя не является
spw(new Widget, loggingDel); // частью типа указателя
```

Дизайн `std::shared_ptr` более гибок. Рассмотрим два указателя `std::shared_ptr<Widget>`, каждый со своим пользовательским удалителем разных типов (например, из-за того, что пользовательские удалители определены с помощью лямбда-выражений):

```
auto customDeleter1 = [](Widget *pw) { ... }; // Пользовательские
auto customDeleter2 = [](Widget *pw) { ... }; // удалители
                                                // разных типов

std::shared_ptr<Widget> pw1(new Widget, customDeleter1);
std::shared_ptr<Widget> pw2(new Widget, customDeleter2);
```

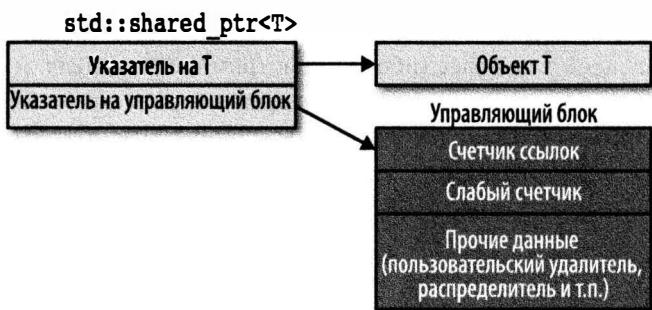
Поскольку `pw1` и `pw2` имеют один и тот же тип, они могут быть помещены в контейнер объектов этого типа:

```
std::vector<std::shared_ptr<Widget>> vpw{ pw1, pw2 };
```

Они также могут быть присвоены один другому и переданы функции, принимающей параметр типа `std::shared_ptr<Widget>`. Ни одно из этих действий не может быть выполнено с указателями `std::unique_ptr`, которые отличаются типами пользовательских удалителей, так как эти типы влияют на тип самого `std::unique_ptr`.

Другим отличием от `std::unique_ptr` является то, что указание пользовательского удалителя не влияет на размер объекта `std::shared_ptr`. Независимо от удалителя объект `std::shared_ptr` имеет размер, равный размеру двух указателей. Это хорошая новость, но она должна привести вас в замешательство. Пользовательские удалители могут быть функциональными объектами, а функциональные объекты могут содержать любое количество данных, а значит, быть любого размера. Как же `std::shared_ptr` может обращаться к удалителю произвольного размера, не используя при этом дополнительной памяти?

А он и не может. Ему приходится использовать дополнительную память, но эта память не является частью объекта `std::shared_ptr`. Она располагается в динамической памяти или, если создатель `std::shared_ptr` воспользуется поддержкой со стороны `std::shared_ptr` пользовательских распределителей памяти, там, где выделит память такой распределитель. Ранее я отмечал, что объект `std::shared_ptr` содержит указатель на счетчик ссылок для объекта, на который он указывает. Это так, но это немного не так, поскольку счетчик ссылок является частью большей структуры данных, известной под названием **управляющий блок** (*control block*). Управляющий блок имеется для каждого объекта, управляемого указателями `std::shared_ptr`. Управляющий блок в дополнение к счетчику ссылок содержит копию пользовательского удалителя, если таковой был указан. Если указан пользовательский распределитель памяти, управляющий блок содержит и его копию. Управляющий блок может также содержать дополнительные данные, включающие, как поясняется в разделе 4.4, вторичный счетчик ссылок, известный как слабый счетчик, но в данном разделе мы его игнорируем. Мы можем представить память, связанную с объектом `std::shared_ptr<T>`, как имеющую следующий вид:



Управляющий блок объекта настраивается функцией, создающей первый указатель `std::shared_ptr` на объект. Как минимум это то, что должно быть сделано. В общем случае функция, создающая указатель `std::shared_ptr` на некоторый объект, не может знать, не указывает ли на этот объект некоторый другой указатель `std::shared_ptr`, так что при создании управляющего блока должны использоваться следующие правила.

- Функция `std::make_shared` (см. раздел 4.4) всегда создает управляющий блок. Она производит новый объект, на который будет указывать интеллектуальный указатель, так что в момент вызова `std::make_shared` управляющий блок для этого объекта, определенно, не существует.
- Управляющий блок создается тогда, когда указатель `std::shared_ptr` создается из указателя с исключительным владением (т.е. `std::unique_ptr` или `std::auto_ptr`). Указатели с исключительным владением не используют управляющие блоки, так что никакого управляющего блока для указываемого объекта не существует. (Как часть своего построения `std::shared_ptr` осуществляет владение указываемым объектом, так что указатель с исключительным владением становится нулевым.)

- Когда конструктор `std::shared_ptr` вызывается с обычным указателем, он создает управляющий блок. Если вы хотите создать `std::shared_ptr` из объекта, у которого уже имеется управляющий блок, вы предположительно передаете в качестве аргумента конструктора `std::shared_ptr` или `std::weak_ptr` (см. раздел 4.3), а не обычный указатель. Конструкторы `std::shared_ptr`, принимающие в качестве аргументов указатели `std::shared_ptr` или `std::weak_ptr`, не создают новые управляющие блоки, поскольку могут воспользоваться управляющими блоками, на которые указывают переданные им интеллектуальные указатели.

Следствием из этих правил является то, что создание более одного `std::shared_ptr` из единственного обычного указателя дает вам бесплатный билет для путешествия в неопределенное поведение, поскольку в результате указываемый объект будет иметь несколько управляющих блоков. Несколько управляющих блоков означают несколько счетчиков ссылок, а несколько счетчиков ссылок означают, что объект будет удален несколько раз (по одному для каждого счетчика ссылок). И все это значит, что приведенный ниже код плох, ужасен, кошмарен:

```
auto pw = new Widget;      // pw – обычный указатель
...
std::shared_ptr<Widget>
    spw1(pw, loggingDel); // Создание управляющего блока для *pw
...
std::shared_ptr<Widget>    // Создание второго
    spw2(pw, loggingDel); // управляющего блока для *pw!
```

Создание обычного указателя `pw`, указывающего на динамически выделенный объект, — плохое решение, поскольку оно противоречит главному совету всей главы: предпочтите интеллектуальные указатели обычным указателям. (Если вы забыли, откуда взялся этот совет, заново прочтите первую страницу данной главы.) Но пока что забудем об этом. Страна, в которой создается `pw`, представляет собой стилистическую мерзость, но она по крайней мере не приводит к неопределенному поведению.

Далее вызывается конструктор для `spw1`, которому передается обычный указатель, так что этот конструктор создает управляющий блок (и тем самым счетчик ссылок) для того, на что он указывает. В данном случае это `*pw` (т.е. объект, на который указывает `pw`). Само по себе это не является чем-то страшным, но далее с тем же обычным указателем вызывается конструктор для `spw2`, и он также создает управляющий блок (а следовательно, и счетчик ссылок) для `*pw`. Объект `*pw`, таким образом, имеет два счетчика ссылок, каждый из которых в конечном итоге примет нулевое значение, и это обязательно приведет к попытке уничтожить объект `*pw` дважды. Второе уничтожение и будет ответственно за неопределенное поведение.

Из этого можно вынести как минимум два урока, касающиеся применения `std::shared_ptr`. Во-первых, пытайтесь избегать передачи обычных указателей конструкторы `std::shared_ptr`. Обычной альтернативой этому является применение функции `std::make_shared` (см. раздел 4.4), но в примере выше мы использовали пользовательские

удалители, а это невозможно при использовании `std::make_shared`. Во-вторых, если вы вынуждены передавать обычный указатель конструктору `std::shared_ptr`, передавайте непосредственно результат оператора `new`, а не используйте переменную в качестве посредника. Если первую часть кода переписать следующим образом,

```
std::shared_ptr<Widget>
spw1(new Widget, // Непосредственное использование new
      loggingDel);
```

то окажется гораздо труднее создать второй `std::shared_ptr` из того же обычного указателя. Вместо этого автор кода, создающего `spw2`, естественно, будет использовать в качестве аргумента инициализации `spw1` (т.е. будет вызывать копирующий конструктор `std::shared_ptr`), и это не будет вести ни к каким проблемам:

```
std::shared_ptr<Widget> // spw2 использует тот же
spw2(spw1);           // управляющий блок, что и spw1
```

Особенно удивительный способ, которым применение переменных с обычными указателями в качестве аргументов конструкторов `std::shared_ptr` может привести к множественным управляющим блокам, включает указатель `this`. Предположим, что наша программа использует указатели `std::shared_ptr` для управления объектами `Widget`, и у нас есть структура данных, которая отслеживает объекты `Widget`, которые были обработаны:

```
std::vector<std::shared_ptr<Widget>> processedWidgets;
```

Далее, предположим, что `Widget` имеет функцию-член, выполняющую эту обработку:

```
class Widget {
public:
    ...
    void process();
};
```

Вот вполне разумно выглядящий подход к написанию `Widget::process`:

```
void Widget::process()
{
    ...
    // Обработка Widget
    processedWidgets.emplace_back(this); // Добавление в список
}                                       // обработанных Widget,
                                         // это неправильно!
```

Комментарий о неправильности кода относится не к использованию `emplace_back`, а к передаче `this` в качестве аргумента. (Если вы не знакомы с `emplace_back`, обратитесь к разделу 8.2.) Этот код будет компилироваться, но он передает обычный указатель (`this`) контейнеру указателей `std::shared_ptr`. Конструируемый таким образом указатель `std::shared_ptr` будет создавать новый управляющий блок для указываемого

Widget (т.е. для объекта `*this`). Это не выглядит ужасным до тех пор, пока вы не понимаете, что, если имеются указатели `std::shared_ptr` вне функции-члена, которые уже указывают на этот объект Widget, вам не избежать неопределенного поведения.

API `std::shared_ptr` включает средство для ситуаций такого рода. Вероятно, его имя — наиболее странное среди всех имен стандартной библиотеки: `std::enable_shared_from_this`. Это шаблон базового класса, который вы наследуете, если хотите, чтобы класс, управляемый указателями `std::shared_ptr`, был способен безопасно создавать `std::shared_ptr` из указателя `this`. В нашем примере Widget будет унаследован от `std::enable_shared_from_this` следующим образом:

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
    ...
    void process();
};
```

Как я уже говорил, `std::enable_shared_from_this` является шаблоном базового класса. Его параметром типа всегда является имя производного класса, так что класс Widget порождается от `std::enable_shared_from_this<Widget>`. Если идея производного класса, порожденного от базового класса, шаблонизированного производным, вызывает у вас головную боль, попытайтесь об этом не думать. Этот код совершенно законный, а соответствующий шаблон проектирования хорошо известен, имеет стандартное имя, хотя и почти такое же странное, как `std::enable_shared_from_this`. Это имя — *Странно повторяющийся шаблон* (*The Curiously Recurring Template Pattern* — CRTP). Если вы хотите узнать о нем побольше, расчехлите свой поисковик, поскольку сейчас мы возвращаемся к нашему барабану по имени `std::enable_shared_from_this`.

Шаблон `std::enable_shared_from_this` определяет функцию-член, которая создает `std::shared_ptr` для текущего объекта, но делает это, не дублируя управляющие блоки. Это функция-член `shared_from_this`, и вы должны использовать ее в функциях-членах тогда, когда вам нужен `std::shared_ptr`, который указывает на тот же объект, что и указатель `this`. Вот как выглядит безопасная реализация `Widget::process`:

```
void Widget::process()
{
    // Как и ранее, обработка Widget
    ...
    // Добавляем std::shared_ptr, указывающий на
    // текущий объект, в processedWidgets
    processedWidgets.emplace_back(shared_from_this());
}
```

Внутри себя `shared_from_this` ищет управляющий блок текущего объекта и создает новый `std::shared_ptr`, который использует этот управляющий блок. Дизайн функции полагается на тот факт, что текущий объект имеет связанный с ним управляющий блок. Чтобы это было так, должен иметься уже существующий указатель `std::shared_ptr`

(например, за пределами функции-члена, вызывающей `shared_from_this()`), который указывает на текущий объект. Если такого `std::shared_ptr` нет (т.е. если текущий объект не имеет связанного с ним управляющего блока), результатом будет неопределенное поведение, хотя обычно `shared_from_this()` генерирует исключение.

Чтобы препятствовать клиентам вызывать функции-члены, в которых используется `shared_from_this()`, до того как на объект будет указывать указатель `std::shared_ptr`, классы, наследуемые от `std::enable_shared_from_this`, часто объявляют свои конструкторы как `private` и заставляют клиентов создавать объекты путем вызова фабричных функций, которые возвращают указатели `std::shared_ptr`. Например, класс `Widget` может выглядеть следующим образом:

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
    // Фабричная функция, пересылающая
    // аргументы закрытому конструктору:
    template<typename... Ts>
    static std::shared_ptr<Widget> create(Ts... params);
    ...
    void process(); // Как и ранее
    ...
private:
    // Конструкторы
};
```

В настоящее время вы можете только смутно припомнить, что наше обсуждение управляющих блоков было мотивировано желанием понять, с какими затратами связано применение `std::shared_ptr`. Теперь, когда мы понимаем, как избегать создания слишком большого количества управляющих блоков, вернемся к нашей первоначальной теме.

Управляющий блок обычно имеет размер в несколько слов, хотя пользовательские удалители и распределители памяти могут его увеличить. Обычная реализация управляющего блока оказывается более интеллектуальной, чем можно было бы ожидать. Она применяет наследование, и при этом даже имеются виртуальные функции. (Все это требуется для того, чтобы обеспечить корректное уничтожение указываемого объекта.) Это означает, что применение указателей `std::shared_ptr` берет на себя также стоимость механизма виртуальной функции, используемой управляющим блоком.

Возможно, после того как вы прочли о динамически выделяемых управляющих блоках, удалителях и распределителях неограниченного размера, механизме виртуальных функций и атомарности работы со счетчиками ссылок, ваш энтузиазм относительно `std::shared_ptr` несколько угас. Это нормально.

Они не являются наилучшим решением для любой задачи управления ресурсами. Но для предоставляемой ими функциональности цена `std::shared_ptr` весьма разумна. В типичных условиях, когда использованы удалитель и распределитель памяти по умолчанию, а `std::shared_ptr` создается с помощью `std::make_shared`, размер управляющего блока составляет около трех слов, и его выделение, по сути, ничего не стоит. (Оно

встроено в выделение памяти для указываемого им объекта. Дополнительная информация об этом приведена в разделе 4.4.) Разыменование `std::shared_ptr` не более дорогостояще, чем разыменование обычного указателя. Выполнение операций, требующих работы со счетчиком ссылок (например, копирующий конструктор или копирующее присваивание, удаление) влечет за собой одну или две атомарные операции, но эти операции обычно отображаются на отдельные машинные команды, так что, хотя они могут быть дороже неатомарных команд, они все равно остаются отдельными машинными командами. Механизм виртуальных функций в управляющем блоке обычно используется только однажды для каждого объекта, управляемого указателями `std::shared_ptr`: когда происходит уничтожение объекта.

В обмен на эти весьма скромные расходы вы получаете автоматическое управление временем жизни динамически выделяемых ресурсов. В большинстве случаев применение `std::shared_ptr` значительно предпочтительнее, чем ручное управление временем жизни объекта с совместным владением. Если вы сомневаетесь, можете ли вы позволить себе использовать `std::shared_ptr`, подумайте, точно ли вам нужно обеспечить совместное владение. Если вам достаточно или даже может быть достаточно исключительного владения, лучшим выбором является `std::unique_ptr`. Его профиль производительности близок к таковому для обычных указателей, а “обновление” `std::unique_ptr` до `std::shared_ptr` выполняется очень легко, так как указатель `std::shared_ptr` может быть создан из указателя `std::unique_ptr`.

Обратное неверно. После того как вы включили управление временем жизни ресурса с помощью `std::shared_ptr`, обратной дороги нет. Даже если счетчик ссылок равен единице, нельзя вернуть владение ресурсом для того, чтобы, скажем, им управлял `std::unique_ptr`. Контракт владения между ресурсом и указателями `std::shared_ptr`, которые указывают на ресурс, написан однозначно — “пока смерть не разлучит нас”. Никаких разводов и раздела имущества не предусмотрено.

Есть еще кое-что, с чем не могут справиться указатели `std::shared_ptr` — массивы. Это еще одно их отличие от указателей `std::unique_ptr`. Класс `std::shared_ptr` имеет API, предназначенное только для работы с указателями на единственные объекты. Не существует `std::shared_ptr<T[]>`. Время от времени “крутые” программисты натыкаются на мысль использовать `std::shared_ptr<T>` для указания на массив, определяя пользовательский удалитель для выполнения освобождения массива (т.е. `delete[]`). Это можно сделать и скомпилировать, но это ужасная идея. С одной стороны, класс `std::shared_ptr` не предоставляет оператор `operator[]`, так что индексирование указываемого массива требует неудобных выражений с применением арифметики указателей. С другой стороны, `std::shared_ptr` поддерживает преобразование указателей на производные классы в указатели на базовые классы, которое имеет смысл только для одиночных объектов, но при применении к массивам оказывается открытой дырой в системе типов. (По этой причине API `std::unique_ptr<T[]>` запрещает такие преобразования.) Что еще более важно, учитывая разнообразие альтернатив встроенным массивам в C++11 (например, `std::array`, `std::vector`, `std::string`), объявление интеллектуального указателя на тупой массив всегда является признаком плохого проектирования.

### Следует запомнить

- `std::shared_ptr` предоставляет удобный подход к управлению временем жизни произвольных ресурсов, аналогичный сборке мусора.
- По сравнению с `std::unique_ptr` объекты `std::shared_ptr` обычно в два раза больше, привносят накладные расходы на работу с управляющими блоками и требуют атомарной работы со счетчиками ссылок.
- Освобождение ресурсов по умолчанию выполняется с помощью оператора `delete`, однако поддерживаются и пользовательские удалители. Тип удалителя не влияет на тип указателя `std::shared_ptr`.
- Избегайте создания указателей `std::shared_ptr` из переменных, тип которых — обычный встроенный указатель.

## 4.3. Используйте `std::weak_ptr` для `std::shared_ptr`-подобных указателей, которые могут быть висячими

Парадоксально, но может быть удобно иметь интеллектуальный указатель, работающий как `std::shared_ptr` (см. раздел 4.2), но который не участвует в совместном владении ресурсом, на который указывает (другими словами, указатель наподобие `std::shared_ptr`, который не влияет на счетчик ссылок объекта). Эта разновидность интеллектуального указателя должна бороться с проблемой, неизвестной указателям `std::shared_ptr`: возможностью того, что объект, на который он указывает, был уничтожен. Истинный интеллектуальный указатель в состоянии справиться с этой проблемой, отслеживая, когда он становится **висячим**, т.е. когда объект, на который он должен указывать, больше не существует. Именно таковым является интеллектуальный указатель `std::weak_ptr`.

Вы можете удивиться, зачем может быть нужен указатель `std::weak_ptr`. Вы, вероятно, удивитесь еще больше, когда познакомитесь с его API. Указатель `std::weak_ptr` не может быть ни разыменован, ни проверен на “нулевость”. Дело в том, что `std::weak_ptr` не является автономным интеллектуальным указателем. Это — дополнение к `std::shared_ptr`.

Их взаимосвязь начинается с самого рождения: указатели `std::weak_ptr` обычно создаются из указателей `std::shared_ptr`. Они указывают на то же место, что и инициализирующие их указатели `std::shared_ptr`, но не влияют на счетчики ссылок объекта, на который указывают:

```
auto spw =                               // После создания spw счетчик
    std::make_shared<Widget>(); // ссылок указываемого Widget
                                // равен 1. (0 std::make_shared
                                // см. раздел 4.4.)  

...
std::weak_ptr<Widget> wpw(spw); // wpw указывает на тот же
                                // Widget, что и spw. Счетчик
```

```
// ссылок остается равным 1
...
spw = nullptr; // Счетчик ссылок равен 0, и
                // Widget уничтожается.
                // wpw становится висячим
```

О висячем `std::weak_ptr` говорят, что он *просрочен* (*expired*). Вы можете проверить это непосредственно:

```
if (wpw.expired()) ... // Если wpw не указывает на объект...
```

Но чаще всего вам надо не просто проверить, не просрочен ли указатель `std::weak_ptr`, но и, если он не просрочен (т.е. не является висячим), обратиться к объекту, на который он указывает. Это проще сказать, чем сделать. Поскольку у указателей `std::weak_ptr` нет операций разыменования, нет и способа написать такой код. Даже если бы он был, разделение проверки и разыменования могло бы привести к состоянию гонки: между вызовом `expired` и разыменованием другой поток мог бы переприсвоить или уничтожить последний `std::shared_ptr`, указывающий на объект, тем самым приводя к уничтожению самого объекта. В этом случае ваше разыменование привело бы к неопределенному поведению.

Что вам нужно — так это атомарная операция, которая проверяла бы просроченность указателя `std::weak_ptr` и, если он не просрочен, предоставляла вам доступ к указываемому объекту. Это делается путем создания указателя `std::shared_ptr` из указателя `std::weak_ptr`. Операция имеет две разновидности, в зависимости от того, что должно произойти в ситуации, когда `std::weak_ptr` оказывается просроченным при попытке создания из него `std::shared_ptr`. Одной разновидностью является `std::weak_ptr::lock`, которая возвращает `std::shared_ptr`. Этот указатель нулевой, если `std::weak_ptr` просрочен:

```
std::shared_ptr<Widget> spwl // Если wpw просрочен,
= wpw.lock(); // spwl — нулевой
auto spw2 = wpw.lock(); // То же самое, но с auto
```

Второй разновидностью является конструктор `std::shared_ptr`, принимающий `std::weak_ptr` в качестве аргумента. В этом случае, если `std::weak_ptr` просрочен, генерируется исключение:

```
std::shared_ptr<Widget> spw3(wpw); // Если wpw просрочен, генерируется std::bad_weak_ptr
```

Но вас, вероятно, интересует, зачем вообще нужен `std::weak_ptr`. Рассмотрим фабричную функцию, которая производит интеллектуальные указатели на объекты только для чтения на основе уникальных значений идентификаторов. В соответствии с советом из раздела 4.1, касающегося возвращаемых типов фабричных функций, она возвращает `std::unique_ptr`:

```
std::unique_ptr<const Widget> loadWidget(WidgetID id);
```

Если `loadWidget` является дорогостоящим вызовом (например, из-за файловых операций ввода-вывода или обращения к базе данных), а идентификаторы часто используются

повторно, разумной оптимизацией будет написание функции, которая делает то же, что и `loadWidget`, но при этом кеширует результаты. Засорение кеша всеми затребованными `Widget` может само по себе привести к проблемам производительности, так что другой разумной оптимизацией является удаление кешированных `Widget`, когда они больше не используются.

Для такой кеширующей фабричной функции возвращаемый тип `std::unique_ptr` не является удачным выбором. Вызывающий код, определенно, получает интеллектуальные указатели на кешированные объекты, и время жизни полученных объектов также определяется вызывающим кодом. Однако кеш также должен содержать указатели на эти же объекты. Указатели кеша должны иметь возможность обнаруживать свое висячее состояние, поскольку когда клиенты фабрики заканчивают работу с объектом, возвращенным ею, этот объект уничтожается, и соответствующая запись кеша становится висячей. Следовательно, кешированные указатели должны представлять собой указатели `std::weak_ptr`, которые могут обнаруживать, что стали висячими. Это означает, что возвращаемым типом фабрики должен быть `std::shared_ptr`, так как указатели `std::weak_ptr` могут обнаруживать, что стали висячими, только когда время жизни объектов управляется указателями `std::shared_ptr`.

Вот как выглядит быстрая и неаккуратная реализация кеширующей версии `loadWidget`:

```
std::shared_ptr<const Widget> fastLoadWidget(WidgetID id)
{
    static std::unordered_map<WidgetID,
        std::weak_ptr<const Widget>> cache;

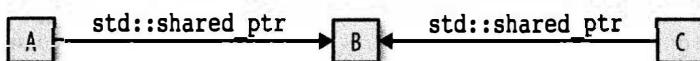
    auto objPtr           // objPtr является std::shared_ptr
        = cache[id].lock(); // для кешированного объекта и
        // нулевым указателем для объекта,
        // отсутствующего в кеше
    if (!objPtr)          // При отсутствии в кеше
        objPtr            // объект загружается
        = loadWidget(id);
    cache[id] = objPtr;   // и кешируется
}
return objPtr;
}
```

Эта реализация использует один из контейнеров C++11, представляющий собой хеш-таблицу (`std::unordered_map`), хотя здесь и не показаны хеширование `WidgetID` и функции сравнения, которые также должны присутствовать в коде.

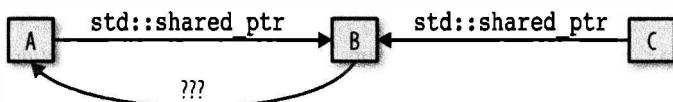
Реализация `fastLoadWidget` игнорирует тот факт, что кеш может накапливать просроченные указатели `std::weak_ptr`, соответствующие объектам `Widget`, которые больше не используются (а значит, были уничтожены). Реализация может быть улучшена, но вместо того чтобы тратить время на вопрос, который не привнесет ничего нового в понимание интеллектуальных указателей `std::weak_ptr`, давайте рассмотрим второе

применение этих указателей: шаблон проектирования Observer (Наблюдатель). Основными компонентами этого шаблона являются субъекты (объекты, которые могут изменяться) и наблюдатели (объекты, уведомляемые при изменении состояний). В большинстве реализаций каждый субъект содержит член-данные, хранящие указатели на его наблюдателей. Это упрощает для субъектов проблемы уведомления об изменении состояний. Субъекты не заинтересованы в управлении временем жизни своих наблюдателей (т.е. тем, когда они должны быть уничтожены), но они очень заинтересованы в том, чтобы, если наблюдатель был уничтожен, субъекты не пытались к нему обратиться. Разумным проектом может быть следующий — каждый субъект хранит контейнер указателей `std::weak_ptr` на своих наблюдателей, тем самым позволяя субъекту определять, не является ли указатель висящим, перед тем как его использовать.

В качестве последнего примера применения `std::weak_ptr` рассмотрим структуру данных с объектами A, B и C в ней, где A и C совместно владеют B, а следовательно, хранят указатели `std::shared_ptr` на нее:



Предположим, что было бы также полезно иметь указатель из B на A. Какую разновидность интеллектуального указателя следует использовать в этом случае?



Есть три варианта.

- **Обычный указатель.** При таком подходе, если уничтожается A, а C продолжает указывать на B, B будет содержать указатель на A, который становится висящим. В нее в состоянии этого определить, а потому B может непреднамеренно этот указатель разыменовать. В результате получается неопределенное поведение.
- **Указатель `std::shared_ptr`.** В этом случае A и B содержат указатели `std::shared_ptr` один на другой. Получающийся цикл `std::shared_ptr` (A указывает на B, а B указывает на A) предохраняет и A, и B от уничтожения. Даже если A и B недостижимы из других структур данных программы (например, поскольку C больше не указывает на B), счетчик ссылок каждого из них равен единице. Если такое происходит, A и B оказываются потерянными для всех практических применений: программа не в состоянии к ним обратиться, а их ресурсы не могут быть освобождены.
- **Указатель `std::weak_ptr`.** Это позволяет избежать обеих описанных выше проблем. Если уничтожается A, указатель в B становится висящим, но B в состоянии это обнаружить. Кроме того, хотя A и B указывают друг на друга, указатель в B не влияет на счетчик ссылок A, а следовательно, не может предотвратить удаление A, когда на него больше не указывает ни один `std::shared_ptr`.

Очевидно, что наилучшим выбором является `std::weak_ptr`. Однако стоит отметить, что необходимость применения указателей `std::weak_ptr` для предотвращения потенциальных циклов из указателей `std::shared_ptr` не является очень распространенным явлением. В строго иерархических структурах данных, таких как деревья, дочерними узлами обычно владеют их родительские узлы. При уничтожении родительского узла должны уничтожаться и его дочерние узлы. В общем случае связи от родительских к дочерним узлам лучше представлять указателями `std::unique_ptr`. Обратные связи от дочерних узлов к родительским можно безопасно реализовывать, как обычные указатели, поскольку дочерний узел никогда не должен иметь время жизни, большее, чем время жизни его родительского узла. Таким образом, отсутствует риск того, что дочерний узел разыменует висячий родительский указатель.

Конечно, не все структуры данных на основе указателей строго иерархичны, и когда приходится сталкиваться с такими неиерархичными ситуациями, как и с ситуациями наподобие кеширования или реализации списков наблюдателей, знайте, что у вас есть такой инструмент, как `std::weak_ptr`.

С точки зрения эффективности `std::weak_ptr`, по сути, такой же, как и `std::shared_ptr`. Объекты `std::weak_ptr` имеют тот же размер, что и объекты `std::shared_ptr`, они используют те же управляющие блоки, что и указатели `std::shared_ptr` (см. раздел 4.2), а операции, такие как создание, уничтожение и присваивание, включают атомарную работу со счетчиком ссылок. Вероятно, это вас удивит, поскольку в начале этого раздела я писал, что указатели `std::weak_ptr` не участвуют в подсчете ссылок. Но это не совсем то, что я написал. Я написал, что указатели `std::weak_ptr` не участвуют в совместном владении объектами, а следовательно, не влияют на счетчик ссылок указываемого объекта. На самом деле в управляющем блоке имеется второй счетчик ссылок, и именно с ним и работают указатели `std::weak_ptr`. Более подробно этот вопрос рассматривается в разделе 4.4.

#### Следует запомнить

- Используйте `std::weak_ptr` как `std::shared_ptr`-образные указатели, которые могут быть висячими.
- Потенциальные применения `std::weak_ptr` включают хеширование, списки наблюдателей и предупреждение циклов указателей `std::shared_ptr`.

## 4.4. Предпочтайте использование `std::make_unique` и `std::make_shared` непосредственному использованию оператора `new`

Начнем с выравнивания игровой площадки для игры `std::make_unique` и `std::make_shared` против обычных указателей. Функция `std::make_shared` является частью C++11, но, увы, `std::make_unique` таковой не является. Она вошла в стандарт только

начиная с C++14. Если вы используете C++11, не переживайте, потому что базовую версию std::make\_unique легко написать самостоятельно. Смотрите сами:

```
template<typename T, typename... Ts>
std::unique_ptr<T> make_unique(Ts&&... params)
{
    return std::unique_ptr<T>(new T(std::forward<Ts>(params)...));
}
```

Как вы можете видеть, make\_unique просто выполняет прямую передачу своих параметров в конструктор создаваемого объекта, создает std::unique\_ptr из обычного указателя, возвращаемого оператором new, и возвращает этот указатель std::unique\_ptr. Функция в данном виде не поддерживает массивы или пользовательские удалители (см. раздел 4.1), но зато демонстрирует, как с минимальными усилиями можно при необходимости создать make\_unique<sup>3</sup>. Только не помещайте вашу версию в пространство имен std, поскольку иначе вы можете столкнуться с коллизией имен при обновлении реализации стандартной библиотеки до C++14.

Функции std::make\_unique и std::make\_shared представляют собой две из трех make-функций, т.е. функций, которые принимают произвольное количество аргументов, выполняют их прямую передачу конструктору объекта, создаваемого в динамической памяти, и возвращают интеллектуальный указатель на этот объект. Третья make-функция — std::allocate\_shared. Она работает почти так же, как и std::make\_shared, за исключением того, что первым аргументом является объект распределителя, использующийся для выделения динамической памяти.

Даже самое тривиальное сравнение создания интеллектуального указателя с помощью make-функции и без участия таковой показывает первую причину, по которой применение таких функций является предпочтительным. Рассмотрим следующий код.

```
auto upwl(std::make_unique<Widget>()); // С make-функцией
std::unique_ptr<Widget> upw2(new Widget); // Без make-функции

auto spwl(std::make_shared<Widget>()); // С make-функцией
std::shared_ptr<Widget> spw2(new Widget); // Без make-функции
```

Я подчеркнул важное отличие: версия с применением оператора new повторяет созданный тип, в то время как make-функции этого не делают. Повторение типа идет вразрез с основным принципом разработки программного обеспечения: избегать дублирования кода. Дублирование в исходном тексте увеличивает время компиляции, может вести к раздатому объектному коду и в общем случае приводит к коду, с которым сложно работать. Зачастую это ведет к несогласованному коду, а несогласованности в коде часто ведут к ошибкам. Кроме того, чтобы набрать на клавиатуре что-то дважды, надо

<sup>3</sup> Для создания полнофункциональной версии make\_unique с минимальными усилиями поищите документ, ставший ее источником, и скопируйте из него ее реализацию. Этот документ — N3656 от 18 апреля 2013 года, его автор — Стивен Т. Лававей (Stephan T. Lavavej).

затратить в два раза больше усилий, чем для единственного набора, а кто из нас не любит сократить эту работу?

Второй причиной для предпочтения make-функций является безопасность исключений. Предположим, что у нас есть функция для обработки Widget в соответствии с некоторым приоритетом:

```
void processWidget(std::shared_ptr<Widget> spw, int priority);
```

Передача std::shared\_ptr по значению может выглядеть подозрительно, но в разделе 8.1 поясняется, что если processWidget всегда делает копию std::shared\_ptr (например, сохраняя ее в структуре данных, отслеживающей обработанные объекты Widget), то это может быть разумным выбором.

Предположим теперь, что у нас есть функция для вычисления приоритета

```
int computePriority();
```

и мы используем ее в вызове processWidget, который использует оператор new вместо std::make\_shared:

```
processWidget(                                // Потенциальная  
    std::shared_ptr<Widget>(new Widget),      // утечка  
    computePriority());                      // ресурса
```

Как указывает комментарий, этот код может приводить к утечке Widget, вызванной применением new. Но почему? И вызывающий код, и вызываемая функция используют указатели std::shared\_ptr, а std::shared\_ptr спроектированы для предотвращения утечек ресурсов. Они автоматически уничтожают то, на что указывают, когда исчезает последний std::shared\_ptr. Если все везде используют указатели std::shared\_ptr, о какой утечке может идти речь?

Ответ связан с тем, как компиляторы транслируют исходный код в объектный. Во время выполнения аргументы функции должны быть вычислены до вызова функции, так что в вызове processWidget до того, как processWidget начнет свою работу, должно произойти следующее.

- Выражение new Widget должно быть вычислено, т.е. в динамической памяти должен быть создан объект Widget.
- Должен быть вызван конструктор std::shared\_ptr<Widget>, отвечающий за управление указателем, сгенерированным оператором new.
- Должна быть вызвана функция computePriority.

Компиляторы не обязаны генерировать код, выполняющий перечисленные действия именно в таком порядке. Выражение new Widget должно быть выполнено до вызова конструктора std::shared\_ptr, поскольку результат этого оператора new используется в качестве аргумента конструктора, но функция computePriority может быть выполнена до указанных вызовов, после них или, что критично, между ними, т.е. компиляторы могут генерировать код для выполнения операций в следующем порядке.

1. Выполнить new Widget.
2. Выполнить computePriority.
3. Вызвать конструктор std::shared\_ptr.

Если сгенерирован такой код и во время выполнения computePriority генерирует исключение, созданный на первом шаге в динамической памяти Widget будет потерян, так как он не будет сохранен в указателе std::shared\_ptr, который, как предполагается, начнет управлять им на третьем шаге.

Применение std::make\_shared позволяет избежать таких проблем. Вызывающий код имеет следующий вид:

```
processWidget(std::make_shared<Widget>(), // Потенциальной
              computePriority());           // утечки ресурсов нет
```

Во время выполнения либо std::make\_shared, либо computePriority будет вызвана первой. Если это std::make\_shared, обычный указатель на созданный в динамической памяти Widget будет безопасно сохранен в возвращаемом указателе std::shared\_ptr до того, как будет вызвана функция computePriority. Если после этого функция computePriority генерирует исключение, деструктор std::shared\_ptr уничтожит объект Widget, которым владеет. А если первой будет вызвана функция computePriority и генерирует при этом исключение, то std::make\_shared даже не будет вызвана, так что не будет создан объект Widget, и беспокоиться будет не о чем.

Если мы заменим std::shared\_ptr и std::make\_shared указателем std::unique\_ptr и функцией std::make\_unique, все приведенные рассуждения останутся в силе. Использование std::make\_unique вместо new так же важно для написания безопасного с точки зрения исключений кода, как и применение std::make\_shared.

Особенностью std::make\_shared (по сравнению с непосредственным использованием new) является повышенная эффективность. Применение std::make\_shared позволяет компиляторам генерировать меньший по размеру и более быстрый код, использующий более компактные структуры данных. Рассмотрим следующее непосредственное применение оператора new:

```
std::shared_ptr<Widget> spw(new Widget);
```

Очевидно, что этот код предполагает выделение памяти, но фактически он выполняет два выделения. В разделе 4.2 поясняется, что каждый указатель std::shared\_ptr указывает на управляющий блок, содержащий, среди прочего, значение счетчика ссылок для указываемого объекта. Память для этого блока управления выделяется в конструкторе std::shared\_ptr. Непосредственное применение оператора new, таким образом, требует одного выделения памяти для Widget и второго — для управляющего блока.

Если вместо этого использовать std::make\_shared,

```
auto spw = std::make_shared<Widget>();
```

то окажется достаточно одного выделения памяти. Дело в том, что функция std::make\_shared выделяет один блок памяти для хранения как объекта Widget, так

и управляющего блока. Такая оптимизация снижает статический размер программы, поскольку код содержит только один вызов распределения памяти и повышает скорость работы выполнимого кода, так как выполняется только одно выделение памяти. Кроме того, применение `std::make_shared` устраняет необходимость в некоторой учетной информации в управляющем блоке, потенциально уменьшая общий объем памяти, требующийся для программы.

Анализ эффективности функции `std::make_shared` в равной мере применим и к `std::allocate_shared`, так что преимущество повышения производительности функции `std::make_shared` распространяется и на нее.

Аргументы в пользу предпочтения `make`-функций непосредственному использованию оператора `new` весьма существенны. Тем не менее, несмотря на их проектные преимущества, безопасность исключений и повышенную производительность, данный раздел говорит о *предпочтительном применении* `make`-функций, но не об их исключительном использовании. Дело в том, что существуют ситуации, когда эти функции не могут или не должны использоваться.

Например, ни одна из `make`-функций не позволяет указывать пользовательские удалители (см. разделы 4.1 и 4.2), в то время как конструкторы `std::unique_ptr` и `std::shared_ptr` это позволяют. Для данного пользовательского удалителя `Widget`

```
auto widgetDeleter = [](Widget* pw) { ... };
```

создание интеллектуального указателя с применением оператора `new` является очень простым:

```
std::unique_ptr<Widget, decltype(widgetDeleter)>
    upw(new Widget, widgetDeleter);
std::shared_ptr<Widget> spw(new Widget, widgetDeleter);
```

Сделать то же самое с помощью `make`-функции невозможно.

Второе ограничение на `make`-функции проистекает из синтаксических деталей их реализации. В разделе 3.1 поясняется, что при создании объекта, тип которого перегружает конструкторы как с параметрами `std::initializer_list`, так и без них, создание объекта с использованием фигурных скобок предпочтет конструктор `std::initializer_list`, в то время как создание объекта с использованием круглых скобок вызывает конструктор, у которого нет параметров `std::initializer_list`. `make`-функции выполняют прямую передачу своих параметров конструктору объекта, но делается ли это с помощью круглых или фигурных скобок? Для некоторых типов ответ на этот вопрос очень важен. Например, в вызовах

```
auto upv = std::make_unique<std::vector<int>>(10, 20);
auto spv = std::make_shared<std::vector<int>>(10, 20);
```

результатирующие интеллектуальные указатели должны указывать на векторы `std::vector` с 10 элементами, значение каждого из которых — 20, или на векторы с двумя элементами, один со значением 10, а второй со значением 20? Или результат непредсказуем?

Хорошая новость в том, что результат все же предсказуем: оба вызова создают векторы std::vector с 10 элементами, значение каждого из которых равно 20. Это означает что в make-функциях прямая передача использует круглые, а не фигурные скобки. Плохая новость в том, что если вы хотите создавать свои указываемые объекты с помощью инициализаторов в фигурных скобках, то должны использовать оператор new непосредственно. Использование make-функции требует способности прямой передачи инициализатора в фигурных скобках, но, как поясняется в разделе 5.8, такие инициализаторы не могут быть переданы прямо. Однако в разделе 5.8 описывается и обходной путь: использование вывода типа auto для создания объекта std::initializer\_list из инициализатора в фигурных скобках (см. раздел 1.2) с последующей передачей созданного объекта через make-функцию:

```
// Создание std::initializer_list
auto initList = { 10, 20 };

// Создание std::vector с помощью конструктора
// с параметром std::initializer_list
auto spv = std::make_shared<std::vector<int>>(initList);
```

Для std::unique\_ptr эти два сценария (пользовательские удалители и фигурные инициализаторы) являются единственными, когда применение make-функции оказывается проблематичным. Что касается std::shared\_ptr и его make-функций, то есть еще два сценария. Оба они являются крайними случаями, но некоторые разработчики постоянно ходят по краю, и вы можете быть одним из них.

Некоторые классы определяют собственные версии operator new и operator delete. Наличие этих функций подразумевает, что глобальные функции выделения и освобождения памяти для объектов этого типа являются неприемлемыми. Зачастую подпрограммы для конкретных классов разрабатываются только для выделения и освобождения блоков памяти, по размеру точно совпадающих с размером объектов класса; например, operator new и operator delete для класса Widget зачатую способны выделять и освобождать только блоки памяти размером sizeof(Widget). Такие подпрограммы плохо подходят для поддержки пользовательского распределения памяти для указателей std::shared\_ptr (с помощью std::allocate\_shared) и их удаления (с помощью пользовательских удалителей), поскольку количество запрашиваемой std::allocate\_shared памяти не совпадает с размером динамически создаваемого объекта (который равен размеру этого объекта плюс размер управляющего блока). Соответственно, применение make-функций для создания объектов типов со специфичными для данного класса версиями operator new и operator delete обычно является плохой идеей.

Преимущества размера и скорости функции std::make\_shared по сравнению с непосредственным применением оператора new вытекают из того факта, что управляющий блок указателя std::shared\_ptr размещается в том же блоке памяти, что и управляемый объект. Когда счетчик ссылок объекта становится равным нулю, объект уничтожается (т.е. вызывается его деструктор). Однако занятая им память не может быть освобождена до тех пор, пока не будет уничтожен и управляющий блок, поскольку блок динамически выделенной памяти содержит как объект, так и управляющий блок.

Как я уже отмечал, управляющий блок содержит, помимо самого счетчика ссылок, некоторую учетную информацию. Счетчик ссылок отслеживает, сколько указателей `std::shared_ptr` ссылаются на управляющий блок, но управляющий блок содержит и второй счетчик ссылок, который подсчитывает, сколько указателей `std::weak_ptr` ссылаются на этот управляющий блок. Этот второй счетчик ссылок известен как *слабый счетчик (weak count)*<sup>4</sup>. Когда указатель `std::weak_ptr` проверяет, не является ли он просроченным (см. раздел 4.2), он делает это путем обращения к счетчику ссылок (но не к слабому счетчику) в управляющем блоке, на который ссылается. Если счетчик ссылок равен нулю (т.е. если указываемый объект не имеет указателей `std::shared_ptr`, указывающих на него, и, таким образом, является удаленным), указатель `std::weak_ptr` является просроченным. В противном случае он просроченным не является.

Пока указатели `std::weak_ptr` указывают на управляющий блок (т.е. слабый счетчик больше нуля), этот управляющий блок должен продолжать существовать. А пока существует управляющий блок, память, его содержащая, должна оставаться выделенной. Таким образом, память, выделенная `make-функцией` для `std::shared_ptr` не может быть освобождена до тех пор, пока не будут уничтожены последний указатель `std::shared_ptr` и последний указатель `std::weak_ptr`, ссылающиеся на объект.

Если время между уничтожением последнего `std::shared_ptr` и последнего `std::weak_ptr` значительно, между уничтожением объекта и занимаемой им памятью может происходить задержка, что особенно важно для типов с большим размером:

```
class ReallyBigType { ... };

auto pBigObj =                               // Создание большого
    std::make_shared<ReallyBigType>(); // объекта с помощью
                                         // std::make_shared

// Создание указателей std::shared_ptr и std::weak_ptr
// на большой объект и работа с ними

// Уничтожение последнего указателя std::shared_ptr на
// этот объект; остаются указатели std::weak_ptr

// Во время этого периода память, ранее занятая большим
// объектом, остается занятой

// Уничтожение последнего указателя std::weak_ptr на
// объект; освобождение памяти, выделенной для
// управляющего блока и объекта
```

<sup>4</sup> На практике значение слабого счетчика не всегда совпадает с количеством указателей `std::weak_ptr`, ссылающихся на управляющий блок, поскольку разработчики библиотеки нашли способы добавлять в слабый счетчик дополнительную информацию, которая упрощает генерацию лучшего кода. В данном разделе мы игнорируем этот факт и считаем, что значение слабого счетчика представляет собой количество указателей `std::weak_ptr`, ссылающихся на управляющий блок.).

При непосредственном использовании new память для объекта ReallyBigType может быть освобождена, как только уничтожается последний указатель std::shared\_ptr, указывающий на него:

```
class ReallyBigType { ... }; // Как и ранее
                           // Создание очень большого объекта с помощью new:std::shared_
ptr<ReallyBigType> pBigObj(new ReallyBigType);

// Как и ранее, создание указателей std::shared_ptr и
// std::weak_ptr на объект и работа с ними

// Уничтожение последнего указателя std::shared_ptr на
// этот объект; остаются указатели std::weak_ptr
// Память, выделенная для объекта, освобождается

// Во время этого периода остается занятой только память,
// ранее выделенная для управляющего блока

// Уничтожение последнего указателя std::weak_ptr на
// объект; освобождение памяти, выделенной для
// управляющего блока
```

Оказавшись в ситуации, когда использование функции std::make\_shared невозможно или неприемлемо, вы можете захотеть защититься от ранее рассматривавшихся нами проблем, связанных с безопасностью исключений. Лучший способ сделать это — обеспечить немедленную передачу результата операции new конструктору интеллектуального указателя в инструкции, которая не делает ничего иного. Это предотвратит создание компилятором кода, который может генерировать исключение между оператором new и вызовом конструктора интеллектуального указателя, который будет управлять объектом, созданным оператором new.

В качестве примера рассмотрим небольшое изменение небезопасного с точки зрения исключений вызова функции processWidget, которую мы рассматривали ранее. В этот раз мы укажем пользовательский удалитель:

```
void processWidget(std::shared_ptr<Widget> spw, // Как и ранее
                  int priority);
void cusDel(Widget *ptr); // Пользовательский удалитель
```

Вот небезопасный с точки зрения исключений вызов:

```
processWidget(
    std::shared_ptr<Widget>(new Widget, cusDel), // потенциальная
    computePriority() // утечка
); // ресурса!
```

**Вспомним: если computePriority вызывается после new Widget, но до конструктора std::shared\_ptr и если computePriority генерирует исключение, происходит утечка динамически созданного Widget.**

Здесь применение пользовательского удалителя препятствует использованию std::make\_shared, так что избежать проблемы можно, поместив создание Widget в динамической памяти и конструирование std::shared\_ptr в собственную инструкцию, а затем вызвав функцию processWidget с передачей ей полученного std::shared\_ptr. Вот и вся суть этого метода, хотя, как мы вскоре увидим, его можно подкорректировать для повышения производительности:

```
std::shared_ptr<Widget> spw(new Widget, cusDel);
processWidget(spw, computePriority()); // Корректно, но не
                                         // оптимально; см. ниже
```

Этот код работает, поскольку std::shared\_ptr предполагает владение обычным указателем, переданным конструктору, даже если этот конструктор генерирует исключение. В данном примере, если конструктор spw генерирует исключение (например, из-за невозможности выделить динамическую память для управляющего блока), он все равно гарантирует вызов cusDel для указателя, полученного в результате операции new Widget.

Небольшое снижение производительности возникает из-за того, что в небезопасном с точки зрения исключений коде мы передавали функции processWidget rvalue

```
processWidget(
    std::shared_ptr<Widget>(new Widget, cusDel), // rvalue
    computePriority()
);
```

а в случае безопасного вызова — передаем lvalue:

```
processWidget(spw, computePriority()); // lvalue
```

Поскольку параметр std::shared\_ptr функции processWidget передается по значению, создание из rvalue влечет за собой только перемещение, в то время как создание из lvalue требует копирования. Для указателя std::shared\_ptr разница может оказаться существенной, так как копирование std::shared\_ptr требует атомарного инкремента счетчика ссылок, в то время как перемещение std::shared\_ptr не требует никаких действий со счетчиком ссылок вообще. Чтобы безопасный с точки зрения исключений код достиг уровня производительности небезопасного кода, нам надо применить std::move к spw для того, чтобы превратить его в rvalue (см. раздел 5.1):

```
processWidget(std::move(spw), // Эффективно и безопасно
              computePriority()); // в смысле исключений
```

Это интересный метод, и его стоит знать, но обычно это не имеет особого значения, поскольку вы будете редко сталкиваться с причинами не использовать make-функцию. Если у вас нет убедительных причин поступать иначе, используйте make-функции.

### Следует запомнить

- По сравнению с непосредственным использованием new, make-функции устраниют дублирование кода, повышают безопасность кода по отношению к исключениям и в случае функций std::make\_shared и std::allocate\_shared генерируют меньший по размеру и более быстрый код.
- Ситуации, когда применение make-функций неприемлемо, включают необходимость указания пользовательских удалителей и необходимость передачи инициализаторов в фигурных скобках.
- Для указателей std::shared\_ptr дополнительными ситуациями, в которых применение make-функций может быть неблагоразумным, являются классы с пользовательским управлением памятью и системы, в которых проблемы с объемом памяти накладываются на использование очень больших объектов и наличие указателей std::weak\_ptr, время жизни которых существенно превышает время жизни указателей std::shared\_ptr.

## 4.5. При использовании идиомы указателя на реализацию определяйте специальные функции-члены в файле реализации

Если вам приходилось бороться со слишком большим временем построения приложения, вы, вероятно, знакомы с идиомой *Pimpl* (pointer to implementation, указатель на реализацию). Это методика, при которой вы заменяете члены-данные класса указателем на класс (или структуру) реализации, помещаете в него члены-данные, использовавшиеся в основном классе, и обращаетесь к ним опосредованно через указатель. Предположим, например, что наш Widget имеет следующий вид:

```
class Widget {           // В заголовочном файле "widget.h"
public:
    Widget();
    ...
private:
    std::string name;
    std::vector<double> data;
    Gadget g1, g2, g3; // Gadget – некий пользовательский тип
};
```

Поскольку члены-данные Widget имеют типы std::string, std::vector и Gadget, для компиляции Widget должны присутствовать соответствующие заголовочные файлы, а это означает, что клиенты Widget должны включать с помощью директивы #include заголовочные файлы string, vector и gadget.h. Эти заголовочные файлы увеличивают время компиляции клиентов Widget, а также делают этих клиентов зависящими от содержимого указанных заголовочных файлов. Если содержимое заголовочного

файла изменяется, клиенты Widget должны быть перекомпилированы. Стандартные заголовочные файлы `string` и `vector` меняются не слишком часто, но заголовочный файл `gadget.h` вполне может оказаться подвержен частым изменениям.

Применение идиомы `Riml` в C++98 могло выполняться с помощью замены членовых-данных Widget обычным указателем на объявленную, но не определенную структуру:

```
class Widget {    // Все еще в заголовочном файле "widget.h"
public:
    Widget();
    ~Widget();    // Деструктор необходим (см. ниже)
    ...
private:
    struct Impl; // Объявление структуры реализации
    Impl *pImpl; // и указателя на нее
};
```

Поскольку Widget больше не упоминает типы `std::string`, `std::vector` и `Gadget`, клиенты Widget больше не обязаны включать соответствующие заголовочные файлы для этих типов. Это ускоряет компиляцию, а кроме того, означает, что если что-то в заголовочных файлах будет изменено, это не затронет клиенты Widget.

Тип, который был объявлен, но не определен, называется *неполным типом*. `Widget::Impl` является таким неполным типом. С неполным типом можно сделать очень немногое, но в это немногое входит объявление указателя на него. Идиома `Riml` использует эту возможность.

Первая часть идиомы `Riml` — объявление члена-данных, который представляет собой указатель на неполный тип. Вторая часть заключается в динамическом создании и уничтожении объекта, хранящего члены-данные, использующиеся в исходном классе. Соответствующий код находится в файле реализации, например для Widget — в файле `widget.cpp`:

```
#include "widget.h"           // Файл реализации "widget.cpp"
#include "gadget.h"
#include <string>
#include <vector>

struct Widget::Impl {          // Определение Widget::Impl
    std::string name;           // с членами-данными, ранее
    std::vector<double> data;   // находившимися в Widget
    Gadget g1, g2, g3;
};

Widget::Widget()               // Создание членов-данных
: pImpl(new Impl)             // для данного объекта Widget
{}
```

```
Widget::~Widget()           // Уничтожение членов-данных
{ delete pImpl; }          // для данного объекта
```

Здесь я привожу директивы `#include`, чтобы было ясно, что общие зависимости от заголовочных файлов для `std::string`, `std::vector` и `Gadget` никуда не исчезли и продолжают существовать. Однако эти зависимости перемещены из файла `widget.h` (видимого и используемого всеми клиентами класса `Widget`) в файл `widget.cpp` (видимый и используемый только реализацией `Widget`). Я также подчеркнул код динамического выделения и освобождения объекта `Impl`. Необходимость освобождения этого объекта при уничтожении `Widget` приводит к необходимости деструктора `Widget`.

Но я показал код C++98, от которого пахнет пылью вековой... Нет, пылью прошлого тысячелетия. Он использует обычные указатели, обычные операторы `new` и `delete`, и вообще весь он слишком сырой<sup>5</sup>. Вся текущая глава построена на идее о том, что интеллектуальные указатели куда предпочтительнее обычных указателей, и если мы хотим динамически создавать объект `Widget::Impl` в конструкторе `Widget` и должны уничтожать его вместе с `Widget`, то для нас отлично подойдет интеллектуальный указатель `std::unique_ptr` (см. раздел 4.1). Заменив обычный указатель `pImpl` указателем `std::unique_ptr`, мы получим следующий код для заголовочного файла

```
class Widget {                                // В файле "widget.h"
public:
    Widget();
    ...
private:
    struct Impl;
    std::unique_ptr<Impl> pImpl; // Интеллектуальный указатель
};                                              // вместо обычного
```

и для файла реализации:

```
#include "widget.h"                         // В файле "widget.cpp"
#include "gadget.h"
#include <string>
#include <vector>

struct Widget::Impl {                        // Как и ранее
    std::string name;
    std::vector<double> data;
    Gadget g1, g2, g3;
};

Widget::Widget()                           // Согласно разделу 4.4
: pImpl(std::make_unique<Impl>()) // создаем std::unique_ptr
{}                                         // с помощью std::make_unique
```

<sup>5</sup> Непереводимая игра слов, основанная на использовании для обычного указателя названия “raw pointer” (дословно — “сырой указатель”). — Примеч. пер.

Вы заметили, что деструктора Widget больше нет? Дело в том, что нет никакого кода, который требуется в нем разместить. std::unique\_ptr автоматически удаляет то, на что указывает, когда он сам (указатель std::unique\_ptr) уничтожается, так что нам не требуется ничего удалять вручную. Это одна из привлекательных сторон интеллектуальных указателей: они устраниют необходимость утруждать свои руки вводом кода для освобождения ресурсов вручную.

Этот код компилируется, но, увы, это не самый тривиальный клиент!

```
#include "widget.h"
Widget w; // Ошибка!
```

Получаемое вами сообщение об ошибке зависит от используемого компилятора, но в общем случае текст упоминает что-то о применении sizeof или delete к неполному типу. Эти операции не входят в число тех, которые можно делать с такими типами.

Эта явная неспособность идиомы R<sup>1</sup>impl использовать std::unique\_ptr вызывает тревогу, поскольку (1) указатели std::unique\_ptr разрекламированы как поддерживающие неполные типы, и (2) идиома R<sup>1</sup>impl — один из наиболее распространенных случаев применения std::unique\_ptr. К счастью, этот код легко сделать работающим. Все, что для этого требуется, — понимание причины проблемы.

Проблема возникает из-за кода, который генерируется при уничтожении w (например, при выходе переменной за пределы области видимости). В этой точке вызывается ее деструктор. Если определение класса использует std::unique\_ptr, мы не объявляем деструктор, так как нам нечего в него поместить. В соответствии с обычными правилами генерации специальных функций-членов компиляторами (см. раздел 3.11) этот деструктор создается вместо нас компилятором. В этот деструктор компилятор вносит код вызова деструктора члена-данных pImpl класса Widget. pImpl представляет собой указатель std::unique\_ptr<Widget::Impl>, т.е. указатель std::unique\_ptr, использующий удалитель по умолчанию. Удалитель по умолчанию является функцией, которая применяет оператор delete к обычному указателю внутри std::unique\_ptr. Однако перед тем как использовать delete, реализации удалителя по умолчанию в C++11 обычно применяют static\_assert, чтобы убедиться, что обычный указатель не указывает на неполный тип. Когда компилятор генерирует код для деструкции Widget w, он в общем случае сталкивается с неудачным static\_assert, что и приводит к выводу сообщения об ошибке. Это сообщение обычно связано с точкой, в которой происходит уничтожение w, поскольку деструктор Widget, подобно всем генерируемым компиляторами специальным функциям-членам, неявно является inline. Сообщение часто указывает на строку, в которой создается w, поскольку она представляет собой исходный текст, явно создающий объект, приводящий впоследствии к неявной деструкции.

Для исправления ситуации надо просто обеспечить полноту типа Widget::Impl в точке, где генерируется код, уничтожающий std::unique\_ptr<Widget::Impl>. Тип становится полным, когда его определение становится видимым, а Widget::Impl определен в файле widget.cpp. Ключом к успешной компиляции является требование, чтобы компилятор видел тело деструктора Widget (т.е. место, где компилятор будет генерировать

код для уничтожения члена-данных std::unique\_ptr) только внутри widget.cpp, после определения Widget::Impl.

Добиться этого просто. Объявим деструктор Widget в widget.h, но не будем определять его там:

```
class Widget {      // Как и ранее, в файле "widget.h"
public:
    Widget();
    ~Widget();      // Только объявление
    ...
private:           // Как и ранее
    struct Impl;
    std::unique_ptr<Impl> pImpl;
};
```

Определим его в widget.cpp после определения Widget::Impl:

```
#include "widget.h"          // Как и ранее, в файле "widget.cpp"
#include "gadget.h"
#include <string>
#include <vector>

struct Widget::Impl {        // Как и ранее, определение
    std::string name;        // Widget::Impl
    std::vector<double> data;
    Gadget g1, g2, g3;
};

Widget::Widget()            // Как и ранее
: pImpl(std::make_unique<Impl>())
{}

Widget::~Widget()          // Определение ~Widget
{}
```

Это хорошо работает и требует небольшого набора текста, но если вы хотите подчеркнуть, что генерируемый компилятором деструктор работает верно, что единственная причина его объявления — генерация его определения в файле реализации Widget, то вы можете определить тело деструктора как = default:

```
Widget::~Widget() = default; // Тот же результат, что и выше
```

Классы, использующие идиому Rimpl, являются естественными кандидатами на поддержку перемещения, поскольку генерируемые компилятором операции перемещения делают именно то, что требуется: выполняют перемещение std::unique\_ptr. Как поясняется в разделе 3.11, объявление деструктора Widget препятствует генерации компилятором операций перемещения, так что, если вы хотите обеспечить их поддержку, вы

должны объявить их самостоятельно. Поскольку генерируемый компилятором версии ведут себя так, как надо, соблазнительно реализовать их следующим образом:

```
class Widget {                                     // В "widget.h"
public:
    Widget();
    ~Widget();
    Widget(Widget&& rhs) = default;           // Идея верна,
    Widget& operator=(Widget&& rhs) = default; // код - нет!
    ...
private:                                         // Как и ранее
    struct Impl;
    std::unique_ptr<Impl> pImpl;
};
```

Этот подход приводит к тем же проблемам, что и объявление класса без деструктора, и по той же самой причине. Генерируемый компилятором оператор перемещающего присваивания должен уничтожить объект, на который указывает pImpl, перед тем как присвоить указателю новое значение, но в заголовочном файле Widget указатель pImpl указывает на неполный тип. Ситуация отличается для перемещающего конструктора. Проблема в том, что компиляторы обычно генерируют код для уничтожения pImpl в том случае, когда в перемещающем конструкторе генерируется исключение, а уничтожение pImpl требует, чтобы тип Impl был полным.

Поскольку проблема точно такая же, как и ранее, то и решение ее такое же — перенос определений перемещающих операций в файл реализации:

```
class Widget {                                     // В файле "widget.h"
public:
    Widget();
    ~Widget();
    Widget(Widget&& rhs);          // Только объявления
    Widget& operator=(Widget&& rhs);
    ...
private:                                         // Как и ранее
    struct Impl;
    std::unique_ptr<Impl> pImpl;
};

#include <string>                                // В файле "widget.cpp"

struct Widget::Impl { ... };                      // Как и ранее

Widget::Widget() // as before
: pImpl(std::make_unique<Impl>())
{}
```

```

Widget::~Widget() = default; // Как и ранее

// Определения:
Widget::Widget(Widget&& rhs) = default;
Widget& Widget::operator=(Widget&& rhs) = default;

```

Идиома Pimpl представляет собой способ снижения зависимости между реализацией класса и его клиентами, но концептуально идиома не меняет то, что представляет собой класс. Исходный класс Widget содержал члены-данные std::string, std::vector и Gadget, так что в предположении, что объекты Gadget, как и объекты std::string и std::vector, могут копироваться, имеет смысл в поддержке классом Widget копирующих операций. Мы должны написать эти функции самостоятельно, поскольку (1) компиляторы не генерируют копирующие операции для классов с типами, поддерживающими только перемещение (наподобие std::unique\_ptr) и (2) даже если бы они генерировались, то такие функции выполняли бы копирование только указателя std::unique\_ptr (т.е. выполняли бы мелкое копирование), а мы хотим копировать то, на что указывает этот указатель (т.е. выполнять глубокое копирование).

В соответствии с ритуалом, который нам теперь хорошо знаком, мы объявляем функции в заголовочном файле и реализуем их в файле реализации:

```

class Widget { // В файле "widget.h"
public:
    ...
    Widget(const Widget& rhs); // Прочее, как ранее
    Widget& operator=(const Widget& rhs); // Только
                                            // объявления
private: // Как и ранее
    struct Impl;
    std::unique_ptr<Impl> pImpl;
};

#include "widget.h" // В "widget.cpp"
...
struct Widget::Impl { ... }; // Как и ранее

Widget::~Widget() = default; // Прочее, как и ранее

Widget::Widget(const Widget& rhs) // Копирующий конструктор
: pImpl(nullptr)
{ if (rhs.pImpl) pImpl = std::make_unique<Impl>(*rhs.pImpl); }

                                            // Копирующее присваивание:
Widget& Widget::operator=(const Widget& rhs)
{
    if (!rhs.pImpl) pImpl.reset();
    else if (!pImpl) pImpl = std::make_unique<Impl>(*rhs.pImpl);
}

```

```

    else *pImpl = *rhs.pImpl;

    return *this;
}

```

Реализация достаточно проста, хотя мы и должны обрабатывать случаи, в которых параметр rhs или, в случае копирующего оператора присваивания, `*this` был перемещен, а потому содержит нулевой указатель `pImpl`. В общем случае мы используем тот факт, что компиляторы создают копирующие операции для `Impl`, и эти операции копируют каждое поле автоматически. Так что мы реализуем копирующие операции `Widget` путем вызова копирующих операций `Widget::Impl`, сгенерированных компилятором. В обеих функциях обратите внимание, как мы следуем совету из раздела 4.4 предпочтительность применения `std::make_unique` непосредственной работе с `new`.

При реализации идиомы `Pimpl` используемым интеллектуальным указателем является `std::unique_ptr`, поскольку указатель `pImpl` внутри объекта (например, внутри `Widget`) имеет исключительное владение соответствующим объектом реализации (например, объектом `Widget::Impl`). Интересно также отметить, что если бы мы использовали для `pImpl` указатель `std::shared_ptr` вместо `std::unique_ptr` (т.е. если бы значения в структуре `Impl` могли совместно использоваться несколькими `Widget`), то нашли бы, что советы из данного раздела больше не применимы. Нам бы не потребовалось объявлять деструктор в `Widget`, а без пользовательского деструктора компиляторы с удовольствием генерировали бы операции перемещения, которые делали бы именно то, что от них требуется. То есть при следующем коде в файле `widget.h`

```

class Widget {                                // В файле "widget.h"
public:
    Widget();                                // Нет объявлений деструктора
                                                // и перемещающих операций

private:
    struct Impl;
    std::shared_ptr<Impl> pImpl; // std::shared_ptr
                                // вместо std::unique_ptr
};

```

и приведенном далее коде клиента, который включает заголовочный файл `widget.h`

```

Widget w1;
auto w2(std::move(w1)); // Перемещающее конструирование w2
w1 = std::move(w2);   // Перемещающее присваивание w1

```

все компилировалось бы и работало именно так, как мы рассчитывали: `w1` был бы создан конструктором по умолчанию, его значение было бы перемещено в `w2`, а затем это значение, в свою очередь, было бы перемещено в `w1`, после чего и `w1`, и `w2` были бы уничтожены (тем самым приводя к уничтожению объекта `Widget::Impl`).

Различие в поведении указателей `std::unique_ptr` и `std::shared_ptr` для `pImpl` вытекает из различий путей, которыми эти интеллектуальные указатели поддерживают

пользовательские удалители. Для `std::unique_ptr` тип удалителя является частью типа интеллектуального указателя, и это позволяет компилятору генерировать меньшие структуры данных времени выполнения и более быстрый код. Следствием этой более высокой эффективности является то, что указываемые типы должны быть полными, когда используются специальные функции-члены, генерируемые компиляторами (например, деструкторы или перемещающие операции). В случае `std::shared_ptr` тип удалителя не является частью типа интеллектуального указателя. Это требует больших структур данных времени выполнения и несколько более медленного кода, но зато указываемые типы не обязаны быть полными при применении специальных функций-членов, генерируемых компиляторами.

При применении идиомы `Rimpl` в действительности нет никакого компромисса между характеристиками `std::unique_ptr` и `std::shared_ptr`, поскольку отношения между классами наподобие `Widget` и `Widget::Impl` представляют собой исключительное владение, и это делает единственно верным выбором в качестве инструмента интеллектуальный указатель `std::unique_ptr`. Тем не менее стоит знать, что в других ситуациях — ситуациях, в которых осуществляется совместное владение (а следовательно, правильным выбором является `std::shared_ptr`), — нет необходимости прыгать через горящие обручи определений функций, которую влечет за собой применение `std::unique_ptr`.

### Следует запомнить

- Идиома `Rimpl` уменьшает время построения приложения, снижая зависимости компиляции между клиентами и реализациями классов.
- Для указателей `rimpl` типа `std::unique_ptr` следует объявлять специальные функции-члены в заголовочном файле, но реализовывать их в файле реализации. Поступайте так, даже если реализации функций по умолчанию являются приемлемыми.
- Приведенный выше совет применим к интеллектуальному указателю `std::unique_ptr`, но не к `std::shared_ptr`.



# Rvalue-ссылки, семантика перемещений и прямая передача

На первый взгляд, семантика перемещения и прямой передачи кажется довольно простой.

- **Семантика перемещения** позволяет компиляторам заменять дорогостоящие операции копирования менее дорогими перемещениями. Так же, как копирующие конструкторы и копирующие операторы присваивания дают вам контроль над тем, что означает копирование объектов, так и перемещающие конструкторы и перемещающие операторы присваивания предоставляют контроль над семантикой перемещения. Семантика перемещения позволяет также создавать типы, которые могут только перемещаться, такие как `std::unique_ptr`, `std::future` или `std::thread`.
- **Прямая передача** делает возможным написание шаблонов функций, которые принимают произвольные аргументы и передают их другим функциям так, что целевые функции получают в точности те же аргументы, что и переданные исходным функциям.

Rvalue-ссылки представляют собой тот клей, который соединяет две довольно разные возможности. Это базовый механизм языка программирования, который делает возможными как семантику перемещения, так и прямую передачу.

С ростом опыта работы с этими возможностями вы все больше понимаете, что ваше первоначальное впечатление было основано только на пресловутой вершине айсберга. Мир семантики перемещения, прямой передачи и rvalue-ссылок имеет больше нюансов, чем кажется на первый взгляд. Например, `std::move` ничего не перемещает, а прямая передача оказывается не совсем прямой. Перемещающие операции не всегда дешевле копирования, а когда и дешевле, то не всегда настолько, как вы думаете; кроме того, они не всегда вызываются в контексте, где перемещение является корректным. Конструкция `type&&` не всегда представляет rvalue-ссылку.

Независимо от того, как глубоко вы закопались в эти возможности, может показаться, что можно долго копать еще глубже. К счастью, эта глубина не безгранична. Эта глава доведет вас до коренной породы. Когда вы докопаетесь до нее, эта часть C++11 будет

выглядеть намного более осмысленной. Например, вы познакомитесь с соглашениями по использованию `std::move` и `std::forward`. Вы почувствуете себя намного более комфортно, сталкиваясь с неоднозначной природой `type&&`. Вы поймете причины удивительно разнообразного поведения перемещающих операций. Все фрагменты мозаики встанут на свои места. В этот момент вы окажетесь там, откуда начинали, потому что семантика перемещений, прямая передача и `rvalue`-ссылки вновь покажутся вам достаточно простыми. Но на этот раз они будут оставаться для вас такими навсегда.

В этой главе особенно важно всегда иметь в виду, что параметр всегда является `lvalue`, даже если его тип — `rvalue`-ссылка. Иными словами, в фрагменте

```
void f(Widget&& w);
```

параметр `w` представляет собой `lvalue`, несмотря на то что его тип — `rvalue`-ссылка на `Widget`. (Если это вас удивляет, вернитесь к обзору `lvalue` и `rvalue`, который содержится во введении.)

## 5.1. Азы `std::move` и `std::forward`

Полезно подойти к `std::move` и `std::forward` с точки зрения того, чего они *не делают*. `std::move` ничего не перемещает. `std::forward` ничего не передает. Во время выполнения они не делают вообще ничего. Они не генерируют выполнимый код — ни одного байта.

`std::move` и `std::forward` являются всего лишь функциями (на самом деле — шаблонами функций), которые выполняют приведения. `std::move` выполняет безусловное приведение своего аргумента к `rvalue`, в то время как `std::forward` выполняет приведение только при соблюдении определенных условий. Это все. Пояснения приводят к новому множеству вопросов, но, по сути, история на этом завершена.

Чтобы сделать историю более конкретной, рассмотрим пример реализации `std::move` в C++11. Она не полностью соответствует деталям стандарта, но очень близка к этому.

```
template<typename T> // В пространстве имен std
typename remove_reference<T>::type&&
move(T&& param)
{
    using ReturnType = // Объявление псевдонима; см. раздел 3.3
        typename remove_reference<T>::type&&;
    return static_cast<ReturnType>(param);
}
```

Я выделил здесь две части кода. Одна — имя функции, потому что спецификация возвращаемого типа достаточно запутанна, и я бы не хотел, чтобы вы в ней заблудились. Вторая — приведение, которое составляет сущность функции. Как вы можете видеть, `std::move` получает ссылку на объект (чтобы быть точным — универсальную ссылку; см. раздел 5.2) и возвращает ссылку на тот же объект.

Часть `&&` возвращаемого типа функции предполагает, что `std::move` возвращает `rvalue`-ссылку, но, как поясняется в разделе 5.6, если тип `T` является `lvalue`-ссылкой, `T&&`

становится lvalue-ссылкой. Чтобы этого не произошло, к `T` применяется свойство типа (см. раздел 3.3) `std::remove_reference`, тем самым обеспечивая применение “`&&`” к типу, не являющемуся ссылкой. Это гарантирует, что `std::move` действительно возвращает rvalue-ссылку, и это важно, поскольку rvalue-ссылки, возвращаемые функциями, являются rvalue. Таким образом, `std::move` приводит свой аргумент к rvalue, и это все, что она делает.

В качестве небольшого отступления скажем, что `std::move` можно реализовать в C++14 меньшими усилиями. Благодаря выводу возвращаемого типа функции (см. раздел 1.3) и шаблону псевдонима `std::remove_reference_t` (см. раздел 3.3) `std::move` можно записать следующим образом:

```
template<typename T> // C++14; находится в
decltype(auto) move(T&& param) // пространстве имен std
{
    using ReturnType = remove_reference_t<T>&&;
    return static_cast<ReturnType>(param);
}
```

Легче для восприятия, не так ли?

Поскольку `std::move` ничего не делает, кроме приведения своего аргумента к rvalue, были предложения дать этому шаблону другое имя, например `rvalue_cast`. Как бы там ни было, у нас имеется имя `std::move`, так что нам важно запомнить, что это имя `std::move` делает и чего не делает. Итак, оно выполняет приведение. И оно ничего не переносит.

Конечно, rvalue являются кандидатами на перемещение, поэтому применение `std::move` для объекта сообщает компилятору, что объект предназначается для перемещения. Вот почему `std::move` имеет такое имя: чтобы легко распознавать объекты, которые могут быть перемещены.

По правде говоря, rvalue *обычно* являются всего лишь кандидатами для перемещения. Предположим, что вы пишете класс, представляющий аннотации. Конструктор класса получает параметр `std::string`, представляющий аннотацию, и копирует значение параметра в член-данные. С учетом информации из раздела 8.1 вы объявляете параметр как передаваемый по значению:

```
class Annotation {
public:
    explicit Annotation(std::string text); // Параметр
                                            // копируемый, так что согласно
                                            // разделу 8.1 он передается по значению
};
```

Но конструктору `Annotation` требуется только прочесть значение `text`. Ему не нужно его модифицировать. В соответствии с освященной веками традицией использования `const` везде, где только можно, вы переписываете свое объявление, делая `text` константой:

```
class Annotation {
public:
```

```
    explicit Annotation(const std::string text);  
};
```

Чтобы избежать дорогостоящей операции копирования `text` в член-данные, вы оставляете в силе совет из раздела 8.1 и применяете `std::move` к `text`, тем самым получая `rvalue`:

```
class Annotation {  
public:  
    explicit Annotation(const std::string text)  
        : value(std::move(text)) // "Перемещение" text в value;  
    { ... } // этот код не делает того,  
    ... // что от него ожидается!  
private:  
    std::string value;  
};
```

Этот код компилируется. Этот код компонуется. Этот код выполняется. Этот код устанавливает значение члена-данных `value` равным содержимому строки `text`. Единственное, что отличает этот код от идеальной реализации ваших намерений, — то, что `text` не перемещается в `value`, а *копируется*. Конечно, `text` приводится к `rvalue` с помощью `std::move`, но `text` объявлен как `const std::string`, так что перед приведением `text` являлся `lvalue` типа `const std::string`, так что результатом приведения является `rvalue` типа `const std::string`, и на протяжении всех этих действий константность сохраняется.

Рассмотрим, как компиляторы определяют, какой из конструкторов `std::string` должен быть вызван. Есть две возможности:

```
class string { // std::string в действительности представляет  
public:       // собой typedef для std::basic_string<char>  
    ...  
    string(const string& rhs); // Копирующий конструктор  
    string(string&& rhs);    // Перемещающий конструктор  
  
};
```

В списке инициализации членов конструктора `Annotation` результатом `std::move(text)` является `rvalue` типа `const std::string`. Это `rvalue` нельзя передать перемещающему конструктору `std::string`, поскольку перемещающий конструктор получает `rvalue`-ссылку на неконстантную `std::string`. Однако это `rvalue` может быть передано копирующему конструктору, поскольку `lvalue`-ссылку на `const` разрешено связывать с константным `rvalue`. Таким образом, инициализация члена использует *копирующий* конструктор `std::string`, несмотря на то что `text` был приведен к `rvalue`! Такое поведение имеет важное значение для поддержания корректности `const`. Перемещение значения из объекта в общем случае модифицирует этот объект, так что язык программирования должен не разрешать передавать константные объекты в функции (такие, как перемещающие конструкторы), которые могут их модифицировать.

Из этого примера следует извлечь два урока. Во-первых, не объявляйте объекты как константные, если хотите иметь возможность выполнять перемещение из них. Запрос перемещения к константным объектам молча трансформируется в копирующие операции. Во-вторых, std::move не только ничего не перемещает самостоятельно, но даже не гарантирует, что приведенный этой функцией объект будет иметь право быть перемещенным. Единственное, что точно известно о результате применения std::move к объекту, — это то, что он является rvalue.

История с std::forward подобна истории с std::move, но тогда как std::move выполняет *безоговорочное* приведение своего аргумента в rvalue, std::forward делает это только при определенных условиях. std::forward является *условным приведением*. Чтобы понять, когда приведение выполняется, а когда нет, вспомним, как обычно используется std::forward. Наиболее распространенным сценарием является тот, когда шаблон функции получает параметр, представляющий собой универсальную ссылку, и который передается другой функции:

```
void process(const Widget& lvalArg); // Обработка lvalue
void process(Widget&& rvalArg);      // Обработка rvalue

template<typename T>                  // Шаблон, передающий
void logAndProcess(T&& param)        // param на обработку
{
    auto now =                         // Получает текущее время
        std::chrono::system_clock::now();
    makeLogEntry("Вызов 'process'", now);
    process(std::forward<T>(param));
}
```

Рассмотрим два вызова logAndProcess, один с lvalue, а другой — с rvalue:

```
Widget w;

logAndProcess(w);           // Вызов с lvalue
logAndProcess(std::move(w)); // Вызов с rvalue
```

В функции logAndProcess параметр param передается функции process. Функция process перегружена для lvalue и rvalue. Вызывая logAndProcess с lvalue, мы, естественно, ожидаем, что lvalue будет передано функции process как lvalue, а вызывая logAndProcess с rvalue, мы ожидаем, что будет вызвана перегрузка process для rvalue.

Однако param, как и все параметры функций, является lvalue. Каждый вызов process внутри logAndProcess будет, таким образом, вызывать перегрузку process для lvalue. Для предотвращения такого поведения нам нужен механизм для приведения param к rvalue тогда и только тогда, когда аргумент, которым инициализируется param — аргумент, переданный logAndProcess, — был rvalue. Именно этим и занимается std::forward. Вот почему std::forward представляет собой *условное приведение*: эта функция выполняет приведение к rvalue только тогда, когда ее аргумент инициализирован rvalue.

Вы можете удивиться, откуда `std::forward` может знать, был ли ее аргумент инициализирован `rvalue`? Например, как в приведенном выше коде `std::forward` может сказать, был ли `param` инициализирован с помощью `lvalue` или `rvalue`? Краткий ответ заключается в том, что эта информация кодируется в параметре `T` шаблона `logAndProcess`. Этот параметр передается `std::forward`, которая восстанавливает закодированную информацию. Детальное описание того, как работает данный механизм, вы найдете в разделе 5.6.

Учитывая, что и `std::move`, и `std::forward` сводятся к приведению и единственная разница между ними лишь в том, что `std::move` всегда выполняет приведение, в то время как `std::forward` — только иногда, вы можете спросить, не можем ли мы обойтись без `std::move` и просто использовать везде `std::forward`. С чисто технической точки зрения ответ утвердительный: `std::forward` может сделать все. Необходимости в `std::move` нет. Конечно, ни одна из этих функций не является действительно *необходимой*, потому что мы могли бы просто вручную написать требуемое приведение, но, я надеюсь, мы сойдемся во мнении, что это будет как минимум некрасиво.

Привлекательными сторонами `std::move` являются удобство, снижение вероятности ошибок и большая ясность. Рассмотрим класс, в котором мы хотели бы отслеживать количество вызовов перемещающего конструктора. Все, что нам надо, — это счетчик, объявленный как `static`, который увеличивался бы при каждом вызове перемещающего конструктора. Полагая, что единственными нестатическими данными класса является `std::string`, вот как выглядит обычный (т.е. использующий `std::move`) способ реализации перемещающего конструктора:

```
class Widget {  
public:  
    Widget(Widget&& rhs)  
        : s(std::move(rhs.s))  
        { ++moveCtorCalls; }  
  
    ...  
private:  
    static std::size_t moveCtorCalls;  
    std::string s;  
};
```

Чтобы реализовать то же поведение с помощью `std::forward`, код должен был бы выглядеть следующим образом:

```
class Widget {  
public:  
    Widget(Widget&& rhs)           // Безусловная,  
        : s(std::forward<std::string>(rhs.s)) // нежелательная  
        { ++moveCtorCalls; }                // реализация  
  
    ...  
};
```

Заметим сначала, что `std::move` требует только аргумент функции (`rhs.s`), в то время как `std::forward` требует как аргумент функции (`rhs.s`), так и аргумент типа

шаблона (`std::string`). Затем обратим внимание на то, что тип, который мы передаем `std::forward`, должен быть не ссылочным, поскольку таково соглашение по кодированию, что передаваемый аргумент является `rvalue` (см. раздел 5.6). Вместе это означает, что `std::move` требует меньшего ввода текста по сравнению с `std::forward` и избавляет от проблем передачи типа аргумента, указывающего, что передаваемый аргумент является `rvalue`. `std::move` устраняет также возможность передачи неверного типа (например, `std::string&`, что привело бы к тому, что член-данные `s` был бы создан с помощью копирования, а не перемещения).

Что еще более важно, так это то, что использование `std::move` выполняет безусловное приведение к `rvalue`, в то время как использование `std::forward` означает приведение к `rvalue` только ссылок, связанных с `rvalue`. Это два совершенно различных действия. Первое из них обычно настраивает перемещение, в то время как второе просто *передает* объект другой функции способом, сохраняющим исходную характеристику объекта (`lvalue` или `rvalue`). Поскольку эти действия совершенно различны, наличие двух разных функций (и разных имен функций) является преимуществом, позволяющим их различать.

### Следует запомнить

- `std::move` выполняет безусловное приведение к `rvalue`. Сама по себе эта функция не перемещает ничего.
- `std::forward` приводит свой аргумент к `rvalue` только тогда, когда этот аргумент связан с `rvalue`.
- Ни `std::move`, ни `std::forward` не выполняют никаких действий времени выполнения.

## 5.2. Отличие универсальных ссылок от `rvalue`-ссылок

Говорят, что истина делает нас свободными, но при соответствующих обстоятельствах хорошо выбранная ложь может оказаться столь же освобождающей. Этот раздел и есть такой ложью. Но поскольку мы имеем дело с программным обеспечением, давайте избегать слова “ложь”, а вместо него говорить, что данный раздел содержит “абстракцию”.

Чтобы объявить `rvalue`-ссылку на некоторый тип `T`, вы пишете `T&&`. Таким образом, представляется разумным предположить, что если вы видите в исходном тексте `T&&`, то имеете дело с `rvalue`-ссылками. Увы, не все так просто.

```
void f(Widget&& param);           // rvalue-ссылка

Widget&& var1 = Widget();          // rvalue-ссылка

auto&& var2 = var1;               // Не rvalue-ссылка

template<typename T>
void f(std::vector<T>&& param); // rvalue-ссылка
```

```
template<typename T>
void f(T&& param); // Не rvalue-ссылка
```

На самом деле “`T&&`” имеет два разных значения. Одно из них — конечно, `rvalue`-ссылка. Такие ссылки ведут себя именно так, как вы ожидаете: они связываются только с `rvalue` и их смысл заключается в идентификации объектов, которые могут быть перемещены.

Другое значение “`T&&`” — либо `rvalue`-ссылка, либо `lvalue`-ссылка. Такие ссылки выглядят в исходном тексте как `rvalue`-ссылки (т.е. “`T&&`”), но могут вести себя так, как если бы они были `lvalue`-ссылками (т.е. “`T&`”). Такая дуальная природа позволяет им быть связанными как с `rvalue` (подобно `rvalue`-ссылкам), так и с `lvalue` (подобно `lvalue`-ссылкам). Кроме того, они могут быть связаны как с константными, так и с неконстантными объектами, как с объектами `volatile`, так и с объектами, не являющимися `volatile`, и даже с объектами, одновременно являющимися `const`, и `volatile`. Они могут быть связаны практически со всем. Такие беспрецедентно гибкие ссылки заслуживают собственного имени, и я называю их *универсальными ссылками*<sup>1</sup>.

Универсальные ссылки возникают в двух контекстах. Наиболее распространенным являются параметры шаблона функции, такие как в приведенном выше примере кода:

```
template<typename T>
void f(T&& param); // param – универсальная ссылка
```

Вторым контекстом являются объявления `auto`, включая объявление из приведенного выше примера кода:

```
auto&& var2 = var1; // var2 – универсальная ссылка
```

Общее в этих контекстах — наличие *вывода типа*. В шаблоне `f` выводится тип `param`, а в объявлении `var2` выводится тип переменной `var2`. Сравните это с приведенными далее примерами (также взятыми из приведенного выше примера кода), в которых вывод типа отсутствует. Если вы видите “`T&&`” без вывода типа, то вы смотрите на `rvalue`-ссылку:

```
void f(Widget&& param); // Вывод типа отсутствует;
                           // param – rvalue-ссылка
Widget&& var1 = Widget(); // Вывод типа отсутствует;
                           // var1 – rvalue-ссылка
```

Поскольку универсальные ссылки являются ссылками, они должны быть инициализированы. Инициализатор универсальной ссылки определяет, какую ссылку она представляет: `lvalue`-ссылку или `rvalue`-ссылку. Если инициализатор представляет собой `rvalue`, универсальная ссылка соответствует `rvalue`-ссылке. Если же инициализатор является `lvalue`, универсальная ссылка соответствует `lvalue`-ссылке. Для универсальных ссылок, которые являются параметрами функций, инициализатор предоставляется в месте вызова:

```
template<typename T>
void f(T&& param); // param является универсальной ссылкой
```

<sup>1</sup> В разделе 5.3 поясняется, что к универсальным ссылкам почти всегда может применяться `std::forward`, так что, когда эта книга готовилась к печати, некоторые члены сообщества C++ начинали именовать универсальные ссылки *передаваемыми ссылками*.

```

Widget w;
f(w);           // В f передается lvalue; тип param -
                // Widget& (т.е. lvalue-ссылка)

f(std::move(w)); // В f передается rvalue; тип param -
                // Widget&& (т.е. rvalue-ссылка)

```

Чтобы ссылка была универсальной, вывод типа необходим, но не достаточен. Вид объявления ссылки также должен быть корректным, и этот вид достаточно ограничен. Он должен в точности иметь вид “`T&&`”. Взглянем еще раз на пример, который мы уже рассматривали ранее:

```

template<typename T>
void f(std::vector<T>&& param); // param - rvalue-ссылка

```

Когда вызывается `f`, тип `T` выводится (если только вызывающий код явно его не укажет, но этот крайний случай мы не рассматриваем). Однако объявление типа `param` не имеет вида “`T&&`”; оно представляет собой `std::vector<T>&&`. Это исключает возможность для `param` быть универсальной ссылкой. Следовательно, `param` является `rvalue-ссылкой`, что ваши компиляторы с удовольствием подтвердят при попытке передать в функцию `f` `lvalue`:

```

std::vector<int> v;
f(v);           // Ошибка! Невозможно связать lvalue
                // с rvalue-ссылкой

```

Даже простого наличия квалификатора `const` достаточно для того, чтобы отобрать у ссылки звание универсальной:

```

template<typename T>
void f(const T&& param); // param - rvalue-ссылка

```

Если вы находитесь в шаблоне и видите параметр функции с типом “`T&&`”, вы можете решить, что перед вами универсальная ссылка. Но вы не должны этого делать, поскольку размещение в шаблоне не гарантирует наличие вывода типа. Рассмотрим следующую функцию-член `push_back` в `std::vector`:

```

template<class T,
         class Allocator = allocator<T>> // Из стандарта C++
class vector {
public:
    void push_back(T&& x);
};

```

Параметр `push_back`, определенно, имеет верный для универсальной ссылки вид, но в данном случае вывода типа нет. Дело в том, что `push_back` не может существовать без конкретного инстанцированного вектора, частью которого он является; а тип этого инстанцирования полностью определяет объявление `push_back`. Другими словами, код

```
std::vector<Widget> v;
```

приводит к следующему инстанцированию шаблона `std::vector`:

```
class vector<Widget, allocator<Widget>> {
public:
    void push_back(Widget&& x); // rvalue-ссылка
};
```

Теперь вы можете ясно видеть, что `push_back` не использует вывода типа. Эта функция `push_back` для `vector<T>` (их две — данная функция перегружена) всегда объявляет параметр типа “rvalue-ссылка на `T`”.

В противоположность этому концептуально подобная функция-член `emplace_back` в `std::vector` выполняет вывод типа:

```
template<class T,
         class Allocator = allocator<T>> // // Из стандарта C++
class vector {
public:
    template <class... Args>
    void emplace_back(Args&&... args);
};
```

Здесь параметр типа `Args` не зависит от параметра типа вектора `T`, так что `Args` должен выводиться при каждом вызове `emplace_back`. (Да, в действительности `Args` представляет собой набор параметров, а не параметр типа, но для нашего рассмотрения его можно рассматривать как параметр типа.)

Тот факт, что параметр типа `emplace_back` имеет имя `Args` и является при этом универсальным указателем, только усиливает мое замечание о том, что универсальная ссылка обязана иметь вид “`T&&`”. Вы не обязаны использовать в качестве имени `T`. Например, приведенный далее шаблон принимает универсальную ссылку, поскольку она имеет правильный вид (“`type&&`”), а тип `param` выводится (опять же, исключая крайний случай, когда вызывающий код явно указывает тип):

```
template<typename MyTemplateType> // param является
void someFunc(MyTemplateType&& param); // универсальной ссылкой
```

Ранее я отмечал, что переменные `auto` также могут быть универсальными ссылками. Чтобы быть более точным, переменные, объявленные с типом `auto&&`, являются универсальными ссылками, поскольку имеет место вывод типа и они имеют правильный вид (“`T&&`”). Универсальные ссылки `auto` не так распространены, как универсальные ссылки, используемые в качестве параметров шаблонов функций, но они также время от времени возникают в C++11. Гораздо чаще они возникают в C++14, поскольку лямбда-выражения C++14 могут объявлять параметры `auto&&`. Например, если вы хотите написать лямбда-выражение C++14 для записи времени, которое занимает вызов произвольной функции, можете сделать это следующим образом:

```

auto timeFuncInvocation =
[] (auto&& func, auto&&... params) // C++14
{
    Запуск таймера;
    std::forward<decltype(func)>(func) (           // Вызов func
        std::forward<decltype(params)>(params)... // с params
    );
    Останов таймера и запись времени;
};

```

Если ваша реакция на код “`std::forward<decltype(ля-ля-ля)>`” внутри лямбда-выражения — “Что за @#\$%!!!”, то, вероятно, вы просто еще не читали раздел 6.3. Не беспокойтесь о нем. Главное для нас сейчас в этом лямбда-выражении — объявленные как `auto&&` параметры. `func` является универсальной ссылкой, которая может быть связана с любым вызываемым объектом, как `lvalue`, так и `rvalue`. `params` представляет собой нуль или несколько универсальных ссылок (т.е. набор параметров, являющихся универсальными ссылками), которые могут быть связаны с любым количеством объектов произвольных типов. В результате, благодаря универсальным ссылкам `auto`, лямбда-выражение `timeFuncInvocation` в состоянии записать время работы почти любого выполнения функции. (Чтобы разобраться в разнице между “любого” и “почти любого”, обратитесь к разделу 5.8.)

Имейте в виду, что весь этот раздел — основы универсальных ссылок — является лож... простите, абстракцией. Лежащая в основе истина, известная как *свертывание ссылок* (reference collapsing), является темой раздела 5.6. Но истина не делает абстракцию менее полезной. Понимание различий между `rvalue`-ссылками и универсальными ссылками поможет вам читать исходные тексты более точно (“Этот `T&&` связывается только с `rvalue` или с чем угодно?”), а также избегать неоднозначностей при общении с коллегами (“Здесь я использовал универсальную ссылку, а не `rvalue`-ссылку...”). Оно также поможет вам в понимании смысла разделов 5.3 и 5.4, которые опираются на указанное различие. Так что примите эту абстракцию, погрузитесь в нее... Так же как законы Ньютона (технически не совсем корректные) обычно куда полезнее и проще общей теории относительности Эйнштейна (“истины”), так и понятие универсальных ссылок обычно предпочтительнее для работы, чем детальная информация о свертывании ссылок.

### Следует запомнить

- Если параметр шаблона функции имеет тип `T&&` для выводимого типа `T` или если объект объявлен с использованием `auto&&`, то параметр или объект является универсальной ссылкой.
- Если вид объявления типа не является в точности `ture&&` или если вывод типа не имеет места, то `ture&&` означает `rvalue`-ссылку.
- Универсальные ссылки соответствуют `rvalue`-ссылкам, если они инициализируются значениями `rvalue`. Они соответствуют `lvalue`-ссылкам, если они инициализируются значениями `lvalue`.

## 5.3. Используйте `std::move` для rvalue-ссылок, а `std::forward` — для универсальных ссылок

Rvalue-ссылки связываются только с объектами, являющимися кандидатами для перемещения. Если у вас есть параметр, представляющий собой rvalue-ссылку, вы знаете, что связанный с ним объект может быть перемещен:

```
class Widget {  
    Widget(Widget&& rhs); // rhs, определенно, ссылается на  
                           // объект, который можно перемещать  
};
```

В этом случае вы захотите передавать подобные объекты другим функциям таким образом, чтобы разрешить им использовать преимущества “правосторонности”. Способ, которым это можно сделать, — привести параметры, связываемые с такими объектами, к rvalue. Как поясняется в разделе 5.1, `std::move` не просто это делает, это та задача, для которой создана эта функция:

```
class Widget {  
public:  
    Widget(Widget&& rhs)      // rhs является rvalue-ссылкой  
    : name(std::move(rhs.name)),  
    p(std::move(rhs.p))  
    { ... }  
    ...  
private:  
    std::string name;  
    std::shared_ptr<SomeDataStructure> p;  
};
```

С другой стороны, универсальная ссылка (см. раздел 5.2) может быть связана с объектом, который разрешено перемещать. Универсальные ссылки должны быть приведены к rvalue только тогда, когда они были инициализированы с помощью `rvalue`. В разделе 5.1 разъясняется, что именно это и делает функция `std::forward`:

```
class Widget {  
public:  
    template<typename T>  
    void setName(T& newName)           // newName является  
    { name = std::forward<T>(newName); } // универсальной ссылкой  
};
```

Короче говоря, rvalue-ссылки при их передаче в другие функции должны быть безусловно приведены к rvalue (с помощью `std::move`), так как они всегда связываются с rvalue, а универсальные ссылки должны приводиться к rvalue при их передаче условно (с помощью `std::forward`), поскольку они только иногда связываются с rvalue.

В разделе 5.1 поясняется, что можно добиться верного поведения rvalue-ссылок и с помощью `std::forward`, но исходный текст при этом становится многословным, подверженным ошибкам и неidiоматичным, так что вы должны избегать применения `std::forward` с rvalue-ссылкам. Еще худшей является идея применения `std::move` к универсальным ссылкам, так как это может привести к неожиданному изменению значений lvalue (например, локальных переменных):

```
class Widget {
public:
    template<typename T>
    void setName(T&& newName)          // Универсальная ссылка.
    { name = std::move(newName); }        // Компилируется, но это
    ...                                    // очень плохое решение!
private:
    std::string name;
    std::shared_ptr<SomeDataStructure> p;
};

std::string getWidgetName(); // Фабричная функция
Widget w;
auto n = getWidgetName();   // n – локальная переменная
w.setName(n);              // Перемещение n в w!
                           // Значение n теперь неизвестно
```

Здесь локальная переменная `n` передается функции `w.setName`. Вызывающий код можно простить за предположение о том, что эта функция по отношению к `n` является операцией чтения. Но поскольку `setName` внутренне использует `std::move` для безусловного приведения своего ссылочного параметра к rvalue, значение `n` может быть перемещено в `w.name`, и `n` вернется из вызова `setName` с неопределенным значением. Этот вид поведения может привести программиста к отчаянию — если не к прямому насилию.

Можно возразить, что `setName` не должен был объявлять свой параметр как универсальную ссылку. Такие ссылки не могут быть константными (см. раздел 5.2), но `setName`, безусловно, не должен изменять свой параметр. Вы могли бы указать, что если перегрузить `setName` для константных значений lvalue и rvalue, то этой проблемы можно было бы избежать, например, таким образом:

```
class Widget {
public:
    void setName(const std::string& newName) // Устанавливается
    { name = newName; }                      // из const lvalue
    void setName(std::string&& newName)      // Устанавливается
    { name = std::move(newName); }            // из rvalue
};

};
```

В данном случае это, безусловно, сработает, но у метода есть и недостатки. Во-первых, требуется вводить исходный текст большего размера (две функции вместо одного

шаблона). Во-вторых, это может быть менее эффективным. Например, рассмотрим следующее применение `setName`:

```
w.setName("Adela Novak");
```

При наличии версии `setName`, принимающей универсальную ссылку, функции `setName` будет передан строковый литерал "Adela Novak", в котором он будет передан оператору присваивания для `std::string` внутри `w`. Таким образом, член-данные `name` объекта `w` будет присвоен непосредственно из строкового литерала; никакого временного объекта `std::string` создаваться не будет. Однако в случае перегруженных версий `setName` будет создан временный объект `std::string`, с которым будет связан параметр функции `setName`, и этот временный объект `std::string` будет перемещен в член-данные объекта `w`. Таким образом, вызов `setName` повлечет за собой выполнение одного конструктора `std::string` (для создания временного объекта), одного перемещающего оператора присваивания `std::string` (для перемещения `newName` в `w.name`) и одного деструктора `std::string` (для уничтожения временного объекта). Это практически наверняка более дорогостоящая последовательность операций, чем вызов только одного оператора присваивания `std::string`, принимающего указатель `const char*`. Дополнительная стоимость может варьироваться от реализации к реализации, и стоит ли беспокоиться о ней, зависит от приложения и библиотеки; однако, скорее всего, в ряде случаев замена шаблона, получающего универсальную ссылку, парой функций, перегруженных для `lvalue`-и `rvalue`-ссылок, приведет к дополнительным затратам времени выполнения.

Однако наиболее серьезной проблемой с перегрузкой для `lvalue` и `rvalue` является не объем или идиоматичность исходного кода и не производительность времени выполнения. Это — плохая масштабируемость проекта. `Widget::setName` принимает только один параметр, так что необходимы только две перегрузки. Но для функций, принимающих большее количество параметров, каждый из которых может быть как `lvalue`, так и `rvalue`, количество перегрузок растет в соответствии с показательной функцией:  $n$  параметров требуют  $2^n$  перегрузок. И это еще не самый худший случай. Некоторые функции (на самом деле — шаблоны функций) принимают *неограниченное количество* параметров, каждый из которых может быть как `lvalue`, так и `rvalue`. Типичными представителями таких функций являются `std::make_shared` и, начиная с C++14, `std::make_unique` (см. раздел 4.4). Попробуйте написать объявления их наиболее часто используемых перегрузок:

```
template<class T, class... Args>      // Из стандарта C++11
shared_ptr<T> make_shared(Arg... args);

template<class T, class... Args>      // Из стандарта C++14
unique_ptr<T> make_unique(Arg... args);
```

Для функций наподобие указанных перегрузка для `lvalue` и `rvalue` не является приемлемым вариантом: единственным выходом является применение универсальных ссылок. А внутри таких функций, уверяю вас, к универсальным ссылкам при их передаче другим функциям следует применять `std::forward`. Вот то, что вы должны делать.

Ну хорошо, обычно должны. В конечном итоге. Но не обязательно изначально. В некоторых случаях вы захотите использовать привязку объекта к rvalue-ссылке или универсальной ссылке более одного раза в одной функции, и вы захотите гарантировать, что перемещения не будет, пока вы явно не укажете его выполнить. В этом случае вы захотите применить `std::move` (для rvalue-ссылок) или `std::forward` (для универсальных ссылок) только к последнему использованию ссылки, например:

```
template<typename T> // text – универсальная
void setSignText(T&& text) // ссылка
{
    sign.setText(text); // Используем text, но не
                        // изменяем его
    auto now =           // Получение текущего времени
        std::chrono::system_clock::now();
    signHistory.add(now,
                    std::forward<T>(text)); // Условное приведение
} // text к rvalue
```

Здесь мы хотим гарантировать, что значение `text` не изменится вызовом `sign.setText`, поскольку мы хотим использовать это значение при вызове `signHistory.add`. Следовательно, `std::forward` применяется только к последнему использованию универсальной ссылки.

Для `std::move` применяются те же рассуждения (т.е. надо применить `std::move` к rvalue-ссылке только при ее последнем использовании), но важно отметить, что в некоторых редких случаях вы захотите вызвать `std::move_if_noexcept` вместо `std::move`. Чтобы узнать, когда и почему, обратитесь к разделу 3.8.

Если вы имеете дело с функцией, осуществляющей возврат *по значению*, и возвращающей объект, привязанный к rvalue-ссылке или универсальной ссылке, вы захотите применять `std::move` или `std::forward` при возврате ссылки. Чтобы понять, почему, рассмотрим функцию `operator+` для сложения двух матриц, где о левой матрице точно известно, что она является rvalue (а следовательно, может повторно использовать свою память для хранения суммы матриц):

```
Matrix // Возврат по значению
operator+(Matrix&& lhs, const Matrix& rhs)
{
    lhs += rhs;
    return std::move(lhs); // Перемещение lhs в
} // возвращаемое значение
```

С помощью приведения `lhs` к rvalue в инструкции `return` (с помощью `std::move`) `lhs` будет перемещен в местоположение возвращаемого функцией значения. Если опустить вызов `std::move`,

```
Matrix // Как и ранее above
operator+(Matrix&& lhs, const Matrix& rhs)
{
```

```
lhs += rhs;
return lhs; // Копирование lhs в
} // возвращаемое значение
```

то тот факт, что `lhs` представляет собой `lvalue`, заставит компиляторы вместо перемещения копировать его в местоположение возвращаемого функцией значения. В предположении, что тип `Matrix` поддерживает перемещающее конструирование, более эффективное, чем копирующее, применение `std::move` в инструкции `return` дает более эффективный код.

Если тип `Matrix` не поддерживает перемещения, приведение его к `rvalue` не повредит, поскольку `rvalue` будет просто скопировано копирующим конструктором `Matrix` (см. раздел 5.1). Если `Matrix` позже будет переделан так, что станет поддерживать перемещение, `operator+` автоматически использует данное преимущество при следующей компиляции. В таком случае ничто не будет потеряно (и возможно, многое будет приобретено) при применении `std::move` к `rvalue`-ссылкам, возвращаемым из функций, которые осуществляют возврат по значению.

Для универсальных ссылок и `std::forward` ситуация схожа. Рассмотрим шаблон функции `reduceAndCopy`, который получает возможно сократимую дробь `Fraction`, сокращает ее, а затем возвращает копию сокращенной дроби. Если исходный объект представляет собой `rvalue`, его значение должно быть перенесено в возвращаемое значение (избегая тем самым стоимости создания копии), но если исходный объект — `lvalue`, должна быть создана фактическая копия:

```
template<typename T>
Fraction                                // Возврат по значению
reduceAndCopy(T&& frac)                  // Универсальная ссылка
{
    frac.reduce();
    return std::forward<T>(frac); // Перемещение rvalue и
}                                         // копирование lvalue в
                                         // возвращаемое значение
```

Если опустить вызов `std::forward`, `frac` будет в обязательном порядке копироваться в возвращаемое значение `reduceAndCopy`.

Некоторые программисты берут приведенную выше информацию и пытаются распространить ее на ситуации, в которых она неприменима. Они рассуждают следующим образом: “если использование `std::move` для параметра, являющегося `rvalue`-ссылкой и копируемого в возвращаемое значение, превращает копирующий конструктор в перемещающий, то я могу выполнить ту же оптимизацию для возвращаемых мною локальных переменных”. Другими словами, они считают, что если дана функция, возвращающая локальную переменную по значению, такая, как следующая:

```
Widget makeWidget() // "Копирующая" версия makeWidget
{
    Widget w;      // Переменная
                    // Настройка w
```

```
    return w;           // "Копирование" w в возвращаемое значение
}
```

то они могут “оптимизировать” ее, превратив “копирование” в перемещение:

```
Widget makeWidget()      // Перемещающая версия makeWidget
{
    Widget w;
    ...
    return std::move(w); // Перемещение w в возвращаемое
}                         // значение (не делайте этого!)
```

Мое обильное использование кавычек должно подсказать вам, что эти рассуждения не лишены недостатков. Но почему? Да потому что Комитет по стандартизации уже прошел этот путь и давно понял, что “копирующая” версия `makeWidget` может избежать необходимости копировать локальную переменную `w`, если будет создавать ее прямо в памяти, выделенной для возвращаемого значения функции. Это оптимизация, известная как *оптимизация возвращаемого значения* (*return value optimization — RVO*) и с самого начала благословленная стандартом C++.

Формулировка такого благословления — сложное дело, поскольку хочется разрешить такое *отсутствие копирования* только там, где оно не влияет на наблюдаемое поведение программы. Перефразируя излишне сухой текст стандарта, это благословение на отсутствие копирования (или перемещения) локального объекта<sup>2</sup> в функции, выполняющей возврат по значению, дается компиляторам, если (1) тип локального объекта совпадает с возвращаемым функцией и (2) локальный объект представляет собой возвращаемое значение. С учетом этого вернемся к “копирующей” версии `makeWidget`:

```
Widget makeWidget() // "Копирующая" версия makeWidget
{
    Widget w;
    ...
    return w;          // "Копирование" w в возвращаемое значение
}
```

Здесь выполняются оба условия, и вы можете доверять мне, когда я говорю вам, что каждый приличный компилятор C++ будет использовать RVO для того, чтобы избежать копирования `w`. Это означает, что “копирующая” версия `makeWidget` на самом деле копирования не выполняет.

Перемещающая версия `makeWidget` делает только то, о чем говорит ее имя (в предложении наличия перемещающего конструктора `Widget`): она перемещает содержимое `w`.

<sup>2</sup> Такими локальными объектами являются большинство локальных переменных (например, такие как `w` в `makeWidget`), а также временные объекты, создаваемые как часть инструкции `return`. Параметры функции на такое звание претендовать не могут. Некоторые программисты различают применение RVO к именованным и неименованным (т.е. временным) локальным объектам, ограничивая термин “RVO” неименованными объектами и называя его применение к именованным объектам *оптимизацией именованных возвращаемых значений* (*named return value optimization — NRVO*).

в местоположение возвращаемого значения `makeWidget`. Но почему компиляторы не используют RVO для устранения перемещения, вновь создавая `w` в памяти, выделенной для возвращаемого значения функции? Ответ прост: они не могут. Условие (2) предусматривает, что RVO может быть выполнена, только если возвращается локальный объект, но в перемещающей версии `makeWidget` это не так. Посмотрим еще раз на инструкцию `return`:

```
return std::move(w);
```

То, что здесь возвращается, не является локальным объектом `w`; это *ссылка на w* — результат `std::move(w)`. Возврат ссылки на локальный объект не удовлетворяет условиям, требующимся для применения RVO, так что компиляторы вынуждены перемещать `w` в местоположение возвращаемого значения функции. Разработчики, пытаясь с помощью применения `std::move` к возвращаемой локальной переменной помочь компиляторам оптимизировать код, на самом деле ограничивают возможности оптимизации, доступные их компиляторам!

Но RVO — это всего лишь оптимизация. Компиляторы *не обязаны* устранивать операции копирования и перемещения даже тогда, когда это им позволено. Возможно, вы пааноик и беспокоитесь о том, что ваши компиляторы будут выполнять операции копирования, просто потому, что они могут это делать. А может, вы настолько глубоко разбираетесь в ситуации, что в состоянии распознать случаи, когда компиляторам трудно применять RVO, например когда различные пути выполнения в функции возвращают разные локальные переменные. (Компиляторы должны генерировать код для построения соответствующей локальной переменной в памяти, выделенной для возвращаемого значения функции, но как компиляторы смогут определить, какая локальная переменная должна использоваться?) В таком случае вы можете быть готовы заплатить цену перемещения как гарантию того, что копирование выполнено не будет. Иначе говоря, вы можете продолжать думать, что применение `std::move` к возвращаемому локальному объекту разумно просто потому, что при этом вы спокойны, зная, что вам не придется платить за копирование.

В этом случае применение `std::move` к локальному объекту все равно остается плохой идеей. Часть стандарта, разрешающая применение RVO, гласит далее, что если условия для применения RVO выполнены, но компиляторы предпочитают не выполнять удаление копирования, то возвращаемый объект *должен рассматриваться как rvalue*. По сути, стандарт требует, чтобы, когда оптимизация RVO разрешена, к возвращаемому локальному объекту либо применялось удаление копирования, либо неявно применялась функция `std::move`. Так что в “копирующей” версии `makeWidget`

```
Widget makeWidget()      // Как и ранее
{
    Widget w;
    ...
    return w;
}
```

компиляторы должны либо устраниить копирование `w`, либо рассматривать функцию, как если бы она была написана следующим образом:

```
Widget makeWidget()
{
    Widget w;
    ...
    return std::move(w); // Рассматривает w как rvalue, поскольку
}                                // удаление копирования не выполняется
```

Ситуация аналогична для параметров функции, передаваемых по значению. Они не имеют права на удаление копирования при их возврате из функции, но компиляторы должны рассматривать их в случае возврата как `rvalue`. В результате, если ваш исходный текст выглядит как

```
Widget makeWidget(Widget w) // Передаваемый по значению параметр
{
    ...
    // имеет тот же тип, что и
    // возвращаемый тип функции
    return w;
}
```

компиляторы должны рассматривать его, как если бы он был написан как

```
Widget makeWidget(Widget w)
{
    ...
    return std::move(w); // w рассматривается как rvalue
}
```

Это означает, что, используя `std::move` для локального объекта, возвращаемого функцией по значению, вы не можете помочь компилятору (он обязан рассматривать локальный объект как `rvalue`, если не выполняет удаления копирования), но вы, определенно, в состоянии ему помешать (препятствуя RVO). Есть ситуации, когда применение `std::move` к локальной переменной может быть разумным (т.е. когда вы передаете ее функции и знаете, что больше вы ее использовать не будете), но эти ситуации не включают применение `std::move` в качестве части инструкции `return`, которая в противном случае претендовала бы на оптимизацию RVO, или возврат параметра, передаваемого по значению.

### Следует запомнить

- Применяйте `std::move` к `rvalue`-ссылкам, а `std::forward` — к универсальным ссылкам, когда вы используете их в последний раз.
- Делайте то же для `rvalue`- и универсальных ссылок, возвращаемых из функций по значению.
- Никогда не применяйте `std::move` и `std::forward` к локальным объектам, которые могут быть объектом оптимизации возвращаемого значения.

## 5.4. Избегайте перегрузок для универсальных ссылок

Предположим, что вам надо написать функцию, которая принимает в качестве параметра имя, записывает в журнал текущие дату и время, а затем добавляет имя в глобальную структуру данных. Вы могли бы начать с функции, которая имеет примерно следующий вид:

```
std::multiset<std::string> names; // Глобальная структура данных

void logAndAdd(const std::string& name)
{
    auto now =                      // Получение текущего времени
        std::chrono::system_clock::now();
    log(now, "logAndAdd");          // Создание журнальной записи
    names.emplace(name);           // Добавление name в глобальную
}                                // структуру данных; emplace
                                    // см. в разделе 8.2
```

Этот код не является неразумным, но он не такой эффективный, каким мог бы быть. Рассмотрим три потенциальных вызова:

```
std::string petName("Darla");
logAndAdd(petName);                // lvalue типа std::string
logAndAdd(std::string("Persephone")); // rvalue типа std::string
logAndAdd("Patty Dog");           // Строковый литерал
```

В первом вызове параметр name функции logAndAdd связывается с переменной petName. Внутри logAndAdd параметр name в конечном итоге передается в вызов names.emplace. Поскольку name является lvalue, он копируется в names. Избежать этого копирования невозможно, так как lvalue (petName) передается в функцию logAndAdd.

В втором вызове параметр name связывается с rvalue (временный объект std::string, явно созданный из строки "Persephone"). Параметр name сам по себе является lvalue, так что он копируется в names, но мы отдаём себе отчет, что, в принципе, это значение может быть перемещено в names. В этом вызове мы платим за копирование, но мы должны быть способны сделать то же с помощью перемещения.

В третьем вызове параметр name опять связывается с rvalue, но в этот раз со времененным объектом std::string, который неявно создается из "Patty Dog". Как и во втором вызове, name копируется в names, но в этот раз аргумент, изначально переданный в logAndAdd, был строковым литералом. Если бы строковый литерал непосредственно передавался в emplace, в создании временного объекта std::string не было бы необходимости вообще. Вместо этого функция emplace использовала бы строковый литерал для создания объекта std::string непосредственно в std::multiset. Таким образом, в этом третьем вызове мы платим за копирование std::string, при том что нет причин платить даже за перемещение, не говоря уже о копировании.

Неэффективность второго и третьего вызовов logAndAdd можно устраниТЬ, переписав эту функцию так, чтобы она принимала универсальную ссылку (см. раздел 5.2) и,

согласно разделу 5.3, передавала ее с помощью `std::forward` функции `emplace`. Результат говорит сам за себя:

```
template<typename T>
void logAndAdd(T&& name)
{
    auto now = std::chrono::system_clock::now();
    log(now, "logAndAdd");
    names.emplace(std::forward<T>(name));
}

std::string petName("Darla");           // Как и ранее

logAndAdd(petName);                   // Как и ранее, копирова-
                                         // ние lvalue в multiset

logAndAdd(std::string("Persephone")); // Перемещение rvalue
                                         // вместо копирования

logAndAdd("Patty Dog");              // Создание std::string
                                         // в multiset вместо
                                         // копирования временного
                                         // std::string
```

Ура, получена оптимальная эффективность!

Если бы это был конец истории, мы могли бы остановиться и гордо удалиться, но я не сказал вам, что клиенты не всегда имеют непосредственный доступ к именам, требующимся `logAndAdd`. Некоторые клиенты имеют только индекс, который `logAndAdd` использует для поиска соответствующего имени в таблице. Для поддержки таких клиентов выполняется перегрузка функции `logAndAdd`:

```
std::string nameFromIdx(int idx); // Возвращает имя,
                                         // соответствующее idx
void logAndAdd(int idx)           // Новая перегрузка
{
    auto now = std::chrono::system_clock::now();
    log(now, "logAndAdd");
    names.emplace(nameFromIdx(idx));
}
```

Разрешение перегрузки работает, как и следовало ожидать:

```
std::string petName("Darla");           // Как и ранее

logAndAdd(petName);                   // Как и ранее, эти вызовы
logAndAdd(std::string("Persephone")); // приводят к использова-
logAndAdd("Patty Dog");              //нию перегрузки для T&&

logAndAdd(22);                       // Вызов int-перегрузки
```

На самом деле разрешение работает, как ожидается, только если вы не ожидаете слишкомного. Предположим, клиент имеет переменную типа `short`, хранящую индекс, и передает ее функции `logAndAdd`:

```
short nameIdx;  
... // Дает значение переменной nameIdx  
logAndAdd(nameIdx); // Ошибка!
```

Комментарий в последней строке, может быть, не слишком понятен, так что позвольте мне пояснить, что же здесь произошло.

Имеется две перегрузки `logAndAdd`. Одна из них, принимающая универсальную ссылку, может вывести тип `T` как `short`, тем самым приводя к точному соответствию. Перегрузка с параметром `int` может соответствовать аргументу `short` только с повышением. Согласно обычным правилам разрешения перегрузки точное соответствие побеждает соответствие с повышением, так что вызывается перегрузка для универсальной ссылки.

В этой перегрузке параметр `name` связывается с переданным значением типа `short`. Таким образом, `name` передается с помощью `std::forward` функции-члену `emplace` объекта `names` (`std::multiset<std::string>`), которая, в свою очередь, послушно передает его конструктору `std::string`. Но конструктора `std::string`, который принимал бы значение `short`, не существует, так что вызов конструктора `std::string` в вызове `multiset::emplace` в вызове `logAndAdd` неудачен. Все дело в том, что перегрузка для универсальной ссылки точнее соответствует аргументу типа `short`, чем перегрузка для `int`.

Функции, принимающие универсальные ссылки, оказываются самыми жадными в C++. Они в состоянии выполнить инстанцирование с точным соответствием практически для любого типа аргумента (несколько видов аргументов, для которых это не так, описаны в разделе 5.8). Именно поэтому сочетание перегрузки и универсальной ссылки почти всегда является плохой идеей: перегрузка для универсальных ссылок годится для гораздо большего количества типов аргументов, чем обычно ожидает разработчик перегрузок.

Простой способ свалиться в эту яму — написать конструктор с прямой передачей. Небольшое изменение функции `logAndAdd` демонстрирует эту проблему. Вместо написания свободной функции, которая принимает либо `std::string`, либо индекс, который можно использовать для поиска `std::string`, представим себе класс `Person` с конструкторами, которые выполняют те же действия:

```
class Person {  
public:  
    template<typename T>  
    explicit Person(T&& n) // Конструктор с прямой передачей  
    : name(std::forward<T>(n)) {} // инициализирует члены-данные  
  
    explicit Person(int idx) // Конструктор с параметром int  
    : name(nameFromIdx(idx)) {}  
    ...  
private:  
    std::string name;  
};
```

Как и в случае с `logAndAdd`, передача целочисленного типа, отличного от `int` (например, `std::size_t`, `short`, `long` и т.п.), будет вызывать перегрузку конструктора для универсальной ссылки вместо перегрузки для `int`, и это будет вести к ошибкам компиляции. Однако проблема гораздо хуже, поскольку в `Person` имеется больше перегрузок, чем видят глаз. В разделе 3.11 поясняется, что при соответствующих условиях C++ будет генерировать как копирующие, так и перемещающие конструкторы, и это так и будет, даже если класс содержит шаблонный конструктор, который при инстанцировании в состоянии дать сигнатуру копирующего или перемещающего конструктора. Если таким образом генерируются копирующий и перемещающий конструкторы для `Person`, класс `Person` будет выглядеть, по сути, следующим образом:

```
class Person {
public:
    template<typename T>           // Конструктор с прямой передачей
    explicit Person(T&& n)
        : name(std::forward<T>(n)) {}

    explicit Person(int idx);      // Конструктор от int

    Person(const Person& rhs);   // Копирующий конструктор
                                  // (сгенерирован компилятором)
    Person(Person&& rhs);       // Перемещающий конструктор
                                  // (сгенерирован компилятором)
};
```

Это приводит к поведению, интуитивно понятному, только если вы потратили на работу с компиляторами и общение с их разработчиками столько времени, что забыли, каково это — быть человеком:

```
Person p("Nancy");
auto cloneOfP(p); // Создание нового объекта Person из p;
                  // этот код не компилируется!
```

Здесь мы пытаемся создать объект `Person` из другого объекта `Person`, что представляется очевидным случаем копирующего конструирования (р является `lvalue`, так что можно выбросить из головы все фантазии на тему “копирования” с помощью операции перемещения). Но этот код не вызывает копирующий конструктор — он вызывает конструктор с прямой передачей. Затем эта функция будет пытаться инициализировать член-данные `std::string` объекта `Person` значением из объекта `Person` (`p`). Класс `std::string` не имеет конструктора, получающего параметр типа `Person`, так что ваш компилятор будет вынужден просто развести руками и наказать вас длинными и непонятными сообщениями об ошибках.

“Но почему, — можете удивиться вы, — вызывается конструктор с прямой передачей, а не копирующий конструктор? Мы же инициализируем `Person` другим объектом `Person`!” Да, это так, но компиляторы приносят присягу свято соблюдать правила C++, а правила, имеющие отношение к данной ситуации, — это правила разрешения вызовов перегруженных функций.

Компиляторы рассуждают следующим образом. `cloneOfP` инициализируется неконстантным `lvalue` (`p`), а это означает, что шаблонный конструктор может быть инстанцирован для получения неконстантного `lvalue` типа `Person`. После такого инстанцирования класс `Person` выглядит следующим образом:

```
class Person {  
public:  
    explicit Person(Person& n)           // Инстанцирован из  
    : name(std::forward<Person&>(n)) {} // шаблона с прямой  
                                         // передачей  
  
    explicit Person(int idx);   // Как и ранее  
    Person(const Person& rhs);  // Копирующий конструктор  
                               // (сгенерирован компилятором)  
};
```

В инструкции

```
auto cloneOfP(p);
```

р может быть передан либо копирующему конструктору, либо инстанцированному шаблону. Вызов копирующего конструктора для точного соответствия типа параметра требует добавления к р модификатора `const`; вызов инстанцированного шаблона никаких добавлений не требует. Таким образом, перегрузка, сгенерированная из шаблона, представляет собой лучшее соответствие, так что компиляторы делают то, для чего предназначены: генерируют вызов той функции, которая соответствует наилучшим образом. “Копирование” неконстантных `lvalue` типа `Person`, таким образом, осуществляется конструктором с прямой передачей, а не копирующим конструктором.

Если мы немного изменим пример, так, чтобы копируемый объект был константным, то увидим совершенно иную картину:

```
const Person cp("Nancy"); // Теперь объект константный  
auto cloneOfP(cp);       // Вызов копирующего конструктора!
```

Поскольку копируемый объект теперь объявлен как `const`, он в точности соответствует типу параметра, получаемого копирующим конструктором. Шаблонизированный конструктор также может быть инстанцирован таким образом, чтобы иметь ту же сигнатуру:

```
class Person {  
public:  
    explicit Person(const Person& n); // Инстанцирован из шаблона  
    Person(const Person& rhs);     // Копирующий конструктор  
                                   // (сгенерирован компилятором)  
};
```

Но это не имеет значения, поскольку одно из правил разрешения перегрузок в C++ гласит, что в ситуации, когда инстанцирование шаблона и нешаблонная функция (т.е. “нормальная” функция) имеют одинаково хорошее соответствие, предпочтение отдается

нормальной функции. Поэтому все козыри оказываются на руках копирующего конструктора (нормальной функции) с той же самой сигнатурой.

(Если вам интересно, почему компиляторы создают копирующий конструктор, если они могут инстанцировать шаблонный конструктор с той же сигнатурой, обратитесь к разделу 3.11.)

Взаимодействие между конструкторами с прямой передачей и генерированными компилятором операциями копирования и перемещения становится еще более сложным, когда в картину включается наследование. В частности, обычные реализации копирующих и перемещающих операций производного класса ведут себя совершенно неожиданно. Взгляните на следующий код:

```
class SpecialPerson: public Person {  
public:  
    SpecialPerson(const SpecialPerson& rhs) // Копирующий  
    : Person(rhs) // конструктор; вызывает конструктор  
    { ... } // базового класса с прямой передачей!  
  
    SpecialPerson(SpecialPerson&& rhs) // Перемещающий  
    : Person(std::move(rhs)) // конструктор; вызывает конструктор  
    { ... } // базового класса с прямой передачей!  
};
```

Как указывают комментарии, копирующий и перемещающий конструкторы производного класса не вызывают копирующий и перемещающий конструкторы базового класса; они вызывают конструктор базового класса с прямой передачей! Чтобы понять, почему, обратите внимание, что функции производного класса используют аргументы типа `SpecialPerson` для передачи в базовый класс, после чего в игру вступает разрешение перегрузок для конструкторов в классе `Person`. В конечном итоге код не будет компилироваться, потому что у `std::string` нет никакого конструктора, принимающего `SpecialPerson`.

Я надеюсь, что теперь я убедил вас, что перегрузка для параметров, являющихся универсальными ссылками, — это то, чего лучше избегать, насколько это возможно. Но если перегрузка для универсальной ссылки — плохая идея, то что же делать, если вам нужна функция, которая выполняет передачу большинства типов аргументов, но при этом должна обрабатывать некоторые из них особым образом? Это яйцо может быть разбито массой способов. Этих способов так много, что я посвятил им целый раздел — раздел 5.5, который идет сразу после того, который вы сейчас читаете. Не останавливайтесь, и вы попадете прямо в него.

### Спедует запомнить

- Перегрузка для универсальных ссылок почти всегда приводит к тому, что данная перегрузка вызывается чаще, чем вы ожидаете.
- Особенно проблематичны конструкторы с прямой передачей, поскольку они обычно соответствуют неконстантным lvalue лучше, чем копирующие конструкторы, и могут перехватывать вызовы из производного класса копирующих и перемещающих конструкторов базового класса.

## 5.5. Знакомство с альтернативами перегрузки для универсальных ссылок

В разделе 5.4 поясняется, что перегрузка для универсальных ссылок может привести к целому ряду проблем как для автономных функций, так и для функций-членов (в особенности для конструкторов). Тем не менее в нем также приводятся примеры, когда такая перегрузка может оказаться полезной, если только она будет вести себя так, как мы хотим! В этом разделе исследуются способы достижения желаемого поведения либо путем проектирования, позволяющего избежать перегрузок для универсальных ссылок, либо путем применения их таким образом, чтобы ограничить типы аргументов, которым они могут соответствовать.

Ниже использованы примеры, представленные в разделе 5.4. Если вы читали его давно или вовсе не читали, просмотрите указанный раздел, прежде чем читать данный.

### Отказ от перегрузки

В первом примере раздела 5.4 функция `logAndAdd` является типичным представителем функций, которые могут избежать недостатков перегрузки для универсальных ссылок, просто используя разные имена для потенциальных перегрузок. Например, рассматривавшиеся перегрузки `logAndAdd` могут быть разделены на `logAndAddName` и `logAndAddNameIdx`. Увы, этот подход не будет работать для второго рассматривавшегося примера — конструктора `Person`, потому что имена конструкторов в языке зафиксированы. Кроме того, кто же захочет отказаться от перегрузки?

### Передача `const T&`

Одной из альтернатив является возврат к C++98 и замена передачи универсальной ссылки передачей `lvalue`-ссылки на `const`. Фактически это первый подход, рассматривавшийся в разделе 5.4. Его недостаток состоит в том, что данный дизайн не столь эффективен, как нам хотелось бы. С учетом всех нынешних наших знаний о взаимодействии универсальных ссылок и перегрузки отказ от некоторой эффективности в пользу простоты может оказаться более привлекательными компромиссом, чем нам казалось изначально.

### Передача по значению

Подход, который часто позволяет добиться производительности без увеличения сложности, заключается в замене передачи параметров по ссылке передачей по значению, как бы противостоящим это ни звучало. Этот дизайн основан на совете из раздела 8.1: рассмотреть вопрос о передаче объектов по значению в случае, когда известно, что они будут копироваться. Поэтому я отложу до указанного раздела подробное обсуждение того, как все это работает и насколько эффективным является это решение. Здесь я просто покажу, как этот подход может использоваться в примере с классом `Person`:

```
class Person {  
public:
```

```

explicit Person(std::string n) // Замена конструктора с T&&;
: name(std::move(n)) {}      // о применении std::move
                             // читайте в разделе 8.1
explicit Person(int idx)     // Как и ранее
: name(nameFromIdx(idx)) {}

...
private:
    std::string name;
};

```

Поскольку конструктора `std::string`, принимающего только целочисленное значение, нет, все аргументы типа `int` и подобных ему (например, `std::size_t`, `short`, `long`), передаваемые конструктору `Person`, будут перенаправлены к перегрузке для `int`. Аналогично все аргументы типа `std::string` (а также аргументы, из которых могут быть созданы объекты `std::string`, например литералы наподобие "Ruth") будут передаваться конструктору, принимающему `std::string`. Здесь для вызывающего кода нет никаких сюрпризов. Возможно, некоторые программисты будут удивлены, что применение 0 или `NULL` в качестве нулевого указателя приведет к вызову перегрузки для `int`, но таким программистам необходимо внимательно прочесть раздел 3.2 и читать его до полного просвещения в данном вопросе.

## Диспетчеризация дескрипторов

Ни передача `lvalue`-ссылки на `const`, ни передача по значению не предоставляют поддержку прямой передачи. Если мотивом использования универсальной ссылки является прямая передача, мы вынуждены использовать универсальную ссылку; у нас просто нет иного выбора. Тем не менее мы не хотим отказываться и от перегрузки. Если мы не отказываемся ни от перегрузок, ни от универсальных ссылок, то как же мы сможем избежать перегрузки для универсальных ссылок?

В действительности это не так трудно. Вызовы перегруженных функций разрешаются путем просмотра всех параметров всех перегрузок, а также всех аргументов в точке вызова с последующим выбором функции с наилучшим общим соответствием — с учетом всех комбинаций “параметр/аргумент”. Параметр, являющийся универсальной ссылкой, обычно обеспечивает точное соответствие для всего, что бы ни было передано, но если универсальная ссылка является частью списка параметров, содержащего другие параметры, универсальными ссылками не являющиеся, достаточно плохое соответствие этих последних параметров может привести к отказу от вызова такой перегрузки. Эта идея лежит в основе подхода *диспетчеризации дескрипторов* (*tag dispatch*), а приведенный ниже пример позволит лучше понять, о чем идет речь.

Применим этот метод к примеру `logAndAdd` из третьего фрагмента кода раздела 5.4. Чтобы вам не пришлось его искать, повторим его здесь:

```

std::multiset<std::string> names; // Глобальная структура данных

template<typename T>           // Делает запись в журнале и
void logAndAdd(T& name)        // добавляет name в names

```

```

{
    auto now = std::chrono::system_clock::now();
    log(now, "logAndAdd");
    names.emplace(std::forward<T>(name));
}

```

Сама по себе эта функция работает отлично, но если мы добавим перегрузку, принимающую значение типа `int`, использующееся для поиска объекта по индексу, то получим проблемы, описанные в разделе 5.4. Цель данного раздела — их избежать. Вместо добавления перегрузки мы реализуем `logAndAdd` заново для делегирования работы двум другим функциям: одной — для целочисленных значений, а другой — для всего прочего. Сама функция `logAndAdd` будет принимать все типы аргументов, как целочисленные, так и нет.

Эти две функции, выполняющие реальную работу, будут называться `logAndAddImpl`, т.е. мы воспользуемся перегрузкой. Одна из этих функций будет принимать универсальную ссылку. Так что у нас будет одновременно и перегрузка, и универсальные ссылки. Но каждая функция будет принимать и второй параметр, указывающий, является ли передаваемый аргумент целочисленным значением. Этот второй параметр и будет средством, избавляющим нас от падений в болото, описанное в разделе 5.4, так как он будет фактором, определяющим выбираемую перегрузку.

Что вы говорите? “Хватит трепа, переходи к делу”? Да хоть сию секунду! Вот почти корректная версия обновленной функции `logAndAdd`:

```

template<typename T>
void logAndAdd(T&& name)
{
    logAndAddImpl(std::forward<T>(name),
                  std::is_integral<T>()); // Не совсем корректно
}

```

Эта функция передает свой параметр в `logAndAddImpl` и при этом передает также аргумент, указывающий, является ли тип параметра (`T`) целочисленным. Как минимум это то, что она должна делать. То же самое она делает и для целочисленных аргументов, являющихся `rvalue`. Но, как поясняется в разделе 5.6, если `lvalue`-аргумент передается универсальной ссылке `name`, то выведенный тип для `T` будет `lvalue`-ссылкой. Так что если в функцию `logAndAdd` передается `lvalue` типа `int`, то тип `T` будет выведен как `int&`. Но это не целочисленный тип — ссылка таковым не является. Это означает, что `std::is_integral<T>` будет иметь ложное значение для любого `lvalue`-аргумента, даже если этот аргумент на самом деле является целочисленным значением.

Понимание данной проблемы равносильно ее решению, поскольку в стандартной библиотеке имеется такое средство, как `std::remove_reference` (см. раздел 3.3), которое делает то, о чем говорит его имя и в чем мы так нуждаемся: удаляет любые квалификаторы ссылок из типа. Так что верный способ написания `logAndAdd` имеет следующий вид:

```

template<typename T>
void logAndAdd(T&& name)
{

```

```

logAndAddImpl(
    std::forward<T>(name),
    std::is_integral<
        typename std::remove_reference<T>::type
    >()
);
}

```

Это в определенной мере трюк. (Кстати, в C++14 можно сэкономить несколько нажатий клавиш, воспользовавшись вместо выделенного текста `std::remove_reference_t<T>`. Подробнее об этом рассказывается в разделе 3.3.)

После принятия этих мер мы можем перенести наше внимание к вызываемой функции, `logAndAddImpl`. Имеются две перегрузки, первая из которых применима к любому нецелочисленному типу (т.е. ко всем типам, для которых значение `std::is_integral<typename std::remove_reference<T>::type>` ложно):

```

// Нецелочисленный аргумент добавляется
// в глобальную структуру данных:
template<typename T>
void logAndAddImpl(T&& name, std::false_type)
{
    auto now = std::chrono::system_clock::now();
    log(now, "logAndAdd");
    names.emplace(std::forward<T>(name));
}

```

Этот код прост, если вы понимаете механику, лежащую в основе выделенного параметра. Концептуально `logAndAdd` передает в функцию `logAndAddImpl` булево значение, указывающее, передан ли функции `logAndAdd` целочисленный тип, но значения `true` и `false` являются значениями *времени выполнения*, а нам для выбора верной версии `logAndAddImpl` необходимо разрешение перегрузки, т.е. явление *времени компиляции*. Это означает, что нам нужен *тип*, соответствующий значению `true`, и другой тип, соответствующий значению `false`. Такая необходимость — настолько распространенное явление, что стандартная библиотека предоставляет то, что нам нужно, под именами `std::true_type` и `std::false_type`. Аргумент, передаваемый в `logAndAddImpl` функцией `logAndAdd`, является объектом типа, унаследованного от `std::true_type`, если `T` — целочисленный тип, и от `std::false_type`, если `T` таковым не является. Конечный результат заключается в том, что эта перегрузка `logAndAddImpl` является реальным кандидатом для вызова в `logAndAdd`, только если `T` не является целочисленным типом.

Вторая перегрузка охватывает противоположный случай, когда `T` представляет собой целочисленный тип. В этом случае `logAndAddImpl` просто ищет имя, соответствующее целочисленному индексу, и передает это имя функции `logAndAdd`:

```

std::string nameFromIdx(int idx); // Как в разделе 5.4

// Целочисленный аргумент: поиск имени и
// вызов с этим именем функции logAndAdd:

```

```
void logAndAddImpl(int idx, std::true_type)
{
    logAndAdd(nameFromIdx(idx));
}
```

Наличие функции `logAndAddImpl` для поиска по индексу соответствующего имени и передача его функции `logAndAdd` (откуда оно будет передано с помощью `std::forward` другой перегрузке функции `logAndAddImpl`) позволяет избежать размещения кода для записи в журнале в обеих перегрузках `logAndAddImpl`.

В таком решении типы `std::true_type` и `std::false_type` являются “дескрипторами”, единственная цель которых — обеспечить разрешение перегрузки требующимся нам способом. Обратите внимание, что нам даже не нужны эти параметры. Они не служат никакой цели во время выполнения, и мы фактически надеемся, что компиляторы распознают, что параметры дескрипторов не используются, и соответствующим образом оптимизируют выполнимый образ программы. (Некоторые компиляторы так и поступают, по крайней мере иногда.) Вызов перегруженных функций реализации в `logAndAdd` “диспетчеризует” передачу работы правильной перегрузке путем создания нужного объекта дескриптора. Отсюда и название этого метода проектирования: *диспетчеризация дескрипторов*. Это стандартный строительный блок шаблонного метапрограммирования, и чем больше вы будете просматривать код внутри современных библиотек C++, тем чаще вы будете с ним сталкиваться.

Для наших целей важно не столько то, как работает диспетчеризация дескрипторов, сколько как она позволяет комбинировать универсальные ссылки и перегрузку без проблем, описанных в разделе 5.4. Функция диспетчеризации — `logAndAdd` — принимает параметр, являющийся неограниченной универсальной ссылкой, но эта функция не перегружается. Перегружается функция реализации — `logAndAddImpl`, — которая принимает параметр, представляющий собой универсальную ссылку, но разрешение вызова этой функции зависит не только от параметра универсальной ссылки, но и от параметра дескриптора, а значения дескрипторов спроектированы таким образом, чтобы было не более одной совпадающей перегрузки. В результате то, какая из перегрузок будет вызвана, определяется дескриптором. Тот факт, что параметр, представляющий собой универсальную ссылку, всегда генерирует точное соответствие своему аргументу, значения не имеет.

## Ограничения шаблонов, получающих универсальные ссылки

Ключевым моментом диспетчеризации дескрипторов является существование (неперегруженной) функции в качестве клиентского API. Эта единственная функция распределяет работу между функциями реализации. Обычно создать такую неперегруженную функцию диспетчеризации несложно, но второй пример, рассмотренный в разделе 5.4, в котором рассматривался конструктор класса `Person` с прямой передачей, является исключением. Компиляторы могут самостоятельно генерировать копирующие и перемещающие конструкторы, так что, если даже мы напишем один конструктор и используем в нем диспетчеризацию дескрипторов, некоторые вызовы конструкторов могут быть

обработаны сгенерированными компиляторами функциями, которые обходят систему диспетчеризации дескрипторов.

По правде говоря, реальная проблема не в том, что генерируемые компиляторами функции иногда обходят диспетчериацию дескрипторов; на самом деле она в том, что они *не всегда* ее обходят. Вы практически всегда хотите, чтобы копирующий конструктор класса обрабатывал запрос на копирование lvalue этого типа, но, как показано в разделе 5.4, предоставление конструктора, принимающего универсальную ссылку, приводит к тому, что при копировании неконстантных lvalue вызывается конструктор с универсальной ссылкой, а не копирующий конструктор. В этом разделе также поясняется, что когда базовый класс объявляет конструктор с прямой передачей, именно этот конструктор обычно вызывается при традиционной реализации производным классом копирующего и перемещающего конструкторов, несмотря на то что корректным поведением является вызов копирующих и перемещающих конструкторов.

Для подобных ситуаций, в которых перегруженная функция, принимающая универсальную ссылку, оказывается более “жадной”, чем вы хотели, но недостаточно жадной, чтобы действовать как единственная функция диспетчериизации, метод диспетчериизации дескрипторов оказывается не тем, что требуется. Вам нужна другая технология, и эта технология — std::enable\_if.

std::enable\_if дает вам возможность заставить компиляторы вести себя так, как если бы определенного шаблона не существовало. Такие шаблоны называют *отключеными* (disabled). По умолчанию все шаблоны *включены*, но шаблон, использующий std::enable\_if, включен, только если удовлетворяется условие, определенное std::enable\_if. В нашем случае мы хотели бы включить конструктор Person с прямой передачей, только если передаваемый тип не является Person. Если переданный тип — Person, то мы хотели бы отключить конструктор с прямой передачей (т.е. заставить компилятор его игнорировать), поскольку при этом для обработки вызова будет применен копирующий или перемещающий конструктор, а это именно то, чего мы хотим, когда один объект типа Person инициализируется другим объектом того же типа.

Способ выражения этой идеи не слишком сложен, но имеет отталкивающий синтаксис, в особенности если вы не встречались с ним ранее. Имеются некоторые шаблоны, располагающиеся вокруг части условия std::enable\_if, так что начнем с него. Вот объявление конструктора с прямой передачей класса Person, который показывает не более чем необходимо для простого использования std::enable\_if. Я покажу только объявление этого конструктора, поскольку применение std::enable\_if не влияет на реализацию функции. Реализация остается той же, что и в разделе 5.4:

```
class Person {  
public:  
    template<typename T,  
             typename = typename std::enable_if<условие>::type>  
    explicit Person(T&& n);  
};
```

Вынужден с прискорбием сообщить, что для того, чтобы разобраться, что происходит в выделенном тексте, вам следует проконсультироваться с другими источниками информации, так как в этой книге у меня просто нет места, чтобы подробно все описать. (В процессе вашего поиска поищите как `std::enable_if`, так и волшебную аббревиатуру “`SFINAE`”, поскольку именно эта технология позволяет работать `std::enable_if`.) Здесь я хочу сосредоточиться на выражении условия, которое управляет тем, является ли конструктор включенным.

Условие, которое мы хотим указать, — что тип `T` не является `Person`, т.е. что шаблонизированный конструктор может быть включенным, только если `T` является типом, отличным от `Person`. Благодаря свойствам шаблонов мы можем определить, являются ли два типа одним и тем же (`std::is_same`), так что создается впечатление, что интересующее нас условие можно записать как `!std::is_same<Person, T>::value`. (Обратите внимание на символ “!” в начале выражения. Мы хотим, чтобы типы `Person` и `T` не совпадали.) Это близко к тому, что нам надо, но не совсем верно, поскольку, как поясняет раздел 5.6, тип, выведенный для универсальной ссылки, инициализированной `lvalue`, всегда является `lvalue`-ссылкой. Это означает, что в коде наподобие

```
Person p("Nancy");
auto cloneOfP(p); // Инициализация с помощью lvalue
```

тип `T` в универсальном конструкторе будет выведен как `Person&`. Типы `Person` и `Person&` — разные, и результат `std::is_same` отражает этот факт: значение `std::is_same<Person, Person&>::value` ложно.

Если разобраться, что означает, что шаблонный конструктор в классе `Person` должен быть включен только тогда, когда `T` не является `Person`, то мы поймем, что, глядя на `T`, мы хотим игнорировать следующее.

- **Ссылки.** С точки зрения определения, должен ли быть включен конструктор с универсальной ссылкой, типы `Person`, `Person&` и `Person&&` должны рассматриваться как идентичные типу `Person`.
- **Модификаторы `const` и `volatile`.** С той же точки зрения типы `const Person`, `volatile Person` и `const volatile Person` должны рассматриваться как идентичные типу `Person`.

Это означает, что нам нужен способ удалить все ссылки, `const` и `volatile` из типа `T` перед тем как выяснить, совпадает ли он с типом `Person`. И вновь на выручку приходит стандартная библиотека, предоставляя шаблон `std::decay`. Тип `std::decay<T>::type` представляет собой то же самое, что и тип `T`, но из него удалены все ссылки и квалифиликаторы `const` и `volatile`. (Я немного вас обманул, потому что `std::decay`, кроме того, превращает массивы и типы функций в указатели (см. раздел 1.1), но для наших целей можно считать, что `std::decay` ведет себя так, как я описал.) Условие, которое должно выполняться для включения рассматриваемого конструктора, имеет вид

```
!std::is_same<Person, typename std::decay<T>::type>::value
```

т.е. Person не совпадает с типом T, без учета всех ссылок и квалификаторов const и volatile. (Как поясняется в разделе 3.3, ключевое слово typename перед std::decay необходимо, поскольку тип std::decay<T>::type зависит от параметра шаблона T.)

Вставка этого условия в шаблон std::enable\_if выше, а также форматирование результата для того, чтобы проще понять взаимоотношения между частями кода, дает следующее объявление конструктора с прямой передачей класса Person:

```
class Person {
public:
    template<
        typename T,
        typename = typename std::enable_if<
            !std::is_same<Person,
                typename std::decay<T>::type
            >::value
        >::type
    >
    explicit Person(T& n);

};
```

Если вы никогда ранее не видели ничего подобного, не пугайтесь. Есть причина, по которой я оставил этот метод напоследок. Если для того, чтобы избежать смешивания универсальных ссылок и перегрузки вы можете использовать один из прочих методов (а это почти всегда возможно), вы должны это сделать. Тем не менее, если вы привыкнете к функциональному синтаксису и множеству угловых скобок, это не так плохо. Кроме того, это позволяет получить поведение, к которому вы стремитесь. С учетом приведенного выше объявления построение объекта Person из другого объекта Person (lvalue или rvalue, с квалификатором const или без него, с квалификатором volatile или без него) никогда не вызовет конструктор, принимающий универсальную ссылку.

Мы добились успеха? Дело сделано?

Пока что нет. Не спешите праздновать. Раздел 5.4 все еще посыпает нам свои приветы. Нам надо заткнуть этот фонтан окончательно.

Предположим, что класс, производный от Person, реализует операции копирования и перемещения традиционным способом:

```
class SpecialPerson: public Person {
public:
    SpecialPerson(const SpecialPerson& rhs) // Копирующий
        : Person(rhs) // конструктор; вызывает конструктор
    { ... } // базового класса с прямой передачей!

    SpecialPerson(SpecialPerson&& rhs) // Перемещающий
        : Person(std::move(rhs)) // конструктор; вызывает конструктор
    { ... } // базового класса с прямой передачей!
};
```

Это тот же код, который вы видели ранее, в конце предыдущего раздела, включая комментарии, увы, оставшиеся справедливыми. Копируя или перемещая объект SpecialPerson, мы ожидаем, что части базового класса будут скопированы или перемещены с помощью копирующего или, соответственно, перемещающего конструктора базового класса. Однако в этих функциях мы передаем объекты SpecialPerson конструкторам базового класса, а поскольку SpecialPerson не совпадает с Person (даже после применения `std::decay`), конструктор с универсальной ссылкой в базовом классе оказывается включенным и без проблем проходит проверку на идеальное совпадение с аргументом SpecialPerson. Это точное соответствие лучше преобразования производного класса в базовый, необходимого для связывания объекта SpecialPerson с параметром Person в копирующем и перемещающем конструкторах класса Person, так что при имеющемся коде копирование и перемещение объектов SpecialPerson будет использовать для копирования и перемещения частей базового класса конструктор с универсальной ссылкой класса Person! Это чудное ощущение дежавю раздела 5.4...

Производный класс просто следует обычным правилам реализации копирующего и перемещающего конструкторов производного класса, поэтому решение этой проблемы находится в базовом классе и, в частности, в условии, которое контролирует включение конструктора с универсальной ссылкой класса Person. Теперь мы понимаем, что надо включать шаблонный конструктор не для любого типа аргумента, отличного от Person, а для любого типа аргумента, отличного как от Person, так и от *типа, производного от Person*. Ох уж это наследование!

Вас уже не должно удивлять обилие всяческих полезных шаблонов в стандартной библиотеке, так что известие о наличии шаблона, который определяет, является ли один класс производным от другого, вы должны воспринять с полным спокойствием. Он называется `std::is_base_of`. Значение `std::is_base_of<T1, T2>::value` истинно, если `T2` — класс, производный от `T1`. Пользовательские типы рассматриваются как производные от самих себя, так что `std::is_base_of<T, T>::value` истинно, если `T` представляет собой пользовательский тип. (Если `T` является встроенным типом, `std::is_base_of<T, T>::value` ложно.) Это удобно, поскольку мы хотим пересмотреть наше условие, управляющее отключением конструктора с универсальной ссылкой класса Person таким образом, чтобы этот конструктор был включен, только если тип `T` после удаления всех ссылок и квалификаторов `const` и `volatile` не являлся ни типом Person, ни классом, производным от Person. Применение `std::is_base_of` вместо `std::is_same` дает нам то, что требуется:

```
class Person {  
public:  
    template<  
        typename T,  
        typename = typename std::enable_if<  
            !std::is_base_of<Person,  
            typename std::decay<T>::type  
>::value
```

```

>::type
>

explicit Person(T&& n);

};

```

Вот теперь работа завершена. Вернее, завершена при условии, что мы пишем код на C++11. При использовании C++14 этот код будет работать, но мы можем использовать псевдонимы шаблонов для std::enable\_if и std::decay, чтобы избавиться от хлама в виде typename и ::type, получая несколько более приятный код:

```

class Person {                                     // C++14
public:
    template<
        typename T,
        typename = std::enable_if_t<           // Меньше кода здесь
            !std::is_base_of<Person,
                std::decay_t<T>               // И здесь
            >::value
        >                                // И здесь
    >

    explicit Person(T&& n);
};

}

```

Ладно, я признаю: я соврал. Мы до сих пор не сделали всю работу. Но мы уже близки к завершению. Соблазнительно близки. Честно!

Мы видели, как использовать std::enable\_if для выборочного отключения конструктора с универсальной ссылкой класса Person для типов аргументов, которые мы хотим обрабатывать с помощью копирующего и перемещающего конструкторов, но мы еще не видели, как применить его для того, чтобы отличать целочисленные аргументы от не являющихся таковыми. В конце концов, таковой была наша первоначальная цель; проблема неоднозначности конструктора была всего лишь неприятностью, подхваченной по дороге.

Все, что нам нужно сделать (и на этот раз “все” действительно означает, что это уже все), — это (1) добавить перегрузку конструктора Person для обработки целочисленных аргументов и (2) сильнее ограничить шаблонный конструктор так, чтобы он был отключен для таких аргументов. Положите эти ингредиенты в кастрюлю со всем остальным, что мы уже обсуждали, варите на медленном огне и наслаждайтесь ароматом успеха:

```

class Person {
public:
    template<
        typename T,
        typename = std::enable_if_t<
            !std::is_base_of<Person, std::decay_t<T>>::value
        >
    >

```

```

66
!std::is_integral<std::remove_reference_t<T>>::value
>
explicit Person(T&& n)      // Конструктор для std::string и
: name(std::forward<T>(n)) // аргументов, приводимых к
{ ... }                      // std::string

explicit Person(int idx)    // Конструктор для
: name(nameFromIdx(idx))    // целочисленных аргументов
{ ... }

...      // Копирующий и перемещающий конструкторы и т.д.

private:
    std::string name;
};

```

Красота! Ну ладно, красота в основном для тех, кто фетишизирует метапрограммирование с использованием шаблонов, но факт остается фактом: этот подход не только справляется с работой, но и делает это с чрезвычайным апломбом. Применение прямой передачи предполагает высокую эффективность, а управление сочетанием универсальных ссылок и перегрузки вместо их запрета позволяет применять этот метод в обстоятельствах (таких, как разработка конструкторов), когда перегрузка неизбежна.

## Компромиссы

Первые три рассмотренные в данном разделе метода — отказ от перегрузки, передача `const T&` и передача по значению — указывают тип каждого параметра в вызываемой функции или функциях. Последние два метода — диспетчеризация дескрипторов и ограничения шаблонов — используют прямую передачу, а следовательно, типы параметров не указывают. Это фундаментальное решение — указывать типы или нет — имеет свои следствия.

Как правило, прямая передача более эффективна, потому что позволяет избежать создания временных объектов исключительно с целью соответствия типу объявления параметра. В случае конструктора `Person` прямая передача допускает передачу строкового литерала, такого как "Nancy", непосредственно в конструктор для `std::string` внутри `Person`, в то время как методы, не использующие прямой передачи, вынуждены создавать временный объект `std::string` из строкового литерала для удовлетворения спецификации параметра конструктора `Person`.

Но прямая передача имеет свои недостатки. Один из них тот, что некоторые виды аргументов не могут быть переданными прямой передачей, несмотря на то что они могут быть переданы функциям, принимающим конкретные типы. Эти сбои прямой передачи исследуются в разделе 5.8.

Второй неприятностью является запутанность сообщений об ошибках, когда клиенты передают недопустимые аргументы. Предположим, например, что клиент, создающий

объект Person, передает строковый литерал, составленный из символов `char16_t` (тип C++11 для представления 16-разрядных символов) вместо `char` (из которых состоит `std::string`):

```
Person p(u"Konrad Zuse"); // "Konrad Zuse" состоит из
                           // символов типа const char16_t
```

При использовании первых трех из рассмотренных в данном разделе подходов компиляторы увидят, что доступные конструкторы могут принимать либо `int`, либо `std::string`, и выведут более или менее понятное сообщение об ошибке, поясняющее, что не существует преобразования из `const char16_t[12]` в `int` или `std::string`.

Однако при подходе с использованием прямой передачи массива `const char16_t` связывается с параметром конструктора без замечаний и жалоб. Оттуда он передается конструктору члена-данных типа `std::string` класса Person, и только в этот момент обнаруживается несоответствие между тем, что было передано (массив `const char16_t`), и тем, что требовалось (любой тип, приемлемый для конструктора `std::string`). В результате получается впечатляющее сообщение об ошибке. Так, в одном из компиляторов оно состояло более чем из 160 строк!

В этом примере универсальная ссылка передается только один раз (из конструктора Person в конструктор `std::string`), но чем более сложна система, тем больше вероятность того, что универсальная ссылка передается через несколько слоев вызовов функций до того, как достигнет точки, в которой определяется приемлемость типа аргумента (или типов). Чем большее количество раз будет передаваться универсальная ссылка, тем более непонятными и громоздкими будут выглядеть сообщения об ошибках, если что-то пойдет не так. Многие разработчики считают, что одно это является основанием для применения универсальных ссылок только там, где особенно важна производительность.

В случае класса Person мы знаем, что параметр универсальной ссылки передающей функции должен выступать в роли инициализатора для `std::string`, так что мы можем использовать `static_assert` для того, чтобы убедиться в его пригодности для этой роли. Свойство типа `std::is_constructible` выполняет проверку времени компиляции того, может ли объект одного типа быть построен из объекта (или множества объектов) другого типа (или множества типов), так что написать такую проверку несложно:

```
class Person {
public:
    template<                                     // Как и ранее
              typename T,
              typename = std::enable_if_t<
                  !std::is_base_of<Person, std::decay_t<T>>::value
                  &&
                  !std::is_integral<std::remove_reference_t<T>>::value
              >
    explicit Person(T&& n)
        : name(std::forward<T>(n))
```

```

    {
        // Проверка возможности создания std::string из T
        static_assert(
            std::is_constructible<std::string, T>::value,
            "Параметр p не может использоваться для "
            "конструирования а std::string"
        );
        // Здесь идет код обычного конструктора
    }
    // Остальная часть класса Person (как ранее)
};


```

В результате при попытке клиентского кода создать объект `Person` из типа, непригодного для построения `std::string`, будет выводиться указанное сообщение об ошибке. К сожалению, в этом примере `static_assert` находится в теле конструктора, а передающий код, являясь частью списка инициализации членов, ему предшествует. Использовавшиеся мною компиляторы выводят ясное и понятное сообщение об ошибке от `static_assert`, но только *после* обычных сообщений (всех этих 160 с лишним строк).

### Следует запомнить

- Альтернативы комбинации универсальных ссылок и перегрузки включают использование различных имен, передачу параметров как lvalue-ссылок на `const`, передачу параметров по значению и использование диспетчеризации дескрипторов.
- Ограничение шаблонов с помощью `std::enable_if` позволяет использовать универсальные ссылки и перегрузки совместно, но управляет условиями, при которых компиляторы могут использовать перегрузки с универсальными ссылками.
- Параметры, представляющие собой универсальные ссылки, часто имеют преимущества высокой эффективности, но обычно их недостатком является сложность использования.

## 5.6. Свертывание ссылок

В разделе 5.1 говорилось, что, когда аргумент передается в шаблонную функцию, выведенный для параметра шаблона тип указывает, является ли аргумент `lvalue` или `rvalue`. В разделе не было упомянуто, что это происходит только тогда, когда аргумент используется для инициализации параметра, являющегося универсальной ссылкой, но тому есть уважительная причина: универсальные ссылки появились только в разделе 5.2. Вместе эти наблюдения об универсальных ссылках и кодировании `lvalue/rvalue` означают, что для шаблона

```

template<typename T>
void func(T&& param);

```

выведенный параметр шаблона T будет включать информацию о том, был ли переданный в param аргумент lvalue или rvalue.

Механизм этого кодирования прост. Если в качестве аргумента передается lvalue, T выводится как lvalue-ссылка. При передаче rvalue вывод типа приводит к тому, что T не является ссылкой. (Обратите внимание: lvalue кодируются как lvalue-ссылки, но rvalue кодируются как *не ссылки*.) Следовательно:

```
Widget widgetFactory(); // Функция, возвращающая rvalue
Widget w;               // Переменная (lvalue)
func(w);                // Вызов функции с lvalue; тип T
                        // представляет собой Widget&
func(widgetFactory()); // Вызов функции с rvalue; тип T
                        // представляет собой Widget
```

В обоих вызовах func передается Widget, но так как один Widget является lvalue, а второй представляет собой rvalue, для параметра шаблона T выводятся разные типы. Это, как вы вскоре увидите, и определяет, чем становятся универсальные ссылки: rvalue- или lvalue-ссылками; а кроме того, это механизм, лежащий в основе работы std::forward.

Прежде чем более внимательно рассмотреть std::forward и универсальные ссылки, мы должны заметить, что ссылка на ссылку в C++ не существует. Попытайтесь объявить ее, и компилятор вынесет вам строгий выговор:

```
int x;
...
auto& & rx = x; // Ошибка! Объявлять ссылки на ссылки нельзя
```

Но рассмотрим, что произойдет, если передать lvalue шаблону функции, принимающую универсальную ссылку:

```
template<typename T>
void func(T&& param); // Как и ранее
func(w);              // Вызов func с lvalue;
                      // T выводится как Widget&
```

Если мы возьмем тип, выведенный для T (т.е. **Widget&**) и используем его для инстанцирования шаблона, то получим следующее:

```
void func(Widget& && param);
```

Ссылка на ссылку! Но компиляторы не возражают. Из раздела 5.2 мы знаем, что, поскольку универсальная ссылка param инициализируется с помощью lvalue, тип param должен быть lvalue-ссылкой, но как компилятор получит результат взятия выведенного типа для T и подстановки его в шаблон, который представляет собой конечную сигнатуру функции?

```
void func(Widget& param);
```

Ответ заключается в *свертывании ссылок* (*reference collapsing*). Да, вам запрещено объявлять ссылки на ссылки, но компиляторы могут создавать их в определенных

контекстах, среди которых — инстанцирование шаблонов. Когда компиляторы генерируют ссылки на ссылки, свертывание ссылок определяет, что будет дальше.

Существуют два вида ссылок (`lvalue` и `rvalue`), так что имеются четыре возможные комбинации “ссылка на ссылку” (`lvalue` на `lvalue`, `lvalue` на `rvalue`, `rvalue` на `lvalue` и `rvalue`). Если ссылка на ссылку возникает в контексте, где это разрешено (например, во время инстанцирования шаблона), то ссылки сворачиваются в единственную ссылку согласно следующему правилу:

Если любая из ссылок является `lvalue`-ссылкой, результат представляет собой `lvalue`-ссылку. В противном случае (т.е. когда обе ссылки являются `rvalue`-ссылками) результат представляет собой `rvalue`-ссылку.

В нашем приведенном выше примере подстановка выведенного типа `Widget&` в шаблон `func` дает `rvalue`-ссылку на `lvalue`-ссылку, и правило свертки ссылок гласит, что результатом является `lvalue`-ссылка.

Свертывание ссылок является ключевой частью механизма, обеспечивающего работу `std::forward`. Как пояснялось в разделе 5.3, `std::forward` применяется к параметрам, являющимся универсальными ссылками, так что обычно его применение имеет следующий вид:

```
template<typename T>
void f(T&& fParam)
{
    ...
    someFunc(std::forward<T>(fParam)); // Некоторая работа
                                         // Передача fParam в
                                         // someFunc
```

Поскольку `fParam` представляет собой универсальную ссылку, мы знаем, что параметр типа `T` будет кодировать информацию о том, являлся ли переданный в аргумент (т.е. выражение, использованное для инициализации `fParam`) `lvalue` или `rvalue`. Работа `std::forward` заключается в приведении `fParam` (`lvalue`) к `rvalue` тогда и только тогда, когда `T` гласит, что переданный в `f` аргумент был `rvalue`, т.е. если `T` не является ссылочным типом.

Вот как можно реализовать `std::forward`, чтобы он выполнял описанные действия:

```
template<typename T>          // В пространстве имен std
T&& forward(typename
            remove_reference<T>::type& param)
{
    return static_cast<T&&>(param);
```

Этот код не совсем отвечает стандарту (я опустил несколько деталей интерфейса), но отличия не играют роли для понимания того, как ведет себя `std::forward`.

Предположим, что аргумент, переданный `f`, является `lvalue` типа `Widget`. Тип `T` будет выведен как `Widget&`, а вызов `std::forward` инстанцирует `std::forward<Widget&>`. Подстановка `Widget&` в реализацию `std::forward` дает следующее:

```
Widget& && forward(typename  
                      remove_reference<Widget&>::type& param)  
{ return static_cast<Widget&&>(param); }
```

Свойство типа `std::remove_reference<Widget&>::type` дает `Widget` (см. раздел 3.3), так что `std::forward` превращается в

```
Widget& && forward(Widget& param)  
{ return static_cast<Widget&&>(param); }
```

К возвращаемому типу и приведению также применяется сворачивание ссылок, и результат представляет собой последнюю версию `std::forward` для вызова:

```
Widget& forward(Widget& param) // В пространстве  
{ return static_cast<Widget>(param); } // имен std
```

Как можно видеть, когда в шаблон функции `f` передается аргумент `lvalue`, `std::forward` инстанцируется для получения и возврата `lvalue`-ссылки. Приведение внутри `std::forward` не делает ничего, поскольку тип `param` уже представляет собой `Widget&`, так что приведение его к `Widget&` ни на что не влияет. Таким образом, `lvalue`-аргумент, переданный `std::forward`, вернет `lvalue`-ссылку. По определению `lvalue`-ссылки являются `lvalue`, так что передача `lvalue` в `std::forward` приводит к возврату `lvalue`, как и предполагалось.

Предположим теперь, что передаваемый `f` аргумент является `rvalue` типа `Widget`. В этом случае выведенный тип параметра типа `T` шаблона `f` будет просто `Widget`. Вызов `std::forward` внутри `f`, таким образом, будет представлять собой `std::forward<Widget>`. Подстановка `Widget` вместо `T` в реализации `std::forward` дает следующее:

```
Widget&& forward(typename  
                      remove_reference<Widget>::type& param)  
{ return static_cast<Widget&&>(param); }
```

Применение `std::remove_reference` к типу `Widget`, не являющемуся ссылкой, дает тот же тип, что и переданный (`Widget`), так что `std::forward` превращается в

```
Widget&& forward(Widget& param)  
{ return static_cast<Widget&&>(param); }
```

Здесь нет ссылок на ссылки, так что нет и свертывания ссылок, и это последняя инстанцированная версия `std::forward` для этого вызова.

Так как `rvalue`-ссылки, возвращаемые из функции, определены как `rvalue`, в этом случае `std::forward` превратит параметр `fParam` (`lvalue`) функции `f` в `rvalue`. Конечным результатом является то, что `rvalue`-аргумент, переданный функции `f`, будет передан функции `someFunc` как `rvalue`, и это именно то, что и должно было произойти.

Наличие в C++14 `std::remove_reference_t` делает возможным реализовать `std::forward` немного более лаконично:

```
template<typename T> // C++14; в
T& forward(remove_reference_t<T>& param) // пространстве
{
    return static_cast<T&>(param); // имен std
}
```

Свертывание ссылок происходит в четырех контекстах. Первый и наиболее распространенный — инстанцирование шаблонов. Второй — генерация типов для переменных `auto`. Детали, по сути, те же, что и для шаблонов, поскольку вывод типа для `auto`-переменных, по сути, совпадает с выводом типов для шаблонов (см. раздел 1.2). Рассмотрим еще раз пример, приводившийся ранее в данном разделе:

```
template<typename T>
void func(T& param);

Widget widgetFactory(); // Функция, возвращающая rvalue
Widget w; // Переменная (lvalue)
func(w); // Вызов функции с lvalue; тип T
          // представляет собой Widget&
func(widgetFactory()); // Вызов функции с rvalue; тип T
                      // представляет собой Widget
```

Это можно имитировать в виде `auto`. Объявление

```
auto&& w1 = w;
```

инициализирует `w1` с помощью `lvalue`, выводя, таким образом, для `auto` тип `Widget&`. Подстановка `Widget&` вместо `auto` в объявление для `w1` дает код со ссылкой на ссылку

```
Widget&& w1 = w;
```

который после сворачивания ссылок принимает вид

```
Widget& w1 = w;
```

В результате `w1` представляет собой `lvalue`-ссылку.

С другой стороны, объявление

```
auto&& w2 = widgetFactory();
```

инициализирует `w2` с помощью `rvalue`, приводя к тому, что для `auto` выводится тип `Widget`, не являющийся ссылкой. Подстановка `Widget` вместо `auto` дает

```
Widget&& w2 = widgetFactory();
```

Здесь нет ссылок на ссылки, так что процесс завершен; `w2` представляет собой `rvalue`-ссылку.

Теперь мы в состоянии по-настоящему понять универсальные ссылки, введенные в разделе 5.2. Универсальная ссылка не является новой разновидностью ссылок, в действительности это *lvalue*-ссылка в контексте, в котором выполняются два условия.

- **Вывод типа отличает lvalue от rvalue.** lvalue типа T выводится как имеющее тип T&, в то время как rvalue типа T дает в качестве выведенного типа T.
- **Происходит свертывание ссылок.**

Концепция универсальных ссылок полезна тем, что избавляет вас от необходимости распознавать наличие контекстов сворачивания, мысленного вывода различных типов для lvalue и rvalue и применения правила свертывания ссылок после мысленной подстановки выведенных типов в контексты, в которых они встречаются.

Я говорил, что имеется четыре контекста, но мы рассмотрели только два из них: инстанцирование шаблонов и генерацию типов auto. Третьим является генерация и использование typedef и объявлений псевдонимов (см. раздел 3.3). Если во время создания или вычисления typedef возникают ссылки на ссылки, для их устранения применяется сворачивание ссылок. Предположим, например, что у нас есть шаблон класса Widget с внедренным typedef для типа rvalue-ссылки

```
template<typename T>
class Widget {
public:
    typedef T& RvalueRefToT;
};
```

и предположим, что мы инстанцируем Widget с помощью типа lvalue-ссылки:

```
Widget<int&> w;
```

Подстановка int& вместо T в шаблоне Widget дает нам следующую конструкцию typedef:

```
typedef int & RvalueRefToT;
```

Сворачивание ссылок приводит этот код к

```
typedef int& RvalueRefToT;
```

Теперь ясно, что имя, которое мы выбрали для typedef, вероятно, не настолько описательно, как мы надеялись: RvalueRefToT представляет собой typedef для lvalue-ссылки, когда Widget инстанцируется типом lvalue-ссылки.

Последним контекстом, в котором имеет место сворачивание ссылок, является использование decltype. Если во время анализа типа, включающего decltype, возникает ссылка на ссылку, она устраняется сворачиванием ссылок. (Информацию о decltype вы найдете в разделе 1.3.)

### Следует запомнить

- Сворачивание ссылок встречается в четырех контекстах: инстанцирование шаблона, генерация типа `auto`, создание и применение `typedef` и объявлений псевдонимов, и `decltype`.
- Когда компиляторы генерируют ссылку на ссылку в контексте сворачивания ссылок, результатом становится единственная ссылка. Если любая из исходных ссылок является `lvalue`-ссылкой, результатом будет `lvalue`-ссылка; в противном случае это будет `rvalue`-ссылка.
- Универсальные ссылки представляют собой `rvalue`-ссылки в контекстах, в которых вывод типов отличает `lvalue` от `rvalue` и происходит сворачивание ссылок.

## 5.7. Считайте, что перемещающие операции отсутствуют, дороги или не используются

Семантика перемещения, пожалуй, самая главная возможность C++11. Вам наверняка приходилось слышать, что “перемещение контейнеров теперь такое же дешевое, как и копирование указателей” или что “копирование временных объектов теперь настолько эффективно, что избегать его равносильно преждевременной оптимизации”. Понять такие настроения легко. Семантика перемещения действительно является очень важной возможностью. Она не просто позволяет компиляторам заменять дорогостоящие операции копирования относительно дешевыми перемещениями, но и требует от них этого (при выполнении надлежащих условий). Возьмите ваш код C++98, перекомпилируйте его с помощью компилятора и стандартной библиотеки C++11 и — о чудо! — ваша программа заработает быстрее.

Семантика перемещения действительно в состоянии осуществить все это — и потому достойна легенды. Однако легенды, как правило, — это результат преувеличения. Цель данного раздела — спустить вас с небес на землю.

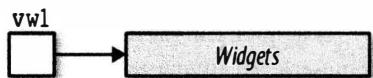
Начнем с наблюдения, что многие типы не поддерживают семантику перемещения. Вся стандартная библиотека C++98 была переработана с целью добавления операций перемещения для типов, в которых перемещение могло быть реализовано быстрее копирования, и реализаций компонентов библиотеки были пересмотрены с целью использования преимуществ новых операций; однако есть вероятность, что вы работаете с кодом, который не был полностью переделан под C++11. Для типов в ваших приложениях (или в используемых вами библиотеках), в которые не были внесены изменения для C++11, мало пользы от наличия поддержки перемещения компилятором. Да, C++11 готов генерировать перемещающие операции для классов, в которых они отсутствуют, но это происходит только для классов, в которых не объявлены копирующие операции, перемещающие операции или деструкторы (см. раздел 3.11). Члены-данные базовых классов типов, в которых перемещения отключены (например, путем удаления перемещающих операций; см. раздел 3.5) также подавляют перемещающие операции, генерируемые

компиляторами. Для типов без явной поддержки перемещения и типов, которые не могут претендовать на перемещающие операции, генерируемые компилятором, нет оснований ожидать что C++11 обеспечит повышение производительности по сравнению с C++98.

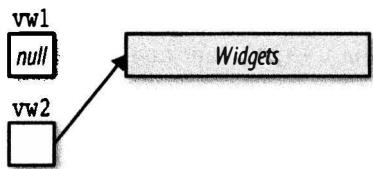
Даже типы с явной поддержкой перемещений не могут обеспечить все, на что вы надеетесь. Например, все контейнеры стандартной библиотеки C++11 поддерживают перемещение, но было бы ошибкой считать, что перемещение является дешевой операцией для всех контейнеров. Для одних контейнеров это связано с тем, что нет никакого действительно дешевого способа перемещения их содержимого. Для других — с тем, что действительно дешевые перемещающие операции предлагаются контейнерами с оговорками, которым не удовлетворяют конкретные элементы контейнера.

Рассмотрим новый контейнер C++11 — `std::array`. Контейнер `std::array`, по сути, представляет собой встроенный массив с STL-интерфейсом. Он фундаментально отличается от других стандартных контейнеров, которые хранят свое содержимое в динамической памяти. Объекты таких типов контейнеров концептуально содержат (в качестве членов-данных) только указатель на динамическую память, хранящую содержимое контейнера. (Действительность более сложна, но для наших целей эти различия не играют роли.) Наличие такого указателя позволяет перемещать содержимое всего контейнера на константное время: просто копируя указатель на содержимое контейнера из исходного контейнера в целевой и делая указатель исходного контейнера нулевым:

```
std::vector<Widget> vw1;
// Размещение данных в vw1
```

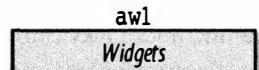


```
// Перемещение vw1 в vw2. Выполняется
// за константное время, изменяя
// только указатели в vw1 и vw2
auto vw2 = std::move(vw1);
```

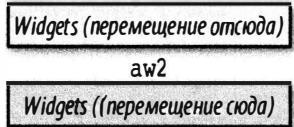


Объекты `std::array` не содержат такого указателя, поскольку данные, содержащиеся в `std::array`, хранятся непосредственно в объекте `std::array`:

```
std::array<Widget, 10000> aw1;
// Размещение данных в vw1
```



```
// Перемещение vw1 в vw2. Выполняется
// за линейное время. Все элементы
// aw1 перемещаются в aw2
auto aw2 = std::move(aw1);
```



Обратите внимание, что все элементы из `aw1` *перемещаются* в `aw2`. В предположении, что `Widget` представляет собой тип, операция перемещения которого выполняется быстрее операции копирования, перемещение `std::array` элементов `Widget` будет более

быстрым, чем копирование того же `std::array`. Поэтому `std::array` предлагает поддержку перемещения. И копирование, и перемещение `std::array` имеют линейное время работы, поскольку должен быть скопирован или перемещен каждый элемент контейнера. Это весьма далеко от утверждения “перемещение контейнеров теперь такое же дешевое, как и копирование указателей”, которое иногда приходится слышать.

С другой стороны, `std::string` предлагает перемещение за константное время и копирование — за линейное. Создается впечатление, что в этом случае перемещение быстрее копирования, но это может и не быть так. Многие реализации строк используют оптимизацию малых строк (small string optimization — SSO). При использовании SSO “малые” строки (например, размером не более 15 символов) хранятся в буфере в самом объекте `std::string`; выделение динамической памяти не используется. Перемещение малых строк при использовании реализации на основе SSO не быстрее копирования, поскольку трюк с копированием только указателя на данные, который в общем случае обеспечивает повышение эффективности, в данном случае не применим.

Мотивацией применения SSO является статистика, указывающая, что короткие строки являются нормой для многих приложений. С помощью внутреннего буфера для хранения содержимого таких строк устраняется необходимость динамического выделения памяти для них, и это, как правило, дает выигрыш в эффективности. Следствием этого выигрыша является то, что перемещение оказывается не быстрее копирования… Хотя для любителей наполовину полного стакана<sup>3</sup> можно сказать, что для таких строк копирование не медленнее, чем перемещение.

Даже для типов, поддерживающих быстрые операции перемещения, некоторые кажущиеся очевидными ситуации могут завершиться созданием копий. В разделе 3.8 поясняется, что некоторые контейнерные операции в стандартной библиотеке предполагают строгие гарантии безопасности исключений и что для гарантии того, что старый код C++98, зависящий от этой гарантии, не станет неработоспособным при переходе на C++11, операции копирования могут быть заменены операциями перемещения, только если известно, что последние не генерируют исключений. В результате, даже если тип предоставляет перемещающие операции, более эффективные по сравнению с соответствующими копирующими операциями, и даже если в определенной точке кода перемещающая операция целесообразна (например, исходный объект представляет собой `rvalue`), компиляторы могут быть вынуждены по-прежнему вызывать копирующие операции, поскольку соответствующая перемещающая операция не объявлена как `noexcept`.

Таким образом, имеется ряд сценариев, в которых семантика перемещения C++11 не пригодна.

- **Отсутствие перемещающих операций.** Объект, из которого выполняется перемещение, не предоставляет перемещающих операций. Запрос на перемещение, таким образом, превращается в запрос на копирование.
- **Перемещение не быстрее.** Объект, из которого выполняется перемещение, имеет перемещающие операции, которые не быстрее копирующих.

<sup>3</sup> Известная история о том, что об одном и том же до середины наполнением стакане пессимисты говорят, что он наполовину пуст, а оптимисты — что он наполовину полон. — Примеч. пер.

- **Перемещение неприменимо.** Контекст, в котором должно иметь место перемещение, требует операцию, не генерирующую исключения, но операция перемещения не объявлена как поэксперт.

Стоит упомянуть также еще один сценарий, когда семантика перемещения не приводит к повышению эффективности.

- **Исходный объект является lvalue.** За очень малыми исключениями (см., например, раздел 5.3) только rvalue могут использоваться в качестве источника перемещающей операции.

Но название этого раздела *предполагает* отсутствие перемещающих операций, их дороживизну или невозможность использования. Это типично для обобщенного кода, например, при написании шаблонов, поскольку вы не знаете всех типов, с которыми придется работать. В таких условиях вы должны быть настолько консервативными в отношении копирования объектов, как будто вы работаете с C++98, — до появления семантики перемещения. Это также случай “нестабильного” кода, т.е. кода, в котором характеристики используемых типов относительно часто изменяются.

Однако зачастую вам известно, какие типы использует ваш код, и вы можете положиться на неизменность их характеристик (например, на поддержку ими недорогих перемещающих операций). В этом случае вам не надо делать такие грустные предположения. Вы просто изучаете детали поддержки операций перемещения, используемых вашими типами. Если эти типы предоставляют недорогие операции перемещения и если вы используете объекты в контекстах, в которых эти операции будут вызываться, можете безопасно положиться на семантику перемещений при замене копирующих операций их менее дорогими перемещающими аналогами.

#### Следует запомнить

- Считайте, что перемещающие операции отсутствуют, дороги или не используются.
- В коде с известными типами или поддержкой семантики перемещения нет необходимости в таких предположениях.

## 5.8. Познакомьтесь с случаями некорректной работы прямой передачи

Одной из ярких звезд на небосклоне C++11 является прямая передача (*perfect forwarding*). Можно сказать, идеально прямая. Но, как говорит наука, даже пространство искривляется, так что есть идеальная прямота, а есть реальная. Прямая передача C++11 очень хороша, но достигает истинного совершенства, только если вы готовы игнорировать небольшие искривления. В данном разделе вы познакомитесь с этими маленькими искривлениями.

Перед тем как перейти к их изучению, стоит посмотреть, что мы подразумеваем под “прямой передачей”. Под передачей подразумевается, что одна функция *передает*

своим параметры другой функции. Цель этого действия заключается в том, чтобы одна функция (которой передаются параметры) получила в точности те же объекты, которые переданы другой функции (которая выполняет передачу). Тем самым исключается передача параметров по значению, поскольку при этом выполняется копирование исходно переданных объектов; мы же хотим, чтобы передающая функция была способна работать с изначально переданными объектами. Указатели также исключаются, поскольку мы не хотим заставлять вызывающий код передавать указатели. Когда речь идет о передаче общего назначения, мы работаем с параметрами, представляющими собой ссылки.

Прямая передача означает, что мы передаем не просто объекты, но и их основные характеристики: их типы, являются они lvalue или rvalue, объявлены они как const или volatile. В сочетании с наблюдением о том, что мы будем иметь дело со ссылочными параметрами, это означает, что мы будем использовать универсальные ссылки (см. раздел 5.2), поскольку только параметры, являющиеся универсальными ссылками, хранят информацию о том, какие аргументы — являющиеся lvalue или rvalue — были им переданы.

Предположим, что у нас есть некоторая функция `f` и мы хотели бы написать функцию (по правде говоря, шаблон функции), которая выполняет передачу ей. Ядро того, что нам надо, имеет следующий вид:

```
template<typename T>
void fwd(T&& param) // Принимает любой аргумент
{
    f(std::forward<T>(param)); // Передача аргумента в f
}
```

Передающие функции по своей природе являются обобщенными. Шаблон `fwd`, например, принимает аргумент любого типа, и он передает все, что бы ни получил. Логическим продолжением этой обобщенности являются передающие функции, являющиеся не просто шаблонами, а шаблонами с произвольным количеством аргументов (вариативными шаблонами (variadic templates)). Вариативная разновидность `fwd` выглядит следующим образом:

```
template<typename... Ts>
void fwd(Ts&&... params) // Принимает любые аргументы
{
    f(std::forward<Ts>(params)...); // Передача аргументов в f
}
```

Эту разновидность вы встретите, помимо прочих мест, в функциях размещения стандартных контейнеров (см. раздел 8.2) и в фабричных функциях для интеллектуальных указателей, `std::make_shared` и `std::make_unique` (см. раздел 4.4).

Для заданной целевой функции `f` и нашей передающей функции `fwd` прямая передача завершается неудачей, если вызов функции `f` с конкретным аргументом выполняет нечто одно, а вызов `fwd` с теми же аргументами — нечто иное:

```
f( expression ); // Если этот вызов выполняет что-то одно,  
fwd( expression ); // а этот – нечто иное, прямая передача  
// функцией fwd функции f неудачна
```

К такой неудаче могут привести несколько видов аргументов. Знать эти аргументы и то, как с ними работать, весьма важно, так что начнем наш тур по аргументам, которые не могут быть переданы с помощью прямой передачи.

## Инициализаторы в фигурных скобках

Предположим, что `f` объявлена следующим образом:

```
void f(const std::vector<int>& v);
```

В этом случае вызов `f` с инициализаторами в фигурных скобках компилируется:

```
f({ 1, 2, 3 }); // OK, "{1, 2, 3}" неявно преобразуется  
// в std::vector<int>
```

Однако передача того же инициализатора в фигурных скобках функции `fwd` не компилируется:

```
fwd({ 1, 2, 3 }); // Ошибка! Код не компилируется!
```

Дело в том, что применение инициализаторов в фигурных скобках — один из случаев, когда прямая передача терпит неудачу.

Все такие случаи отказов имеют одну и ту же причину. В непосредственном вызове `f` (таком, как `f({1,2,3})`) компиляторы видят аргументы, переданные в точке вызова, и видят типы параметров, объявленные функцией `f`. Они сравнивают аргументы в точке вызова с объявлениями параметров на предмет совместимости и при необходимости выполняют неявное преобразование, чтобы вызов был успешным. В приведенном выше примере они генерируют временный объект типа `std::vector<int>` из `{1,2,3}`, так что к параметру `v` функции `f` привязывается объект типа `std::vector<int>`.

При косвенном вызове `f` с помощью шаблона передающей функции `fwd` компиляторы больше не сравнивают аргументы, переданные в точке вызова `fwd`, с объявлениями параметров в `f`. Вместо этого они выводят типы аргументов, переданных в `fwd`, и сравнивают выведенные типы с объявлениями параметров в `f`. Прямая передача оказывается неудачной, если происходит что-то из следующего.

- **Компиляторы неспособны вывести тип** одного или нескольких параметров `fwd`. В этом случае код не компилируется.
- **Компиляторы выводят “неверный” тип** одного или нескольких параметров `fwd`. Здесь “неверный” может означать как то, что инстанцирование `fwd` не компилируется с выведенными типами, так и то, что вызов `f` с использованием выведенных типов `fwd` ведет себя не так, как непосредственный вызов `f` с аргументами, переданными в `fwd`. Одним источником такого отклонения в поведении могла бы быть ситуация, когда у `f` имеется перегрузка, и из-за “некорректного” вывода типов эта перегрузка `f`, вызываемая в `fwd`, отличалась бы от перегруженной функции `f`, используемой при непосредственном вызове.

В приведенном выше вызове “`fwd({1, 2, 3})`” проблема заключается в том, что передача инициализатора в фигурных скобках параметру шаблона функции, не объявленному как `std::initializer_list`, заставляет его быть, как предписывает стандарт, “не выводимым контекстом”. На простом человеческом языке это означает, что компиляторам запрещено выводить тип для выражения `{1, 2, 3}` в вызове `fwd`, поскольку параметр `fwd` не объявлен как `std::initializer_list`. Не имея возможности вывести тип параметра `fwd`, компиляторы, понятно, вынуждены отклонять такой вызов.

Интересно, что в разделе 1.2 поясняется, что вывод типа для переменных `auto`, инициализированных с помощью инициализатора в фигурных скобках, успешен. Такие переменные считаются объектами типа `std::initializer_list`, и это обеспечивает простой обходной путь для случаев, когда тип передающей функции должен быть выведен как `std::initializer_list`: объявить локальную переменную как `auto`, а затем передать ее в передающую функцию:

```
auto il = { 1, 2, 3 }; // Тип il выводится как
                      // std::initializer_list<int>
fwd(il);           // OK, прямая передача il в f
```

## 0 и NULL в качестве нулевых указателей

В разделе 3.2 поясняется, что, когда вы пытаетесь передать в шаблон 0 или `NULL` в качестве нулевого указателя, вывод типа для переданного аргумента дает вместо типа указателя целочисленный тип (обычно `int`). В результате ни 0, ни `NULL` не может быть передано с помощью прямой передачи как нулевой указатель. Решение проблемы простое: передавать `nullptr` вместо 0 и `NULL`. Детальную информацию вы можете найти в разделе 3.2.

## Целочисленные члены-данные `static const` и `constexpr` без определений

В качестве общего правила не требуется определять в классах целочисленные члены-данные `static const` и `constexpr`; одних объявлений вполне достаточно. Дело в том, что компиляторы выполняют *распространение const* для значений таких членов, тем самым устранив необходимость выделять для них память. Например, рассмотрим такой код:

```
class Widget {
public:
    // Объявление MinVals:
    static constexpr std::size_t MinVals = 28;
};

// Объявления MinVals нет

std::vector<int> WidgetData;
WidgetData.reserve(Widget::MinVals); // Использование MinVals
```

Здесь мы используем `Widget::MinVals` (далее — просто `MinVals`) для указания начальной емкости `widgetData`, даже несмотря на то, что определения `MinVals` нет. Компиляторы обходят отсутствующее определение (как и должны это делать) подстановкой значения `28` во все места, где упоминается `MinVals`. Тот факт, что для значения `MinVals` не выделена память, проблемой не является. Если берется адрес `MinVals` (например, кто-то создает указатель на `MinVals`), то `MinVals` требует места в памяти (чтобы указателю было на что указывать), и тогда приведенный выше код хотя и будет компилироваться, не будет компоноваться до тех пор, пока не будет предоставлено определение `MinVals`.

С учетом этого представим, что `f` (которой функция `fwd` передает аргумент) объявлен следующим образом:

```
void f(std::size_t val);
```

Вызов `f` с `MinVals` проблемы не представляет, поскольку компиляторы просто заменяют `MinVals` его значением:

```
f(Widget::MinVals); // OK, рассматривается как "f(28)"
```

Увы, все не так хорошо, если попытаться вызвать `f` через `fwd`:

```
fwd(Widget::MinVals); // Ошибка! Не должно компоноваться
```

Этот код компилируется, но не должен компоноваться. Если это напоминает вам происходящее при взятии адреса `MinVals`, это хорошо, потому что проблема в обоих случаях одна и та же.

Хотя нигде в исходном коде не берется адрес `MinVals`, параметром `fwd` является универсальная ссылка, а ссылки в коде, сгенерированном компилятором, обычно рассматриваются как указатели. В бинарном коде программы указатели и ссылки, по сути, представляют собой одно и то же. На этом уровне можно считать, что ссылки — это просто указатели, которые автоматически разыменовываются. В таком случае передача `MinVals` по ссылке фактически представляет собой то же, что и передача по указателю, а раз так, то должна иметься память, на которую этот указатель указывает. Передача целочисленных членов-данных `static const` и `constexpr` по ссылке в общем случае требует, чтобы они были определены, и это требование может привести к неудачному применению прямой передачи там, где эквивалентный код без прямой передачи будет успешен.

Возможно, вы обратили внимание на некоторые юркие слова, употребленные мною выше. Я сказал, что код “не должен” компоноваться. Ссылки “обычно” рассматриваются как указатели. Передача целочисленных членов-данных `static const` и `constexpr` по ссылке “в общем случае” требует, чтобы они были определены. Это похоже на то, как будто я что-то знаю, но ужасно не хочу вам сказать...

Это потому, что так и есть. В соответствии со стандартом передача `MinVals` по ссылке требует, чтобы этот член-данные был определен. Но не все реализации выполняют это требование. Так что в зависимости от ваших компиляторов и компоновщиков вы можете обнаружить, что в состоянии выполнить прямую передачу целочисленных членов-данных `static const` и `constexpr`, которые не были определены. Если это так — поздравляю, но нет причин ожидать, что такой код будет переносим. Чтобы сделать его

переносимым, просто добавьте определение интересующего вас целочисленного члена-данных, объявленного как static const или constexpr. Для MinVals это определение имеет следующий вид:

```
constexpr std::size_t Widget::MinVals; // В .cpp-файле Widget
```

Обратите внимание, что в определении не повторяется инициализатор (28 в случае MinVals). Не переживайте об этом. Если вы забудете предоставить инициализатор в обоих местах, компиляторы будут жаловаться, тем самым напоминая вам о необходимости указать его только один раз.

## Имена перегруженных функций и имена шаблонов

Предположим, что поведение нашей функции f (которой мы хотим передать аргументы через шаблон fwd) может настраиваться путем передачи ей некоторой функции, выполняющей определенную работу. В предположении, что эта функция получает и возвращает int, функция f может быть объявлена следующим образом:

```
void f(int (*pf)(int)); // pf – функция обработки
```

Стоит заметить, что f может также быть объявлена с помощью более простого синтаксиса без указателей. Такое объявление может иметь следующий вид, несмотря на то что оно означает в точности то же, что и объявление выше:

```
void f(int pf(int)); // Объявление той же f, что и выше
```

В любом случае теперь предположим, что у нас есть перегруженная функция processVal:

```
int processVal(int value);  
int processVal(int value, int priority);
```

Мы можем передать processVal функции f,

```
f(processVal); // Без проблем
```

но это выглядит удивительным. Функции f в качестве аргумента требуется указатель на функцию, но processVal не является ни указателем на функцию, ни даже функцией; это имя двух разных функций. Однако компиляторы знают, какая processVal нужна: та, которая соответствует типу параметра функции f. Таким образом, они могут выбрать processVal, принимающую один int, а затем передать адрес этой функции в функцию f.

Все это работает благодаря тому, что объявление f позволяет компиляторам вывести требующуюся версию функции processVal. Однако fwd представляет собой шаблон функции, не имеющий информации о том, какой тип ему требуется, а потому компиляторы не в состоянии определить, какая из перегрузок должна быть передана:

```
fwd(processVal); // Ошибка! Какая processVal?
```

Само по себе имя processVal не имеет типа. Без типа не может быть вывода типа, а без вывода типа мы получаем еще один случай неудачной прямой передачи.

Та же проблема возникает и если мы пытаемся использовать шаблон функции вместо перегруженного имени функции или в дополнение к нему. Шаблон функции представляет не единственную функцию, он представляет множество функций:

```
template<typename T>
T workOnVal(T param) // Шаблон для обработки значений
{ ... }

fwd(workOnVal); // Ошибка! Какое инстанцирование workOnVal?
```

Получить функцию прямой передачи наподобие fwd, принимающую имя перегруженной функции или имя шаблона, можно, вручную указав перегрузку (или инстанцирование), которую вы хотите передать. Например, вы можете создать указатель на функцию того же типа, что и параметр f, инициализировать этот указатель с помощью processVal или workOnVal (тем самым обеспечивая выбор корректной версии processVal или генерацию корректного инстанцирования workOnVal) и передать его шаблону fwd:

```
using ProcessFuncType =           // Псевдоним типа (раздел 3.3)
    int (*) (int);
ProcessFuncType processValPtr = // Определяем необходимую
    processVal;                 // сигнатуру processVal
fwd(processValPtr);           // OK
fwd(static_cast<ProcessFuncType>(workOnVal)); // Также OK
```

Конечно, это требует знания типа указателя на функцию, передаваемого шаблоном fwd. Разумно предположить, что функция с прямой передачей это документирует. В конце концов, функции с прямой передачей предназначены для принятия *всего* что угодно, так что если нет никакой документации, говорящей, что должно быть передано, то откуда вы узнаете, что им передавать?

## Битовые поля

Последний случай неудачной прямой передачи — когда в качестве аргумента функции используется битовое поле. Чтобы увидеть, что это означает на практике, рассмотрим заголовок IPv4, который может быть смоделирован следующим образом:<sup>4</sup>

```
struct IPv4Header {
    std::uint32_t version:4,
                  IHL:4,
                  DSCP:6,
                  ECN:2,
                  totalLength:16;
};
```

<sup>4</sup>Здесь предполагается, что битовые поля располагаются от младшего бита к старшему. C++ это не гарантирует, но компиляторы часто предлагают механизм, который позволяет программисту управлять схемой размещения битовых полей.

Если наша многострадальная функция `f` (являющаяся целевой для нашей функции прямой передачи `fwd`) объявлена как принимающая параметр `std::size_t`, то вызов, скажем, с полем `totalLength` объекта `IPv4Header` компилируется без проблем:

```
void f(std::size_t sz); // Вызываемая функция
IPv4Header h;
...
f(h.totalLength); // Все в порядке
```

Однако попытка передать `h.totalLength` в `f` через `fwd` — это совсем другая история:  
`fwd(h.totalLength); // Ошибка!`

Проблема заключается в том, что параметр функции `fwd` представляет собой ссылку, а `h.totalLength` — неконстантное битовое поле. Это может показаться не столь уж плохим, но стандарт C++ осуждает это сочетание в непривычно ясной форме: “неконстантная ссылка не может быть привязана к битовому полю”. Тому есть превосходная причина. Битовые поля могут состоять из произвольных частей машинных слов (например, биты 3–5 из 32-битного `int`), но непосредственно их адресовать нет никакой возможности. Ранее я упоминал, что ссылки и указатели представляют собой одно и то же на аппаратном уровне, и просто так же, как нет никакого способа создать указатели на отдельные биты (C++ утверждает, что наименьшей сущностью, которую вы можете адресовать, является `char`), нет никакого способа связать ссылку с произвольными битами.

Обходной путь для невозможности прямой передачи битовых полей становится простым, как только вы осознаете, что любая функция, принимающая битовое поле в качестве аргумента, на самом деле получает копию значения битового поля. В конце концов, никакая функция не может привязать ссылку к битовому полю, поскольку не существует указателей на битовые поля. Единственная разновидность параметров, которым могут передаваться битовые поля, — это параметры, передаваемые по значению, и, что интересно, ссылки на `const`. В случае передачи по значению вызываемая функция, очевидно, получает копию значения в битовом поле, и оказывается, что в случае параметра, являющегося ссылкой на `const`, стандарт требует, чтобы эта ссылка в действительности была привязана к копии значения битового поля, сохраненного в объекте некоторого стандартного целочисленного типа (например, `int`). Ссылки на `const` не привязываются к битовым полям, они привязываются к “нормальным” объектам, в которые копируются значения битовых полей.

Итак, ключом к передаче битового поля в функцию с прямой передачей является использование того факта, что функция, в которую осуществляется передача, всегда получает копию значения битового поля. Таким образом, вы можете сделать копию самостоятельно и вызвать передающую функцию, передав ей копию. В нашем примере с `IPv4Header` код, осуществляющий этот подход, имеет следующий вид:

```
// Копирование значения битового поля; см. раздел 2.2
auto length = static_cast<std::uint16_t>(h.totalLength);
fwd(length); // Передача копии
```

## Резюме

В большинстве случаев прямая передача работает так, как разрекламировано. Вы редко должны задумываться о ней. Но когда она не работает (разумно выглядящий код не компилируется или, хуже того, компилируется, но работает не так, как вы ожидаете), важно знать о несовершенствах прямой передачи. Не менее важно знать, как эти несовершенства обойти. В большинстве случаев это довольно просто.

### Следует запомнить

- Прямая передача неудачна, когда вывод типа не удается выполнить или когда он выводит неверный тип.
- Разновидностями аргументов, которые приводят к неудачам при прямой передаче, являются инициализаторы в фигурных скобках, нулевые указатели, выраженные как 0 или NULL, целочисленные члены-данные, объявленные как const static и не имеющие определений, имена шаблонов и перегруженных функций и битовые поля.



# Лямбда-выражения

Лямбда-выражения, иногда называемые просто *лямбдами* (*lambdas*), существенно изменили правила игры в программировании на C++. Это несколько удивительно, так как они не внесли в язык никаких новых возможностей выражения идей. Все, на что способны лямбда-выражения, вы в состоянии сделать вручную, разве что ценой немного больших усилий по вводу с клавиатуры. Но лямбда-выражения представляют собой очень удобное средство создания функциональных объектов, оказывающее огромное влияние на повседневную разработку программного обеспечения на C++. Без лямбда-выражений алгоритмы “*\_if*” из STL (например, `std::find_if`, `std::remove_if`, `std::count_if` и др.) обычно работают с самыми тривиальными предикатами, но при доступности лямбда-выражений применение этих алгоритмов резко возрастает. То же самое верно и для алгоритмов, настраиваемых с помощью пользовательской функции сравнения (например, `std::sort`, `std::nth_element`, `std::lower_bound` и др.). Вне STL лямбда-выражения позволяют быстро создавать пользовательские удалители для `std::unique_ptr` и `std::shared_ptr` (см. разделы 4.1 и 4.2) и делают столь же простыми спецификации предикатов для переменных условий в потоковом API (см. раздел 7.5). Помимо использования в объектах стандартной библиотеки, лямбда-выражения облегчают определение функций обратного вызова “на лету”, функций адаптации интерфейса и контекстно-зависимых функций для разовых вызовов. Лямбда-выражения действительно делают C++ более приятным языком программирования.

Терминология, связанная с лямбда-выражениями, может обескуражить. *Лямбда-выражение* является именно тем, что написано: выражением. Это часть исходного текста. В приведенном ниже фрагменте выделенная полужирным шрифтом часть представляет собой лямбда-выражение.

```
std::find_if(container.begin(), container.end(),
    [] (int val) { return 0 < val && val < 10; });
```

- *Замыкание* (*closure*) представляет собой объект времени выполнения, создаваемый лямбда-выражением. В зависимости от режима захвата замыкания хранят копии ссылок на захваченные данные. В приведенном выше вызове `std::find_if` замыкание представляет собой объект, передаваемый во время выполнения в `std::find_if` в качестве третьего аргумента.

- Класс замыкания (closure class) представляет собой класс, из которого инстанцируется замыкание. Каждое лямбда-выражение заставляет компиляторы генерировать уникальный класс замыкания. Инструкции внутри лямбда-выражения становятся выполнимыми инструкциями функций-членов их класса замыкания.

Лямбда-выражения часто используются для создания замыкания, которое применяется только в качестве аргумента функции. Именно этот случай представлен в приведенном выше примере `std::find_if`. Однако в общем случае замыкания могут копироваться, так что обычно можно иметь несколько замыканий типа замыкания, соответствующего одному лямбда-выражению. Например, в коде

```
{  
    int x;                                // x – локальная переменная  
    ...  
    auto c1 =                               // c1 – копия замыкания,  
        [x](int y) {                      // сгенерированного  
            return x * y > 55; }; // лямбда-выражением  
    auto c2 = c1;                          // c2 – копия c1  
    auto c3 = c2;                          // c3 – копия c2  
}
```

`c1`, `c2` и `c3` являются копиями замыкания, порожденного лямбда-выражением.

Неформально вполне приемлемо размытие границы между лямбда-выражениями, замыканиями и классами замыканий. Но в последующих разделах очень часто важно различать, что существует во время компиляции (лямбда-выражения и классы замыканий), что — во время выполнения (замыкания) и как они соотносятся одно с другим.

## 6.1. Избегайте режимов захвата по умолчанию

В C++11 имеются два режима захвата: по ссылке и по значению. Захват по умолчанию по ссылке может привести к висячим ссылкам. Захват по умолчанию по значению сбазняет считать, что вы не подвержены этой проблеме (это не так), и думать, что ваши замыкания являются самодостаточными автономными замыканиями (но они могут и не быть таковыми).

Так выглядят основные положения данного раздела. Если вы в большей степени творец, чем ремесленник, вы, вероятно, захотите узнать побольше — так давайте начнем с опасности захвата по ссылке.

Захват по ссылке приводит к тому, что замыкание содержит ссылку на локальную переменную или на параметр, доступный в области видимости, где определено лямбда-выражение. Если время жизни замыкания, созданного из лямбда-выражения, превышает время жизни локальной переменной или параметра, ссылка в замыкании оказывается висячей. Предположим, например, что у нас есть контейнер с функциями фильтрации, каждая из которых принимает `int` и возвращает `bool`, указывающий, удовлетворяет ли переданное значение фильтру:

```
// См. using в разделе 3.3, std::function – в 2.1:  
using FilterContainer = std::vector<std::function<bool(int)>>;  
  
FilterContainer filters; // Функции фильтрации
```

Мы можем добавить фильтр для чисел, кратных 5, следующим образом:

```
filters.emplace_back( // См. emplace_back в разделе 8.2  
    [](int value) { return value % 5 == 0; }  
);
```

Однако может быть так, что нам нужно вычислять делитель во время выполнения, т.е. мы не можем жестко “просить” значение 5 в лямбда-выражении. Поэтому добавление фильтра может выглядеть следующим образом:

```
void addDivisorFilter()  
{  
    auto calc1 = computeSomeValue1();  
    auto calc2 = computeSomeValue2();  
    auto divisor = computeDivisor(calc1, calc2);  
    filters.emplace_back( // Опасно! Ссылка на divisor повиснет!  
        [&](int value) { return value % divisor == 0; }  
    );  
}
```

Этот код — бомба замедленного действия. Лямбда-выражение ссылается на локальную переменную divisor, но эта переменная прекращает свое существование после выхода из addDivisorFilter. Этот выход осуществляется сразу после выхода из filters.emplace\_back, так что добавленная в filters функция, по сути, является мертворожденной. Применение этого фильтра ведет к неопределенному моменту практически с момента создания.

Та же проблема имеет место и при явном захвате divisor по ссылке,

```
filters.emplace_back(  
    [&divisor](int value) // Опасно! Ссылка на  
    { return value % divisor == 0; } // divisor все равно  
); // повисает!
```

но при явном захвате проще увидеть, что жизнеспособность лямбда-выражения зависит от времени жизни divisor. Кроме того, использование имени divisor напоминает нам о необходимости убедиться, что divisor существует как минимум столько же времени, сколько и замыкание лямбда-выражения. Это более конкретное напоминание, чем обобщенное “убедитесь, что ничего не висит”, о чём гласит конструкция [&].

Если вы знаете, что замыкание будет использовано немедленно (например, будучи переданным в алгоритм STL) и не будет копироваться, нет никакого риска, что ссылки, которые оно хранит, переживут локальные переменные и параметры в среде, где создано это лямбда-выражение. Вы могли бы утверждать, что в этом случае нет риска получить висячие ссылки, а следовательно, нет причин избегать режима захвата по ссылке

по умолчанию. Например, наше фильтрующее лямбда-выражение может использоваться только как аргумент в алгоритме C++11 `std::all_of`, который проверяет, все ли элементы диапазона удовлетворяют условию:

```
template<typename C>
void workWithContainer(const C& container)
{
    auto calc1 = computeSomeValue1();           // Как и ранее
    auto calc2 = computeSomeValue2();           // Как и ранее
    auto divisor = computeDivisor(calc1, calc2); // Как и ранее

    // Тип элементов в контейнере:
    using ContElemT = typename C::value_type;

    using std::begin;                         // Для обобщенности;
    using std::end;                           // см. раздел 3.7

    if (std::all_of(
        begin(container),                  // Все значения
        end(container),                  // в контейнере
        [ & ](const ContElemT& value)      // кратны divisor?
        { return value % divisor == 0; }))
    {
        ...
    }
} else {
    ...
}
```

Да, все так, это безопасно, но эта безопасность довольно неустойчива. Если выяснится, что это лямбда-выражение полезно в других контекстах (например, как функция, добавленная в контейнер `filters`) и будет скопировано и вставлено в контекст, где это замыкание может пережить `divisor`, вы вновь вернетесь к повисшим ссылкам, и в выражении захвата не будет ничего, что напомнило бы вам о необходимости анализа времени жизни `divisor`.

С точки зрения долгосрочной перспективы лучше явно перечислять локальные переменные, от которых зависит лямбда-выражение.

Кстати, возможность применения `auto` в спецификации параметров лямбда-выражений в C++14 означает, что приведенный выше код можно записать проще при использовании C++14. Определение псевдонима типа `ContElemT` можно убрать, а условие `if` может быть переписано следующим образом:

```
if (std::all_of(begin(container), end(container),
    [&](const auto& value)           // C++14
    { return value % divisor == 0; }))
```

Одним из способов решения нашей проблемы с локальной переменной divisor может быть применение режима захвата по умолчанию по значению, т.е. мы можем добавить лямбда-выражение к filters следующим образом:

```
filters.emplace_back(           // Теперь
    [=] (int value)           // divisor
    { return value % divisor == 0; } // не может
);                           // зависнуть
```

Для данного примера этого достаточно, но в общем случае захват по умолчанию по значению не является лекарством от висящих ссылок, как вам могло бы показаться. Проблема в том, что если вы захватите указатель по значению, то скопируете его в замыкания, возникающие из лямбда-выражения, но не сможете предотвратить освобождение объекта, на который он указывает (и соответственно, повисания), внешним кодом.

“Этого не может случиться! — возразите вы. — После прочтения главы 4 я работаю только с интеллектуальными указателями. Обычные указатели используют только несчастные программисты на C++98”. Это может быть правдой, но это не имеет значения, потому что на самом деле вы используете обычные указатели, а они могут быть удалены. Да, в современном стиле программирования на C++ в исходном коде это незаметно, но это так.

Предположим, что одна из задач, которые могут решать Widget, — добавление элементов в контейнер фильтров:

```
class Widget {
public:
    ...                         // Конструкторы и т.п.
    void addFilter() const; // Добавление элемента в filters
private:
    int divisor;               // Используется в фильтре
};
```

Widget::addFilter может быть определен следующим образом:

```
void Widget::addFilter() const
{
    filters.emplace_back(
        [=] (int value) { return value % divisor == 0; }
    );
}
```

Для блаженно непосвященных код выглядит безопасным. Лямбда-выражение зависит от divisor, но режим захвата по умолчанию по значению гарантирует, что divisor копируется в любое замыкание, получающееся из лямбда-выражения, так ведь?

Нет. Совершенно не так. Ужасно не так! Смертельно не так!

Захваты применяются только к нестатическим локальным переменным (включая параметры), видимым в области видимости, в которой создано лямбда-выражение. В теле Widget::addFilter переменная divisor не является локальной переменной,

это — член-данные класса Widget. Она не может быть захвачена. Если отменить режим захвата по умолчанию, код компилироваться не будет:

```
void Widget::addFilter() const
{
    filters.emplace_back(      // Ошибка! divisor недоступна!
        [](int value) { return value % divisor == 0; }
    );
}
```

Кроме того, если сделана попытка явного захвата divisor (по значению или по ссылке — значения не имеет), захват не компилируется, поскольку divisor не является локальной переменной или параметром:

```
void Widget::addFilter() const
{
    filters.emplace_back(      // Ошибка! Нет захватываемой
        [divisor](int value) // локальной переменной divisor!
        { return value % divisor == 0; }
    );
}
```

Если захват по умолчанию по значению не захватывает divisor, а без захвата по умолчанию по значению код не компилируется, то что же происходит?

Объявление связано с неявным использованием обычного указателя: this. Каждая нестатическая функция-член получает указатель this, и вы используете этот указатель всякий раз при упоминании члена-данных этого класса. В любой функции-члене Widget, например, компиляторы внутренне заменяют каждое использование divisor на this->divisor. В версии Widget::addFilter с захватом по умолчанию по значению

```
void Widget::addFilter() const
{
    filters.emplace_back(
        [=](int value) { return value % divisor == 0; }
    );
}
```

this->divisor захватывается указатель this объекта Widget, а не divisor. Компиляторы рассматривают этот код так, как будто он написан следующим образом:

```
void Widget::addFilter() const
{
    auto currentObjectPtr = this;
    filters.emplace_back(
        [currentObjectPtr](int value)
        { return value % currentObjectPtr->divisor == 0; }
    );
}
```

Понимание этого равносильно пониманию того, что жизнеспособность замыканий, вытекающих из этого лямбда-выражения, связана со временем жизни объекта Widget, копии указателя this которого в них содержатся. В частности, рассмотрим код, который в соответствии с главой 4 использует только интеллектуальные указатели:

```
using FilterContainer = std::vector<std::function<bool(int)>>; // Как и ранее

FilterContainer filters; // Как и ранее

void doSomeWork()
{
    auto pw = std::make_unique<Widget>(); // Создание Widget;
    // std::make_unique см. в
    // разделе 4.4
    pw->addFilter(); // Добавление фильтра
    // с Widget::divisor

} // Уничтожение Widget; filters хранит висячий указатель!
```

Когда выполняется вызов doSomeWork, создается фильтр, зависящий от объекта Widget, созданного std::make\_unique, т.е. фильтр, который содержит копию указателя на этот Widget, — указатель this объекта Widget. Этот фильтр добавляется в filters, но по завершении работы doSomeWork объект Widget уничтожается, так как std::unique\_ptr управляет его временем жизни (см. раздел 4.1). С этого момента filters содержит элемент с висячим указателем.

Эта конкретная проблема может быть решена путем создания локальной копии член-данных, который вы хотите захватить, и захвата этой копии:

```
void Widget::addFilter() const
{
    auto divisorCopy = divisor; // Копирование
    filters.emplace_back( // члена-данных
        [divisorCopy](int value) // Захват копии
        { return value % divisorCopy == 0; } // Ее использование
    );
}
```

Чтобы быть честным, скажу, что при таком подходе захват по умолчанию по значению также будет работать:

```
void Widget::addFilter() const
{
    auto divisorCopy = divisor; // Копирование
    filters.emplace_back( // члена-данных
        [=](int value) // Захват копии
        { return value % divisorCopy == 0; } // Ее использование
    );
}
```

но зачем искушать судьбу? Режим захвата по умолчанию делает возможным случайный захват `this`, когда вы думаете, в первую очередь, о захвате `divisor`.

В C++14 имеется лучший способ захвата члена-данных, заключающийся в использовании обобщенного захвата лямбда-выражения (см. раздел 6.2):

```
void Widget::addFilter() const
{
    filters.emplace_back(           // C++14:
        [divisor = divisor](int value) // Копирование divisor
            // в замыкание
        { return value % divisor == 0; } // Использование копии
    );
}
```

Однако такого понятия, как режим захвата по умолчанию для обобщенного захвата лямбда-выражения, не существует, так что даже в C++14 остается актуальным совет данного раздела — избегать режимов захвата по умолчанию.

Дополнительным недостатком режимов захвата по умолчанию является то, что они могут предполагать самодостаточность соответствующих замыканий и их изолированность от изменений внешних данных. В общем случае это не так, поскольку лямбда-выражения могут зависеть не только от локальных переменных и параметров (которые могут быть захвачены), но и от объектов со *статическим временем хранения*. Такие объекты определены в глобальной области видимости или области видимости пространства имен или объявлены как `static` внутри классов, функций или файлов. Эти объекты могут использоваться внутри лямбда-выражений, но не могут быть захвачены. Тем не менее спецификация режима захвата по умолчанию может создать именно такое впечатление. Рассмотрим преобразованную версию функции `addDivisorFilter`, с которой мы встречались ранее:

```
void addDivisorFilter()
{
    static auto calc1
        = computeSomeValue1();           // Статический
    static auto calc2
        = computeSomeValue2();           // Статический

    static auto divisor =           // Статический
        computeDivisor(calc1, calc2);

    filters.emplace_back(
        [=](int value)           // Ничего не захватывает
        { return value % divisor == 0; } // Ссылка на статическую
    );                           // переменную

    ++divisor;                   // Изменение divisor
}
```

Случайный читатель этого кода может быть прощен за то, что, видя [=], может подумать “Отлично, лямбда-выражение делает копию всех объектов, которые использует, и поэтому оно является самодостаточным”. Но это не так. Это лямбда-выражение не использует никакие нестатические локальные переменные, поэтому ничего не захватывает-ся. Вместо этого код лямбда-выражения обращается к статической переменной divisor. Когда в конце каждого вызова addDivisorFilter выполняется увеличение divisor, все лямбда-выражения, которые были добавлены в filters с помощью данной функции, будут демонстрировать новое поведение (соответствующее новому значению divisor). С практической точки зрения это лямбда-выражение захватывает divisor по ссылке, а это выглядит противоречащим объявленному захвату по умолчанию по значению. Если держаться подальше от захвата по умолчанию по значению, можно уменьшить риск невер-ного понимания такого кода.

### Следует запомнить

- Захват по умолчанию по ссылке может привести к висячим ссылкам.
- Захват по умолчанию по значению восприимчив к висячим указателям (особенно к this) и приводит к ошибочному предположению о самодостаточности лямбда-вы-ражений.

## 6.2. Используйте инициализирующий захват для перемещения объектов в замыкания

Иногда ни захват по значению, ни захват по ссылке не является тем, что вы хотите. Если у вас имеется объект, который можно только перемещать (например, std::unique\_ptr или std::future) и который вы хотите передать замыканию, C++11 не предлагает вам никакого способа для этого. Если у вас есть объект, который гораздо де-шевле переместить, чем копировать (например, большинство контейнеров стандартной библиотеки), и вы хотели бы передать его в замыкание, то гораздо эффективнее пере-местить его, чем копировать. И вновь C++11 не предоставляет вам способа сделать это.

Но только C++11. C++14 — совершенно другая история. Он предлагает непосред-ственную поддержку перемещения объектов в замыкания. Если ваш компилятор соот-ветствует стандарту C++14, радуйтесь и читайте дальше. Если же вы работаете с компи-ляторами C++11, вы тоже должны радоваться и читать дальше — потому что и в C++11 имеются способы приблизиться к перемещающему захвату.

Отсутствие перемещающего захвата было признано недостатком даже при принятии C++11. Казалось бы, простейшим путем было его добавление в C++14, но Комитет по стандартизации пошел иным путем. Он добавил новый механизм, который настоль-ко гибкий, что захват путем перемещения является всего лишь одним из вариантов его работы. Новая возможность называется инициализирующим захватом (init capture). Он может делать почти все, что могут делать захваты в C++11, и еще многое. Единственное, что нельзя выразить с помощью инициализирующего захвата (и от чего, как поясняется

в разделе 6.1, вам надо держаться подальше), — это режим захвата по умолчанию. (Для ситуаций, охватываемых захватами C++11, инициализирующий захват несколько многословнее, так что там, где справляется захват C++11, совершенно разумно использовать именно его.)

Применение инициализирующего захвата делает возможным указать

1. имя члена-данных в классе замыкания, сгенерированном из лямбда-выражения, и
2. выражение инициализации этого члена-данных.

Вот как можно использовать инициализирующий захват для перемещения std::unique\_ptr в замыкание:

```
class Widget {                                     // Некоторый полезный тип
public:
    ...
    bool isValidated() const;
    bool isProcessed() const;
    bool isArchived() const;
private:
};

auto pw = std::make_unique<Widget>();           // Создание Widget;
                                                // std::make_unique
                                                // см. в разделе 4.4
                                                // Настройка *pw
...
auto func = [pw = std::move(pw)]                // Инициализация члена
{ return pw->isValidated()                     // в замыкании с помощью
&& pw->isArchived(); }; // std::move(pw)
```

Выделенный текст представляет собой инициализирующий захват. Слева от знака равенства = находится имя члена-данных в классе замыкания, который вы определяете, а справа — инициализирующее выражение. Интересно, что область видимости слева от “=” отличается от области видимости справа. Область видимости слева — это область видимости класса замыкания. Область видимости справа — та же, что и определяемого лямбда-выражения. В приведенном выше примере имя pw слева от = ссылается на члены-данные в классе замыкания, в то время как имя pw справа ссылается на объект, объявленный выше лямбда-выражения, т.е. на переменную, инициализированную вызовом std::make\_unique. Так что “pw = std::move(pw)” означает “создать член-данные pw в замыкании и инициализировать этот член-данные результатом применения std::move к локальной переменной pw”.

Как обычно, код в теле лямбда-выражения находится в области видимости класса замыкания, так что использованные в нем pw относятся к члену-данным класса замыкания.

Комментарий “настройка \*pw” в этом примере указывает, что после создания Widget с помощью std::make\_unique и до того, как интеллектуальный указатель std::unique\_ptr на этот Widget будет захвачен лямбда-выражением, Widget

некоторым образом модифицируется. Если такая настройка не нужна, т.е. если объект Widget, созданный с помощью std::make\_unique, находится в состоянии, пригодном для захвата лямбда-выражением, локальная переменная pw не нужна, поскольку членовые класса замыкания может быть непосредственно инициализирован с помощью std::make\_unique:

```
auto func = [pw = std::make_unique<Widget>()] // Инициализация
    { return pw->isValidated() // члена-данных в замыкании
      && pw->isArchived(); }; // результатом вызова make_unique
```

Из этого должно быть ясно, что понятие захвата в C++14 значительно обобщено по сравнению с C++11, поскольку в C++11 невозможно захватить результат выражения. Поэтому еще одним назначением инициализирующего захвата является *обобщенный захват лямбда-выражения* (generalized lambda capture).

Но что если один или несколько используемых вами компиляторов не поддерживают инициализирующий захват C++14? Как выполнить перемещающий захват в языке, в котором нет поддержки перемещающего захвата?

Вспомните, что лямбда-выражение — это просто способ генерации класса и создания объекта этого типа. Нет ничего, что можно сделать с лямбда-выражением и чего нельзя было бы сделать вручную. Например, код на C++14, который мы только что рассмотрели, может быть записан на C++11 следующим образом:

```
class IsValAndArch {
public:
    using DataType = std::unique_ptr<Widget>;
    explicit IsValAndArch(DataType&&ptr) // Применение std::move
        : pw(std::move(ptr)) {} // описано в разделе 5.3
    bool operator()() const
        { return pw->isValidated() && pw->isArchived(); }
private:
    DataType pw;
};

auto func = IsValAndArch(std::make_unique<Widget>());
```

Это требует больше работы, чем написание лямбда-выражения, но это не меняет того факта, что если вам нужен класс C++11, поддерживающий перемещающую инициализацию своих членов-данных, то ваше желание отделено от вас лишь некоторым временем за клавиатурой.

Если вы хотите придерживаться лямбда-выражений (с учетом их удобства это, вероятно, так и есть), то перемещающий захват можно эмулировать в C++11 с помощью

1. **перемещения захватываемого объекта в функциональный объект с помощью std::bind и**
2. **передачи лямбда-выражению ссылки на захватываемый объект.**

Если вы знакомы с `std::bind`, код достаточно прост. Если нет, вам придется немного привыкнуть к нему, но игра стоит свеч.

Предположим, вы хотите создать локальный `std::vector`, разместить в нем соответствующее множество значений, а затем переместить его в замыкание. В C++14 это просто:

```
std::vector<double> data;           // Объект, перемещаемый
                                    // в замыкание
...
auto func = [data = std::move(data)] // Инициализирующий захват
    { /* Использование данных */ };
```

Я выделил ключевые части этого кода: тип объекта, который вы хотите перемещать (`std::vector<double>`), имя этого объекта (`data`) и выражение инициализации для инициализирующего захвата (`std::move(data)`). Далее следует эквивалент этого кода на C++11, в котором я выделил те же ключевые части:

```
std::vector<double> data;           // Как и ранее
...
auto func =
    std::bind(                      // Эмуляция в C++11
        [] (const std::vector<double>& data) // инициализирующего
            { /* Использование данных */ },      // захвата
        std::move(data)
    );
```

Подобно лямбда-выражениям, `std::bind` создает функциональные объекты. Я называю функциональные объекты, возвращаемые `std::bind`, *bind-объектами*. Первый аргумент `std::bind` — вызываемый объект. Последующие аргументы представляют передаваемые этому объекту значения.

Bind-объект содержит копии всех аргументов, переданных `std::bind`. Для каждого `lvalue`-аргумента соответствующий объект в bind-объекте создается копированием. Для каждого `rvalue` он создается перемещением. В данном примере второй аргумент представляет собой `rvalue` (как результат применения `std::move`; см. раздел 5.1), так что `data` перемещается в bind-объект. Это перемещающее создание является сутью эмуляции перемещающего захвата, поскольку перемещение `rvalue` в bind-объект есть обходной путь для перемещения `rvalue` в замыкание C++11.

Когда bind-объект “вызывается” (т.е. выполняется его оператор вызова функции), сохраненные им аргументы, первоначально переданные в `std::bind`, передаются в вызываемый объект. В данном примере это означает, что когда вызывается bind-объект `func`, лямбда-выражению, переданному в `std::bind`, в качестве аргумента передается созданная в `func` перемещением копия `data`.

Это лямбда-выражение то же самое, что и использованное нами в C++14, за исключением добавленного параметра `data`. Этот параметр представляет собой `lvalue`-ссылку на копию `data` в bind-объекте. (Это не `rvalue`-ссылка, поскольку, хотя выражение, использованное для инициализации копии `data` (“`std::move(data)`”), является `rvalue`, сама по себе копия `data` представляет собой `lvalue`.) Таким образом, применение `data` внутри

лямбда-выражения будет работать с копией `data` внутри `bind`-объекта, созданной перемещением.

По умолчанию функция-член `operator()` в классе замыкания, сгенерированном из лямбда-выражения, является `const`. Это приводит к тому, что все члены-данные в замыкании в теле лямбда-выражения являются константными. Однако созданная перемещением копия `data` внутри `bind`-объекта не является константной, так что, чтобы предотвратить модификацию этой копии `data` внутри лямбда-выражения, параметр лямбда-выражения объявляется как указатель на `const`. Если лямбда-выражение было объявлено как `mutable`, `operator()` в его классе замыкания не будет объявлен как `const`, так что целесообразно опустить `const` в объявлении параметра лямбда-выражения:

```
auto func =  
    std::bind(  
        [] (std::vector<double>& data) mutable // инициализирующего  
        { /* uses of data */ }, // захвата для лямбда-  
        std::move(data) // выражения, объяв-  
    ); // ленного mutable
```

Поскольку `bind`-объект хранит копии всех аргументов, переданных `std::bind`, `bind`-объект в нашем примере содержит копию замыкания, произведенного из лямбда-выражения, являющегося первым аргументом этого объекта. Следовательно, время жизни замыкания совпадает со временем жизни `bind`-объекта. Это важно, поскольку это означает, что, пока существует замыкание, существует и `bind`-объект, содержащий объект, захваченный псевдоперемещением.

Если вы впервые столкнулись с `std::bind`, вам может понадобиться учебник или справочник по C++11, чтобы все детали этого обсуждения встали на свои места в вашей голове. Вот основные моменты, которые должны быть понятными.

- Невозможно выполнить перемещение объекта в замыкание C++11, но можно выполнить перемещение объекта в `bind`-объект C++11.
- Эмуляция захвата перемещением в C++11 состоит в перемещении объекта в `bind`-объект с последующей передачей перемещенного объекта в лямбда-выражение по ссылке.
- Поскольку время жизни `bind`-объекта совпадает с таковым для замыкания, можно рассматривать объекты в `bind`-объекте так, как будто они находятся в замыкании.

В качестве второго примера применения `std::bind` для эмуляции перемещающего захват рассмотрим пример кода C++14, который мы видели ранее и который создает `std::unique_ptr` в замыкании:

```
auto func = [pw = std::make_unique<Widget>()] // Как и ранее,  
    { return pw->isValidated() // создает pw  
     && pw->isArchived(); }; // в замыкании
```

А вот как выглядит его эмуляция на C++11:

```
auto func = std::bind(
    [] (const std::unique_ptr<Widget>& pw)
{ return pw->isValidated()
    && pw->isArchived(); },
    std::make_unique<Widget>()
);
```

Забавно, что я показываю, как использовать `std::bind` для обхода ограничений лямбда-выражений в C++11, поскольку в разделе 6.4 я выступаю как сторонник применения лямбда-выражений вместо `std::bind`. Однако в данном разделе поясняется, что в C++11 имеются ситуации, когда может пригодиться `std::bind`, и это одна из них. (В C++14 такие возможности, как инициализирующий захват и параметры `auto`, устраниют такие ситуации.)

#### Следует запомнить

- Для перемещения объектов в замыкания используется инициализирующий захват C++14.
- В C++11 инициализирующий захват эмулируется с помощью написания классов вручную или применения `std::bind`.

### 6.3. Используйте параметры `decltype` для `auto&&` для передачи с помощью `std::forward`

Одной из самых интересных возможностей C++14 являются *обобщенные лямбда-выражения* — лямбда-выражения, в спецификации параметров которых используется ключевое слово `auto`. Реализация этой возможности проста: `operator()` в классе замыкания лямбда-выражения является шаблоном. Например, для лямбда-выражения

```
auto f = [] (auto x) { return normalize(x); };
```

оператор вызова функции класса замыкания имеет следующий вид:

```
class SomeCompilerGeneratedClassName {
public:
    template<typename T>           // См. возвращаемый тип auto
    auto operator()(T x) const // в разделе 1.3
    { return normalize(x); }           // Прочая функциональность
};                                // класса замыкания
```

В этом примере единственное, что делает лямбда-выражение с параметром `x`, — это передает его функции `normalize`. Если `normalize` рассматривает значения `lvalue` не так, как значения `rvalue`, это лямбда-выражение написано некорректно, поскольку оно всегда передает функции `normalize` `lvalue` (параметр `x`), даже если переданный в лямбда-выражение аргумент представляет собой `rvalue`.

Корректным способом написания лямбда-выражения является прямая передача `x` в `normalize`. Это требует внесения в код двух изменений. Во-первых, `x` должен быть универсальной ссылкой (см. раздел 5.2), а во-вторых, он должен передаваться в `normalize` с использованием `std::forward` (см. раздел 5.3). Концептуально это требует тривиальных изменений:

```
auto f = [] (auto&& x)
{
    return normalize(std::forward<???>(x));
};
```

Однако между концепцией и реализацией стоит вопрос о том, какой тип передавать в `std::forward`, т.е. вопрос определения того, что должно находиться там, где я написал “???”.

Обычно, применяя прямую передачу, вы находитесь в шаблонной функции, принимающей параметр типа `T`, так что вам надо просто написать `std::forward<T>`. В обобщенном лямбда-выражении такой параметр типа `T` вам недоступен. Имеется `T` в шаблонизированном операторе `operator()` в классе замыкания, сгенерированном лямбда-выражением, но сослаться на него из лямбда-выражения невозможно, так что никак не помогает.

В разделе 5.6. поясняется, что если `lvalue`-аргумент передается параметру, являющемуся универсальной ссылкой, то типом этого параметра становится `lvalue`-ссылка. Если же передается `rvalue`, параметр становится `rvalue`-ссылкой. Это означает, что вне лямбда-выражения мы можем определить, является ли переданный аргумент `lvalue` или `rvalue`, рассматривая тип параметра `x`. Ключевое слово `decltype` дает нам возможность сделать это (см. раздел 1.3). Если было передано `lvalue`, `decltype(x)` даст тип, являющийся `lvalue`-ссылкой. Если же было передано `rvalue`, `decltype(x)` даст тип, являющийся `rvalue`-ссылкой.

В разделе 5.6 поясняется также, что при вызове `std::forward` соглашения требуют, чтобы для указания `lvalue` аргументом типа была `lvalue`-ссылка, а для указания `rvalue` — тип, не являющийся ссылкой. В нашем лямбда-выражении, если `x` привязан к `lvalue`, `decltype(x)` даст `lvalue`-ссылку. Это отвечает соглашению. Однако, если `x` привязан к `rvalue`, `decltype(x)` вместо типа, не являющегося ссылкой, даст `rvalue`-ссылку. Но взглянем на пример реализации `std::forward` в C++14 из раздела 5.6:

```
template<typename T> // В пространстве
T& forward(remove_reference_t<T>& param) // имен std
{
    return static_cast<T&>(param);
}
```

Если клиентский код хочет выполнить прямую передачу `rvalue` типа `Widget`, он обычно инстанцирует `std::forward` типом `Widget` (т.е. типом, не являющимся ссылочным), и шаблон `std::forward` дает следующую функцию:

```
Widget&& forward(Widget& param) // Инстанцирование для
{ // std::forward, когда
```

```
    return static_cast<Widget&&>(param); // T является Widget
}
```

Но рассмотрим, что произойдет, если код клиента намерен выполнить прямую передачу того же самого rvalue типа `Widget`, но вместо следования соглашению об определении `T` как не ссылочного типа определит его как `rvalue-ссылку`, т.е. рассмотрим, что случится, если `T` определен как `Widget&&`. После начального инстанцирования `std::forward` и применения `std::remove_reference_t`, но до свертывания ссылок (еще раз обратитесь к разделу 5.6) `std::forward` будет выглядеть следующим образом:

```
Widget&& forward(Widget& param)           // Инстанцирование
{
    return static_cast<Widget&& &>(param); // T, равном Widget&&
}                                              // (до сворачивания ссылок)
```

Применение правила сворачивания ссылок о том, что `rvalue-ссылка` на `rvalue-ссылку` становится одинарной `rvalue-ссылкой`, приводит к следующему инстанцированию:

```
Widget&& forward(Widget& param)           // Инстанцирование
{
    return static_cast<Widget&&>(param); // T, равном Widget&&
}                                              // (после сворачивания ссылок)
```

Если вы сравните это инстанцирование с инстанцированием, получающимся в результате вызова `std::forward` с `T`, равным `Widget`, то вы увидите, что они идентичны. Это означает, что инстанцирование `std::forward` типом, представляющим собой `rvalue-ссылку`, дает тот же результат, что и инстанцирование типом, не являющимся ссылочным.

Это чудесная новость, так как `decltype(x)` дает `rvalue-ссылку`, когда в качестве аргумента для параметра `x` нашего лямбда-выражения передается `rvalue`. Выше мы установили, что, когда в наше лямбда-выражение передается `lvalue`, `decltype(x)` дает соответствующий соглашению тип для передачи в `std::forward`, и теперь мы понимаем, что для `rvalue` `decltype(x)` дает тип для передачи `std::forward`, который не соответствует соглашению, но тем не менее приводит к тому же результату, что и тип, соответствующий соглашению. Так что как для `lvalue`, так и для `rvalue` передача `decltype(x)` в `std::forward` дает нам желаемый результат. Следовательно, наше лямбда-выражение с прямой передачей может быть записано следующим образом:

```
auto f = [](auto&& x)
{
    return normalize(std::forward<decltype(x)>(x));
};
```

Чтобы это лямбда-выражение принимало любое количество параметров, нам, по сути, надо только шесть точек, поскольку лямбда-выражения в C++14 могут быть с переменным числом аргументов:

```
auto f = [](auto&&... xs)
{
    return normalize(std::forward<decltype(xs)>(xs)...);
};
```

### Следует запомнить

- Используйте для параметров `auto&&` при их прямой передаче с помощью `std::forward` ключевое слово `decltype`.

## 6.4. Предпочитайте лямбда-выражения применению `std::bind`

`std::bind` в C++11 является преемником `std::bind1st` и `std::bind2nd` из C++98, но неформально этот шаблон является частью стандартной библиотеки еще с 2005 года. Именно тогда Комитет по стандартизации принял документ, известный как TR1, который включал спецификацию `bind`. (В TR1 `bind` находился в отдельном пространстве имен, так что обращаться к нему надо было как `std::tr1::bind`, а не `std::bind`, а кроме того, некоторые детали его интерфейса были иными.) Эта история означает, что некоторые программисты имеют за плечами десятилетний опыт использования `std::bind`. Если вы один из них, вы можете не быть склонными отказываться от столь долго верой и правдой служившего вам инструмента. Это можно понять, и все же в данном случае лучше его сменить, поскольку лямбда-выражения C++11 почти всегда являются лучшим выбором, чем `std::bind`. Что касается C++14, то здесь лямбда-выражения являются настоящим кладом.

В этом разделе предполагается, что вы хорошо знакомы с `std::bind`. Если это не так, вы, вероятно, захотите получить базовые знания о нем, прежде чем продолжить чтение. Что ж, это похвально, тем более что никогда не знаешь, не встретишься ли с применением `std::bind` в коде, который придется читать или поддерживать.

Как и в разделе 6.2, я называю функциональные объекты, возвращаемые `std::bind`, *bind-объектами*.

Наиболее важная причина, по которой следует предпочитать лямбда-выражения, заключается в их большей удобочитаемости. Например, предположим, что у нас есть функция для настройки будильника:

```
// Тип для момента времени (см. синтаксис в разделе 3.3)
using Time = std::chrono::steady_clock::time_point;

// См. "enum class" в разделе 3.4
enum class Sound { Beep, Siren, Whistle };

// Тип для продолжительности промежутка времени
using Duration = std::chrono::steady_clock::duration;

// В момент t издать звук s продолжительностью d
void setAlarm(Time t, Sound s, Duration d);
```

Далее предположим, что в некоторой точке программы мы определили, что хотим, чтобы будильник был отключен в течение часа, после чего подал сигнал продолжительностью

30 с. Сам звук остается неопределенным. Мы можем написать лямбда-выражение, которое изменяет интерфейс `setAlarm` так, что необходимо указать только звук:

```
// setSoundL ("L" означает "лямбда-выражение") — функциональный
// объект, позволяющий указать сигнал будильника, который должен
// звучать через час в течение 30 с
auto setSoundL =
    [] (Sound s)
{
    // Делает доступными компоненты std::chrono
    using namespace std::chrono;

    setAlarm(steady_clock::now() + hours(1), // Будильник через
        s,                                // час, звучит
        seconds(30));                      // 30 секунд
};
```

Я выделил вызов `setAlarm` в лямбда-выражении. Он выглядит, как обычный вызов функции, и даже читатель с малым опытом в лямбда-выражениях может понять, что переданный лямбда-выражению параметр `s` передается в качестве аргумента функции `setAlarm`.

В C++14 этот код можно упростить, используя стандартные суффиксы для секунд (`s`, `миллисекунд` (`ms`), часов (`h`) и других единиц, основанных на поддержке пользовательских литералов в C++14. Эти суффиксы определены в пространстве имен `std::literals`, так что приведенный выше код переписывается как

```
auto setSoundL =
    [] (Sound s)
{
    using namespace std::chrono;
    using namespace std::literals;          // Суффиксы C++14

    setAlarm(steady_clock::now() + 1h, // C++14, смысл
        s,                            // тот же, что
        30s);                        // и выше
};
```

Наша первая попытка написать соответствующий вызов `std::bind` приведена ниже. Она содержит ошибки, которые мы вскоре исправим, но главное — что правильный код более сложен, и даже эта упрощенная версия приводит к ряду важных вопросов:

```
using namespace std::chrono;           // Как и ранее
using namespace std::literals;
using namespace std::placeholders;       // Необходимо для "_1"
auto setSoundB =                         // "B" означает "bind"
    std::bind(setAlarm,
        steady_clock::now() + 1h, // Ошибка! См. ниже
        _1,
        30s);
```

Я хотел бы выделить вызовы `setAlarm`, как делал это в лямбда-выражении, но здесь нет вызова, который можно было бы выделить. Читатели этого кода просто должны знать, что вызов `setSoundB` приводит к вызову `setAlarm` со временем и продолжительностью, указанными в вызове `std::bind`. Для непосвященных заполнитель `_1` выглядит магически, но даже знающие читатели должны в уме отобразить число в заполнителе на позицию в списке параметров `std::bind`, чтобы понять, что первый аргумент вызова `setSoundB` передается в `setAlarm` в качестве второго аргумента. Тип этого аргумента в вызове `std::bind` не определен, так что читатели должны проконсультироваться с объявлением `setAlarm`, чтобы выяснить, какой аргумент передается в `setSoundB`.

Но, как я уже говорил, этот код не совсем верен. В лямбда-выражении очевидно, что выражение `"steady_clock::now() + 1h"` представляет собой аргумент `setAlarm`. Оно будет вычислено при вызове `setAlarm`. Это имеет смысл: мы хотим, чтобы будильник заработал через час после вызова `setAlarm`. Однако в вызове `std::bind` выражение `"steady_clock::now() + 1h"` передается в качестве аргумента в `std::bind`, а не в `setAlarm`. Это означает, что выражение будет вычислено при вызове `std::bind` и полученное время будет храниться в сгенерированном `bind`-объекте. В результате будильник сработает через час *после вызова* `std::bind`, а не через час *после вызова* `setAlarm`!

Решение данной проблемы требует указания для `std::bind` отложить вычисление выражения до вызова `setAlarm`, и сделать это можно с помощью вложения еще двух вызовов `std::bind` в первый:

```
auto setSoundB =
    std::bind(setAlarm,
              std::bind(std::plus<>(),
                        std::bind(steady_clock::now),
                        1h),
              _1,
              30s);
```

Если вы знакомы с шаблоном `std::plus` из C++98, вас может удивить то, что между угловыми скобками не указан тип, т.е. что код содержит `"std::plus<>"`, а не `"std::plus<type>"`. В C++14 аргумент типа шаблона для шаблонов стандартных операторов в общем случае может быть опущен, так что у нас нет необходимости указывать его здесь. C++11 такой возможности не предоставляет, так что в C++11 эквивалентный лямбда-выражению `std::bind` имеет следующий вид:

```
using namespace std::chrono;           // Как и ранее
using namespace std::placeholders;

auto setSoundB =
    std::bind(setAlarm,
              std::bind(std::plus<steady_clock::time_point>(),
                        std::bind(steady_clock::now),
                        hours(1)),
```

```
_1,  
seconds(30));
```

Если в этот момент лямбда-выражение не выглядит для вас гораздо более привлекательным, вам, пожалуй, надо сходить к окулисту.

При перегрузке `setAlarm` возникают новые вопросы. Предположим, что перегрузка получает четвертый параметр, определяющий громкость звука:

```
enum class Volume { Normal, Loud, LoudPlusPlus };  
  
void setAlarm(Time t, Sound s, Duration d, Volume v);
```

Лямбда-выражение продолжает работать, как и ранее, поскольку разрешение перегрузки выбирает трехаргументную версию `setAlarm`:

```
auto setSoundL = // Как и ранее  
    [](Sound s)  
    {  
        using namespace std::chrono;  
        setAlarm(steady_clock::now() + 1h, // OK, вызывает  
                 s,                      // setAlarm с тремя  
                 30s);                  // аргументами  
    };
```

Вызов же `std::bind` теперь не компилируется:

```
auto setSoundB =           // Ошибка! Какая из  
    std::bind(setAlarm,      // функций setAlarm?  
              std::bind(std::plus<>(),  
                        std::bind(steady_clock::now),  
                        1h),  
              _1,  
              30s);
```

Проблема в том, что у компиляторов нет способа определить, какая из двух функций `setAlarm` должна быть передана в `std::bind`. Все, что у них есть, — это имя функции, а одно имя приводит к неоднозначности.

Чтобы вызов `std::bind` компилировался, `setAlarm` должна быть приведена к корректному типу указателя на функцию:

```
using SetAlarm3ParamType = void(*)(Time t, Sound s, Duration d);  
  
auto setSoundB =           // Теперь все в порядке  
    std::bind(static_cast<SetAlarm3ParamType>(setAlarm),  
              std::bind(std::plus<>(),  
                        std::bind(steady_clock::now),  
                        1h),  
              _1,  
              30s);
```

Но это привносит еще одно различие между лямбда-выражениями и `std::bind`. Внутри оператора вызова функции для `setSoundL` (т.е. оператора вызова функции класса заимствования лямбда-выражения) вызов `setAlarm` представляет собой обычный вызов функции, который может быть встроен компиляторами:

```
setSoundL(Sound::Siren); // Тело setAlarm может быть встроено
```

Однако вызов `std::bind` получает указатель на функцию `setAlarm`, а это означает, что внутри оператора вызова функции `setSoundB` (т.е. оператора вызова функции `bind`-объекта) имеет место вызов `setAlarm` с помощью указателя на функцию. Компиляторы менее склонны к встраиванию вызовов функций, выполняемых через указатели, а это означает, что вызовы `setAlarm` посредством `setSoundB` будут встроены с куда меньшей вероятностью, чем вызовы посредством `setSoundL`:

```
setSoundB(Sound::Siren); // Тело setAlarm вряд ли будет встроено
```

Таким образом, вполне возможно, что применение лямбда-выражений приводит к генерации более быстрого кода, чем применение `std::bind`.

Пример `setAlarm` включает только простой вызов функции. Если вы хотите сделать что-то более сложное, то перевес в пользу лямбда-выражений только увеличится. Рассмотрим, например, такое лямбда-выражение C++14, которое выясняет, находится ли аргумент между минимальным (`lowVal`) и максимальным (`highVal`) значениями, где `lowVal` и `highVal` являются локальными переменными.

```
auto betweenL =
[lowVal, highVal]
(const auto& val) // C++14
{ return lowVal <= val && val <= highVal; };
```

`std::bind` может выразить то же самое, но эта конструкция является примером кода, безопасного только потому, что никто не в состоянии его понять:

```
using namespace std::placeholders; // Как и ранее
auto betweenB =
std::bind(std::logical_and<>(),
          std::bind(std::less_equal<>(), lowVal, _1),
          std::bind(std::less_equal<>(), _1, highVal));
```

В C++11 мы должны указывать сравниваемые типы, и вызов `std::bind` принимает следующий вид:

```
auto betweenB = // Версия C++11
std::bind(std::logical_and<bool>(),
          std::bind(std::less_equal<int>(), lowVal, _1),
          std::bind(std::less_equal<int>(), _1, highVal));
```

Конечно, в C++11 лямбда-выражение не может получать параметр `auto`, так что здесь тоже надо указывать тип:

```
auto betweenL = // Версия C++11
    [lowVal, highVal]
    (int val)
    { return lowVal <= val && val <= highVal; };
```

В любом случае, я надеюсь, все согласятся, что лямбда-версия не только более короткая, но и более понятная и легче поддерживаемая.

Ранее я отмечал, что для тех, у кого нет опыта работы с std::bind, заполнители (например, \_1, \_2 и др.) выглядят магией. Но непонятно не только поведение заполнителей. Предположим, у нас есть функция для создания сжатых копий Widget

```
enum class CompLevel { Low, Normal, High }; // Уровень сжатия

Widget compress(const Widget& w,           // Создание сжатой
                CompLevel lev);        // копии w
```

и мы хотим создать функциональный объект, который позволяет нам указывать, насколько сильно должен быть сжат конкретный объект Widget w. Представленное ниже применение std::bind создает такой объект:

```
Widget w;

using namespace std::placeholders;

auto compressRateB = std::bind(compress, w, _1);
```

Теперь, когда мы передаем w в std::bind, он должен храниться для последующего вызова compress. Он сохраняется в объекте compressRateB, но как именно — по значению или по ссылке? Это важно, потому что, если w изменяется между вызовами std::bind и compressRateB, хранение w по ссылке будет отражать это изменение, в то время как сохранение по значению — нет.

Ответ заключается в том, что оно хранится по значению<sup>1</sup>, но единственный способ узнать это — просто запомнить: в вызове std::bind нет никаких признаков этого. Сравните этот подход с лямбда-выражением, в котором захват w по значению или по ссылке выполняется явно:

```
auto compressRateL = // Захват w по значению;
    [w](CompLevel lev) // lev передается по значению
    { return compress(w, lev); };
```

Не менее явно в лямбда-выражении передаются и параметры. Здесь очевидно, что параметр lev передается по значению. Таким образом,

```
compressRateL(CompLevel::High); // arg передается по значению
```

<sup>1</sup> std::bind всегда копирует свои аргументы, но вызывающий код может лобиться эффекта сохранения аргумента по ссылке путем применения std::ref. Результат вызова

```
auto compressRateB = std::bind(compress, std::ref(w), _1);
```

состоит в том, что compressRateB действует так, как если бы сохранялась ссылка на объект w, а не его копия.

Но как аргумент передается в объект, получающийся с помощью `std::bind`?

```
compressRateB(CompLevel::High); // Как передается arg?
```

И вновь единственный способ знать, как работает `std::bind`, — это запомнить. (Ответ заключается в том, что все аргументы, передаваемые `bind`-объектам, передаются по ссылке, поскольку оператор вызова функции для таких объектов использует прямую передачу.)

По сравнению с лямбда-выражениями код, использующий `std::bind`, менее удобочитаем, менее выразителен и, возможно, менее эффективен. В C++14 нет обоснованных случаев применения `std::bind`. Однако в C++11 применение `std::bind` может быть оправдано в двух ограниченных ситуациях.

- **Захват перемещением.** Лямбда-выражения C++11 не предоставляют возможности захвата перемещением, но его можно эмулировать с помощью комбинации лямбда-выражения и `std::bind`. Детали описываются в разделе 6.2, в котором также поясняется, что в C++14 поддержка инициализирующего захвата в лямбда-выражениях устраняет необходимость в такой эмуляции.
- **Полиморфные функциональные объекты.** Поскольку оператор вызова функции `bind`-объекта использует прямую передачу, он может принимать аргументы любого типа (с учетом ограничений на прямую передачу, описанных в разделе 5.8). Это может быть полезным, когда вы хотите связать объект с шаблонным оператором вызова функции. Например, для класса

```
class PolyWidget {  
public:  
    template<typename T>  
    void operator()(const T& param) const;
```

```
};
```

`std::bind` может связать `PolyWidget` следующим образом:

```
PolyWidget pw;
```

```
auto boundPW = std::bind(pw, _1);
```

После этого `boundPW` может быть вызван с разными типами аргументов:

```
boundPW(1930);      // Передача int в PolyWidget::operator()  
boundPW(nullptr);  // Передача nullptr в PolyWidget::operator()  
boundPW("Rosebud"); // Передача строкового литерала
```

Выполнить это с помощью лямбда-выражений C++11 невозможно. Однако в C++14 этого легко достичь с помощью лямбда-выражения с параметром `auto`:

```
auto boundPW = [pw] (const auto& param) // C++14
{ pw(param); };
```

Конечно, это крайние случаи, и они встречаются все реже, поскольку поддержка C++14 лямбда-выражений компиляторами становится все более распространенной.

Когда `bind` был неофициально добавлен в C++ в 2005 году, это было существенным усовершенствованием по сравнению с его предшественником 1998 года. Однако добавление поддержки лямбда-выражений в C++11 привело к устареванию `std::bind`, а с момента появления C++14 для него, по сути, не осталось применений.

### Следует запомнить

- Лямбда-выражения более удобочитаемы, более выразительны и могут быть более эффективными по сравнению с `std::bind`.
- Только в C++11 `std::bind` может пригодиться для реализации перемещающего захвата или для связывания объектов с шаблонными операторами вызова функции.

# Параллельные вычисления

Одним из наибольших триумфов C++11 является включение параллелизма в язык программирования и библиотеку. Программисты, знакомые с другими потоковыми API (например, pthreads или Windows threads), иногда удивляются сравнительно спартанскому набору возможностей, предлагаемому C++, но это связано с тем, что по большей части поддержка параллельности в C++ имеет вид ограничений для разработчиков компиляторов. Получаемые в результате языковые гарантии означают, что впервые в истории C++ программисты могут писать многопоточные приложения со стандартным поведением на всех платформах. Это создает прочный фундамент, на котором могут быть построены выразительные библиотеки, а элементы параллелизма в стандартной библиотеке (задачи, фьючерсы, потоки, мьютексы, переменные условий, атомарные объекты и прочие) являются лишь началом того, что обязательно станет более богатым набором инструментов для разработки параллельного программного обеспечения на C++.

В последующих разделах нужно иметь в виду, что стандартная библиотека содержит два шаблона для фьючерсов: `std::future` и `std::shared_future`. Во многих случаях различия между ними не важны, так что я часто говорю просто о *фьючерсах*, подразумевая при этом обе разновидности.

## 7.1. Предпочитайте программирование на основе задач программированию на основе потоков

Если вы хотите выполнить функцию `doAsyncWork` асинхронно, у вас есть два основных варианта. Вы можете создать `std::thread` и запустить с его помощью `doAsyncWork`, тем самым прибегнув к подходу *на основе потоков*:

```
int doAsyncWork();  
std::thread t(doAsyncWork);
```

Вы также можете передать `doAsyncWork` в `std::async`, воспользовавшись стратегией, известной как подход *на основе задач*:

```
auto fut = std::async(doAsyncWork); // "fut" от "future"
```

В таких вызовах функциональный объект, переданный в `std::async` (например, `doAsyncWork`), рассматривается как *задача* (*task*).

Подход на основе задач обычно превосходит свой аналог на основе потоков, и небольшие фрагменты кода, с которыми вы встретились выше, показывают некоторые причины этого. Здесь `doAsyncWork` дает возвращаемое значение, в котором, как мы можем разумно предположить, заинтересован вызывающий `doAsyncWork` код. В случае вызова на основе потоков нет простого способа к нему обратиться. При подходе на основе потоков нет простого способа получить доступ к вызову. В случае же подхода на основе задач это можно легко сделать, поскольку фьючерс, возвращаемый `std::async`, предлагает функцию `get`. Эта функция `get` еще более важна, если `doAsyncWork` генерирует исключение, поскольку `get` обеспечивает доступ и к нему. При подходе на основе потоков в случае генерации функцией `doAsyncWork` исключения программа аварийно завершается (с помощью вызова `std::terminate`).

Более фундаментальным различием между подходами на основе потоков и на основе задач является воплощение более высокого уровня абстракции в последнем. Он освобождает вас от деталей управления потоками, что, кстати, напомнило мне о необходимости рассказать о трех значениях слова *поток* (*thread*) в параллельном программировании на C++.

- *Аппаратные потоки* являются потоками, которые выполняют фактические вычисления. Современные машинные архитектуры предлагают по одному или по нескольким аппаратным потокам для каждого ядра процессора.
- *Программные потоки* (известные также как *потоки ОС* или *системные потоки*) являются потоками, управляемыми операционной системой<sup>1</sup> во всех процессах и планируемыми для выполнения аппаратными потоками. Обычно можно создать программных потоков больше, чем аппаратных, поскольку, когда программный поток заблокирован (например, при вводе-выводе или ожидании мьютекса или переменной условия), пропускная способность может быть повышена путем выполнения других, незаблокированных потоков.
- `std::thread` представляют собой объекты в процессе C++, которые действуют как дескрипторы для лежащих в их основе программных потоков. Некоторые объекты `std::thread` представляют "нулевые" дескрипторы, т.е. не соответствуют программным потокам, поскольку находятся в состоянии, сконструированном по умолчанию (следовательно, без выполняемой функции); потоки, из которых выполнено перемещение (после перемещения объект `std::thread`, в который оно произошло, действует как дескриптор для соответствующего программного потока); потоки, у которых выполняемая ими функция завершена; а также потоки, у которых разорвана связь между ними и обслуживающими их программными потоками.

Программные потоки являются ограниченным ресурсом. Если вы попытаетесь создать их больше, чем может предоставить система, будет сгенерировано исключение `std::system_error`. Это так, даже если функция, которую вы хотите запустить, не генерирует исключений. Например, даже если `doAsyncWork` объявлена как `noexcept`,

```
int doAsyncWork() noexcept; // См. noexcept в разделе 3.8
```

<sup>1</sup> В предположении, что таковая имеется. В некоторых встроенных системах ее нет.

следующая инструкция может сгенерировать исключение:

```
std::thread t(doAsyncWork); // Генерация исключения, если  
                           // больше нет доступных потоков
```

Хорошо написанное программное обеспечение должно каким-то образом обрабатывать такую возможность, но как? Один вариант — запустить `doAsyncWork` в текущем потоке, но это может привести к несбалансированной нагрузке и, если текущий поток является потоком GUI, к повышенному времени реакции системы на действия оператора. Другой вариант — ожидание завершения некоторых существующих программных потоков с последующей попыткой создания нового объекта `std::thread`, но может быть и так, что существующие потоки ожидают действий, которые должна выполнить функция `doAsyncWork` (например, ее результата или уведомления переменной условия).

Даже если вы не исчерпали потоки, у вас могут быть проблемы с *превышением подписки* (*oversubscription*). Это происходит, когда имеется больше готовых к запуску (т.е. незаблокированных) программных потоков, чем аппаратных. Когда это случается, планировщик потоков (который обычно представляет собой часть операционной системы) выполняет разделение времени для выполнения программных потоков аппаратными. Когда время работы одного потока завершается и начинается время работы второго, выполняется переключение контекстов. Такие переключения контекстов увеличивают накладные расходы по управлению потоками и могут оказаться в особенности дорогостоящими, когда аппаратный поток, назначаемый программному, оказывается выполняемым другим ядром, не тем, что ранее. В этом случае (1) кэши процессора обычно оказываются с данными, не имеющими отношения к данному программному потоку, и (2) запуск “нового” программного потока на этом ядре “загрязняет” кэши процессора, заполняя их данными, не имеющими отношения к “старым” потокам, которые выполнялись этим ядром и, вероятно, будут выполняться им снова.

Избежать превышения подписки сложно, поскольку оптимальное отношение программных и аппаратных потоков зависит от того, как часто запускаются программные потоки, и может изменяться динамически, например, при переходе программы от работы по вводу-выводу к вычислениям. Наилучшее отношение программных и аппаратных потоков зависит также от стоимости переключения контекстов и от того, насколько эффективно программные потоки используют кэши процессора. Кроме того, количество аппаратных потоков и подробная информация о кешах процессора (например, насколько они велики и какова их относительная скорость) зависят от архитектуры компьютера, так что, даже если вы настроите свое приложение так, чтобы избежать превышения подписки (сохраняя занятость аппаратного обеспечения) на одной платформе, нет никакой гарантии, что ваше решение будет хорошо работать и на других видах машин.

Ваша жизнь станет легче, если вы переложите свои проблемы на чужие плечи, и это плечо вам готов подставить `std::async`:

```
auto fut = std::async(doAsyncWork); // Управление потоками лежит  
                                   // на реализации стандартной  
                                   // библиотеки
```

Этот вызов переносит ответственность за управление потоками на реализацию стандартной библиотеки C++. Например, вероятность получения исключения нехватки потоков значительно снижается, поскольку этот вызов, вероятно, никогда его не сгенерирует. “Как такое может быть? — удивитесь вы. — Если я запрошу больше потоков, чем может предоставить система, то какая разница, как это будет сделано — через создание std::thread или вызовом std::async?” Это имеет значение, поскольку std::async, будучи вызвана в данном виде (т.е. со стратегией запуска по умолчанию; см. раздел 7.2), не гарантирует создания нового программного потока. Она скорее разрешает планировщику организовать для указанной функции (в нашем примере — doAsyncWork) возможность запуска потоком, запрашивающим результат doAsyncWork (например, в потоке, вызывающем get или wait для объекта fut), и интеллектуальные планировщики воспользуются предоставленной свободой, если в системе превышена подпись или не хватает потоков.

Если вы попытаетесь проделать этот “запуск в потоке, запрашивающем результат” самостоятельно, то я замечу, что это может привести к вопросам о дисбалансе нагрузки, и эти вопросы не исчезнут просто потому, что std::async и планировщик времени выполнения возьмутся за дело вместо вас. Тем не менее, когда дело доходит до балансировки нагрузки, планировщик времени выполнения, скорее всего, будет иметь более полную картину происходящего на машине, чем вы, потому что он управляет потоками всех процессов, а не только вашего.

При применении std::async время отклика потока GUI может остаться проблематичным, поскольку планировщику неизвестно, какой из ваших потоков имеет повышенные требования к этому параметру. В таком случае вы можете захотеть передать в std::async стратегию запуска std::launch::async. Это гарантирует, что интересующая вас функция будет действительно выполнена другим потоком (см. раздел 7.2).

Современные планировщики потоков во избежание превышения подписки используют пулы потоков всей системы и повышают балансировку загрузки между аппаратными ядрами с помощью специальных алгоритмов. Стандарт C++ не требует применения пулов потоков или конкретных алгоритмов, и, честно говоря, есть некоторые технические аспекты спецификации параллельности в C++11, которые делают его применение более трудным, чем хотелось бы. Тем не менее некоторые производители используют преимущества указанных методов в своих реализациях стандартной библиотеки, и следует ожидать продолжения прогресса в данной области. Если вы примете для параллельного программирования подход на основе задач, вы будете автоматически пользоваться ее преимуществами, которые будут возрастать с ее распространенностью. С другой стороны, программируя непосредственно с помощью std::thread, вы берете на себя бремя борьбы с исчерпанием потоков, превышением подписки и балансировкой загрузки (не упомянутая уже о том, насколько ваши решения этих проблем будут совместимы с решениями в программах, работающих в других процессах на том же компьютере).

По сравнению с программированием на основе потоков программирование на основе задач спасает вас от управления потоками вручную и обеспечивает естественный способ получения результатов асинхронно выполненных функций (т.е. возвращаемых значений

или исключений). Тем не менее существуют ситуации, в которых применение потоков может быть более подходящим. Они включают в себя следующее.

- **Вам нужен доступ к API, лежащей в основе реализации потоков.** В C++ API параллельных вычислений обычно реализуется с помощью низкоуровневого платформо-зависимого API, обычно pthreads или Windows Threads. Эти API в настоящее время предлагают больше возможностей, чем C++. (Например, C++ не имеет понятий приоритетов или родственности потоков.) Для предоставления доступа к API реализации потоков `std::thread` обычно предлагает функцию-член `native_handle`. Такая функциональность отсутствует в `std::future` (т.е. в том, что возвращает `std::async`).
- **Вам требуется возможность оптимизации потоков в вашем приложении.** Это может произойти, например, если вы разрабатываете серверное программное обеспечение с известным профилем выполнения, которое может быть развернуто как единственный процесс на машине с фиксированными аппаратными характеристиками.
- **Вам требуется реализовать поточную технологию, выходящую за рамки API параллельных вычислений в C++,** например пулы потоков на платформах, на которых ваши реализации C++ их не предоставляют.

Однако это нестандартные ситуации. В большинстве случаев вы должны выбирать программирование на основе задач, а не на основе потоков.

#### Следует запомнить

- API `std::thread` не предлагает способа непосредственного получения возвращаемых значений из асинхронно выполняемых функций, и, если такие функции генерируют исключения, программа завершается.
- Программирование на основе потоков требует управления вручную исчерпанием потоков, превышением подписки, балансом загрузки и адаптацией к новым платформам.
- Программирование на основе задач с помощью `std::async` со стратегией запуска по умолчанию решает большинство перечисленных проблем вместо вас.

## 7.2. Если важна асинхронность, указывайте `std::launch::async`

Вызывая `std::async` для выполнения функции (или иного выполнимого объекта), вы в общем случае планируете выполнять ее асинхронно. Но вы не обязательно требуете это от `std::async`. В действительности вы запрашиваете выполнение функции в соответствии со *стратегией запуска* `std::async`. Имеются две стандартные стратегии, представленные перечислителями в перечислении с областью видимости `std::launch`. (Читайте информацию о перечислениях с областью видимости в разделе 3.4.) В предположении, что для выполнения в `std::async` передается функция `f`:

- **стратегия** `std::launch::async` означает, что `f` должна выполняться асинхронно, т.е. в другом потоке;
- **стратегия** `std::launch::deferred` означает, что `f` может выполняться только тогда, когда для фьючерса, возвращенного `std::async`, вызывается функция-член `get` или `wait`<sup>2</sup>, т.е. выполнение `f` откладывается до тех пор, пока не будет выполнен такой вызов. Когда вызываются функции-члены `get` или `wait`, функция `f` выполняется синхронно, т.е. вызывающая функция блокируется до тех пор, пока `f` не завершит работу. Если не вызывается ни `get`, ни `wait`, `f` не выполняется.

Возможно, это окажется удивительным, но стратегия запуска `std::async` по умолчанию — используемая в случае, если вы не указали таковую — не является ни одной из перечисленных. На самом деле это стратегия, которая представляет собой сочетание описанных с помощью оператора “или”. Два приведенных далее вызова имеют один и тот же смысл:

```
auto fut1 = std::async(f);           // Выполнение f со стратегией
                                    // запуска по умолчанию
auto fut2 = std::async(            // Выполнение f
    std::launch::async |          // асинхронное
    std::launch::deferred,        // или отложенное
    f);
```

Таким образом, стратегия по умолчанию позволяет `f` быть выполненной как асинхронно, так и синхронно. Как указано в разделе 7.1, эта гибкость позволяет `std::async` и компонентам управления потоками стандартной библиотеки брать на себя ответственность за создание и уничтожение потоков, предупреждение превышения подписки и баланс загрузки. Все это делает параллельное программирование с помощью `std::async` таким удобным.

Но применение `std::async` со стратегией запуска по умолчанию имеет некоторые интересные последствия. Для потока `t`, выполняющего приведенную ниже инструкцию, справедливы следующие утверждения.

```
auto fut = std::async(f); // Выполнение f со стратегией
                        // запуска по умолчанию
```

- **Невозможно предсказать, будет ли f выполнятся параллельно с t**, поскольку выполнение `f` может быть отложено планировщиком.
- **Невозможно предсказать, будет ли f выполнятся потоком, отличным от того, в котором вызываются функции-члены get или wait объекта fut**. Если этот

<sup>2</sup> Это упрощение. Значение имет не фьючерс, для которого вызывается `get` или `wait`, а совместно используемое состояние, на которое ссылается фьючерс. (В разделе 7.4 обсуждается взаимосвязь фьючерсов и совместно используемых состояний.) Поскольку `std::future` поддерживают перемещение, а также могут использоваться для построения `std::shared_future` и поскольку `std::shared_future` могут копироваться, объект фьючерса, ссылающийся на совместно используемое состояние, возникающее из вызова `std::async`, в который была передана `f`, вероятно, будет отличаться от фьючерса, возвращенного `std::async`. Однако обычно просто говорят о вызове функций-членов `get` или `wait` фьючерса, возвращенного из `std::async`.

поток —  $t$ , отсюда вытекает невозможность предсказать, будет ли  $f$  выполнятся потоком, отличным от  $t$ .

- **Может быть невозможно предсказать, будет ли  $f$  выполнена вообще**, поскольку может оказаться невозможно гарантировать, что функции-члены `get` или `wait` объекта `fut` будут вызваны на всех путях выполнения программы.

Гибкость планирования стратегии запуска по умолчанию часто плохо комбинируется с использованием переменных `thread_local`, поскольку она означает, что если  $f$  читает или записывает такую локальную память потока (`thread-local storage — TLS`), то невозможно предсказать, к переменным какого потока будет обращение:

```
auto fut = std::async(f); // TLS для f может принадлежать
                           // независимому потоку, но может
                           // принадлежать и потоку, вызывающему
                           // get или wait объекта fut
```

Это также влияет на циклы на основе `wait` с использованием тайм-аутов, поскольку вызов `wait_for` или `wait_until` для откладываемой задачи (см. раздел 7.1) дает значение `std::future_status::deferred`. Это означает, что приведенный далее цикл, который выглядит как в конечном итоге завершающийся, может оказаться бесконечным:

```
using namespace std::literals; // Сuffixes длительности C++14;
                               // см. раздел 6.4
void f()                      // f ждет 1 с, затем выполняется
{                            // возврат из функции
    std::this_thread::sleep_for(1s);
}
auto fut = std::async(f);      // (Концептуально) асинхронное
                               // выполнение функции f
while (fut.wait_for(100ms) != // Цикл до завершения f...
       std::future_status::ready)
{
    // ... которого никогда не будет!
}
```

Если функция  $f$  выполняется параллельно с потоком, вызывающим `std::async` (т.е. если выбранная для  $f$  стратегия запуска — `std::launch::async`), нет никаких проблем (в предположении, что  $f$  в конечном итоге завершится), но если выполнение  $f$  откладывается, `fut.wait_for` всегда будет возвращать `std::future_status::deferred`. Это возвращаемое значение никогда не станет равным `std::future_status::ready`, так что цикл никогда не завершится.

Такого рода ошибки легко упустить во время разработки и модульного тестирования, потому что они могут проявляться только при больших нагрузках. При этом возможно превышение подписки или исчерпание потоков, и в такой ситуации задача, скорее всего, будет отложена. В конечном итоге, если аппаратному обеспечению не угрожает превышение подписки или исчерпание потоков, у системы времени выполнения нет никаких оснований для того, чтобы не запланировать параллельное выполнение задачи.

Исправить ошибку просто: следует проверить фьючерс, соответствующий вызову `std::async`, и выяснить, не отложена ли данная задача. Если отложена, то надо избегать входа в цикл на основе тайм-аутов. К сожалению, нет непосредственного способа выяснить у фьючерса, отложена ли задача. Вместо этого вы должны вызывать функцию на основе тайм-аута, такую как `wait_for`. В этом случае вы в действительности не хотите ничего ожидать, а хотите просто проверить, не возвращает ли она значение `std::future_status::deferred`, так что можно вызывать эту функцию с нулевым временем ожидания:

```
auto fut = std::async(f); // Как и ранее
if (fut.wait_for(0s) ==      // Если задача отложена...
    std::future_status::deferred)
{
    ...
}
else {                      // Если задача не отложена -
    while(fut.wait_for(100ms) !=      // бесконечный цикл
          std::future_status::ready) { // невозможен (если f в
                                         // конце концов завершается)
        ...
    }
    // Задача не отложена и не готова,
    // так что она выполняется параллельно
}
// fut выполнен
}
```

Из всего изложенного получается, что применение `std::async` со стратегией запуска по умолчанию отлично работает, пока выполняются следующие условия.

- Задача не обязана работать параллельно с потоком, вызывающим `get` или `wait`.
- Не имеет значения, переменные `thread_local` какого потока читаются или записываются.
- Либо гарантируется вызов `get` или `wait` для фьючерса, возвращаемого `std::async`, либо ситуация, когда задача не выполняется совсем, является приемлемой.
- Код с использованием `wait_for` или `wait_until` учитывает возможность отложенного состояния задачи.

Если не выполняется любое из этих условий, то вы, вероятно, захотите гарантировать, что `std::async` обеспечит асинхронное выполнение задачи. Для этого в качестве первого аргумента в нее надо передать значение `std::launch::async`:

```
auto fut = std::async(std::launch::async, f); // Асинхронный
                                              // запуск f
```

На практике удобно иметь функцию, работающую как `std::async`, но автоматически использующую стратегию запуска `std::launch::async`. Такую функцию несложно написать. Вот как выглядит версия этой функции для C++11:

```

template<typename F, typename... Ts>
inline
std::future<typename std::result_of<F(Ts...)>::type>
reallyAsync(F&& f, Ts&&... params)           // Возврат фьючерса
{                                                 // для асинхронного
    return std::async(std::launch::async, // вызова f(params...)
                      std::forward<F>(f),
                      std::forward<Ts>(params)...);
}

```

Эта функция получает вызываемый объект `f` и нуль или более параметров `params` и выполняет их прямую передачу (см. раздел 5.3) в `std::async`, передавая также значение `std::launch::async` в качестве стратегии вызова. Подобно `std::async`, она возвращает `std::future` для результата выполнения `f` с параметрами `params`. Определить тип этого результата просто, так как его нам дает свойство типа `std::result_of`. (Информацию о свойствах типов вы найдете в разделе 3.3.)

Функция `reallyAsync` используется так же, как и `std::async`:

```

auto fut = reallyAsync(f); // Асинхронный запуск f; генерирует
                           // исключение, если это делает
                           // std::async

```

В C++14 возможность вывода возвращаемого типа `reallyAsync` сокращает объявление функции:

```

template<typename F, typename... Ts>
inline
auto                                         // C++14
reallyAsync(F&& f, Ts&&... params)
{
    return std::async(std::launch::async,
                      std::forward<F>(f),
                      std::forward<Ts>(params)...);
}

```

Эта версия делает кристально ясным то, что `reallyAsync` не делает ничего, кроме вызова `std::async` со стратегией запуска `std::launch::async`.

### Следует запомнить

- Стратегия запуска по умолчанию для `std::async` допускает как асинхронное, так и синхронное выполнение задачи.
- Эта гибкость ведет к неопределенности при обращении к переменным `thread_local`, к тому, что задача может никогда не быть выполнена, и влияет на логику программы для вызовов `wait` на основе тайм-аутов.
- Если асинхронное выполнение задачи критично, указывайте стратегию запуска `std::launch::async`.

## 7.3. Делайте `std::thread` неподключаемым на всех путях выполнения

Каждый объект `std::thread` может находиться в двух состояниях: *подключаемом* (*joinable*) и *неподключаемом* (*unjoinable*)<sup>3</sup>. Подключаемый `std::thread` соответствует асинхронному потоку выполнения, который выполняется или может выполняться. Например, таковым является `std::thread`, соответствующий потоку, который заблокирован или ожидает запуска планировщиком. Объекты `std::thread`, соответствующие потокам, которые выполняются до полного завершения, также рассматриваются как подключаемые.

Неподключаемые объекты `std::thread` являются именно тем, что вы и ожидаете, а именно — объектами `std::thread`, которые не являются подключаемыми. Неподключаемые объекты `std::thread` включают следующие.

- **Объекты `std::thread`, созданные конструкторами по умолчанию.** Такие `std::thread` не имеют выполняемой функции, а значит, не соответствуют никакому потоку выполнения.
- **Объекты `std::thread`, из которых выполнено перемещение.** В результате перемещения поток выполнения, соответствующий `std::thread`, становится соответствующим другому объекту `std::thread`.
- **Объекты `std::thread`, для которых выполнена функция-член `join`.** После выполнения функции-члена `join` объект `std::thread` больше не соответствует потоку выполнения, который при этом вызове полностью завершается.
- **Объекты `std::thread`, для которых выполнена функция-член `detach`.** Функция-член `detach` разрывает связь между объектом `std::thread` и соответствующим ему потоком выполнения.

Одной из причин, по которым так важна подключаемость `std::thread`, является то, что при вызове деструктора для подключаемого объекта программа завершает свою работу. Предположим, например, что у нас есть функция `doWork`, которая получает функцию фильтрации `filter` и максимальное значение `maxVal` в качестве параметров. Функция `doWork` выполняет проверку, чтобы убедиться, что все условия, необходимые для ее работы, выполнены, а затем выполняет вычисления со всеми значениями от 0 до `maxVal`, проходящими через фильтр. Если фильтрация и определение выполнения условий для функции `doWork` требуют большого времени выполнения, может быть разумным выполнить эти два действия параллельно.

Мы бы предпочли воспользоваться советом из раздела 7.1 и программировать параллельность на основе задач, но предположим, что нам надо установить приоритет потока, выполняющего фильтрацию. Как пояснялось в разделе 7.1, для этого требуется

<sup>3</sup> Часто их переводят как *объединяемое* и *необъединяемое*, но, на наш взгляд, термин *подключение* лучше отражает смысл происходящего с потоком, связанным с объектом `std::thread`, при вызове функции `join`, как и отключение для происходящего при вызове функции `detach`. — Примеч. пер.

системный дескриптор потока, а он доступен только через API `std::thread`; API задач (т.е. фьючерсы) такой возможности не предоставляет. Поэтому мы работаем с потоками, а не с задачами.

Мы могли бы начать с кода наподобие следующего:

```
// См. constexpr в разделе 3.9
constexpr auto tenMillion = 10000000;

// Возвращает значение, указывающее, были ли выполнены
// вычисления; std::function см. в разделе 2.1
bool doWork(std::function<bool(int)> filter,
            int maxVal = tenMillion)
{
    // Значения, удовлетворяющие фильтру:
    std::vector<int> goodVals;
    // Заполнение goodVals:
    std::thread t([&filter, maxVal, &goodVals]
    {
        for (auto i = 0; i <= maxVal; ++i)
            { if (filter(i)) goodVals.push_back(i); }
    });

    // Используем системный дескриптор потока для
    // установки приоритета:
    auto nh = t.native_handle();

    if (conditionsAreSatisfied()) {
        t.join();                                // Завершаем t
        performComputation(goodVals);
        return true;                            // Вычисление выполнено
    }

    return false;                           // Вычисление не выполнено
}
```

Перед тем как пояснить, почему этот код проблематичен, я замечу, что инициализирующее значение `tenMillion` в C++14 можно сделать более удобочитаемым, воспользовавшись возможностью C++14 использовать апостроф для разделения цифр:

```
constexpr auto tenMillion = 10'000'000;      // C++14
```

Замечу также, что установка приоритета `t` после его запуска напоминает забивание двери конюшни после того, как лошадь уже убежала. Лучше начинать с `t` в приостановленном состоянии (тем самым делая возможным изменение его приоритета до выполнения вычислений), но я не хочу отвлекать вас этим кодом. Потерпите до раздела 7.5, в нем показано, как начинать работать с потоками в приостановленном состоянии.

Вернемся к функции `doWork`. Если вызов `conditionsAreSatisfied()` возвращает `true`, все в порядке, но если он вернет значение `false` или сгенерирует исключение, объект `t` будет подключаемым в момент вызова деструктора при окончании работы `doWork`. Это приведет к завершению работы программы.

Вы можете удивиться, почему деструктор `std::thread` ведет себя столь неподобающееся. Да просто потому, что два других варианта, описанных далее, еще хуже.

- **Неявный вызов `join`.** В этом случае деструктор `std::thread` будет ожидать завершения соответствующего асинхронного потока. Звучит разумно, но может привести к аномалиям производительности, которые будет трудно отследить. Например, было бы нелогичным, чтобы функция `doWork` ожидала применения фильтра ко всем значениям, если вызов `conditionsAreSatisfied()` уже вернул значение `false`.
- **Неявный вызов `detach`.** В этом случае деструктор `std::thread` разрывает связь между объектом `std::thread` и его потоком, отключая последний от объекта. Поток продолжает выполняться. Это звучит не менее разумно, чем применение `join`, но проблемы отладки, к которым это может привести, делают этот вариант еще худшим. Например, в функции `doWork` локальная переменная `goodVals` захватывается лямбда-выражением по ссылке. Она также изменяется в лямбда-выражении (с помощью вызова `push_back`). Предположим теперь, что во время асинхронного выполнения лямбда-выражения вызов `conditionsAreSatisfied()` вернул `false`. В таком случае функция `doWork` должна завершиться, а ее локальные переменные (включая `goodVals`) должны быть уничтожены. Ее кадр стека удаляется, и выполнение потока продолжается с точки вызова `doWork`.

Инструкции после этой точки могут в некоторой иной точке выполнять вызовы других функций, и как минимум одна из них может занять место в стеке, ранее занятом кадром стека `doWork`. Назовем эту функцию `f`. Во время работы `f` лямбда-выражение из `doWork` продолжает асинхронно выполнятся и может вызвать `push_back` для памяти в стеке, которая ранее использовалась для локальной переменной `goodVals`, а теперь принадлежит кадру стека `f`. Такой вызов может изменить память, использовавшуюся для `goodVals`, а это означает, что с точки зрения `f` содержимое памяти в ее кадре стека может внезапно измениться! Только представьте себе, что вам придется отлаживать *такое*.

Комитет по стандартизации решил, что последствия уничтожения подключаемого потока достаточно неприятны, чтобы полностью их запретить (указав, что уничтожение подключаемого потока приводит к завершению программы).

Это решение возлагает на вас ответственность за то, что если вы используете объект `std::thread`, то должны сделать его неподключаемым на всех путях из области видимости, в которой он определен. Но охват всех путей может быть весьма сложным. Он включает как нормальный выход из области видимости, так и такие выходы, как с помощью `return`, `continue`, `break`, `goto` или исключения. Этих путей может быть великое множество.

Всякий раз, когда надо выполнить некоторое действие на всех путях, ведущих из блока, естественным подходом является размещение этого действия в деструкторе локального объекта. Такие объекты известны как *объекты RAI*, а соответствующие классы — *классы RAI*. (*RAI* означает “Resource Acquisition Is Initialization”, “захват ресурса есть инициализация”, хотя на самом деле речь идет о методе деструкции, а не инициализации.) *RAI*-классы — распространенное явление в стандартной библиотеке. Примерами являются контейнеры *STL* (деструктор каждого контейнера уничтожает его содержимое и освобождает память), стандартные интеллектуальные указатели (в разделах 4.1–4.3 поясняется, что деструктор `std::unique_ptr` вызывает удалитель для объекта, на который указывает, а деструкторы `std::shared_ptr` и `std::weak_ptr` уменьшают счетчики ссылок), объекты `std::fstream` (их деструкторы закрывают соответствующие файлы) и многое другое. Тем не менее стандартного *RAI*-класса для объектов `std::thread` нет, вероятно, потому что Комитет по стандартизации, отвергнув и `join`, и `detach` как варианты по умолчанию, просто не знал, что же должен делать такой класс.

К счастью, такой класс несложно написать самостоятельно. Например, приведенный далее класс позволяет вызывающей функции указать, должна ли быть вызвана функция-член `join` или `detach` при уничтожении объекта *ThreadRAI* (объект *RAI* для `std::thread`):

```
class ThreadRAII {
public:
    enum class DtorAction { join, detach }; // См.enum class
                                            // в разделе 3.4
    ThreadRAII(std::thread& t, DtorAction a) // Деструктор для
        : action(a), t(std::move(t)) {}          // t выполняет a
    ~ThreadRAII()                            // См. ниже
    {
        if (t.joinable()) {                  // проверка на
            if (action == DtorAction::join) { // подключаемость
                t.join();
            } else {
                t.detach();
            }
        }
    }
    std::thread& get() { return t; }           // См. ниже
private:
    DtorAction action;
    std::thread t;
};
```

Я надеюсь, что этот код самодостаточен и не требует особых пояснений, тем не менее может быть полезно остановиться на следующих моментах.

- Конструктор принимает только `rvalue std::thread`, поскольку мы хотим перемещать передаваемый объект `std::thread` в объект `ThreadRAII`. (Вспомните, что объекты `std::thread` не копируются.)
- Порядок параметров выбран интуитивно понятным для программиста (сначала указывается поток, а затем — способ его деструкции), но список инициализации членов соответствует порядку объявлений членов-данных. Этот порядок размещает объект `std::thread` последним. В этом классе порядок не имеет значения, но в общем случае возможна ситуация, когда инициализация одного члена-данных зависит от другого, а поскольку объекты `std::thread` могут запускаться на выполнение немедленно после их инициализации, объявлять их последними в классе — неплохая привычка. Это гарантирует, что в момент их создания все члены-данные, им предшествующие, уже инициализированы, и, таким образом, асинхронно выполняемый поток, соответствующий объекту `std::thread`, может безопасно к ним обращаться.
- `ThreadRAII` предоставляет функцию `get`, обеспечивающую доступ к соответствующему объекту `std::thread`. Это аналог функций `get`, предоставляемых стандартными интеллектуальными указателями и обеспечивающих доступ к их базовым обычным указателям. Предоставление `get` позволяет избежать необходимости дублировать в классе `ThreadRAII` весь интерфейс `std::thread`, а также означает, что объекты `ThreadRAII` могут использоваться в контекстах, где требуются объекты `std::thread`.
- Перед тем как деструктор `ThreadRAII` вызовет функцию-член объекта `t` типа `std::thread`, он проверяет, является ли этот объект подключаемым. Это необходимо, поскольку применение `join` или `detach` к неподключаемому объекту приводит к неопределенному поведению. Может быть так, что клиент создает `std::thread`, затем создает из него объект `ThreadRAII`, использует функцию-член `get` для получения доступа к `t`, а затем выполняет перемещение из `t` или вызывает для него `join` или `detach`. Каждое из этих действий делает `t` неподключаемым.

```
if (t.joinable()) {
    if (action == DtorAction::join) {
        t.join();
    } else {
        t.detach();
    }
}
```

Если в приведенном фрагменте вас беспокоит возможность условия гонки из-за того, что между вызовами `t.joinable()` и `join` или `detach` другой поток может сделать `t` неподключаемым, то ваша интуиция заслуживает похвалы, но ваши опасения в данном случае беспочвенны. Объект `std::thread` может изменить состояние с подключаемого на неподключаемое только путем вызова функции-члена, например `join`, `detach` или операции перемещения. В момент вызова деструктора `ThreadRAII` никакие другие потоки не должны вызывать функцию-член для этого объекта. При

наличии одновременных вызовов, определенно, имеется условие гонки, но не внутри деструктора, а в клиентском коде, который пытается вызвать одновременно две функции-члена объекта (деструктор и что-то еще). В общем случае одновременные вызовы функций-членов для одного объекта безопасны, только если все они являются константными функциями-членами (см. раздел 3.10).

Использование ThreadRAII в нашей функции doWork может выглядеть следующим образом:

```
bool doWork(std::function<bool(int)> filter, // Как и ранее
            int maxVal = tenMillion)
{
    std::vector<int> goodVals;                      // Как и ранее
    ThreadRAII t( // use RAII object
        std::thread([&filter, maxVal, &goodVals]
    {
        for (auto i = 0; i <= maxVal; ++i)
            if (filter(i)) goodVals.push_back(i);
    }),
    ThreadRAII::DtorAction::join                      // Действие RAII
);

auto nh = t.get().native_handle();
...
if (conditionsAreSatisfied())
{
    t.get().join();
    performComputation(goodVals);
    return true;
}
return false;
}
```

В этом случае мы выбрали использование `join` для асинхронно выполняющегося потока в деструкторе ThreadRAII, поскольку, как мы видели ранее, применение `detach` может привести к настоящим кошмарам при отладке. Мы также видели ранее, что применение `join` может вести к аномалиям производительности (что, откровенно говоря, также может быть неприятно при отладке), но выбор между неопределенным поведением (к которому ведет `detach`), завершением программы (при использовании обычного `std::thread`) и аномалиями производительности предопределен — мы выбираем меньшее из зол.

Увы, раздел 7.5 демонстрирует, что применение ThreadRAII для выполнения `join` при уничтожении `std::thread` иногда может привести не к аномалии производительности, а к полному “зависанию” программы. “Правильным” решением этого типа проблем было бы сообщить асинхронно выполняющемуся лямбда-выражению, что в его услугах мы больше не нуждаемся и оно должно поскорее завершиться. Увы, C++11 не поддерживает *прерываемые потоки*. Их можно реализовать вручную, но данный вопрос выходит за рамки нашей книги<sup>4</sup>.

<sup>4</sup> Вы можете обратиться к книге Энтони Вильямса (Anthony Williams) *C++ Concurrency in Action* (Manning Publications, 2012), раздел 9.2.

В разделе 3.11 поясняется, что, поскольку ThreadRAII объявляет деструктор, в нем нет генерируемых компилятором перемещающих операций, но причин, по которым ThreadRAII не должен быть перемещаемым, тоже нет. Если бы компилятор генерировал такие функции, то они демонстрировали бы верное поведение, так что просто попросим компилятор их все же сгенерировать:

```
class ThreadRAII {
public:
    enum class DtorAction { join, detach };           // Как и ранее

    ThreadRAII(std::thread&& t, DtorAction a)        // Как и ранее
        : action(a), t(std::move(t)) {}

    ~ThreadRAII()
    {
        // Как и ранее
    }

    ThreadRAII(ThreadRAII&)=default;                 // Поддержка
    ThreadRAII& operator=(ThreadRAII&)=default; // перемещения

    std::thread& get() { return t; }                   // Как и ранее

private:
    DtorAction action;
    std::thread t;
};
```

#### Следует запомнить

- Делайте `std::thread` неподключаемыми на всех путях выполнения.
- Применение `join` при уничтожении объекта может привести к трудно отлаживаемым аномалиям производительности.
- Применение `detach` при уничтожении объекта может привести к трудно отлаживаемому неопределенному поведению.
- Объявляйте объекты `std::thread` в списке членов-данных последними.

## 7.4. Помните о разном поведении деструкторов дескрипторов потоков

В разделе 7.3 вы узнали, что подключаемый `std::thread` соответствует базовому системному потоку выполнения. Фьючерс для неотложенной задачи (см. раздел 7.2) имеет схожую связь с системным потоком. А раз так, и объекты `std::thread`, и объекты фьючерсов можно рассматривать как *дескрипторы* (*handles*) системных потоков.

С этой точки зрения интересно, что `std::thread` и фьючерсы совершенно по-разному ведут себя в деструкторах. Как упоминалось в разделе 7.3, уничтожение подключаемого `std::thread` завершает работу программы, поскольку две очевидные альтернативы — неявный вызов `join` и неявный вызов `detach` — оказываются еще более плохим выбором. Однако деструктор фьючерса ведет себя так, как если бы иногда выполнялся неявный вызов `join`, иногда — неявный вызов `detach`, а иногда — ни то ни другое. Он никогда не приводит к завершению работы программы. Поведение этого дескриптора потока заслуживает более внимательного рассмотрения.

Начнем с наблюдения, что фьючерс представляет собой один из концов канала связи, по которому вызываемая функция передает результаты вызывающей<sup>5</sup>. Вызываемая функция (обычно работающая асинхронно) записывает результат вычислений в коммуникационный канал (обычно с помощью объекта `std::promise`), а вызывающая функция читает результат с помощью фьючерса. Вы можете представлять это для себя следующим образом (пунктирные стрелки показывают поток информации от вызываемой функции к вызывающей):



Но где же хранится результат вызываемой функции? Вызываемая функция может завершиться до того, как будет вызвана функция-член `get` соответствующего фьючерса, так что результат не может быть сохранен в `std::promise` вызываемой функции. Этот объект, будучи локальным по отношению к вызываемой функции, уничтожается по ее завершении.

Результат не может храниться и во фьючерсе вызывающей функции, поскольку (среди прочих причин) `std::future` может быть использован для создания объекта `std::shared_future` (тем самым передавая владение результатом вызываемой функции от `std::future` в `std::shared_future`), который затем, после уничтожения исходного `std::future`, может быть многократно копирован. С учетом того, что не все типы результата могут быть скопированы (например, существуют только перемещаемые типы) и что результат должен существовать до тех пор, пока как минимум последний фьючерс на него ссылается, какой из потенциально многих фьючерсов, соответствующих вызываемой функции, должен содержать ее результат?

Поскольку ни объекты, связанные с вызываемой функцией, ни объекты, связанные с вызывающей функцией, не являются подходящими местами для хранения результата вызываемой функции, они должны храниться где-то вне этих объектов. Такое местоположение известно как *общее состояние* (*shared state*). Это общее состояние обычно представлено объектом в динамической памяти, но его тип, интерфейс и реализация в стандарте языка не указаны. Авторы стандартной библиотеки могут реализовывать общие состояния так, как хотят.

<sup>5</sup> В разделе 7.5 поясняется, что разновидность канала связи фьючерса может быть использована и для других целей. Однако в этом разделе мы будем рассматривать только его применение в качестве механизма для передачи результата из вызываемой функции вызывающей.

Мы можем представить себе отношения между вызываемой функцией, вызывающей функцией и общим состоянием следующим образом (пунктирными стрелками вновь представлен поток информации):



Существование общего состояния имеет важное значение, потому что поведение деструктора фьючерса — тема этого раздела — определяется общим состоянием, связанным с этим фьючерсом. В частности, справедливо следующее.

- **Деструктор последнего фьючерса, ссылающегося на общее состояние для неотложенной задачи, запущенной с помощью std::async, блокируется до тех пор, пока задача не будет завершена.** По сути, деструктор такого фьючерса неявно выполняет `join` для потока, асинхронно выполняющего задачу.
- **Деструкторы всех прочих фьючерсов просто уничтожают объект фьючерса.** Для асинхронно выполняющихся задач это сродни неявному отключению базового потока с помощью вызова `detach`. Для отложенных задач, для которых данный фьючерс является последним, это означает, что отложенная задача никогда не будет выполнена.

Эти правила выглядят сложнее, чем есть на самом деле. Мы имеем дело с простым “нормальным” поведением и одним исключением. Нормальное поведение заключается в том, что деструктор фьючерса уничтожает объект фьючерса. Вот и все. Он ничего не подключает, ничего не отключает, он ничего не запускает. Он просто уничтожает члены-данные фьючерса. (Ну, на самом деле он делает еще одно дело — уменьшает значение счетчика ссылок общего состояния, которое управляетя как ссылающимися на него фьючерсами, так и объектами `std::promise` вызываемой функции. Этот счетчик ссылок позволяет библиотеке знать, когда можно уничтожать общее состояние. Информацию по счетчикам ссылок вы можете найти в разделе 4.2.)

Иключение из этого нормального поведения осуществляется только для фьючерса, для которого выполняются все перечисленные далее условия.

- Он ссылается на общее состояние, созданное вызовом `std::async`.
- Стратегия запуска задачи — `std::launch::async` (см. раздел 7.2), либо потому, что она выбрана системой времени выполнения, либо потому, что была явно указана в вызове `std::async`.
- Фьючерс является последним фьючерсом, ссылающимся на общее состояние. Это всегда справедливо для `std::future`. Для `std::shared_future`, если при уничтожении фьючерса на то же самое общее состояние ссылаются другие `std::shared_future`, поведение уничтожаемого фьючерса — нормальное (т.е. просто уничтожаются его члены-данные).

Только когда все эти условия выполнены, деструктор фьючерса демонстрирует особое поведение, и это поведение состоит в блокировке до тех пор, пока не будет завершена асинхронно выполняющаяся задача. С практической точки зрения это равносильно неявному вызову `join` для потока с запущенной задачей, созданной с помощью `std::async`.

Часто приходится слышать, что это исключение из нормального поведения деструктора фьючерса резюмируется как “Фьючерс из `std::async` блокируется в своем деструкторе”. В качестве первого приближения это так, но иногда нам надо что-то большее, чем первое приближение. Теперь вы знаете правду во всей ее красе и славе.

Ваше удивление может принять и иную форму. Например, “Не понимаю, почему имеется особое правило для общих состояний для неотложенных задач, запущенных с помощью `std::async`”. Это разумный вопрос. Я могу сказать, что Комитет по стандартизации хотел избежать проблем, связанных с неявным вызовом `detach` (см. раздел 7.3), но не хотел одобрять такую радикальную стратегию, как обязательное завершение программы (как сделано для подключаемых `std::thread`; см. тот же раздел 7.3), так что в качестве компромисса был принят неявный вызов `join`. Это решение не без противоречий, и были серьезные предложения отказаться от этого поведения в C++ 14. В конце концов никакие изменения сделаны не были, так что поведение деструкторов фьючерсов в C++11 согласуется с таковым в C++14.

API для фьючерсов не предлагает способа определения, ссылается ли фьючерс на общее состояние, возникающее из вызова `std::async`, так что невозможно узнать, будет ли заблокирован некоторый объект фьючерса в своем деструкторе для ожидания завершения асинхронно выполняющейся задачи. Это имеет некоторые интересные последствия.

```
// Этот контейнер может блокироваться в деструкторе, поскольку
// один или несколько содержащихся в нем фьючерсов могут
// ссылаться на общее состояние неотложенного задания,
// запущенного с помощью std::async
std::vector<std::future<void>> futs; // См. std::future<void>
                                         // в разделе 7.5
class Widget {                         // Объекты Widget могут
public:                                // блокироваться в их
    ...                                  // деструкторах
private:
    std::shared_future<double> fut;
};
```

Конечно, если у вас есть способ узнать, что данный фьючерс *не* удовлетворяет условиям, приводящим к специальному поведению деструктора (например, в силу логики программы), вы можете быть уверены, что этот фьючерс не приведет к блокировке деструктора. Например, претендовать на особое поведение могут только общие состояния, получающиеся в результате вызовов `std::async`, но есть и иные способы создания этих общих состояний. Один из них — использование `std::packaged_task`. Объект `std::packaged_task` подготавливает функцию (или иной вызываемый объект) к асинхронному выполнению, “заворачивая” ее таким образом, что ее результат помещается

в общее состояние. Фьючерс, ссылающийся на это общее состояние, может быть получен с помощью функции `get_future` объекта `std::packaged_task`.

```
int calcValue();           // Выполняемая функция
std::packaged_task<int()> pt(calcValue); // Заворачивание calcValue для
                                         // асинхронного выполнения
auto fut = pt.get_future(); // Получение фьючерса для pt
```

В этой точке мы знаем, что фьючерс `fut` не ссылается на общее состояние, созданное вызовом `std::async`, так что его деструктор будет вести себя нормально.

Будучи созданным, объект `pt` типа `std::packaged_task` может быть запущен в потоке. (Он может быть запущен и с помощью вызова `std::async`, но если вы хотите выполнить задачу с использованием `std::async`, то нет смысла создавать `std::packaged_task`, поскольку `std::async` делает все, что делает `std::packaged_task` до того, как планировщик начинает выполнение задачи.)

Объекты `std::packaged_task` не копируются, так что когда `pt` передается в конструктор `std::thread`, он должен быть приведен к `gvalue` (с помощью `std::move`; см. раздел 5.1):

```
std::thread t(std::move(pt)); // Выполнение pt потоком t
```

Этот пример дает некоторое представление о нормальном поведении деструкторов фьючерсов, но его легче увидеть, если собрать весь код вместе в одном блоке:

```
{ // Начало блока
    std::packaged_task<int()>
        pt(calcValue);
    auto fut = pt.get_future();
    std::thread t(std::move(pt));
    // См. ниже
} // Конец блока
```

Наиболее интересный код скрывается за троеточием "...", следующим за созданием объекта `t` типа `std::thread` и предшествующим концу блока. Имеются три основные возможности.

- **С `t` ничего не происходит.** В этом случае `t` в конце области видимости будет неподключаемым. Это приведет к завершению программы (см. раздел 7.3).
- **Для `t` вызывается функция-член `join`.** В этом случае фьючерсу `fut` не требуется блокировать деструктор, так как вызов `join` уже имеется в вызывающем коде.
- **Для `t` вызывается функция-член `detach`.** В этом случае фьючерсу `fut` не требуется вызывать `detach` в деструкторе, поскольку вызывающий код уже сделал это.

Другими словами, когда у вас есть фьючерс, соответствующий общему состоянию, получившемуся из-за применения `std::packaged_task`, обычно не требуется принимать специальную стратегию деструкции, так как решение о прекращении выполнения программы, подключении или отключении потока принимается в коде, работающем с потоком `std::thread`, в котором выполняется `std::packaged_task`.

### Следует запомнить

- Деструкторы фьючерсов обычно просто уничтожают данные-члены фьючерсов.
- Последний фьючерс, ссылающийся на общее состояние неотложенной задачи, запущенной с помощью `std::async`, блокируется до завершения этой задачи.

## 7.5. Применяйте фьючерсы `void` для одноразовых сообщений о событиях

Иногда требуется, чтобы одна задача могла сообщить другой, выполняющейся асинхронно, о том, что произошло некоторое событие, поскольку вторая задача не может продолжать работу, пока это событие не произойдет. Например, пока не будет инициализирована структура данных, не будет завершено некоторое вычисление или не будет обнаружен сигнал от датчика. Какой в этом случае способ межпоточного сообщения является наилучшим?

Очевидный подход заключается в применении *переменной условия*. Если назвать задачу, которая обнаруживает условие, *задачей обнаружения*, а задачу, которая на него реагирует, — *задачей реакции*, то выразить стратегию просто: задача реакции ожидает переменную условия, а поток задачи обнаружения выполняет ее уведомление при наступлении события. При

```
std::condition_variable cv; // Переменная условия события  
std::mutex m;           // Мьютекс для использования с cv
```

код задачи обнаружения прост настолько, насколько это возможно:

```
...                                // Обнаружение события  
cv.notify_one();                  // Уведомление задачи реакции
```

Если требуется уведомить несколько задач реакции, можно заменить `notify_one` на `notify_all`, но пока что будем считать, что у нас только одна задача реакции.

Код задачи реакции немного сложнее, поскольку перед вызовом `wait` для переменной условия он должен блокировать мьютекс с помощью объекта `std::unique_lock`. (Блокировка мьютекса перед ожиданием переменной условия типична для многопоточных библиотек. Необходимость блокировки мьютекса с помощью объекта `std::unique_lock` является частью API C++11.) Вот как выглядит концептуальный подход:

```
...                                // Подготовка к реакции  
{  
    std::unique_lock<std::mutex> lk(m);          // Блокировка мьютекса  
    cv.wait(lk);                                // Ожидание уведомления;  
                                                // неверно!  
                                                // Реакция на событие  
                                                // (m заблокирован)  
}                                                // Закрытие критического раздела;
```

```
// разблокирование m с
// помощью деструктора lk
// Продолжение реакции
// (m разблокирован)
```

Первой проблемой при таком подходе является то, что часто называют *кодом с душком* (code smell): даже если команда работает, что-то выглядит не совсем верным. В нашем случае запах исходит от необходимости применения мьютексов. Мьютексы используются для управления доступом к совместно используемым данным, но вполне возможно, что для задач обнаружения и реакции такой посредник не требуется. Например, задача обнаружения может отвечать за инициализацию глобальной структуры данных, которая затем передается для использования задаче реакции. Если задача обнаружения никогда не обращается к структуре данных после ее инициализации и если задача реакции никогда не обращается к ней до того, как задача обнаружения укажет, что структура данных готова к использованию, эти две задачи оказываются не связанными логикой программы одна с другой. При этом нет никакой необходимости в мьютексе. Тот факт, что подход с использованием переменной условия требует применения мьютексов, оставляет тревожащий запашок подозрительного дизайна.

Даже если пропустить этот вопрос, все равно остаются две проблемы, которым, определенно, следует уделить внимание.

- **Если задача обнаружения уведомляет переменную условия до вызова `wait` задачей реакции, задача реакции “зависнет”.** Чтобы уведомление переменной условия активизировало другую задачу, эта другая задача должна находиться в состоянии ожидания переменной условия. Если вдруг задача обнаружения выполняет уведомление до того, как задача реакции выполняет `wait`, эта задача реакции пропустит уведомление и будет ждать его вечно.
- **Вызов `wait` приводит к ложным пробуждениям.** В потоковых API (во многих языках программирования, не только в C++) не редкость ситуация, когда код, ожидающий переменную условия, может быть пробужден, даже если переменная условия не была уведомлена. Такое пробуждение называется *ложным пробуждением* (spurious wakeup). Корректный код обрабатывает такую ситуацию, проверяя, что ожидаемое условие в действительности выполнено, и это делается первым, немедленно после пробуждения. API переменных условия C++ делает это исключительно простым, поскольку допускает применение лямбда-выражений (или иных функциональных объектов), которые проверяют условие, переданное в `wait`. Таким образом, вызов `wait` в задаче реакции может быть записан следующим образом:

```
cv.wait(lk,
[] { return Произошло ли событие; });
```

Применение этой возможности требует, чтобы задача реакции могла выяснить, истинно ли ожидаемое ею условие. Но в рассматриваемом нами сценарии ожидаемым условием является наступление события, за распознавание которого

отвечает поток обнаружения. Поток реакции может быть не в состоянии определить, имело ли место ожидаемое событие. Вот почему он ожидает переменную условия!

Имеется много ситуаций, когда сообщение между задачами с помощью переменной условия вполне решает стоящую перед программистом проблему, но не похоже, что перед нами одна из них.

Многие разработчики используют совместно используемый булев флаг. Изначально этот флаг имеет значение `false`. Когда поток обнаружения распознает ожидаемое событие, он устанавливает этот флаг:

```
std::atomic<bool> flag(false); // Совместно используемый флаг;
                                // std::atomic см. в разделе 7.6
...
flag = true;                  // Сообщение задаче обнаружения
```

Со своей стороны поток реакции просто опрашивает флаг. Когда он видит, что флаг установлен, он знает, что ожидаемое событие произошло:

```
...                      // Подготовка к реакции
while (!flag);          // Ожидание события
                        // Реакция на событие
```

Этот подход не страдает ни одним из недостатков проекта на основе переменной условия. Нет необходимости в мьютексе, не проблема, если задача обнаружения устанавливает флаг до того, как задача реакции начинает опрос, и нет ничего подобного ложным пробуждениям. Хорошо, просто замечательно.

Куда менее замечательно выглядит стоимость опроса в задаче реакции. Во время ожидания флага задача, по сути, заблокирована, но продолжает выполняться. А раз так, она занимает аппаратный поток, который мог бы использоваться другой задачей, требует стоимости переключения контекста при каждом начале и завершении выделенных потоку временных промежутков и заставляет работать ядро, которое в противном случае могло бы быть отключено для экономии энергии. При настоящей блокировке задача не делает ничего из перечисленного. Это является преимуществом подхода на основе переменных условия, поскольку блокировка задачи при вызове `wait` является истинной.

Распространено сочетание подходов на основе переменных условия и флагов. Флаг указывает, произошло ли интересующее нас событие, но доступ к флагу синхронизирован мьютексом. Поскольку мьютекс предотвращает параллельный доступ к флагу, не требуется, как поясняется в разделе 7.6, чтобы флаг был объявлен как `std::atomic`; вполне достаточно простого `bool`. Задача обнаружения в этом случае может иметь следующий вид:

```
std::condition_variable cv;           // Как и ранее
std::mutex m;                       //
bool flag(false);                   // Не std::atomic
...
{
    std::lock_guard<std::mutex> g(m); // Блокировка m
                                        // конструктором g
```

```

    flag = true;                                // Сообщаем задаче реакции
}                                              // (часть 1)
cv.notify_one();                             // Снятие блокировки m
                                              // деструктором g
                                              // Сообщаем задаче реакции
                                              // (часть 2)

```

А вот задача реакции:

```

...                                         // Подготовка к реакции
{                                           // Как и ранее
    std::unique_lock<std::mutex> lk(m); // Как и ранее
    cv.wait(lk, []{ return flag; });   // Применение лямбда-
                                         // выражения во избежание
                                         // ложных пробуждений
                                         // Реакция на событие
                                         // (m заблокирован)
}
                                         // Продолжение реакции
                                         // (m разблокирован)

```

Этот подход позволяет избежать проблем, которые мы обсуждали. Он работает независимо от того, вызывает ли задача реакции `wait` до уведомления задачей обнаружения, он работает при наличии ложных пробуждений и не требует опроса флага. Тем не менее душок остается, потому что задача обнаружения взаимодействует с задачей реакции очень любопытным способом. Уведомляемая переменная условия говорит задаче реакции о том, что, вероятно, произошло ожидаемое событие, но задаче реакции необходимо проверить флаг, чтобы быть в этом уверенной. Установка флага говорит задаче реакции, что событие, определенно, произошло, но задача обнаружения по-прежнему обязана уведомить переменную условия о том, чтобы задача реакции активизировалась и проверила флаг. Этот подход работает, но не кажется очень чистым.

Альтернативный вариант заключается в том, чтобы избежать переменных условия, мьютексов и флагов с помощью вызова `wait` задачей реакции для фьючерса, установленного задачей обнаружения. Это может показаться странной идеей. В конце концов, в разделе 7.4 поясняется, что фьючерс представляет принимающий конец канала связи от вызываемой функции к (обычно асинхронной) вызывающей функции, а между задачами обнаружения и реакции нет отношений “вызываемая–вызывающая”. Однако в разделе 7.4 также отмечается, что канал связи, передающий конец которого представляет собой `std::promise`, а принимающий — фьючерс, может использоваться для большего, чем простой обмен информацией между вызываемой и вызывающей функциями. Такой канал связи может быть использован в любой ситуации, в которой необходима передача информации из одного места вашей программы в другое. В нашем случае мы воспользуемся им для передачи информации от задачи обнаружения задаче реакции, и информация, которую мы будем передавать, — о том, что произошло интересующее нас событие.

Проект прост. Задача обнаружения имеет объект `std::promise` (т.е. передающий конец канала связи), а задача реакции имеет соответствующий фьючерс. Когда задача обнаружения видит, что произошло ожидаемое событие, она устанавливает объект `std::promise` (т.е. выполняет запись в канал связи). Тем временем задача реакции выполняет вызов `wait` своего фьючерса. Этот вызов `wait` блокирует задачу реакции до тех пор, пока не будет установлен объект `std::promise`.

И `std::promise`, и фьючерсы (т.е. `std::future` и `std::shared_future`) являются шаблонами, требующими параметр типа. Этот параметр указывает тип данных, передаваемый по каналу связи. Однако в нашем случае никакие данные не передаются. Единственное, что представляет интерес для задачи реакции, — что ее фьючерс установлен. Нам нужно указать для шаблонов `std::promise` и фьючерса тип, показывающий, что по каналу связи не будут передаваться никакие данные. Таким типом является `void`. Задача обнаружения, таким образом, будет использовать `std::promise<void>`, а задача реакции — `std::future<void>` или `std::shared_future<void>`. Задача обнаружения устанавливает свой объект `std::promise<void>`, когда происходит интересующее нас событие, а задача реакции ожидает с помощью вызова `wait` своего фьючерса. Даже несмотря на то, что задача реакции не получает никаких данных от задачи обнаружения, канал связи позволит задаче реакции узнать, что задача обнаружения “записала” `void`-данные с помощью вызова `set_value` своего объекта `std::promise`.

Так что для данного

```
std::promise<void> p; // Коммуникационный канал
```

код задачи обнаружения тривиален:

```
... // Обнаружение события  
p.set_value(); // Сообщение задаче реакции
```

Код задачи реакции не менее прост:

```
... // Подготовка к реакции  
p.get_future().wait(); // Ожидание фьючерса,  
// соответствующего p  
// Реакция на событие
```

Этот подход, как и использование флагов, не требует мьютексов, работает независимо от того, устанавливает ли задача обнаружения свой объект `std::promise` до того, как задача реакции вызывает `wait`, и невосприимчива к ложным пробуждениям. (Этой проблеме подвержены только переменные условия.) Подобно подходу на основе переменных условия задача реакции оказывается истинно заблокированной после вызова `wait`, так что во время ожидания не потребляет системные ресурсы. Идеально, нет?

Не совсем. Конечно, подход на основе фьючерсов обходит описанные неприятности, но ведь есть и другие. Например, в разделе 7.4 поясняется, что между `std::promise` и фьючерсом находится общее состояние, а общие состояния обычно выделяются динамически. Поэтому следует предполагать, что данное решение приводит к расходам на динамическое выделение и освобождение памяти.

Вероятно, еще более важно то, что `std::promise` может быть установлен только один раз. Канал связи между `std::promise` и фьючерсом является одноразовым механизмом: он не может быть использован многократно. Это существенное отличие от применения переменных условия и флагов, которые могут использоваться для связи много раз. (Переменная условия может быть уведомлена неоднократно, а флаг может сбрасываться и устанавливаться вновь.)

Ограничение однократности не столь тяжкое, как можно подумать. Предположим, что вы хотите создать системный поток в приостановленном состоянии. То есть вы хотели бы заплатить все накладные расходы, связанные с созданием потока, заранее, с тем, чтобы как только вы будете готовы выполнить что-то в этом потоке, вы сможете делать это сразу, без задержки. Или, может быть, вы захотите создать поток в приостановленном состоянии с тем, чтобы можно было настроить его перед выполнением. Такая настройка может включать, например, установку приоритета. API параллельных вычислений C++ не предоставляет способ выполнить такие действия, но объекты `std::thread` предлагают функцию-член `native_handle`, результат которой призван предоставить доступ к API многопоточности используемой платформы (обычно потокам POSIX или Windows). Низкоуровневые API часто позволяют настраивать такие характеристики потоков, как приоритет или сродство.

В предположении, что требуется приостановить поток только один раз (после создания, но до запуска функции потока), можно воспользоваться `void`-фьючерсом. Вот как выглядит эта методика.

```
std::promise<void> p;
void react(); // Функция потока реакции
void detect(); // Функция потока обнаружения
{
    std::thread t([]) // Создание потока
    {
        p.get_future().wait(); // Приостановлен до
        react(); // установки фьючерса
    });
    // Здесь t приостановлен
    // до вызова react
    p.set_value(); // Запуск t (и тем самым вызов react)
    ...
    t.join(); // Выполнение дополнительной работы
    // Делаем t неподключаемым
} // (см. раздел 7.3)
```

Поскольку важно, чтобы поток `t` стал неподключаемым на всех путях выполнения, ведущих из `detect`, применение RAII-класса наподобие приведенного в раздела 7.3 класса `ThreadRAII` выглядит целесообразным. На ум приходит следующий код:

```
void detect()
{
    ThreadRAII tr( // RAII-объект
        std::thread([])
```

```

    {
        p.get_future().wait();
        react();
    }),
ThreadRAII::DtorAction::join // Рискованно! (См. ниже)
);
...
// Поток в tr приостановлен
p.set_value(); // Поток в tr разблокирован
}

```

Выглядит безопаснее, чем на самом деле. Проблема в том, что, если в первой области “...” (с комментарием “Поток в tr приостановлен”) будет сгенерировано исключение, для `p` никогда не будет вызвана функция `set_value`. Это означает, что вызов `wait` в лямбда-выражении никогда не завершится. А это, в свою очередь, означает, что поток, выполняющий лямбда-выражение, никогда не завершается, а это представляет собой проблему, поскольку RAII-объект `tr` сконфигурирован для выполнения `join` для этого потока в деструкторе. Другими словами, если в первой области кода “...” будет сгенерировано исключение, эта функция “зависнет”, поскольку деструктор `tr` никогда не завершится.

Имеются способы решения этой проблемы, но я оставлю их в священном виде упражнения для читателя<sup>6</sup>. Здесь я хотел бы показать, как исходный код (т.е. без применения `ThreadRAII`) может быть расширен для приостановки и последующего продолжения не одной задачи реакции, а нескольких. Это простое обобщение, поскольку ключом является применение `std::shared_future` вместо `std::future` в коде `react`. Как вы уже знаете, функция-член `share` объекта `std::future` передает владение его общим состоянием объекту `std::shared_future`, созданному `share`, а после этого код пишется почти сам по себе. Единственной тонкостью является то, что каждый поток реакции требует собственную копию `std::shared_future`, которая ссылается на общее состояние, так что объект `std::shared_future`, полученный от `share`, захватывается по значению лямбда-выражением, запускаемым в потоке реакции:

```

std::promise<void> p; // Как и ранее
void detect() // Теперь для нескольких
{
    auto sf=p.get_future().share(); // Тип sf -
                                    // std::shared_future<void>
    std::vector<std::thread> vt; // Контейнер для потоков
                                // реакции
    for (int i = 0; i < threadsToRun; ++i) {
        // Ожидание локальной копии sf;
        // см. emplace_back в разделе 8.2:
        vt.emplace_back([sf]{ sf.wait();
                            react(); });
    }
}

```

---

<sup>6</sup> Разумной отправной точкой для начала изучения этого вопроса является запись в моем блоге от 24 декабря 2013 года в *The View From Aristeia*, “ThreadRAII + Thread Suspension = Trouble?”

```

    }

        // detect "зависает", если
        // здесь генерируется
        // исключение!
    p.set_value();           // Продолжение всех потоков

    ...
    for (auto& t : vt) {
        t.join();            // Все потоки делаются
                            // неподключаемыми;
    }                      // см. "auto&" в разделе 1.2
}

}

```

Примечателен тот факт, что дизайн с помощью фьючерсов позволяет добиться описанного эффекта, так что поэтому следует рассматривать возможность его применения там, где требуется одноразовое сообщение о событии.

### Следует запомнить

- Для простого сообщения о событии дизайн с применением переменных условия требует избыточных мьютексов, накладывает ограничения на относительное выполнение задач обнаружения и реакции и требует от задачи реакции проверки того, что событие в действительности имело место.
- Дизайн с использованием флага устраняет эти проблемы, но использует опрос, а не блокировку.
- Переменные условия и флаги могут быть использованы совместно, но получающийся механизм сообщений оказывается несколько неестественным.
- Применение объектов `std::promise` и фьючерсов решает указанные проблемы, но этот подход использует динамическую память для общих состояний и ограничен одноразовым сообщением.

## 7.6. Используйте `std::atomic` для параллельности, `volatile` — для особой памяти

Бедный квалификатор `volatile!` Такой неверно понимаемый... Его даже не должно быть в этой главе, потому что он не имеет ничего общего с параллельным программированием. Но в других языках (например, в Java и C#) он полезен для такого программирования, и даже в C++ некоторые компиляторы перенасыщены `volatile` с семантикой, делающей его применимым для параллельного программирования (но только при компиляции этими конкретными компиляторами). Таким образом, имеет смысл обсудить `volatile` в главе, посвященной параллельным вычислениям, хотя бы для того, чтобы развеять окружающую его путаницу.

Возможность C++, которую программисты периодически путают с `volatile` и которая, безусловно, относится к данной главе, — это шаблон `std::atomic`. Инстанцирования этого шаблона (например, `std::atomic<int>`, `std::atomic<bool>`,

`std::atomic<Widget*>` и т.п.) предоставляют операции, которые другими потоками будут гарантированно восприниматься как атомарные. После создания объекта `std::atomic` операции над ним ведут себя так, как будто они выполняются внутри критического раздела, защищенного мьютексом, но эти операции обычно реализуются с помощью специальных машинных команд, которые значительно эффективнее применения мьютексов.

Рассмотрим код с применением `std::atomic`:

```
std::atomic<int> ai(0); // Инициализация ai значением 0
ai = 10;                // Атомарное присваивание ai значения 10
std::cout << ai;        // Атомарное чтение значения ai
++ai;                  // Атомарный инкремент ai до 11
--ai;                  // Атомарный декремент ai до 10
```

В процессе выполнения данных инструкций другие потоки, читающие `ai`, могут увидеть только значения 0, 10 и 11. Никакие другие значения невозможны (конечно, в предположении, что это единственный поток, модифицирующий `ai`).

Следует отметить два аспекта этого примера. Во-первых, в инструкции `"std::cout << ai;"` тот факт, что `ai` представляет собой `std::atomic`, гарантирует только то, что атомарным является чтение `ai`. Нет никакой гарантии атомарности всей инструкции. Между моментом чтения значения `ai` и вызовом оператора `operator<<` для записи в поток стандартного вывода другой поток может изменить значение `ai`. Это не влияет на поведение инструкции, поскольку `operator<<` для `int` использует передачу выводимого параметра типа `int` по значению (таким образом, выведенное значение будет тем, которое прочитано из `ai`), но важно понимать, что во всей этой инструкции атомарным является только чтение значения `ai`.

Второй важный аспект этого примера заключается в поведении двух последних инструкций — инкремента и декремента `ai`. Каждая из этих операций является операцией чтения-изменения-записи (`read-modify-write` — RMW), выполняемой атомарно. Это одна из приятнейших характеристик типов `std::atomic`: если объект `std::atomic` создан, все его функции-члены, включая RMW-операции, будут гарантированно рассматриваться другими потоками как атомарные.

В противоположность этому такой же код, но использующий квалификатор `volatile`, в многопоточном контексте не гарантирует почти ничего:

```
volatile vi(0);      // Инициализация vi значением 0
vi = 10;              // Присваивание vi значения 10
std::cout << vi;      // Чтение значения vi
++vi;                // Инкремент vi до 11
--vi;                // Декремент vi до 10
```

Во время выполнения этого кода, если другой поток читает значение `vi`, он может прощать что угодно, например `-12, 68, 4090727`, — любое значение! Такой код обладает неопределенным поведением, потому что эти инструкции изменяют `vi`, и если другие потоки читают `vi` в тот же момент времени, то эти одновременные чтения и записи памяти не защищены ни `std::atomic`, ни с помощью мьютексов, а это и есть определение гонки данных.

В качестве конкретного примера того, как отличаются поведения `std::atomic` и `volatile` в многопоточной программе, рассмотрим простые счетчики каждого вида, увеличиваемые несколькими потоками. Инициализируем каждый из них значением 0:

```
std::atomic<int> ac(0); // "счетчик atomic"  
volatile int vc(0); // "счетчик volatile"
```

Затем увеличим каждый счетчик по разу в двух одновременно работающих потоках:

```
/*---- Поток 1 ----*/ /*---- Поток 2 ----*/  
    ++ac;           ++ac;  
    ++vc;           ++vc;
```

По завершении обоих потоков значение `ac` (т.е. значение `std::atomic`) должно быть равно 2, поскольку каждый инкремент осуществляется как атомарная, неделимая операция. Значение `vc`, с другой стороны, не обязано быть равным 2, поскольку его инкремент может не быть атомарным. Каждый инкремент состоит из чтения значения `vc`, увеличения прочитанного значения и записи результата обратно в `vc`. Но для объектов `volatile` не гарантируется атомарность всех трех операций, так что части двух инкрементов `vc` могут чередоваться следующим образом.

1. Поток 1 считывает значение `vc`, равное 0.
2. Поток 2 считывает значение `vc`, все еще равное 0.
3. Поток 1 увеличивает 0 до 1 и записывает это значение в `vc`.
4. Поток 1 увеличивает 0 до 1 и записывает это значение в `vc`.

Таким образом, окончательное значение `vc` оказывается равным 1, несмотря на два инкремента.

Это не единственный возможный результат. В общем случае окончательное значение `vc` непредсказуемо, поскольку переменная `vc` включена в гонку данных, а стандарт однозначно утверждает, что гонка данных ведет к неопределенному поведению; это означает, что компиляторы могут генерировать код, выполняющий буквально все что угодно. Конечно, компиляторы не используют эту свободу для чего-то вредоносного. Но они могут выполнить оптимизации, вполне корректные при отсутствии гонки данных, и эти оптимизации приведут к неопределенному и непредсказуемому поведению при наличии гонки данных.

RMW-операции — не единственная ситуация, в которой применение `std::atomic` ведет к успешным параллельным вычислениям, а `volatile` — к неудачным. Предположим, что одна задача вычисляет важное значение, требуемое для второй задачи. Когда первая задача вычисляет значение, она должна сообщить об этом второй задаче. В разделе 7.5 поясняется, что одним из способов, которым один поток может сообщить о доступности требуемого значения другому потоку, является применение `std::atomic<bool>`. Код в задаче, выполняющей вычисление значения, имеет следующий вид:

```
std::atomic<bool> valAvailable(false);  
auto impValue = computeImportantValue(); // Вычисление значения
```

```
valAvailable = true; // Сообщение об этом
// другому потоку
```

Как люди, читая этот код, мы знаем, что критично важно, чтобы присваивание `imptValue` имело место до присваивания `valAvailable`, но все компиляторы видят здесь просто пару присваиваний независимым переменным. В общем случае компиляторы имеют право переупорядочить такие независимые присваивания. Иначе говоря, последовательность присваиваний (где `a`, `b`, `x` и `y` соответствуют независимым переменным)

```
a = b;
x = y;
```

компиляторы могут переупорядочить следующим образом:

```
x = y;
a = b;
```

Даже если такое переупорядочение выполнено не будет, это может сделать аппаратное обеспечение (или сделать его видимым таковым для других ядер, если таковые имеются в наличии), поскольку иногда это может сделать код более быстрым.

Однако применение `std::atomic` накладывает ограничения на разрешенные переупорядочения кода, и одно такое ограничение заключается в том, что никакой код, предшествующий в исходном тексте записи переменной `std::atomic`, не может иметь место (или выглядеть таковым для других ядер) после нее<sup>7</sup>. Это означает, что в нашем коде

```
auto imptValue = computeImportantValue(); // Вычисление значения
valAvailable = true; // Сообщение об этом
// другому потоку
```

компиляторы должны не только сохранять порядок присваиваний `imptValue` и `valAvailable`, но и генерировать код, который гарантирует, что так же поведет себя и аппаратное обеспечение. В результате объявление `valAvailable` как `std::atomic` гарантирует выполнение критичного требования к упорядоченности — что значение `imptValue` должно быть видимо всеми потоками как измененное не позже, чем значение `valAvailable`.

Объявление `valAvailable` как `volatile` не накладывает такое ограничение на переупорядочение кода:

```
volatile bool valAvailable(false);
auto imptValue = computeImportantValue();
```

<sup>7</sup> Это справедливо только для `std::atomic`, использующих *последовательную согласованность*, которая является применяемой по умолчанию (и единственной) моделью согласованности для объектов `std::atomic`, использующих показанный в этой книге синтаксис. C++11 поддерживает также модели согласованности с более гибкими правилами переупорядочения кода. Такие *слабые* (или *смягченные*) модели делают возможным создание программного обеспечения, работающего более быстро на некоторых аппаратных архитектурах, но применение таких моделей дает программное обеспечение, которое гораздо труднее правильно понимать и поддерживать. Тонкие ошибки в коде с ослабленной атомарностью не являются редкостью даже для экспертов, так что вы должны придерживаться, насколько это возможно, последовательной согласованности.

```
valAvailable = true; // Другие потоки могут увидеть это
                    // присваивание до присваивания imptValue!
```

Здесь компиляторы могут изменить порядок присваиваний переменным `imptValue` и `valAvailable`, но даже если они этого не сделают, они могут не генерировать машинный код, который предотвратит возможность аппаратному обеспечению сделать так, что другие ядра увидят изменение `valAvailable` до изменения `imptValue`.

Эти две проблемы — отсутствие гарантии атомарности операции и недостаточные ограничения на переупорядочение кода — поясняют, почему `volatile` бесполезен для параллельного программирования, но не поясняют, для чего же этот квалификатор полезен. В двух словах — чтобы сообщать компиляторам, что они имеют дело с памятью, которая не ведет себя нормально.

“Нормальная”, “обычная” память обладает тем свойством, что если вы записываете в нее значение, то оно остается неизменным, пока не будет перезаписано. Так что если у меня есть обычный `int`

```
int x;
```

и компилятор видит последовательность операций

```
auto y = x; // Чтение x
y = x;      // Чтение x еще раз
```

то он может оптимизировать генерируемый код, убрав присваивание переменной `y`, поскольку оно является излишним из-за инициализации `y`.

Обычная память обладает также тем свойством, что если вы запишете значение в ячейку памяти, никогда не будете его читать, а потом запишете туда же что-то еще, то первую запись можно не выполнять, потому что записанное ею значение никогда не используется. С учетом этого в коде

```
x = 10; // Запись x
x = 20; // Запись x еще раз
```

компиляторы могут убрать первую инструкцию. Это означает, что если у нас имеется код

```
auto y = x; // Чтение x
y = x;      // Чтение x еще раз
x = 10;     // Запись x
x = 20;     // Запись x еще раз
```

то компиляторы могут рассматривать его, как если бы он имел следующий вид:

```
auto y = x; // Чтение x
x = 20;     // Запись x
```

Чтобы вас не мучило любопытство, кто в состоянии написать такой код с избыточными чтениями и лишними записями (технически известными как избыточные загрузки (*redundant loads*) и *бессмысленные сохранения* (*dead stores*)), отвечу: нет, люди не пишут непосредственно такой код, по крайней мере я очень на это надеюсь. Однако после того как компиляторы получают разумно выглядящий код и выполняют инстанцирования шаблонов, встраивание кода и различные виды переупорядочивающих оптимизаций,

в результате не так уже редко получаются и избыточные загрузки, и бессмысленные сохранения, от которых компиляторы могут избавиться.

Такие оптимизации корректны, только если память ведет себя нормально. “Особая” память так себя не ведет. Наиболее распространенным видом особой памяти, вероятно, является память, используемая для *отображенного на память ввода-вывода*. Вместо чтения и записи обычной памяти, местоположения в такой особой памяти в действительности сообщаются с периферийными устройствами, например внешними датчиками или мониторами, принтерами, сетевыми портами и т.п. Давайте с учетом этого снова рассмотрим код с, казалось бы, избыточными чтениями:

```
auto y = x; // Чтение x  
y = x; // Чтение x еще раз
```

Если x соответствует, скажем, значению, которое передает датчик температуры, то второе чтение x избыточным не является, поскольку температура между первым и вторым чтениями может измениться.

Похожа ситуация с записями, кажущимися излишними. Например, если в коде

```
x = 10; // Запись x  
x = 20; // Запись x еще раз
```

переменная x соответствует управляющему порту радиопередатчика, может оказаться, что этот код выполняет некоторые команды с радиопередатчиком, и значение 10 соответствует команде, отличной от имеющей код 20. Оптимизация, убирающая первое присваивание, могла бы изменить последовательность команд, отправляемых радиопередатчику.

Квалификатор `volatile` представляет собой способ сообщить компиляторам, что мы имеем дело с такой особой памятью. Для компилятора это означает “не выполнять никаких оптимизаций над операциями с этой памятью”. Так что если переменная x соответствует особой памяти, она должна быть объявлена как `volatile`:

```
volatile int x;
```

Рассмотрим влияние этого квалификатора на последовательность нашего исходного кода:

```
auto y = x; // Чтение x  
y = x; // Чтение x еще раз (не может быть устранено)  
x = 10; // Запись x (не может быть устранена)  
x = 20; // Запись x еще раз
```

Это именно то, чего мы хотим, когда x представляет собой отображение в память (или отображается в ячейке памяти, совместно используемой разными процессами и т.п.).

Вопрос на засыпку: какой тип у в последнем фрагменте кода: `int` или `volatile int`?

<sup>8</sup> Тип у получается с помощью вывода `auto`, так что используются правила, описанные в разделе 1.2. Эти правила предписывают, чтобы для объявления типов, не являющихся ссылочными или типами указателей (что и выполняется в случае у), квалификаторы `const` и `volatile` были опущены. Следовательно, типом у является просто `int`. Это означает, что избыточные чтения и записи у могут быть удалены. В приведенном примере компиляторы должны выполнять инициализацию, и присваивание у, поскольку x объявлена как `volatile`, так что второе чтение x может давать другое значение, отличное от первого.

Тот факт, что кажущиеся избыточными загрузки и бессмысленные сохранения должны оставаться на месте при работе с особой памятью, объясняет, кстати, почему для такого рода работы не подходят объекты `std::atomic`. Компиляторам разрешается устранять такие избыточные операции у `std::atomic`. Код написан не в точности так же, как и для `volatile`, но если мы на минуту отвлечемся от этого и сосредоточимся на том, что компиляторам разрешается делать, то можно сказать, что концептуально компиляторы могут, получив код

```
std::atomic<int> x;
auto y = x; // Концептуально читает x (см. ниже)
y = x;      // Концептуально читает x еще раз (см. ниже)
x = 10;     // Записывает x
x = 20;     // Записывает x еще раз
```

оптимизировать его до

```
auto y = x; // Концептуально читает x (см. ниже)
x = 20;     // Записывает x
```

Очевидно, что это неприемлемое поведение при работе с особой памятью.

Но если `x` имеет тип `std::atomic`, ни одна из этих инструкций компилироваться не будет:

```
auto y = x; // Ошибка!
y = x;      // Ошибка!
```

Дело в том, что копирующие операции в `std::atomic` удалены (см. раздел 3.5). И не зря. Рассмотрим, что произошло бы, если бы инициализация `y` значением `x` компилировалась. Поскольку `x` имеет тип `std::atomic`, тип `y` был бы также выведен как `std::atomic` (см. раздел 1.2). Ранее я отмечал, что одна из лучших возможностей `std::atomic` заключается в атомарности всех их операций, но чтобы копирующеее конструирование `y` из `x` было атомарным, компиляторы должны генерировать код для чтения `x` и записи `y` как единую атомарную операцию. В общем случае аппаратное обеспечение не в состоянии это сделать, так что копирующеее конструирование типами `std::atomic` не поддерживается. Копирующеее присваивание удалено по той же причине, а потому присваивание `x` переменной `y` также не компилируется. (Перемещающие операции не объявлены в `std::atomic` явно, так что в соответствии с правилами генерации специальных функций, описанных в разделе 3.11, `std::atomic` не предоставляет ни перемещающего конструирования, ни перемещающего присваивания.)

Можно получить значение `x` в переменную `y`, но это требует использования функций-членов `load` и `store` типа `std::atomic`. Функция-член `load` атомарно считывает значение `std::atomic`, а функция-член `store` атомарно его записывает. Для инициализации `y` значением `x`, после которой выполняется размещение значения `x` в `y`, код должен иметь следующий вид:

```
std::atomic<int> y(x.load()); // Чтение x  
y.store(x.load()); // Чтение x еще раз
```

Этот код компилируется, но тот факт, что чтение x (с помощью x.load()) является отдельным от инициализации или сохранения значения у вызовом функции, делает очевидным то, что нет никаких оснований ожидать, что целая инструкция будет выполнятьсь как единая атомарная операция.

Компиляторы могут “оптимизировать” приведенный код, сохраняя значение x в регистре вместо двойного его чтения:

```
register = x.load(); // Чтение x в регистр  
  
std::atomic<int> y(register); // Инициализация у  
// значением регистра  
y.store(register); // Сохранение значения  
// регистра в у
```

В результате, как можно видеть, чтение из x выполняется только один раз, и этой разновидности оптимизации следует избегать при работе с особой памятью. (Эта оптимизация не разрешена при работе с переменными volatile.)

Таким образом, ситуация должна быть очевидна.

- std::atomic применяется в параллельном программировании, но не для доступа к особой памяти.
- volatile применяется для доступа к особой памяти, но не в параллельном программировании.

Поскольку std::atomic и volatile служат разным целям, они могут использоваться совместно:

```
volatile std::atomic<int> vai; // Операции над vai атомарны  
// и не могут быть удалены  
// при оптимизации
```

Этот может оказаться полезным, когда vai соответствует ячейке отображаемого на память ввода-вывода, обращение к которой выполняется несколькими потоками.

Последнее примечание: некоторые разработчики предпочитают использовать функции-члены load и store типа std::atomic даже там, где это не требуется, поскольку это четко указывает в исходном тексте на то, что данные переменные не являются “обычными”. Подчеркивание этого факта не является необоснованным. Доступ к std::atomic обычно гораздо медленнее, чем к переменным, не являющимся std::atomic, и мы уже видели, что использование std::atomic предотвращает определенное переупорядочение кода, которое иначе было бы разрешено. Вызовы загрузок и сохранений std::atomic могут тем самым помочь в идентификации потенциальных узких мест масштабируемости. С точки зрения корректности *отсутствие вызова store у переменной, предназначеннной для передачи информации в другие потоки* (например, флаг, указывающий доступность

данных), может означать, что эта переменная не объявлена как `std::atomic`, хотя должна быть таковой.

Однако в большей степени это вопрос стиля, и как таковой он не имеет отношения к выбору между `std::atomic` и `volatile`.

### Следует запомнить

- `std::atomic` применяется для обращения нескольких потоков к данным без использования мьютексов. Это инструмент параллельного программирования.
- `volatile` применяется для памяти, чтения и записи которой не должны удаляться при оптимизации. Это инструмент для работы с особой памятью.

# Тонкости

Для каждого общего метода или возможности C++ имеются условия, когда его применение является разумным, и обстоятельства, в которых его не следует применять. Обычно описание того, когда имеет смысл применение некоторого общего метода или возможности, достаточно простое, но в данной главе описываются два исключения из этого правила: общий метод (передача по значению) и общая возможность (размещение (*emplacement*)). Решение об их применении зависит от такого большого количества факторов, что лучший совет, какой я могу дать, сводится к *рассмотрению* возможности их применения. Тем не менее оба они являются важными составляющими эффективного современного программирования на C++, и приведенная в разделах этой главы информация необходима для принятия вами решения об их применении в своих программах.

## 8.1. Рассмотрите передачу по значению для копируемых параметров, которые легко перемещаются и всегда копируются

Некоторые параметры функций предназначаются для копирования<sup>1</sup>. Например, функция-член `addName` может копировать свой параметр в закрытый контейнер. Для эффективности такая функция должна копировать аргументы, являющиеся `lvalue`, но перемещать аргументы, являющиеся `rvalue`:

```
class Widget {  
public:  
    void addName(const std::string& newName) // lvalue;  
    { names.push_back(newName); } // копируем  
  
    void addName(std::string&& newName) // rvalue;  
    { names.push_back(std::move(newName)); } // перемещаем  
    // См. применение std::move в разделе 5.3
```

<sup>1</sup> В данном разделе “копирование” параметра в общем случае означает его использование как источника для операций копирования или перемещения. Помните, что в C++ нет терминологического различия между копированием, выполняемым с помощью операции копирования, и копированием с помощью операции перемещения.

```
...
private:
    std::vector<std::string> names;
};
```

Этот способ работает, но требует двух функций, выполняющих, по сути, одни и те же действия. Это несколько раздражает: надо объявить две функции, реализовать две функции, документировать две функции и наконец поддерживать также две функции! Тьфу...

Кроме того, две функции будут и в объектном коде, что при определенных обстоятельствах тоже может напрягать. В данном конкретном случае обе функции, вероятно, будут встраиваемыми, так что этот вопрос не встанет, но если функции не встраиваемые, то в объектном коде будут они обе.

Альтернативный подход заключается в том, чтобы сделать addName шаблоном функции, получающей универсальную ссылку (см. раздел 5.2):

```
class Widget {
public:
    template<typename T>          // Получаем как lvalue,
    void addName(T&& newName) // так и rvalue; lvalue
    {                           // копируем, rvalue перемещаем
        names.push_back(std::forward<T>(newName));
    } // Применение std::forward описано в разделе 5.3
};
```

Это приводит к уменьшению количества исходного текста, с которым приходится работать, но применение универсальных ссылок влечет за собой другие сложности. Будучи шаблоном, реализация addName обычно должна располагаться в заголовочном файле. В объектном коде такой подход может дать несколько функций, так как инстанцирование будет выполняться по-разному не только для lvalue и rvalue, но и для std::string и типов, преобразуемых в std::string (см. раздел 5.3). В то же время имеются типы аргументов, которые не могут быть переданы с помощью универсальной ссылки (см. раздел 5.8), и если клиент передаст аргументы некорректного типа, сообщения компилятора об ошибках могут быть приводящими в трепет (см. раздел 5.5).

Было бы неплохо, если бы имелся способ написания функций наподобие addName, таких, чтобы lvalue копировались, rvalue перемещались, при этом (в исходном тексте и объектном коде) имелась бы только одна функция и при этом можно было избежать неприятностей, связанных с универсальными ссылками. И такой способ есть. Все, что от вас требуется, — забыть об одном из первых правил, с которыми вы познакомились как программист на C++. Это правило, гласящее, что следует избегать передачи пользовательских типов по значению. Для параметров наподобие newName в функциях наподобие addName передача по значению может быть вполне разумной стратегией.

Перед тем как начать выяснение, почему передача по значению может быть хорошим решением для newName и addName, посмотрим, как она может быть реализована:

```

class Widget {
public:
    void addName(std::string newName)           // lvalue или
        { names.push_back(std::move(newName)); } // rvalue;
                                                // перемещаем его
};

```

Единственной неочевидной частью этого кода является применение `std::move` к параметру `newName`. Обычно `std::move` используется с `rvalue`-ссылками, но в данном случае мы знаем, что (1) `newName` представляет собой объект, полностью независимый от того, что передает вызывающая функция, так что изменение `newName` не влияет на вызывающую функцию, и (2) это последнее применение `newName`, так что его перемещение никак не влияет на остальную часть функции.

Тот факт, что существует только одна функция `addName`, поясняет, как мы избегаем дублирования кода — как исходного, так и объектного. Мы не используем универсальную ссылку, так что данный подход не ведет к увеличению заголовочных файлов, странным неприятностям или непонятным сообщениям об ошибках. Но что можно сказать об эффективности такого дизайна? Мы же выполняем передачу *по значению*. Не слишком ли она дорога?

В C++98 можно держать pari, что так и есть. Независимо от того, что передает вызывающая функция, параметр `newName` создается с помощью *копирующего конструктора*. Однако в C++11 `newName` будет создаваться с помощью копирующего конструирования только для `lvalue`. В случае `rvalue` этот объект создается с помощью *перемещающего конструктора*. Вот, взгляните:

```

Widget w;
...
std::string name("Bart");
w.addName(name);           // Вызов addName с lvalue
...
w.addName(name + "Jenne"); // Вызов addName с rvalue
                           // (см. ниже)

```

В первом вызове `addName` (при передаче `name`) параметр `newName` инициализируется значением `lvalue`. Поэтому объект `newName` создается путем копирования, так же, как это было бы в C++98. Во втором вызове `newName` инициализируется объектом `std::string`, полученным в результате вызова оператора `operator+` для `std::string` (т.е. выполнения операции добавления). Этот объект представляет собой `rvalue`, и `newName`, таким образом, создается перемещением.

Итак, `lvalue` копируются, а `rvalue` перемещаются, как мы и хотели. Здорово, правда?

Здорово, но есть несколько моментов, которые следует иметь в виду. Для облегчения понимания вспомним три рассмотренные версии функции `addName`:

```

class Widget { // Подход 1: перегрузка для lvalue и rvalue
public:
    void addName(const std::string& newName)

```

```

{ names.push_back(newName); }
void addName(std::string&& newName)
{ names.push_back(std::move(newName)); }

...
private:
    std::vector<std::string> names;
};

class Widget { // Подход 2: применение универсальной ссылки
public:
    template<typename T>
    void addName(T&& newName)
    { names.push_back(std::forward<T>(newName)); }

};

class Widget { // Подход 3: передача по значению
public:
    void addName(std::string newName)
    { names.push_back(std::move(newName)); }

};

```

Я буду говорить о первых двух версиях как о “подходе с передачей ссылки”, поскольку они обе передают параметры по ссылке.

Вот два сценария вызова, которые мы рассмотрели:

```

Widget w;
...
std::string name("Bart");
w.addName(name);           // Передача lvalue
...
w.addName(name + "Jenne"); // Передача rvalue

```

Давайте теперь рассмотрим стоимость (в операциях копирования и перемещения) добавления имени в Widget для приведенных сценариев и каждой из трех рассмотренных реализаций addName. Мы будем игнорировать оптимизирующие возможности компиляторов по удалению копирований и перемещений, поскольку такая оптимизация зависит от контекста и компилятора и с практической точки зрения на суть анализа не влияет.

- **Перегрузка.** Независимо от передачи lvalue или rvalue аргумент вызывающей функции связан со ссылкой по имени newName. Это ничего не стоит в смысле операций копирования и перемещения. В перегрузке для lvalue newName копируется в Widget::names. В перегрузке для rvalue объект перемещается. Итоговая стоимость: одно копирование для lvalue, одно перемещение для rvalue.
- **Применение универсальной ссылки.** Как и в случае перегрузки, аргумент вызывающей функции связан со ссылкой newName. Эта операция бесплатна.

Благодаря использованию `std::forward` lvalue-аргументы `std::string` копируются в `Widget::names`, в то время как rvalue-аргументы `std::string` перемещаются. Итоговая стоимость для аргументов `std::string` такая же, как и при перегрузке: одно копирование для lvalue, одно перемещение для rvalue.

В разделе 5.3 поясняется, что если вызывающая функция передает аргумент, отличный от `std::string`, он будет передан в конструктор `std::string`, и это может привести к нулевому количеству копирований и перемещений `std::string`. Таким образом, функции, получающие универсальные ссылки, оказываются уникально эффективными. Однако это не влияет на выполняемый нами анализ, так что для простоты мы будем предполагать, что вызывающая функция всегда передает аргументы `std::string`.

- **Передача по значению.** Независимо от передачи lvalue или rvalue должен быть сконструирован параметр `newName`. Если передано lvalue, это стоит одно копирование, если передано rvalue — одно перемещение. В теле функции `newName` безусловно перемещается в `Widget::names`. Итоговая стоимость, таким образом, равна одному копированию и одному перемещению для lvalue и двум перемещениям для rvalue. По сравнению с подходом с передачей ссылки мы получаем одно лишнее перемещение как для lvalue, так и для rvalue.

Взглянем еще раз на название раздела:

Рассмотрите передачу по значению для копируемых параметров, которые легко перемещаются и всегда копируются.

Оно сформулировано таким образом не без причины. Точнее, не без четырех причин.

1. Вы должны всего лишь *рассмотреть* использование передачи по значению. Да, при этом требуется написать всего лишь одну функцию. Да, при этом в объектном коде генерируется только одна функция. Да, вы избегаете проблем, связанных с применением универсальных ссылок. Но стоимость этого решения выше, чем стоимость альтернативных вариантов, и, как вы увидите далее, в некоторых случаях есть стоимость, которую мы еще не рассматривали.
2. Рассмотрите передачу по значению только для *копируемых параметров*. Параметры, не соответствующие этому условию, должны иметь типы, являющиеся только перемещаемыми, поскольку если они не копируемые, а функция всегда делает копию, такая копия должна создаваться с помощью перемещающего конструктора<sup>2</sup>. Вспомним, что преимущества передачи по значению перед перегрузкой заключаются в том, что при передаче по значению достаточно написать только одну функцию. Но для только перемещаемых типов нет необходимости предоставлять перегрузку для lvalue, поскольку копирование lvalue влечет вызов копирующего конструктора, который у таких типов отсутствует. Это означает, что требуется

<sup>2</sup> Для таких предложений хорошо бы иметь терминологию, отличающую копирование с помощью копирующего конструктора от копирования с помощью перемещающего конструктора.

поддержка только аргументов, являющихся rvalue, и в таком случае решение на основе “перегрузки” требует только одну перегрузку, принимающую rvalue-ссылку.

Рассмотрим класс с данными-членом `std::unique_ptr<std::string>` и функцию установки для него. Тип `std::unique_ptr` является только перемещаемым типом, так что подход с использованием “перегрузки” состоит из единственной функции.

```
class Widget {
public:
    ...
    void setPtr(std::unique_ptr<std::string>&& ptr)
    { p = std::move(ptr); }
private:
    std::unique_ptr<std::string> p;
};
```

Вызывающая функция может использовать ее следующим образом:

```
Widget w;
...
w.setPtr(std::make_unique<std::string>("Modern C++"));
```

Здесь rvalue `std::unique_ptr<std::string>`, возвращаемое из `std::make_unique` (см. раздел 4.4), передается по ссылке в `setPtr`, где оно перемещается в данные-член `p`. Общая стоимость составляет одно перемещение.

Если бы `setPtr` принимала параметры по значению,

```
class Widget {
public:
    ...
    void setPtr(std::unique_ptr<std::string> ptr)
    { p = std::move(ptr); }
    ...
};
```

то тот же вызов создавал бы параметр `ptr` перемещением, а затем `ptr` был бы перемещен в данные-член `p`. Общая стоимость составила бы два перемещения — в два раза больше, чем при подходе с “перегрузкой”.

3. Передачу по значению стоит рассматривать только для параметров с недорогим перемещением. Когда перемещение дешевое, стоимость дополнительного перемещения может быть приемлемой, но если это не так, то излишнее перемещение становится аналогичным излишнему копированию, а важность устранения излишнего копирования и была основной причиной появления правила C++98 о нежелательности применения передачи по значению.
4. Вы должны рассматривать передачу по значению только для параметров, которые всегда копируются. Чтобы увидеть, почему это важно, предположим, что перед копированием параметра в контейнер паттерн функция `addName` проверяет, не слишком ли короткое (или длинное) имя передано. Если это так, запрос на добавление

имени игнорируется. Реализация с передачей по значению может быть написана следующим образом:

```
class Widget {  
public:  
    void addName(std::string newName)  
    {  
        if ((newName.length() >= minLen) &&  
            (newName.length() <= maxLen))  
        {  
            names.push_back(std::move(newName));  
        }  
    }  
    ...  
private:  
    std::vector<std::string> names;  
};
```

Эта функция берет на себя создание и уничтожение newName, даже если в names ничего не добавляется. Это цена, которую платить с передачей по ссылке платить не должен.

Даже когда вы имеете дело с функцией, выполняющей безусловное копирование копируемого типа с дешевым перемещением, бывают моменты, когда передача по значению может оказаться неприемлемой. Это связано с тем, что функция может копировать параметр двумя способами: с помощью конструирования (т.е. с помощью копирующего конструктора или перемещающего конструктора) и с помощью присваивания (т.е. с помощью оператора копирующего или перемещающего присваивания). Функция addName использует конструирование: ее параметр newName передается функции vector::push\_back, и внутри этой функции newName конструируется копированием в новом элементе в конце вектора std::vector. Для функций, которые используют конструирование для копирования своего параметра, анализ, который мы видели ранее, завершен: использование передачи по значению приводит к дополнительному перемещению как для lvalue-аргументов, так и для rvalue-аргументов.

Когда параметр копируется с использованием присваивания, ситуация становится более сложной. Предположим, например, что у нас есть класс, представляющий пароли. Поскольку пароль может изменяться, мы предоставляем функцию установки changeTo. Используя стратегию передачи по значению, реализовать Password можно следующим образом:

```
class Password {  
public:  
    explicit Password(std::string pwd) // Передача по значению;  
    : text(std::move(pwd)) {}           // конструирование text  
  
    void changeTo(std::string newPwd) // Передача по значению;
```

```
{ text = std::move(newPwd); } // присваивание text  
...  
private:  
    std::string text; // Текст пароля  
};
```

Хранение паролей в виде обычного текста приведет специалистов по безопасности в не-  
истовство, но мы их проигнорируем и рассмотрим следующий код:

```
std::string initPwd("Supercalifragilisticexpialidocious");  
Password p(initPwd);
```

Здесь нет никаких сюрпризов: `p.text` конструируется с использованием заданного паро-  
ля, а применение передачи по значению в конструкторе приводит к стоимости переме-  
щающего конструирования `std::string`, которое может оказаться излишним при при-  
менении перегрузки или прямой передачи. Все в порядке.

Пользователь этой программы может быть не столь оптимистичным насчет пароля,  
так как слово “Supercalifragilisticexpialidocious” можно найти в словаре. А потому он пред-  
принимает действия, которые ведут к выполнению следующего кода:

```
std::string newPassword = "Beware the Jabberwock";  
p.changeTo(newPassword);
```

Лучше новый пароль старого или нет — вопрос сложный, но это проблемы пользователя.  
Нашей же проблемой является то, что необходимость функции `changeTo` использовать  
присваивание (а не конструирование) для копирования параметра `newPwd`, вероятно,  
приведет к росту стоимости стратегии передачи параметра по значению.

Аргумент, переданный функции `changeTo`, представляет собой `lvalue` (`newPassword`),  
так что, когда конструируется параметр `newPwd`, вызывается копирующий конструктор  
`std::string`. Этот конструктор выделяет память для хранения нового пароля. Затем  
`newPwd` присваивается с перемещением переменной `text`, что приводит к освобождению  
памяти, которая ранее принадлежала этой переменной `text`. Таким образом, в `changeTo`  
выполняются два действия по управлению динамической памятью: одно выделяет па-  
мять для хранения нового пароля, а второе освобождает память, в которой хранился ста-  
рый пароль.

Но в данном случае старый пароль (“Supercalifragilisticexpialidocious”) длиннее нового  
 (“Beware the Jabberwock”), так что нет необходимости в выделении и освобождении па-  
мяти вовсе. Если бы использовался подход с перегрузкой, вероятно, никакие выделения  
и освобождения не выполнялись бы:

```
class Password {  
public:  
    ...  
    void changeTo(const std::string& newPwd) // Перегрузка для  
    {                                         // lvalue  
        text = newPwd; // При text.capacity() >= newPwd.size()  
                      // можно использовать память text
```

```
    }  
    ...  
private:  
    std::string text; // Как выше  
};
```

В этом сценарии стоимость передачи по значению включает дополнительное выделение и освобождение памяти — стоимость, которая, скорее всего, превысит стоимость операции по перемещению `std::string` на порядки.

Интересно, что если старый пароль короче нового, то обычно невозможно избежать при присваивании действий по выделению и освобождению памяти, и в этом случае передача по значению будет выполняться практически с той же скоростью, что и передача по ссылке. Таким образом, стоимость копирования параметров с помощью присваивания может зависеть от объектов, участвующих в присваивании! Этот вид анализа применим к любому типу параметров, который хранит данные в динамически выделенной памяти. Не все типы таковы, но многие — включая `std::string` и `std::vector` — обладают этим свойством.

Это потенциальное увеличение стоимости в общем случае применимо только к передаче аргументов, являющихся `lvalue`, поскольку необходимость выделения и освобождения памяти обычно возникает только тогда, когда выполняется истинное копирование (не перемещение). В случае `rvalue`-аргументов перемещений почти всегда достаточно.

Получается, что дополнительная стоимость передачи по значению (по сравнению с передачей по ссылке) для функций, копирующих параметр с использованием присваивания, зависит от передаваемого типа, соотношения `lvalue`- и `rvalue`-аргументов и от того, использует ли тип динамическую память (и, если использует, то от реализации операторов присваивания для данного типа и вероятности того, что память, связанная с целевым объектом присваивания, как минимум того же размера, что и память присваиваемого объекта). В случае `std::string` она также зависит от того, использует ли реализация оптимизацию малых строк (SSO; см. раздел 5.7), и если использует, то помещаются ли присваиваемые значения в буфер SSO.

Так что, как я говорил, при копировании параметров с помощью присваивания анализ стоимости передачи по значению становится весьма сложным. Обычно наиболее практический подход состоит в стратегии презумпции виновности (“виновен, пока не доказано иное”), в соответствии с которым вы используете перегрузку или универсальную ссылку, а не передачу по значению, пока не будет показано, что передача по значению дает приемлемо эффективный код для используемого вами типа параметра.

Итак, для программного обеспечения, которое должно быть насколько это возможно быстрым, передача по значению может оказаться неподходящей стратегией, поскольку важным может быть даже устранение дешевых перемещений. Кроме того, не всегда понятно, сколько же перемещений имеют место на самом деле. В примере `Widget::addName` передача по значению приводит только к одной лишней операции перемещения; однако предположим, что `Widget::addName` вызывает `Widget::validateName`, в которую параметр также передается по значению. (Возможно, имеется причина для того, чтобы всегда

копировать параметр, например, для хранения его в структуре данных всех проверенных значений.) Предположим также, что функция `validateName` вызывает третью функцию с передачей ей параметра по значению...

Как видите, название раздела не зря такое неопределенное. Когда имеется цепочка вызовов функций, каждая из которых использует передачу по значению, поскольку “его стоимость составляет только одно недорогое перемещение”, стоимость всей цепочки может стать такой, что вы не сможете спокойно ее терпеть. При использовании передачи по ссылке вы избегаете такого накопления накладных расходов.

Есть еще одно соображение, не имеющее отношения к производительности, но которое стоит иметь в виду. Передача по значению, в отличие от передачи по ссылке, подвержена *проблеме срезки*. Эта проблема хорошо известна в C++98 и многократно проанализирована во множестве книг, так что я не буду подробно на ней останавливаться. Но если вам нужна функция, которая должна принимать параметры типа базового класса и любых производных от него типов, то вы не должны объявлять параметр этого типа как передаваемый по значению, так как при этом будут “срезаться” характеристики объекта производного класса, передаваемого функции:

```
class Widget { ... }; // Базовый класс
class SpecialWidget: public Widget { ... }; // Производный класс
void processWidget(Widget w); // Функция для любого вида Widget,
                             // включая производные типы;
...
SpecialWidget sw; // подвержена проблеме срезки
...
processWidget(sw); // processWidget видит Widget,
                   // а не SpecialWidget!
```

Если вы не знакомы с проблемой срезки, поищите информацию в Интернете или поинтересуйтесь у друзей; информации о ней предостаточно. Вы узнаете, что срезка — это еще одна причина ( помимо проблемы эффективности), по которой передача по значению имеет такую плохую репутацию в C++98. Как видите, имеются веские причины для того, чтобы вбивать в головы новичкам в программировании на C++: не передавайте объекты пользовательских типов по значению!

C++11 не отменяет мудрость C++98, касающуюся передачи по значению. В общем случае передача по значению по-прежнему влечет за собой снижение производительности, которого следует избегать, и по-прежнему может приводить к срезке. Новым в C++11 является различие аргументов, являющихся `lvalue` и `rvalue`. Реализация функций, которые используют преимущества семантики перемещения для `rvalue` копируемых типов, требует либо перегрузки, либо применения универсальных ссылок, и оба эти подхода имеют свои недостатки. В частном случае копируемых легко перемещаемых типов, передаваемых в функцию, которая всегда их копирует и где срезка не является проблемой, передача по значению может быть простой в реализации альтернативой, почти столь же эффективной, как и ее конкуренты с передачей по ссылке, но при этом не отягощенной их недостатками.

### Следует запомнить

- Для копируемых и легко перемещаемых параметров, которые всегда копируются, передача по значению может быть почти столь же эффективной, как и передача по ссылке, более простой в реализации и генерировать меньший объектный код.
- Для lvalue-аргументов передача по ссылке (например, копирующее конструирование), за которой следует перемещающее присваивание, может оказаться существенно более дорогостоящей, чем передача по ссылке с последующим копирующим присваиванием.
- Передача по значению подвержена проблеме срезки, так что обычно не годится для типов параметров базовых классов.

## 8.2. Рассмотрите применение размещения вместо вставки

Если у вас есть, скажем, контейнер, хранящий строки `std::string`, представляется логичным, что при добавлении нового элемента с помощью функции вставки (т.е. `insert`, `push_front`, `push_back` или для `std::forward_list` — `insert_after`) тип передаваемого функции элемента представляет собой `std::string`. В конце концов, именно этот тип хранится в контейнере.

Несмотря на всю логичность, это не всегда верно. Рассмотрим следующий код:

```
std::vector<std::string> vs; // Контейнер std::string
vs.push_back("xyzzy");      // Добавление строкового литерала
```

Здесь контейнер хранит строки `std::string`, но в действительности вы передаете в функцию `push_back` строковый литерал, т.е. последовательность символов в двойных кавычках. Строковый литерал не является `std::string`, и это означает, что переданный вами в функцию `push_back` аргумент имеет тип, отличный от типа элементов, хранящихся в контейнере.

Функция `push_back` класса `std::vector` перегружена для lvalue и rvalue следующим образом:

```
template <class T,                                     // Из стандарта
          class Allocator = allocator<T>> // C++11
class vector {
public:
    ...
    void push_back(const T& x);                  // Вставка lvalue
    void push_back(T&& x);                      // Вставка rvalue
};
```

В вызове

```
vs.push_back("xyzzy");
```

компиляторы видят несоответствие между типом аргумента (`const char[6]`) и типом параметра, получаемого функцией `push_back` (ссылка на `std::string`). Они разрешают это несоответствие путем генерации кода для создания временного объекта `std::string` из строкового литерала и передачи этого временного объекта функции `push_back`. Другими словами, они рассматривают вызов так, как будто он записан следующим образом:

```
vs.push_back(std::string("xyzzy")); // Создание временного
// объекта std::string и передача его функции push_back
```

Этот код компилируется и выполняется, и все расходятся счастливыми и довольными. Все, кроме свихнувшихся на производительности программистов, которые обнаруживают, что этот код не настолько эффективен, насколько должен быть.

Они понимают, что для создания нового элемента в контейнере, содержащем строки `std::string`, должен быть вызван конструктор `std::string`, но приведенный выше код делает не один вызов конструктора, а два, а также вызывает деструктор `std::string`. Вот что происходит во время выполнения вызова `push_back`.

1. Из строкового литерала "xyzzy" создается временный объект `std::string`. Этот объект не имеет имени; назовем его `temp`. Создание `temp` представляет собой первое конструирование `std::string`. Поскольку это временный объект, `temp` представляет собой `rvalue`.
2. Объект `temp` передается в `rvalue`-перегрузку `push_back`, где он связывается с параметром `x`, представляющим собой `rvalue`-ссылку. Затем в памяти `std::vector` создается копия `x`. Это *второе* конструирование действительно создает новый объект внутри `std::vector`. (Конструктор, использованный для копирования `x` в `std::vector`, представляет собой перемещающий конструктор, поскольку `x`, будучи `rvalue`-ссылкой, приводится к `rvalue` перед копированием. Информацию о приведении параметров, являющихся `rvalue`-ссылками, в `rvalue` можно найти в разделе 5.3.)
3. Непосредственно после возврата из `push_back` уничтожается объект `temp`; при этом вызывается деструктор `std::string`.

Фанаты производительности не в состоянии помочь, но замечают, что если бы был способ взять строковый литерал и непосредственно передать его в код шага 2, который конструирует объект `std::string` внутри `std::vector`, то можно было бы избежать конструирования и удаления `temp`. Это могло бы оказаться максимально эффективным подходом.

Поскольку вы программист на C++, шанс, что вы фанат производительности, явно выше среднего. Если вы не из таких, то, пожалуй, все равно им симпатизируете. (Если же производительность вас не интересует, может, вы просто ошиблись дверью? Python находится дальше по коридору...) Так что я рад сообщить вам, что есть способ сделать именно то, что требуется для достижения максимальной эффективности в вызове `push_back`. Это — не вызывать `push_back`. Функция `push_back` неправильная. Вам нужна функция `emplace_back`.

Функция `emplace_back` делает именно то, что мы хотим: использует переданный аргумент для конструирования `std::string` непосредственно внутри `std::vector`, не прибегая ни к каким временным объектам:

```
vs.emplace_back("xyzzy"); // Создает std::string в vs  
                         // непосредственно из "xyzzy"
```

`emplace_back` использует прямую передачу, так что до тех пор, пока вы не столкнетесь с одним из ограничений прямой передачи (раздел 5.8), можете передавать любое количество аргументов с любой комбинацией типов. Например, если вы хотите создать `std::string` в `vs` с помощью конструктора `std::string`, получающего символ и количество его повторений, то вы пишете следующий исходный текст:

```
vs.emplace_back(50, 'x'); // Вставка std::string из  
                         // 50 символов 'x'
```

Функция `emplace_back` доступна во всех стандартных контейнерах, которые поддерживают `push_back`. Аналогично каждый стандартный контейнер, который поддерживает `push_front`, поддерживает и `emplace_front`. И каждый стандартный контейнер, поддерживающий `insert` (т.е. все контейнеры, кроме `std::forward_list` и `std::array`), поддерживает `emplace`. Ассоциативные контейнеры предоставляют `emplace_hint` в качестве дополнения к функциям `insert`, которые получают итератор “подсказки”, а у `std::forward_list` имеется `emplace_after`, соответствующий его `insert_after`.

Что позволяет функциям размещения превзойти функции вставки, так это их более гибкий интерфейс. Функции вставки получают *вставляемые объекты*, в то время как функции размещения получают *аргументы конструктора вставляемых объектов*. Это отличие позволяет функциям размещения избегать создания и уничтожения временных объектов, которые могут требоваться функциям вставки.

Поскольку функции размещения может быть передан аргумент типа, хранимого в контейнере (аргумент, таким образом, заставляет функцию выполнить копирующее или перемещающее конструирование), размещение может использоваться даже тогда, когда функция вставки не требует временного объекта. В таком случае вставка и размещение делают, по сути, одно и то же. Например, для

```
std::string queenOfDisco("Donna Summer");
```

оба приведенных далее вызова корректны, и оба приводят к одному и тому же результату:

```
vs.push_back(queenOfDisco);      // Копирующее конструирование  
                                 // queenOfDisco в конце vs  
vs.emplace_back(queenOfDisco); // То же самое
```

Таким образом, размещающие функции могут делать все то же самое, что и функции вставки. Иногда они делают это более эффективно и как минимум теоретически не должны делать менее эффективно. Так почему же мы не используем их все время?

Потому что теоретически разницы между теорией и практикой нет, а практически — есть. При текущих реализациях стандартной библиотеки имеются ситуации, когда, как и ожидается, размещение превосходит вставку, но — увы! — есть ситуации, когда вставка работает быстрее. Такие ситуации непросто охарактеризовать, поскольку они зависят от типов передаваемых аргументов, используемых контейнеров, местоположения вставки или размещения в контейнере, безопасности исключений конструкторов типов, содержащихся в контейнере, и для контейнеров, в которых запрещены дубликаты (т.е. `std::set`, `std::map`, `std::unordered_set`, `std::unordered_map`), от того, содержится ли уже в контейнере вставляемое значение. А потому следует пользоваться обычным советом по повышению производительности: для определения того, какой метод работает быстрее, надо сравнивать их реальную производительность в конкретных условиях.

Это, конечно, не очень приятно, так что вы будете рады узнать, что есть эвристический алгоритм, который может помочь вам определить ситуации, когда, скорее всего, имеет смысл использовать функции размещения. Если все приведенные далее утверждения справедливы, размещение почти наверняка будет опережать вставку.

- **Добавляемое значение конструируется в контейнере, а не присваивается.** Пример, с которого начал данный раздел (добавление `std::string` со значением "xyzzy" в `std::vector` `vs`), демонстрирует значение, добавляемое в конец вектора `vs` — в место, где пока что нет никакого объекта. Таким образом, новое значение должно быть сконструировано в `std::vector`. Если мы пересмотрим пример так, что новая строка `std::string` будет направляться в местоположение, уже занятое объектом, это будет совсем другая история:

```
std::vector<std::string> vs;           // Как и ранее
...                                     // Добавление элементов в vs
vs.emplace(vs.begin(), "xyzzy"); // Добавление "xyzzy" в начало vs
```

При таком коде только редкие реализации будут конструировать добавляемый объект `std::string` в памяти, занятой `vs[0]`. Большинство реализаций используют перемещающее присваивание в указанное место. Но перемещающее присваивание требует наличия перемещаемого объекта, а это означает, что необходимо создание временного объекта. Поскольку основное преимущество размещения над вставкой заключается в том, что не создаются и не уничтожаются временные объекты, при добавлении значения в контейнер с помощью присваивания преимущества размещения исчезают.

Увы, выполняется ли добавление значения в контейнер путем конструирования или перемещения, в общем случае зависит от реализации. И вновь на помощь может прийти эвристика.

Контейнеры на основе узлов почти всегда используют для добавления новых значений конструирование, а большинство стандартных контейнеров являются именно таковыми. Исключениями являются `std::vector`, `std::deque` и `std::string`. (Контейнер `std::array` также не является таковым, но он не поддерживает ни вставку, ни размещение, поэтому упоминать о нем здесь нет смысла.) В контейнерах, не основанных на узлах, можно рассчитывать на использование

функцией `emplace_back` для размещения нового значения конструирования вместо присваивания; то же самое можно сказать и о функции `emplace_front` контейнера `std::deque`.

- **Типы передаваемых аргументов отличаются от типа, хранящегося в контейнере.** И вновь, преимущество размещения по сравнению со вставкой в общем случае связано с тем фактом, что его интерфейс не требует создания и уничтожения временного объекта при передаче аргументов типа, отличного от типа, хранящегося в контейнере. Когда в контейнер `container<T>` добавляется объект типа `T`, нет причин ожидать, что размещение окажется быстрее вставки, поскольку для удовлетворения интерфейса вставки не требуется создание временного объекта.
- **Маловероятно, что контейнер отвергнет новое значение как дубликат.** Это означает, что либо контейнер разрешает наличие дубликатов, либо большинство передаваемых значений уникальны. Это важно, поскольку для того, чтобы определить наличие дубликата, реализации размещения обычно создают узел с новым значением, а затем сравнивают его с имеющимися узлами контейнера. Если добавляемое значение в контейнере отсутствует, узел встраивается в контейнер. Однако, если такое значение уже есть в контейнере, размещение прерывается, а узел уничтожается, так что впустую расходуется стоимость создания и уничтожения объекта. Такие узлы для функций размещения создаются более часто, чем для функций вставки.

Приведенные вызовы, с которыми мы уже сталкивались в данном разделе, удовлетвряют всем перечисленным критериям. Они работают быстрее соответствующих вызовов `push_back`.

```
vs.emplace_back("xyzzy"); // Конструирует новое значение в конце
                           // контейнера; тип аргумента отличен от
                           // типа, хранимого в контейнере;
                           // контейнер не отвергает дубликаты
vs.emplace_back(50, 'x'); // То же самое
```

При принятии решения об использовании функций размещения стоит иметь в виду еще пару вопросов. Первый из них связан с управлением ресурсами. Предположим, что у вас есть контейнер с объектами `std::shared_ptr<Widget>`

```
std::list<std::shared_ptr<Widget>> ptrs;
```

и вы хотите добавить `std::shared_ptr`, который должен быть освобожден с помощью пользовательского удалителя (см. раздел 4.2). В разделе 4.4 поясняется, что по возможности вы должны использовать для создания `std::shared_ptr` функцию `std::make_shared`, но в нем же указано, что существуют ситуации, когда это невозможно. Одна из таких ситуаций — когда вы хотите указать пользовательский удалитель. В этом случае для получения обычного указателя, которым будет управлять интеллектуальный указатель `std::shared_ptr`, вы должны непосредственно использовать оператор `new`.

Если пользовательский удалитель представляет собой функцию

```
void killWidget(Widget* pWidget);
```

то код, использующий функцию вставки, может выглядеть следующим образом:

```
ptrs.push_back(std::shared_ptr<Widget>(new Widget, killWidget));
```

Он может также принять и такой вид, означающий то же самое:

```
ptrs.push_back({ new Widget, killWidget });
```

В любом случае перед вызовом `push_back` будет создан временный объект `std::shared_ptr`. Параметром `push_back` является ссылка на `std::shared_ptr`, так что должен быть объект, на который ссылается этот параметр.

Функция `emplace_back` должна избегать создания временного объекта `std::shared_ptr`, но в этом случае ценность временного объекта гораздо выше его стоимости. Рассмотрим следующую потенциальную последовательность событий.

1. В любом из приведенных выше вызовов конструируется временный объект `std::shared_ptr<Widget>`, хранящий простой указатель, являющийся результатом операции “`new Widget`”. Назовем этот объект `temp`.
2. Функция `push_back` получает `temp` по ссылке. В процессе выделения памяти для узла списка, который должен содержать копию `temp`, генерируется исключение нехватки памяти.
3. При выходе исключения за пределы `push_back` объект `temp` уничтожается. Поскольку это единственный интеллектуальный указатель `std::shared_ptr`, указывающий на управляемый им объект `Widget`, он автоматически удаляет последний, в данном случае с помощью вызова `killWidget`.

Несмотря на происшедшую генерацию исключения нет никаких утечек: `Widget`, созданный с помощью “`new Widget`” в вызове `push_back`, освобождается деструктором объекта `std::shared_ptr`, который был создан для управления им (объектом `temp`). Все отлично.

Рассмотрим теперь, что произойдет при вызове `emplace_back` вместо `push_back`:

```
ptrs.emplace_back(new Widget, killWidget);
```

1. Обычный указатель, являющийся результатом выполнения “`new Widget`”, передается с помощью прямой передачи в точку внутри `emplace_back`, где выделяется память для узла списка. При попытке выделения памяти генерируется исключение нехватки памяти.
2. При выходе исключения за пределы `emplace_back` обычный указатель, который был единственным средством доступа к `Widget` в динамической памяти, оказывается потерянным. Происходит утечка `Widget` (и всех ресурсов, которыми владеет этот объект).

В этом сценарии все совсем *не* отлично, и класс `std::shared_ptr` в этом не повинен. Та же самая проблема возникнет при использовании `std::unique_ptr` с пользовательским удалителем. По существу, эффективность классов управления ресурсами,

таких как `std::shared_ptr` и `std::unique_ptr`, основана на *немедленной* передаче ресурсов (таких, как обычные указатели, возвращаемые оператором `new`) конструкторам управляющих ресурсами объектов. Тот факт, что функции `std::make_shared` и `std::make_unique` автоматизируют этот процесс, является одной из причин, по которым эти функции так важны.

В вызовах функций вставки контейнеров, хранящих управляющие ресурсами объекты (например, `std::list<std::shared_ptr<Widget>>`), типы параметров функций в общем случае гарантируют, что между захватом ресурса (например, использованием оператора `new`) и конструированием управляющего ресурсом объекта ничего не происходит. В функциях размещения прямая передача откладывает создание управляющих ресурсами объектов до тех пор, пока они не смогут быть сконструированы в памяти контейнера, и тем самым открывают окно, генерация исключений в котором может привести к утечке ресурсов. Все стандартные контейнеры подвержены этой проблеме. При работе с контейнерами, хранящими управляющие ресурсами объекты, вы должны принять меры, гарантирующие, что при выборе функции размещения вместо функции вставки вы не заплатите за повышение эффективности безопасностью исключений.

Откровенно говоря, вы в любом случае не должны передавать выражения наподобие `"new Widget"` в функции `emplace_back` и `push_back`, как и в большинство любых других функций, поскольку, как поясняется в разделе 4.4, это ведет к возможным проблемам с безопасностью исключений, одну из которых мы только что рассмотрели. Для предотвращения неприятностей требуется, чтобы получение указателя от `"new Widget"` и преобразование его в управляющий ресурсом объект выполнялось в одной инструкции, а уже затем этот объект передавался как `rvalue` функции, которой вы хотели изначально передавать `"new Widget"`. (Более детально этот подход рассматривается в разделе 4.4.) Таким образом, код, использующий `push_back`, должен быть записан скорее как

```
std::shared_ptr<Widget> spw(new Widget, // Создание Widget и
                           killWidget); // передача его spw
ptrs.push_back(std::move(spw));          // Добавление spw
                                         // как rvalue
```

Версия с использованием `emplace_back` аналогична:

```
std::shared_ptr<Widget> spw(new Widget, killWidget);
ptrs.emplace_back(std::move(spw));
```

В любом случае данный подход включает стоимость создания и удаления `spw`. С учетом того, что мотивацией для применения функций размещения вместо функций вставки является устранение стоимости создания и уничтожения временного объекта типа, хранящегося в контейнере (а `spw` концептуально и является таким объектом), функции размещения вряд ли превзойдут функции вставки при добавлении в контейнер объектов, управляющих ресурсами (если вы будете следовать хорошо проверенной практике и гарантировать, что между захватом ресурса и превращением его в управляющий объект не будет выполняться никаких действий).

Вторым важным аспектом функций размещения является их взаимодействие с конструкторами, объявленными как `explicit`. Предположим, что вы создаете контейнер объектов регулярных выражений C++11:

```
std::vector<std::regex> regexes;
```

Отвлекшись на скору ваших коллег о том, как часто следует проверять свой Facebook, вы случайно написали следующий, казалось бы, бессмысленный код:

```
regexes.emplace_back(nullptr);
```

Вы не заметили ошибку при вводе, компилятор скомпилировал его без замечаний, так что вам пришлось потратить немало времени на отладку. В какой-то момент вы обнаружили, что вставляете в контейнер регулярных выражений нулевой указатель. Но как это возможно? Указатели не являются регулярными выражениями, и если вы попытаетесь написать что-то вроде

```
std::regex r = nullptr; // Ошибка! Не компилируется!
```

компилятор отвергнет такой код. Интересно, что будет отвергнут и вызов `push_back` вместо `emplace_back`:

```
regexes.push_back(nullptr); // Ошибка! Не компилируется!
```

Такое любопытное поведение поясняется тем, что объекты `std::regex` могут быть построены из символьных строк. Это делает корректным такой полезный код, как

```
std::regex upperCaseWord("[A-Z]+");
```

Создание `std::regex` из символьной строки может иметь достаточно высокую стоимость, так что для минимизации вероятности непреднамеренных расходов конструктор `std::regex`, принимающий указатель `const char*`, объявлен как `explicit`. Именно поэтому не компилируются следующие строки:

```
std::regex r = nullptr; // Ошибка! Не компилируется!
```

```
regexes.push_back(nullptr); // Ошибка! Не компилируется!
```

В обоих случаях требуется неявное преобразование указателя в `std::regex`, а объявление конструктора как `explicit` его предотвращает.

Однако в вызове `emplace_back` мы передаем не объект `std::regex`, а аргументы конструктора объекта `std::regex`. Это не рассматривается как запрос неявного преобразования. Компилятор трактует этот код так, как если бы вы написали

```
std::regex r(nullptr); // Компилируется
```

Если лаконичный комментарий “Компилируется” кажется вам лишенным энтузиазма, то это хорошо, потому что, несмотря на компилируемость, данный код имеет неопределенное поведение. Конструктор `std::regex`, принимающий указатель `const char*`, требует, чтобы этот указатель был ненулевым, а `nullptr` подчеркнуто нарушает данное требование. Если вы напишете и скомпилируете такой код, лучшее, на что вы можете

надеяться, — аварийное завершение программы во время выполнения. Если вы не такой счастливчик, то вам предстоит получить немалый опыт работы с отладчиком.

На минутку оставляя без внимания `push_back` и `emplace_back`, обратим внимание на то, что очень похожие синтаксисы инициализации дают совершенно разные результаты:

```
std::regex r1 = nullptr; // Ошибка! Не компилируется  
std::regex r2(nullptr); // Компилируется
```

В официальной терминологии стандарта синтаксис, использованный для инициализации `r1` (со знаком равенства), соответствует *инициализации копированием* (*copy initialization*). Синтаксис же, использованный для инициализации `r2` (с круглыми скобками, хотя могут использоваться и фигурные), дает то, что называется *прямой инициализацией* (*direct initialization*). Инициализация копированием не может использовать конструкторы, объявленные как `explicit`, в то время как прямая инициализация — может. Вот почему строка с инициализацией `r1` не компилируется, в отличие от строки с инициализацией `r2`.

Но вернемся к `push_back` и `emplace_back` и в более общем случае — к функциям вставки и размещения. Функции размещения используют прямую инициализацию, т.е. могут пользоваться конструкторами, объявленными как `explicit`. Функции вставки применяют инициализацию копированием, а потому использовать такие конструкторы не могут.

```
regexes.emplace_back(nullptr); // Компилируется. Прямая  
// инициализация разрешает использовать конструктор  
// explicit std::regex, получающий указатель  
regexes.push_back(nullptr); // Ошибка! Копирующая  
// инициализация такие конструкторы не использует
```

Урок, который следует извлечь из данного материала, состоит в том, что при использовании размещающих функций необходимо быть особенно осторожным и убедиться, что функциям передаются правильные аргументы, поскольку в ходе анализа кода будут рассмотрены даже конструкторы, объявленные как `explicit`.

### Следует запомнить

- В принципе, функции размещения должны быть более эффективными, чем соответствующие функции вставки, и не должны быть менее эффективными.
- На практике они чаще всего более быстрые, когда (1) добавляемое значение конструируется в контейнере, а не присваивается; (2) типы передаваемых аргументов отличаются от типа, хранящегося в контейнере; и (3) контейнер не отвергает дубликаты уже содержащихся в нем значений.
- Функции размещения могут выполнять преобразования типов, отвергаемые функциями вставки.



# Предметный указатель

## Символы

= default, 120

= delete, 85

## A

array<>, 209

async<>() , 245; 247

стратегии запуска, 249

atomic<>, 113; 272

auto, 23; 49

auto\_ptr<>, 126

## B

bind<>, 232; 237

## C

const, 106

constexpr, 105

const\_iterator, 95

## D

decltype, 23; 36; 235

## E

enable\_if<>, 195

enum, 78

explicit, 298

## F

false\_type, 193

final, 92

forward<>() , 169; 204

function<>, 51

## I

initializer\_list<>, 34; 63

is\_base\_of<>, 198

is\_constructible<>, 201

## L

lvalue, 16

## M

make\_shared(), 146

make\_unique(), 146

move(), 166

mutable, 112

## N

поexcept, 99

условный, 102

nullptr, 69

## O

override, 91

## P

packaged\_task<>, 263

## R

rvalue, 16

## S

shared\_ptr<>, 133

## T

thread<>() , 245

thread\_local, 251

true\_type, 193

typedef, 73

typename, 75

## U

unique\_ptr<>, 126

using, 74

## V

vector<>

bool, 55

ошибка дизайна, 68

volatile, 272; 277

## W

weak\_ptr<>, 142

## Б

Большая тройка, 119

## В

Вывод типа, 53

auto, 31

decltype, 38

шаблона, 23

## Г

Гарантия безопасности исключения

базовая, 18

строгая, 18

**Д**  
Диспетчеризация дескрипторов, 191

**З**  
Завершающий возвращаемый тип, 37  
Зависимый тип, 75  
Замыкание, 19; 221  
    класс, 222

**И**  
Идиома  
    Pimpl, 132; 155  
    RAII, 257  
    захват ресурса есть инициализация, 257  
    указателя на реализацию, 132; 155  
    явной типизации инициализатора, 58

Инициализация  
    копированием, 299  
    прямая, 299  
    унифицированная, 62  
    фигурная, 62; 213  
        и vector<>, 67

Исключение  
    и деструктор, 103  
    спецификация, 98

Исключительное владение, 127

Итератор, 95

**К**  
Класс  
    замыкания, 222  
    перечислений, 78  
    шаблонный, 19

Конструктор  
    перемещающий, 117

Контекстное ключевое слово, 92

Контракт, 103

**Л**  
Лямбда-выражение, 221  
    инициализирующий захват, 229  
    и прямая передача, 235  
    обобщенное, 234  
    обобщенный захват, 231  
    режим захвата, 222

**М**  
Массив, 28  
Метапрограммирование, 76; 194; 200

**Н**  
Неопределенное поведение, 20; 41; 258; 274  
и предусловие, 104

**О**  
Объект  
    вызываемый, 18  
    со статическим временем хранения, 228  
    функциональный, 18  
Объявление, 19  
    псевдонима, 74  
Определение, 19  
Оптимизация  
    возвращаемого значения, 181  
    избыточных чтений и записей, 276  
    малых строк, 210; 289

**П**  
Параллельные вычисления, 245  
ложное пробуждение, 266  
локальная память потока, 251  
общее состояние, 261  
переменная условия, 265  
поток, 246  
Перекрытие, 88  
Перемещение, 117  
ограничения, 211  
почленное, 117; 210  
Перечисление, 78  
    базовый тип, 79  
    с областью видимости, 78  
Превышение подписки, 247  
Преобразование типа  
    неявное сужающее, 63  
Прокси-класс, 56  
    и auto, 57  
Прямая передача, 18; 165; 212; 235

**Р**  
Размещение, 293

**С**  
Свертывание ссылок. 175; 202; 203  
Совместное владение, 133  
Срезка, 290  
Ссылка  
    передаваемая, 172  
    универсальная, 171; 190  
        и перегрузка, 184

ограничения, 194

Ссылочный квалификатор, 89; 92

Счетчик ссылок, 133

## T

Тип

базовый перечисления, 79

зависимый, 75

литеральный, 108

неполный, 156

свойства, 76

только перемещаемый, 127; 165

## Y

Указатель

интеллектуальный, 20

на функцию, 30

обычный, 20

Унифицированная инициализация, 62

## Ф

Функциональный объект. См. Объект функциональный

Функция

make, 147

аргумент, 18

параметр, 18; 166

lvalue, 169

перекрытие, 88

сигнатура, 19

удаленная, 85

член, специальная, 116

шаблонная, 19

Фьючерс, 245

## Ш

Шаблон

вариативный, 213

выражения, 57

класса, 19

отключенный, 195

проектирования

наблюдатель, 145

прокси, 57

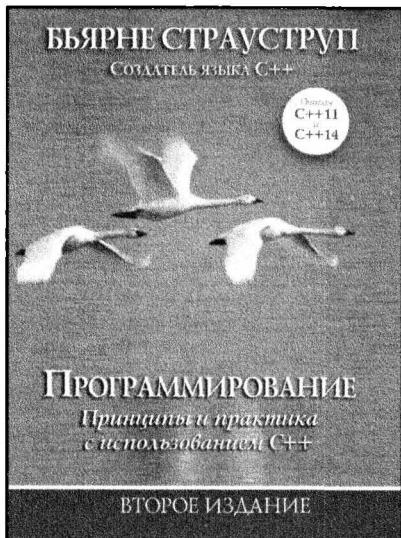
странный повторяющийся шаблон, 139

функции, 19

# ПРОГРАММИРОВАНИЕ. ПРИНЦИПЫ И ПРАКТИКА С ИСПОЛЬЗОВАНИЕМ С++

**ВТОРОЕ ИЗДАНИЕ**

**Бъярне Страуструп**



[www.williamspublishing.com](http://www.williamspublishing.com)

Эта книга — учебник по программированию. Несмотря на то что его автор — создатель языка С++, книга не посвящена этому языку; он играет в большей степени иллюстративную роль. Книга задумана как вводный курс по программированию с примерами программных решений на языке С++ и описывает широкий круг понятий и приемов программирования, необходимых для того, чтобы стать профессиональным программистом. В первую очередь книга адресована начинающим программистам, но она будет полезна и профессионалам, которые найдут в ней много новой информации, а главное, смогут узнать точку зрения создателя языка С++ на современные методы программирования.

**ISBN 978-5-8459-1949-6** в продаже

## Эффективный и современный C++

Освоение C++11 и C++14 – это больше, чем просто ознакомление с вводимыми этими стандартами возможностями (например, объявлениями типов `auto`, семантикой перемещения, лямбда-выражениями или поддержкой многопоточности). Вопрос в том, как использовать их эффективно, чтобы создаваемые программы были корректны, эффективны и переносимы, а также чтобы их легко можно было сопровождать. Именно этим вопросам и посвящена данная книга, описывающая создание по-настоящему хорошего программного обеспечения с использованием C++11 и C++14 – т.е. с использованием современного C++.

### В книге рассматриваются следующие темы.

- Преимущества и недостатки инициализации с помощью фигурных скобок, спецификации `noexcept`, прямой передачи и функций `make` интеллектуальных указателей
- Связь между `std::move`, `std::forward`, rvalue-ссылками и универсальными ссылками
- Методы написания понятных, корректных, эффективных лямбда-выражений
- Чем `std::atomic` отличается от `volatile`, как они используются и как соотносятся с API параллельных вычислений C++
- Какие из лучших методов "старого" программирования на C++ (т.е. C++98) должны быть пересмотрены при работе с современным C++

Эффективный и современный C++, следуя принципам более ранних книг Скотта Мейерса, охватывает совершенно новый материал. Эта книга достойна занять свое место на полке каждого программиста на современном C++.

Более чем 20 лет книги **Скотта Мейерса** серии Эффективный C++ являются критерием уровня книг по программированию на C++. Понятное пояснение сложного технического материала принесло ему всемирную известность. Он всегда самый желанный гость на международных конференциях, а его услуги консультанта широко востребованы во всем мире.

**Скотт Мейерс** имеет степень доктора философии (Ph.D.) в области компьютерных наук в Университете Брауна (Brown University).

ПРОГРАММИРОВАНИЕ/C++

После изучения основ C++ я перешел к изучению того, как применять C++ в промышленном программировании, с помощью серии книг Скотта Мейерса Эффективный C++. Эффективный и современный C++ – наиболее важная из книг серии, предлагающая ключевые рекомендации, стили и идиомы, позволяющие эффективно использовать современный C++. Вы еще не купили эту книгу? Сделайте это прямо сейчас.

Герб Саттер,  
глава Комитета ISO по стандартизации C++, специалист в области архитектуры программного обеспечения на C++ в Microsoft



[www.williamspublishing.com](http://www.williamspublishing.com)

