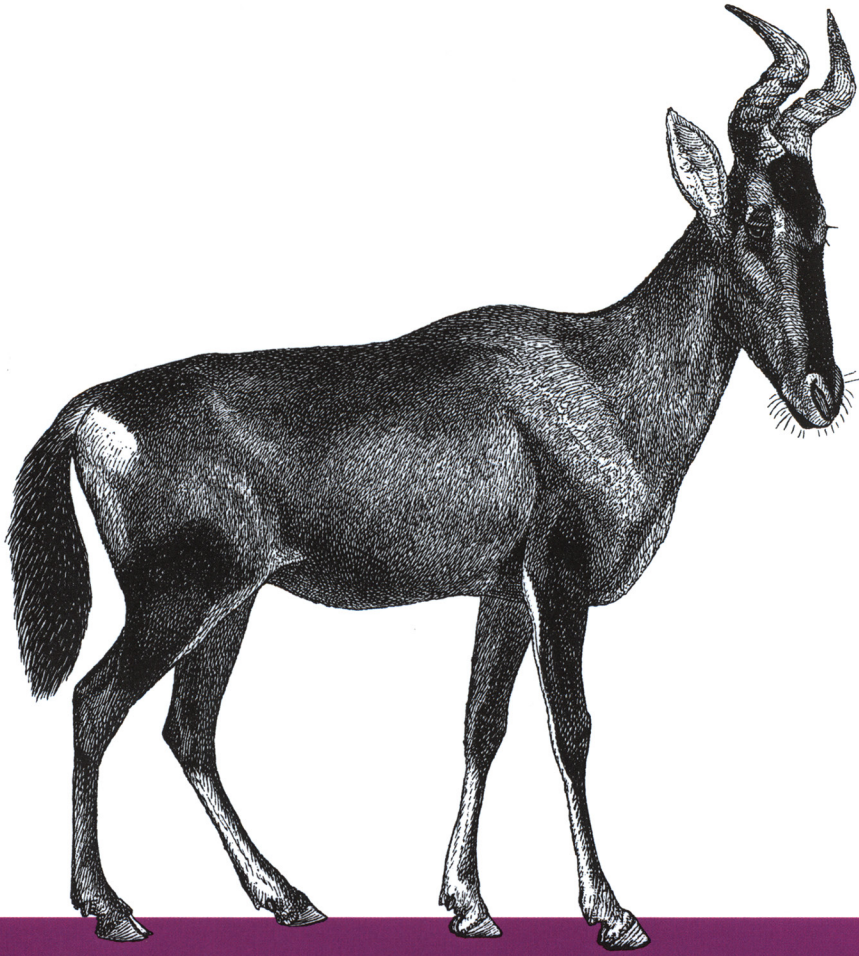
O'REILLY®



Оптимизация программ на C++

ПРОВЕРЕННЫЕ МЕТОДЫ ПОВЫШЕНИЯ ПРОИЗВОДИТЕЛЬНОСТИ

Курт Гантерот

Оптимизация программ на C++

Optimized C++

Kurt Guntheroth

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Оптимизация программ на C++

ПРОВЕРЕННЫЕ МЕТОДЫ ДЛЯ ПОВЫШЕНИЯ ПРОИЗВОДИТЕЛЬНОСТИ

Курт Гантерот

Москва • Санкт-Петербург • Киев
2017

ББК 32.973.26-018.2.75
Г19
УДК 681.3.07

Компьютерное издательство "Диалектика"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского и редакция канд. техн. наук *И.В. Красикова*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Гантерот, Курт.
Г19 Оптимизация программ на C++. Проверенные методы для повышения производительности. : Пер. с англ. — СпБ. : ООО "Альфа-книга", 2017. — 400 с. : ил. — Парал. тит. англ.

ISBN 978-5-9908910-6-7 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Курт Гантерот

Оптимизация программ на C++
Проверенные методы для повышения производительности

Литературный редактор *Л.Н. Красножон*
Верстка *Л.В. Чернокозинская*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. Гордиенко*

Подписано в печать 06.03.2017. Формат 70x100/16.
Гарнитура Times.
Усл. печ. л. 32,25. Уч.-изд. л. 24.41.
Тираж 300 экз. Заказ № 1488.

Отпечатано в АО «Первая Образцовая типография»
Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д.1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8(499)270-73-59

ООО "Альфа-книга", 195027, Санкт-Петербург, Магнитогорская ул., д. 30

ISBN 978-5-9908910-6-7 (рус.)

ISBN 978-1-491-92206-4 (англ.)

Оглавление

Содержание

Все благодарят супруг за помощь в работе над книгой. Я знаю, что это банально. Однако именно благодаря моей жене Рене Остлер (Renee Ostler) появилась эта книга. Рене месяцами позволяла мне работать над ней, обеспечивая мне время и место для работы и даже беседуя со мной об оптимизации программ на C++ — несмотря на то, что это не ее тема, просто чтобы меня поддержать. Этот проект стал важным для нее, потому что он был важен для меня. Я не могу просить у нее большего.

Предисловие

Приветствую вас! Меня зовут Курт, и я кодоголик[1].

Я пишу программы более 35 лет. Я никогда не работал в Microsoft, Google, Facebook, Apple или в каких-то иных знаменитых фирмах. Но, не считая нескольких коротких перерывов, я ежедневно в течение всего этого времени пишу код. Последние 20 лет я пишу почти исключительно на C++ и общаюсь с очень яркими разработчиками на этом языке программирования. Все это позволяет мне написать книгу об оптимизации кода на C++. За свою жизнь я написал *много* прозы, включая спецификации, руководства, комментарии, заметки и сообщения в блоге (`http://oldhandsblog.blogspot.com`). Поэтому меня всегда удивляет тот факт, что не больше половины ярких, компетентных программистов, с которыми я работал, в состоянии соединить в одно целое пару предложений на обычном человеческом языке.

Одна из моих любимых цитат — из письма Исаака Ньютона, в котором он пишет: "Если я и видел дальше других, то только потому, что стоял на плечах гигантов". Я не только стоял на плечах гигантов, но и читал их книги: элегантные небольшие книги, такие как книга Брайана Кернигана (Brian Kernighan) и Денниса Ритчи (Dennis Ritchie) *Язык программирования C*; интеллектуальные и просвещающие книги наподобие книг серии Скотта Мейерса (Scott Meyers) *Эффективный C++*; сложные, расширяющие сознание книги, такие как *Современное проектирование на C++* Андрея Александреску (Andrei Alexandrescu); точные и выверенные книги, как *Справочное руководство по языку программирования C++* Бьярне Страуструпа (Bjarne Stroustrup) и Маргарет Эллис (Margaret Ellis). Читая эти книги, я даже не думал, что когда-нибудь сам напишу книгу. Но в один прекрасный день я внезапно понял, что должен это сделать.

Но зачем писать книгу о настройке производительности программ на C++?

В начале XXI века C++ оказался под постоянными нападками. Поклонники C указывали на программы на C++, производительность которых уступала предположительно эквивалентному коду, написанному на C. Известные корпорации с большими бюджетами расхваливали собственные патентованные объектно-ориентированные языки, утверждая, что C++ слишком трудно использовать и что их

[1] Аллюзия на собрания групп психологической поддержки алкоголиков. — *Примеч. пер.*

языки — настоящие инструменты будущего. Университеты выбирали для преподавания Java, потому что этот язык имел массу бесплатных инструментальных средств. В результате всего этого шума крупные компании сделали большие ставки на кодирование веб-сайтов и операционных систем на Java, C# или PHP. Казалось, что наступил закат C++. Это было тяжелое время для тех, кто считал, что C++ был и остается мощным, полезным инструментом.

Затем произошли забавные вещи. Производительность ядер процессоров перестала расти, в отличие от постоянного роста рабочих нагрузок. И эти же компании принялись нанимать программистов на C++ для решения проблем масштабирования. Стоимость переписывания кода "с нуля" на C++ оказалась меньше расходов на электроэнергию в центрах данных. Внезапно C++ вновь стал популярен.

Среди языков программирования, широко использовавшихся в начале 2016 года, по сути, только C++ предлагает разработчикам массу вариантов реализаций, начиная от автоматизированной поддержки для тонкого ручного управления кодом. C++ позволяет разработчикам получить полный контроль над производительностью программ, делающий возможной оптимизацию кода.

Есть не так уж много книг, посвященных оптимизации кода на C++. Одной из них является педантичное исследование Балка (Bulka) и Мэйхью (Mayhew) *Optimizing C++*. Похоже, у этих авторов такой же опыт работы, как и у меня, и они открыли массу тех же принципов оптимизации, что и я. Для читателей, которые заинтересованы в другом взгляде на вопросы, поднимаемые в моей книге, их книга является хорошим подспорьем. Кроме того, вопросы оптимизации часто освещает в своих книгах Скотт Мейерс.

Материала по оптимизации программ достаточно, чтобы написать не одну, а десяток книг. Я попытался выбрать тот материал, который часто встречался в моей личной работе или который предлагает существенное повышение производительности. Многие читатели имеют собственный опыт борьбы за производительность, и они могут задаться вопросом, почему я ничего не сказал о стратегиях, которые творили чудеса в их программах. Увы, я могу ответить одно — *так мало времени, так много надо рассказать*![2]

Я жду ваши комментарии, замечания об ошибках и описания ваших стратегий оптимизации по адресу `antelope_book@guntheroth.com`.

Я люблю свое ремесло программиста. Мне нравится бесконечно "вылизывать" каждый новый цикл или интерфейс. Это смесь науки и искусства, которую может оценить только столь же увлеченный программист. Элегантно закодированная функция полна своей внутренней красоты, а использование идиом программирования полно мудрости. К сожалению, на каждую поэму программного обеспечения наподобие стандартной библиотеки шаблонов Степанова приходится по 10 тысяч однообразно-безвкусных томов неодухотворенного кода.

Главное назначение этой книги — помочь каждому читателю не только оценить красоту хорошо настроенного и оптимизированного программного обеспечения, но и самостоятельно его создать!

[2] Парафраз припева песни из сериала "Так мало времени" ("So little time"). — *Примеч. пер.*

Извинения за код

Хотя я пишу и оптимизирую код на C++ более 20 лет, большая часть кода, содержащегося в этой книге, была разработана специально для нее. Как весь новый код, он, безусловно, содержит дефекты. Я заранее приношу мои извинения за него.

Я в течение многих лет программировал для Windows, Linux и различных встроенных систем. Код, представленный в этой книге, разработан для Windows. Код и сама книга, несомненно, демонстрируют склонность к Windows. Тем не менее методы оптимизации кода C++, которые иллюстрируются с помощью Visual Studio в Windows, применимы и в Linux, Mac OS X или любой другой среде C++. Однако точные цифры, связанные с различными методами оптимизации, зависят от компилятора, реализации стандартной библиотеки и процессора, на котором выполняется код. Оптимизация — наука экспериментальная. Принятие советов по оптимизации на веру чревато различными неприятными сюрпризами.

Я знаю, что совместимость кода с различными компиляторами, а также с операционной системой Unix и встраиваемыми системами может оказаться сложной задачей, так что я заранее прошу прощения, если приведенный в книге код не компилируется в вашей любимой системе. Поскольку эта книга не о переносимости кода, я позволил себе жертвовать переносимостью в пользу простоты и понятности кода.

Используемый в книге стиль форматирования кода — не мой любимый, но в связи с тем, что мне надо было размещать на странице как можно больше строк кода, я вынужден был отказаться от своих предпочтений.

Использование примеров кода

Сопутствующие материалы (примеры кода, решения и т.п.) доступны по адресу `www.guntheroth.com`.

Эта книга призвана помочь вам выполнить вашу работу. В общем случае вы можете использовать примеры кода из нее в своих программах и документации. Вам не нужно связываться с издательством для получения разрешения, если только вы не воспроизводите значительную часть кода. Например, для написания программы, в которой используется несколько фрагментов кода из этой книги, разрешение получать не нужно. Однако для продажи или распространения на CD-ROM примеров из книг издательства O'Reilly необходимо отдельное разрешение. Ссылаться на книгу или пример кода в ответах на вопросы можно и без разрешения. Но для включения значительного объема кода из этой книги в документацию вашего продукта следует получить разрешение.

Мы не требуем точного указания источника при использовании примеров кода, но были бы признательны за него. Обычно достаточно названия книги, имени автора, названия издательства и ISBN.

Если вы полагаете, что использование примеров кода выходит за рамки описанных разрешений, не постесняйтесь связаться с нами по адресу `permissions@oreilly.com`.

Соглашения, использованные в книге

В книге использованы некоторые типографские соглашения.

Обычный текст

Используется для обычного текста, пунктов меню и названий клавиш (таких, как <Alt> и <Ctrl>).

Курсив

Используется для новых терминов, важных концепций и т.п.

`Моноширинный шрифт`

Используется для листингов, а также в тексте для элементов программного кода (имен переменных, функций, инструкций и т.д.), URL, имен файлов и т.п.

Об авторе

Курт Гантерот — программист более чем с 35-летним стажем, интенсивно работающий с языком программирования C++ более 25 лет. Он разрабатывал программное обеспечение для Windows, Linux и различных встроенных устройств.

Все свободное время Курт проводит с женой и четырьмя сыновьями. Живет в Сиэттле, штат Вашингтон.

Об изображении на обложке

На обложке книги изображена антилопа каама (*Alcelaphus buselaphus caama*), обитающая на равнинах и в лесах Юго-Западной Африки. Это большая антилопа, принадлежащая к семейству полорогих (*Bovidae*). Антилопы обоих полов обладают витыми рогами, достигающими 60 сантиметров в длину. Животные также обладают завидными обонянием и слухом. Бегая зигзагом, чтобы спастись от хищников, они развивают скорость до 55 километров в час. Хотя львы, леопарды и гепарды иногда охотятся на представителей этого вида, обычно они предпочитают других жертв.

Многие из животных, изображенных на обложках книг издательства O'Reilly, находятся под угрозой исчезновения; все они имеют важное значение для мира. Чтобы узнать больше о том, как вы можете им помочь, перейдите на сайт по адресу `animals.oreilly.com`.

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@dialektika.com`
WWW: `http://www.dialektika.com`

Наши почтовые адреса:
в России: 195027, Санкт-Петербург, Магнитогорская ул., д. 30, ящик 116
в Украине: 03150, Киев, а/я 152

ГЛАВА 1

Обзор оптимизации

У всего мира в настоящее время колоссальный аппетит на вычисления. Где бы ни выполнялся код, в часах, телефоне, на планшете, рабочей станции, в суперкомпьютере или глобальной сети центров обработки данных, существует множество программ, которые должны выполняться все время. Поэтому недостаточно просто точно преобразовать блестящую идею, возникшую в вашей голове, в строки кода. Недостаточно даже просто отлаживать код в поисках дефектов до тех пор, пока он не будет постоянно правильно работать. Приложение может оказаться слишком медленным для того типа оборудования, которое ваши клиенты могут себе позволить. Компания может ограничить вас крошечным процессором для экономии электроэнергии. Вы можете драться с конкурентами за пропускную способность или за количество кадров в секунду. Словом, в жизни всегда есть место оптимизации.

Эта книга — об оптимизации, конкретнее — об оптимизации программ C++, с особым акцентом на модели поведения кода C++. Одни методы из этой книги применимы и к другим языкам программирования, но я не пытался объяснять предлагаемые методы в универсальной форме. Другие оптимизации, эффективные для кода C++, никак не влияют на производительность (или просто невозможны) на других языках.

Эта книга — о получении правильного кода, который воплощает в себе лучшие практики проектирования C++, и об его преобразовании в правильный код, который по-прежнему воплощает в себе хороший дизайн C++, но при этом работает быстрее и потребляет меньше ресурсов практически на любом компьютере. Многие возможности оптимизации доступны потому, что некоторые мимоходом использованные функции C++ медленно работают и потребляют много ресурсов. Такой код, будучи корректным, является необдуманным, учитывающим только небольшое количество знаний о современных микропроцессорных устройствах или мало заботящимся о стоимости различных конструкций C++. Другие виды оптимизации доступны благодаря возможностям тонкого управления памятью и копирования, предлагаемым языком программирования C++.

Эта книга — *не* о том, как кропотливо кодировать подпрограммы на языке ассемблера, подсчитывать такты процессора или выяснить, сколько команд последний процессор Intel в состоянии выполнить одновременно. Есть разработчики, которые годами работают с одной платформой (хорошим примером является Xbox), так что у них есть время и необходимость освоить эти темные искусства. Однако большинство

разработчиков пишут для телефонов, планшетов или персональных компьютеров, которые содержат бесконечное множество микропроцессоров (некоторые из них еще даже не разработаны). Разработчики программного обеспечения для встроенных устройств также сталкиваются с различными процессорами и архитектурами. Попытка изучить конкретный процессор может парализовать всю работу такого программиста. Я не рекомендую этот путь. Оптимизация, зависящая от процессора, просто не оказывается плодотворным решением для большинства приложений, которые по определению должны работать на различных процессорах.

Эта книга *не* посвящена также изучению операционных систем — выяснению, как именно некоторая операция выполняется в Windows, OS X и Linux, а равно и в каждой из встроенных систем. Книга посвящена тому, что можно сделать в C++, включая стандартную библиотеку языка. Выход за рамки C++ для выполнения оптимизации может усложнить просмотр или комментарии оптимизированного кода. Не следует относиться к этому легкомысленно. Эта книга — об обучении тому, *как* следует оптимизировать код. В этой области любой статичный каталог методов или функций обречен, так как постоянно разрабатываются новые алгоритмы и становятся доступными новые возможности языка. Вместо этого в настоящей книге содержится несколько рабочих примеров постепенного улучшения кода, так что читатель знакомится с постепенным процессом настройки производительности и развивает свое мышление в этом направлении.

Эта книга — об оптимизации самого процесса кодирования. Памятуя о стоимости выполнения, разработчики могут писать код, который эффективен с самого начала. С ростом опыта для написания быстрого кода обычно не требуется больше времени, чем для написания более медленного кода.

Наконец это книга о чудесах производительности, которыми будут восхищены ваши коллеги. Оптимизация — это то, что вы всегда можете делать как разработчик и чем вы всегда можете гордиться.

Оптимизация — часть разработки программного обеспечения

Оптимизация является частью кодирования. В традиционном процессе разработки программного обеспечения оптимизация выполняется после завершения кода, на этапе интеграции и тестирования проекта, когда можно наблюдать производительность всей программы. В процессе гибкой разработки для оптимизации может быть выделен один или несколько спринтов после того, как необходимая функциональность уже разработана.

Цель оптимизации — улучшить поведение корректно работающей программы так, чтобы она удовлетворяла потребностям клиентов в отношении скорости, пропускной способности, объема памяти, потребления электроэнергии и т.д. Таким образом, оптимизация так же важна, как и процесс кодирования функциональных возможностей. Неприемлемо низкая производительность является такой же проблемой для пользователей, как и ошибки или отсутствие функциональных возможностей.

Одно важное различие между исправлением ошибок и настройкой производительности заключается в том, что производительность является непрерывной переменной. Функция либо закодирована, либо нет. Ошибка либо есть, либо отсутствует. Однако производительность может быть очень плохой или очень хорошей или находиться где-то между этими двумя крайностями. Оптимизация также представляет собой итеративный процесс, в котором каждый раз улучшается самая медленная часть программы, после чего появляется новая самая медленная часть.

Оптимизация — в высшей степени экспериментальная наука, требующая научного мышления в большей степени, чем некоторые другие задачи кодирования. Для успешной оптимизации необходимы наблюдение за поведением программы, выдвижение проверяемых гипотез на основе этих наблюдений и проведение экспериментов, которые приводят к измерениям, поддерживающим или опровергающим выдвинутые гипотезы. Опытные разработчики часто верят в свой опыт и интуицию при разработке оптимального кода. Но если постоянно не проверять свои интуитивные решения, они зачастую оказываются неверными. Мой личный опыт написания тестовых программ для этой книги привел к ряду результатов, противоречащих моей интуиции. Эксперименты взамен интуиции — одна из тем этой книги.

Эффективность оптимизации

Разработчикам трудно судить о воздействии отдельных решений на общую производительность большой программы. Таким образом, практически все полные программы содержат значительные возможности оптимизации. Даже код, созданный опытными группами за длительное время, зачастую можно ускорить на 30–100%. В случае более быстрого кодирования или менее опытной команды улучшение производительности может составлять от 3 до 10 раз — я лично сталкивался с такими ситуациями. Увеличение скорости более чем в 10 раз путем настройки кода оказывается куда менее вероятным. Однако выбор лучшего алгоритма или более подходящей структуры данных может обеспечить и такую разницу в производительности.

Оптимизируйте!

Многие учебники по оптимизации начинаются с предупреждения: *не делайте этого*! Не оптимизируйте, а если вам и нужно оптимизировать, то не делайте этого до конца проекта и не делайте оптимизацию сверх необходимой. Например, знаменитый ученый Дональд Кнут (Donald Knuth) сказал об оптимизации:

> *Мы не должны помнить о малой эффективности, скажем, около 97 процентов времени: преждевременная оптимизация является корнем всех зол.*
>
> — Donald Knuth, *Structured Programming with go to Statements*, ACM Computing Surveys 6(4), December 1974, p. 268. CiteSeerX: 10.1.1.103.6084 (`http://bit.ly/knuth-1974`)

А вот что сказал Уильям А. Вульф (William A. Wulf):

> *Во имя эффективности (но не обязательно ее достижения) совершается больше вычислительных грехов, чем по любой иной причине — включая слепую глупость.*
>
> — "A Case Against the GOTO," *Proceedings of the 25th National ACM Conference* (1972): 796

Совет не оптимизировать стал высшей мудростью, непререкаемой даже для многих опытных кодировщиков, которые рефлекторно содрогаются, когда речь заходит о настройке производительности. Я думаю, что циничное выпячивание этой рекомендации слишком часто используется для оправдания отсутствия привычки даже к минимальному анализу, результатом которого мог бы стать значительно более быстрый код. Я также думаю, что некритическое принятие этой рекомендации ответственно за огромное количество потраченного впустую процессорного времени, многие часы пользовательского разочарования и слишком большое время, затраченное на переделку кода, который должен был быть более эффективным с самого начала.

Мой совет менее догматичен. Оптимизация — это нормально. Вы можете изучать идиомы эффективного программирования и применять их постоянно, даже если не знаете, для какого кода производительность является критичной. Эти идиомы и есть "хороший C++". Если же кто-то спросит, почему вы не написали что-то "просто" и неэффективно, можете ответить, что "написание моего кода занимает столько же времени, сколько и написание медленного, расточительного кода. Так почему я должен преднамеренно писать неэффективный код?"

Что не нормально — так это отсутствие какого-либо прогресса в течение многих дней, потому что вы не можете решить, какой алгоритм будет лучше, если не уверены, что это имеет какое-то значение. Не нормально — потратить недели, кодируя что-то на языке ассемблера, поскольку вы *предполагаете*, что этот код может быть критичным в плане производительности, а затем свести на нет все усилия, вызывая этот код как функцию C++, в то время как компилятор может встроить его для вас. Не нормально — требовать от команды писать половину программы на языке C просто потому, что "все знают, что C быстрее", в то время как на самом деле неизвестно ни что C действительно быстрее, ни что C++ не такой быстрый. Другими словами, следует по-прежнему применять все лучшие практики разработки программного обеспечения. Оптимизация — не повод для нарушения правил.

Не нормально — тратить кучу времени на оптимизацию, когда вы не знаете, где именно у вас есть проблемы с производительностью. В главе 3, "Измерение производительности", вводится правило 90/10, гласящее, что только около 10% кода программы критично в смысле производительности. Таким образом, не является необходимым или полезным вмешательство в каждую строку с целью улучшения производительности программы. Поскольку лишь 10% программы оказывает значительное влияние на производительность, ваши шансы выбора случайным образом хорошей отправной точки для повышения производительности оказываются слишком малы. В главе 3, "Измерение производительности", описаны инструменты, помогающие определить, где находятся действительно важные точки кода.

Когда я учился в колледже, преподаватель предупреждал, что оптимальные алгоритмы могут иметь более высокую начальную стоимость, чем более простые. Таким образом, их следует использовать только для очень больших наборов данных. Хотя это, вполне возможно, верно для ряда сложных алгоритмов, мой опыт показывает, что оптимальные алгоритмы для задач простого поиска и сортировки требуют не так уж много времени для написания и дают улучшение производительности даже на небольших наборах данных.

Мне также рекомендовали разрабатывать программы, используя алгоритмы, которые проще всего кодировать, а затем при необходимости возвращаться и оптимизировать, если программа работает слишком медленно. Хотя это, несомненно, хороший совет — постоянно добиваться улучшений, но после того, как вы несколько раз написали оптимальный поиск или сортировку, эта работа становится ничуть не сложнее написания более медленного алгоритма. Вы можете с самого начала писать и отлаживать только один, оптимальный алгоритм.

Пожалуй, самый страшный враг оптимальности — общепризнанная мудрость. Например, "все знают", что оптимальный алгоритм сортировки имеет время работы $O(n \cdot log\ n)$, где n — размер набора данных (в главе 5, "Оптимизация алгоритмов", вы познакомитесь с O-записями и сложностью алгоритмов). Это ценное знание, которое удержит программиста от веры в то, что написанная им сортировка вставками со временем работы $O(n^2)$ оптимальна... но далеко не такое ценное, если заодно удержит программиста от поиска в литературе информации о том, что есть более быстрая поразрядная сортировка со временем работы $O(n \cdot log_r n)$ (где r — основание системы счисления или количество корзин сортировки), что сортировка распределением (flashsort) быстрее $O(n)$ для случайно распределенных данных или что быстрая сортировка, несмотря на превосходство в среднем над другими методами сортировки, в наихудшем случае имеет время работы $O(n^2)$. В свое время Аристотель утверждал, что у женщин зубов меньше, чем у мужчин (*The History of Animals*, Book II, part 1 (`http://bit.ly/aristotle-animals`)), и прошло около 1500 лет, прежде чем кто-то оказался достаточно любознательным, чтобы заглянуть в несколько ртов и пересчитать зубы. Противоядием для общепризнанной мудрости является научный метод в форме эксперимента. В главе 3, "Измерение производительности", рассматриваются инструменты для измерения производительности программного обеспечения и эксперименты для проверки оптимизаций.

В мире разработки программного обеспечения существуют также общепризнанные мудрости, не имеющие отношения к оптимизации. Так, утверждают, что даже если ваш код сегодня работает медленно, то, поскольку каждый год появляются все более и более быстрые процессоры, решение проблем с производительностью придет само — через некоторое время. Как и большинство общепризнанных мудростей, этот самородок мысли никогда не соответствовал действительности. Это еще могло показаться верным в 1980- и 1990-е годы, когда доминировали настольные компьютеры и автономные приложения, а скорость одноядерных процессоров вырастала в два раза каждые 18 месяцев. Но сейчас, когда многоядерные процессоры становятся все более мощными в совокупности, производительность отдельных ядер улучшается очень не намного, а то и вовсе снижается. Сегодняшние программы должны

работать и на мобильных платформах, на которых скорость выполнения ограничивается продолжительностью жизни аккумуляторов и выделением тепла. Кроме того, в то время как новые клиенты могут оказаться с более быстрыми компьютерами, обычно мало что делается для повышения производительности имеющегося оборудования. У существующих клиентов рабочие нагрузки со временем только растут. Так что рост скорости для существующих клиентов может получаться только путем оптимизации новых версий программного обеспечения.

Наносекунда туда, наносекунда сюда

> *Миллиард туда, миллиард сюда — так вскоре мы будем говорить о реальных деньгах.*
>
> — Часто ложно приписывается сенатору Эверетту Дирксону (Everett Dirkson) (1898–1969), который утверждал, что никогда не говорил этой фразы, хотя и высказывал похожие мысли

Настольные компьютеры удивительно быстрые. Они могут выполнять новую команду каждую наносекунду (а то и быстрее). Каждые 10^{-9} секунд! Очень соблазнительно считать, что оптимизация не имеет значения для таких быстрых компьютеров.

Проблема такого образа мышления заключается в том, что чем быстрее процессор, тем быстрее накапливаются ненужные команды. Если не нужны 50% команд, выполняемых программой, то такая программа может ускориться в два раза путем устранения ненужных команд, независимо от того, как быстро они выполняются.

Ваши коллеги, которые говорят, что "эффективность не имеет значения", могут также подразумевать, что она не имеет значения для некоторых приложений, которые отвечают на запросы человека и работают на настольных компьютерах, — мол, время реакции и так очень мало. Но эффективность очень много значит для малых встраиваемых и мобильных процессоров с ограничениями памяти, питания или скорости. Это также важный вопрос для серверных программ, работающих на больших машинах. Эффективность имеет важное значение для любого приложения, которое должно работать в условиях ограниченных ресурсов (памяти, мощности, скорости процессора). Вопросы эффективности важны и в случаях достаточно большой рабочей нагрузки, распределенной между несколькими компьютерами.

За 50 лет производительность компьютеров выросла на шесть порядков. И тем не менее мы все равно говорим об оптимизации. И скорее всего, вопросы оптимизации останутся актуальными и в будущем.

Стратегии оптимизации кода на C++

> *Соберем обычных подозреваемых.*
>
> — Капитан Луи Рено (Клод Рейнс), к/ф *Касабланка*, 1942

Сочетание возможностей C++ предоставляет бесконечное количество вариантов реализации, начиная с полной автоматизации и выразительности, с одной стороны,

и закачивая полным тонким контролем производительности — с другой. Именно эта возможность выбора позволяет настроить программы на C++ для удовлетворения требованиям производительности.

C++ имеет своих "обычных подозреваемых" в смысле горящих мест для оптимизации, включая вызовы функций, выделения памяти и циклы. Ниже приведено несколько способов повышения производительности программ C++, который одновременно является и планом данной книги. Все советы поразительно просты. Все они были опубликованы раньше. Но, как всегда, дьявол прячется в подробностях. Примеры и эвристики из этой книги помогут вам лучше распознавать возможности оптимизации, когда вы их увидите.

Используйте компилятор получше; используйте компилятор лучше

Компиляторы C++ — сложные компьютерные программы. Каждый компилятор принимает различные решения о том, какой машинный код должен быть сгенерирован для тех или иных инструкций C++. Компиляторы видят различные возможности для оптимизации и производят различные выполнимые файлы из одного и того же исходного кода. Если вы пытаетесь выжать всю возможную производительность из своего кода, возможно, вам стоит попробовать несколько компиляторов, чтобы выяснить, какой из них даст наиболее быстрый выполнимый файл для вашего кода.

Наиболее важный совет при выборе компилятора C++ — используйте *компилятор, соответствующий стандарту C++11*. Этот стандарт реализует rvalue-ссылки и семантику перемещения, которые устраняют многие операции копирования, неизбежные в предыдущих версиях C++. (Семантика перемещения рассматривается в разделе "Реализация семантики перемещения" главы 6, "Оптимизация переменных в динамической памяти").

Иногда *использовать лучший компилятор* на самом деле означает *использовать компилятор лучше*. Например, если приложение кажется вам медленно работающим, взгляните на параметры компилятора, чтобы убедиться, что оптимизатор включен. Это совершенно очевидно, но я не могу даже прикинуть, сколько раз я давал этот совет программистам, которые впоследствии признавались, что после этого их код стал работать гораздо быстрее. Во многих случаях больше ничего и не требуется. Один лишь компилятор в состоянии сделать программу в несколько раз быстрее, если, конечно, хорошенько его об этом попросить.

По умолчанию у большинства компиляторов какая-либо оптимизация не включена. Без оптимизации время компиляции программы немного короче. Это имело значение в 1990-х годах, но в настоящее время компиляторы и компьютеры так быстры, что дополнительная плата за оптимизацию оказывается весьма незначительной. Кроме того, при выключенной оптимизации отладка также проще, потому что поток выполнения точно соответствует исходному коду. Оптимизатор может вынести код из цикла, удалить некоторые вызовы функций или полностью удалить некоторые переменные. Многие компиляторы вообще не выдают отладочную информацию при включенной оптимизации. Другие компиляторы оказываются более доброжелательными в этом отношении, но понять, что делает программа, наблюдая ход ее выполнения в отладчике, может оказаться непросто. Ряд компиляторов позволяет

включать и выключать отдельные методы оптимизации в отладочном варианте, чтобы не слишком усложнять отладку. Простое включение встраивания функций может иметь значительное влияние на программы C++, поскольку хороший стиль C++ включает написание множества небольших функций-членов для доступа к переменным-членам классов.

В документации, которая поставляется с компилятором C++, содержится обширное описание доступных параметров и прагм оптимизации. Эта документация подобна руководству пользователя, которое поставляется с новым автомобилем. Вы можете сесть в свой новый автомобиль и повести его и без чтения руководства, но есть масса информации, которая может помочь вам использовать этот большой и сложный инструмент более эффективно.

Если вам достаточно повезло и вы разрабатываете приложения для архитектуры x86 под управлением Windows или Linux, то у вас есть выбор из нескольких отличных активно развивающихся компиляторов. Microsoft выпустила три версии Visual C++ за пять лет, предшествовавших написанию этой книги. GCC выходит со скоростью более одной версии в год.

По состоянию на начало 2016 года имелось общее согласие в том, что компилятор Intel C++ создает наиболее короткий код для Linux и Windows, что компилятор GNU C++ — GCC — имеет более низкую производительность, но отличное соответствие стандарту и что Microsoft Visual C++ находится между ними. Я хотел бы помочь вам в принятии решения, приведя небольшую диаграмму, которая показывала бы, на сколько процентов код, создаваемый Intel C++, быстрее кода, генерируемого GCC, но это зависит от конкретного кода и конкретной версии. Intel C++ стоит более тысячи долларов, но предлагает 30-дневную бесплатную пробную версию. Имеются бесплатные экспресс-версии Visual C++. В Linux компилятор GCC всегда был и остается бесплатным. Несложно провести небольшой эксперимент, испытывая каждый компилятор с вашим кодом, чтобы увидеть, дает ли какой-то из них особое преимущество в производительности.

Использование лучших алгоритмов

Самый большой эффект дает выбор оптимального алгоритма. Эти усилия могут повысить производительность программы самым драматическим образом. Они могут повлиять на код так же, как кардинальное обновление старенького компьютера. К сожалению, так же, как и модернизация компьютера, большая часть оптимизаций повышает производительность на константный множитель. Многие усилия по оптимизации дают улучшение 30–100%. Если вам повезет, производительность может утроиться. Но качественный скачок в производительности маловероятен, если только вы не находите более эффективный алгоритм.

Глупо героически оптимизировать плохой алгоритм. Изучение и использование оптимальных алгоритмов для поиска и сортировки открывают широкий путь для оптимального кода. Неэффективная процедура поиска или сортировки может совершенно испортить программу. Настройка оптимизации без изменения алгоритма ведет к сокращению времени выполнения на постоянный множитель. Переход к более эффективному алгоритму может сократить время выполнения на множитель,

который тем больше, чем больше ваш набор данных. Даже на небольших наборах данных из десятков элементов оптимальный поиск или сортировка может сохранить кучу времени, если поиск данных выполняется часто. В главе 5, "Оптимизация алгоритмов", содержится ряд указаний о том, как выглядят оптимальные алгоритмы.

Из истории оптимизационных войн

Еще в дни 8-дюймовых дискет и процессоров с частотой 1 МГц некоторый разработчик делал программу для управления радиостанциями. Одной из составных частей этой программы была подготовка отсортированного журнала песен, проигранных в течение дня. Проблема заключалась в том, что сортировка происшедшего за 24 часа занимала около 27 часов, что было явно неприемлемо. Чтобы заставить сортировку работать быстрее, разработчик предпринял героические усилия. Он применил методы обратной инженерии к компьютеру и использовал недокументированные средства микрокода, но время выполнения оставалось все еще неприемлемым — 17 часов. Отчаявшись, он обратился за помощью на фирму — производитель компьютера (на которой я в то время работал).

Я спросил разработчика, какой алгоритм сортировки он использует. Он ответил, что использует сортировку слиянием. Но сортировка слиянием относится к семейству оптимальных алгоритмов сортировки. Сколько же записей он сортировал? "Несколько тысяч", — ответил он. Это было невообразимо. Система, которую он использовал, должна была сортировать его данные менее чем за час. Я попросил разработчика подробно описать примененный алгоритм сортировки. Я не могу вспомнить в деталях, как он мне отвечал, но главное — то, что он называл сортировкой слиянием, было на самом деле сортировкой вставками. Сортировка вставками — плохой выбор, особенно если учесть, что ее время работы пропорционально квадрату количества сортируемых записей (см. табл. 5.3). Он знал, что какой-то алгоритм называется сортировкой слиянием и что он считается оптимальным. Вот он и описал сортировку вставками, используя слова "сортировка слиянием".

Когда я написал для этого клиента программу реальной сортировки слиянием, его данные стали сортироваться за 45 минут.

Возможность использовать оптимальные алгоритмы не связана с конкретным размером данных. Есть много прекрасных книг, освещающих эту тему, и их можно изучать всю жизнь. Жаль, что в этой книге я могу только слегка коснуться темы оптимальных алгоритмов.

Раздел "Шаблоны оптимизации" главы 5, "Оптимизация алгоритмов", охватывает несколько важных методик повышения производительности; к ним относятся *предвычисления* (перемещение вычислений со времени выполнения на время компоновки,

компиляции или разработки), *отложенные вычисления* (перемещение вычисления в точку, где не используемый в ряде случаев результат действительно необходим) и *кеширование* (сохранение и повторное использование результатов дорогостоящих вычислений). В главе 7, "Оптимизация инструкций", имеется много примеров применения этих методов на практике.

Использование лучших библиотек

Стандартные библиотеки шаблонов и времени выполнения, которые поставляются с компиляторами C++, должны быть простыми в обслуживании, обобщенными и очень надежными. Сюрпризом для программистов может оказаться то, что эти библиотеки не обязательно настроены для получения оптимальной скорости работы. Еще более удивительным может оказаться то, что даже после 30 лет развития C++ библиотеки, которые поставляются с коммерческими компиляторами C++, по-прежнему содержат ошибки и могут не соответствовать текущим стандартам C++ или даже стандарту, действовавшему во время выпуска компилятора. Это осложняет задачу измерений или рекомендации оптимизаций и делает непереносимым любой опыт оптимизации, который, как им кажется, имеют разработчики. Глава 8, "Использование лучших библиотек", посвящена именно этим вопросам.

Умение работать со стандартной библиотекой C++ является одним из важнейших навыков разработчика. Эта книга содержит рекомендации по алгоритмам поиска и сортировки (глава 9, "Оптимизация сортировки и поиска"), оптимальных идиом использования контейнерных классов (глава 10, "Оптимизация структур данных"), ввода-вывода (глава 11, "Оптимизация ввода-вывода"), параллелизма (глава 12, "Оптимизация параллельности") и управления памятью (глава 13, "Оптимизация управления памятью").

Для выполнения важных функций, таких как управление памятью (см. раздел "Высокопроизводительные диспетчеры памяти" главы 13, "Оптимизация управления памятью"), имеются открытые библиотеки, которые обеспечивают сложные реализации, могущие быть более быстрыми и более функциональными, чем стандартные библиотеки времени выполнения C++. Преимущество этих альтернативных библиотек заключается в том, что можно легко добавить их в существующий проект и обеспечить немедленное увеличение скорости.

Среди множества других есть много общедоступных библиотек проекта Boost (`http://www.boost.org`) и Google Code (`https://code.google.com`), которые предоставляют библиотеки для ввода-вывода, управления окнами, обработки строк (см. раздел "Применение более новой реализации строки" главы 4, "Оптимизация использования строк") и параллелизма (см. раздел "Библиотеки для параллельных вычислений" главы 12, "Оптимизация параллельности") и которые не являются заменой стандартных библиотек, а просто предлагают улучшенную производительность и повышенную функциональность. Эти библиотеки обладают преимуществом — повышенной скоростью — за счет ряда компромиссов проектирования, отличных от используемых в стандартной библиотеке.

Наконец, можно разработать библиотеку специально для проекта, которая ослабляет некоторые из ограничений безопасности и надежности стандартной библиотеки

в обмен на преимущество в скорости. Все эти темы рассматриваются в главе 8, "Использование лучших библиотек".

Вызовы функций являются дорогостоящими по нескольким причинам (см. раздел "Стоимость вызовов функций" главы 7, "Оптимизация инструкций"). API хорошей библиотеки функций предоставляют функции, которые отражают идиомы использования этих API, так что пользователю не приходится прибегать к чрезмерно частым вызовам базовых функций. Например, интерфейс, который получает символы и предоставляет только функцию `get_char()`, требует вызова этой функции для каждого символа. Если же интерфейс предоставляет также функцию `get_buffer()`, можно избежать необходимости вызова функции для каждого символа.

Библиотеки функций и классов позволяют скрыть сложность, которая иногда является ценой хорошо настроенной программы. Библиотеки должны компенсировать стоимость вызова функций, делая свою работу с максимальной эффективностью. Библиотечные функции часто оказываются на дне глубоко вложенных цепочек вызовов, что усиливает эффект повышения производительности.

Уменьшение количества выделений памяти и копирований

Уменьшение количества вызовов диспетчера памяти является столь эффективной оптимизацией, что разработчик может быть успешным оптимизатором, зная только один этот трюк. Несмотря на то, что стоимость большинства языковых возможностей C++ не превышает нескольких команд, стоимость каждого обращения к диспетчеру памяти измеряется тысячами инструкций.

Поскольку строки являются очень важной (и очень дорогостоящей) частью многих программ C++, я посвятил им целую главу в качестве тематического исследования в области оптимизации. Глава 4, "Оптимизация использования строк", вводит и оправдывает многие концепции оптимизации в знакомом контексте обработки строк. Глава 6, "Оптимизация переменных в динамической памяти", посвящена снижению стоимости динамического выделения памяти без отказа от таких полезных идиом программирования C++, как строки и контейнеры стандартной библиотеки.

Один вызов функции копирования буфера также может потребовать тысяч тактов процессора. Таким образом, уменьшение количества операций копирования является очевидным способом ускорения кода. Большое количество копирований выполняется в связи с выделением памяти, поэтому исправление одной проблемы часто позволяет устранить другую. Еще одними "горячими точками" в смысле копирования являются конструкторы и операторы присваивания и ввода-вывода. Эта тема подробно рассматривается в главе 6, "Оптимизация переменных в динамической памяти".

Устранение вычислений

Если не считать распределение памяти и вызовы функций, стоимость одной инструкции C++ обычно незначительна. Но выполнение одного и того же кода миллион раз в цикле, или всякий раз, когда программа обрабатывает событие, может оказаться существенным. Большинство программ имеет один или несколько циклов

обработки событий и одну или несколько функций, которые обрабатывают символы. Идентификация и оптимизация этих циклов почти всегда оказывается плодотворной. В главе 7, "Оптимизация инструкций", предложено несколько советов о том, как найти часто выполняемый код. Можно спорить, что он всегда будет находиться в цикле.

В литературе по оптимизации содержится обилие методов эффективного использования отдельных инструкций C++. Многие программисты считают, что знание этих трюков является хлебом и маслом оптимизации. Но дело в том, что, если только этот код не выполняется очень часто, удаление из него одного или двух обращений к памяти не приводит к измеримой разнице в общей производительности. В главе 3, "Измерение производительности", содержатся методы, позволяющие определить, какие части программы выполняются чаще всего, чтобы уменьшить объем вычислений именно в этих местах.

Современные компиляторы C++ делают действительно выдающуюся работу по поиску и выполнению таких локальных усовершенствований. Поэтому разработчикам не следует пытаться менять все вхождения `i++` на `++i`, раскрывать все циклы и объяснять каждому коллеге, что такое устройство Даффа и почему оно такое замечательное. Тем не менее я делаю беглый обзор всего этого изобилия в главе 7, "Оптимизация инструкций".

Использование лучших структур данных

Выбор наиболее подходящей структуры данных имеет огромное влияние на производительность. Отчасти это связано с тем, что алгоритмы для вставки, итерации, сортировки и извлечения записей имеют стоимость выполнения, которая зависит от структуры данных. Кроме того, различные структуры данных по-разному используют диспетчер памяти. Кроме того, структура данных может иметь (или не иметь) хорошую локальность кеша. В главе 10, "Оптимизация структур данных", исследуются производительность, поведение и компромиссы между структурами данных, предоставляемыми стандартной библиотекой C++. В главе 9, "Оптимизация сортировки и поиска", обсуждается использование алгоритмов стандартной библиотеки для реализации табличных структур данных на основе простых векторов и массивов C.

Увеличение параллельности

Большинство программ вынуждены ожидать завершения некоторых действий в утомительном мире физической реальности. Они должны ждать, пока файлы будут считаны с механических дисков, страницы будут загружены из Интернета или пользователи медленными пальцами нажмут нужные клавиши. Все время, пока программы блокируются в ожидании такого события, упускаются возможности для выполнения некоторых вычислений.

Современные компьютеры имеют несколько ядер процессора, доступных для выполнения команд. Если распределить работу среди нескольких ядер, она может быть завершена быстрее.

Наряду с параллельным выполнением имеются инструменты для синхронизации параллельных потоков выполнения таким образом, чтобы они могли обмениваться данными. Эти инструменты также могут быть использованы хорошо или плохо. Глава 12, "Оптимизация параллельности", рассматривает некоторые соображения по поводу эффективного управления синхронизацией параллельных потоков.

Оптимизация управления памятью

Диспетчер памяти, являющийся частью библиотеки времени выполнения C++, которая управляет выделением динамической памяти, — это очень часто выполняемый код во многих программах на C++. Язык C++ имеет обширный интерфейс управления памятью, хотя большинство разработчиков никогда не используют все его возможности. В главе 13, "Оптимизация управления памятью", показаны некоторые способы улучшения производительности управления памятью.

Резюме

Данная книга помогает разработчику идентифицировать и использовать следующие возможности улучшения производительности кода.

- *Использовать лучшие компиляторы и их оптимизаторы.*
- *Использовать оптимальные алгоритмы.*
- *Использовать лучшие библиотеки и использовать их наиболее эффективно.*
- *Снизить количество распределений памяти.*
- *Снизить количество копирований.*
- *Устранить лишние вычисления.*
- *Использовать оптимальные структуры данных.*
- *Увеличить степень параллельности.*
- *Оптимизировать управление памятью.*

Как я уже говорил, дьявол прячется в подробностях. Итак, начнем.

ГЛАВА 2

Оптимизация, влияющая на поведение компьютера

Ложь, умение рассказывать прекрасные истории, каких никогда не случалось, составляет истинную цель Искусства.[1]

— Оскар Уайльд (Oscar Wilde), "Упадок лжи", *Намерения* (1891)

Цель настоящей главы — обеспечить минимум справочной информации об оборудовании компьютера для мотивации оптимизаций, описанных в этой книге, чтобы читателю не пришлось сходить с ума, углубляясь в 600-страничный справочник по процессору. Здесь приводится поверхностный обзор архитектуры процессора, позволяющий выделить некоторые эвристики для оптимизации. Очень нетерпеливый читатель может пропустить эту главу и вернуться к ней позже, когда другие главы будут ссылаться на материал из данной главы. Тем не менее приводимая здесь информация является важной и полезной для понимания остального материала книги.

Микропроцессорные устройства в настоящее время невероятно разнообразны. Они варьируются от дешевых встроенных устройств, состоящих всего лишь из нескольких тысяч логических вентилей с тактовой частотой ниже 1 МГц, до настольных устройств с миллиардами вентилей и частотами, измеряемыми в гигагерцах. Мощные ЭВМ могут иметь размер большой комнаты, содержать тысячи независимых исполнительных устройств и потреблять электроэнергию, достаточную для освещения небольшого города. Заманчиво считать, что ничто не связывает это изобилие вычислительных устройств, но в действительности они имеют очень много общего. В конце концов, если бы между ними не было никакого сходства, было бы невозможно компилировать код на C++ для множества процессоров, для которых имеются соответствующие компиляторы.

Все широко используемые компьютеры выполняют команды, хранящиеся в памяти. Эти команды обрабатывают данные, которые также хранятся в памяти. Память делится на множество небольших *слов* по нескольку битов. Несколько драгоценных слов памяти представляют собой *регистры*, которые именуются в машинных

[1] Перевод А. Зверева.

командах непосредственно. Большинство же слов памяти именуются с использованием их числового *адреса*. Определенный регистр в каждом компьютере содержит адрес следующей выполняемой команды. Если память рассматривать как книгу, то *выполняемый адрес* выглядит как палец, указывающий на следующее читаемое слово. *Исполнительное устройство* (именуемое также процессором, ядром, ЦПУ и кучей других слов) считывает поток команд из памяти и действует в соответствии с ними. Команды говорят исполнительному устройству, какие данные читать (загружать, выбирать) из памяти, что с ними делать и где в памяти записывать (запоминать, сохранять) результаты. Компьютер состоит из устройств, которые подчиняются физическим законам. Чтение или запись каждого адреса памяти занимает некоторое ненулевое количество времени, как и выполнение некоторых действий над считанными данными.

За пределами этой базовой схемы, знакомой любому первокурснику, генеалогическое дерево компьютерных архитектур демонстрирует буйный рост и изобилие ветвей. Поскольку компьютерные архитектуры весьма изменчивы, трудно сформулировать какие-либо строгие числовые правила в отношении поведения аппаратного обеспечения. Современные процессоры выполняют так много различных взаимодействующих операций, чтобы ускорить выполнение команд, что какие-то конкретные сроки их выполнения в общем случае указать невозможно. С учетом того, что многие разработчики даже не знают точно, на каких процессорах будет работать их код, лучшее, что можно ожидать в смысле оптимизации, — это некоторые эвристики.

Ложь о компьютерах, в которую верит C++

Конечно, программа на C++ по крайней мере делает вид, что верит в изложенную в предыдущем разделе версию простой модели компьютера. Существует память, адресуемая в байтах размера `char`, которая является по существу бесконечной. Существует специальный адрес, именуемый `nullptr`, который отличается от любого допустимого адреса памяти. Целое число `0` преобразуется в `nullptr`, хотя `nullptr` не обязательно указывает на адрес `0`. Существует единый концептуальный адрес выполнения, указывающий на инструкцию исходного кода, выполняемую в настоящее время. Инструкции выполняются в том порядке, в котором они написаны, с учетом действия операторов управления потоком выполнения C++.

C++ знает, что в действительности компьютеры сложнее, чем эта простая модель. Поэтому из сияющих внутренностей реализации C++ сквозь крышку пробиваются следующие сверкающие лучи.

- Программа на C++ должна лишь вести себя так, "как если бы" инструкции выполнялись в указанном порядке. Компилятор C++ и сам компьютер имеют право изменять порядок выполнения, чтобы добиться ускорения работы программы, если при этом не меняется смысл выполняемых вычислений.
- Что касается стандарта C++11, то C++ больше не считает, что существует только один адрес выполнения. Стандартная библиотека C++ теперь поставляется с возможностями запуска и остановки потоков выполнения и синхронизации

доступа к памяти между этими потоками. До C++11 программисты лгали компилятору C++ о потоках, что зачастую приводило к трудно отлаживаемым проблемам.

- Некоторые адреса памяти на самом деле вместо обычной памяти могут быть регистрами устройства. Значения в некоторых адресах могут изменяться между двумя последовательными чтениями одного и того же адреса в одном и том же потоке, отражая некоторые действия со стороны аппаратного обеспечения. Такие места описаны в C++ с помощью ключевого слова `volatile`. Объявление переменной как `volatile` требует от компилятора извлекать новую копию переменной всякий раз, когда она используется, вместо того, чтобы оптимизировать программу путем сохранения значения в регистре и его повторного использования. Могут быть также объявлены указатели на `volatile`-память.
- C++11 предлагает магическое заклинание `std::atomic<>`, которое заставляет память некоторое время вести себя так, как если бы это в действительности было простое линейное хранилище байтов, без учета всех сложностей современных микропроцессоров с их многочисленными потоками выполнения, многослойными кешами памяти и т.д. Некоторые разработчики считают, что для этого служит ключевое слово `volatile`, но они сильно ошибаются.

Операционная система также врет программам и их пользователям. На самом деле вся цель операционной системы — сообщить каждой программе набор очень убедительной лжи. Среди наиболее важной лжи — операционная система хочет убедить каждую программу, что она для компьютера единственная, что предоставляемая ей физическая память бесконечна и что есть бесконечное число процессоров, доступных для запуска потоков программы.

Операционная система умело использует аппаратное обеспечение компьютеров, чтобы скрывать свою ложь, так что программа на C++ не имеет иного выбора, кроме как поверить ей. Эта ложь обычно не слишком сильно влияет на работу программы, за исключением замедления ее работы. Однако она может осложнить выполнение измерений производительности программы.

Правда о компьютерах

Модели C++ соответствуют только простейшие микропроцессоры и некоторые древние ЭВМ. Что действительно важно для оптимизации — это то, что аппаратное обеспечение памяти реальных компьютеров очень медленное по сравнению со скоростью выполнения команд, что обращение к памяти в действительности не побайтное, что память не является простым линейным массивом одинаковых ячеек и что она имеет конечную емкость. Реальные компьютеры могут иметь более чем один адрес команды. Реальные компьютеры быстрые не потому, что они быстро выполняют каждую команду, а потому, что выполнение нескольких команд перекрывается, и сложные схемы следят за тем, чтобы такие перекрывающиеся команды вели себя так, как если бы они выполнялись одна за другой.

Медленная память

Основная память компьютера очень медленная по сравнению с его внутренними логическими вентилями и регистрами. Она медленная настолько, что процессор настольного компьютера может выполнить сотни команд за время, необходимое для извлечения из основной памяти единственного слова данных.

Следствием этого для оптимизации является то, что *доступ к памяти доминирует над другими действиями процессора*, включая выполнение команд.

Узкое место фон Неймана

Интерфейс к основной памяти является узким местом, которое ограничивает скорость работы. Это узкое место даже имеет имя. Оно называется узким местом (или бутылочным горлышком) фон Неймана, в честь знаменитого пионера компьютерной архитектуры и математика Джона фон Неймана (John von Neumann) (1903–1957).

Например, компьютер, оснащенный памятью DDR2 с частотой 1000 МГц (типичной для компьютеров несколько лет назад), имеет теоретическую пропускную способность 2 млрд слов в секунду, или 500 пс на слово. Но это не значит, что компьютер может считывать или записывать случайное слово данных каждые 500 пс.

Начнем с того, что за один такт (половина такта часов с частотой 1000 МГц) может выполняться только последовательный доступ. Доступ к не последовательным местам в памяти выполняется примерно за 6–10 тактов.

Далее, за доступ к шине памяти конкурируют несколько действий. Процессор постоянно проводит выборку очередных выполняемых команд из памяти. Контроллер кеш-памяти выполняет выборку блоков данных памяти для кеша и сбрасывает записанные строки кеша. Контроллер DRAM также требует циклы для обновления заряда в динамических ячейках устройства памяти. Числа ядер многоядерного процессора достаточно, чтобы гарантировать перенасыщенность шины памяти. Фактическая скорость, с которой данные могут быть прочитаны из основной памяти в определенное ядро, составляет более 20–80 нс на одно слово.

Закон Мура позволяет каждый год получать все больше и больше ядер в процессорах. Но чтобы обеспечить более быстрый интерфейс основной памяти, этого мало. Таким образом, удвоение числа ядер в будущем будет оказывать на производительность отрицательное влияние. Ядра будут бороться за доступ к памяти. Надвигающееся ограничение производительности называется *стеной памяти*.

Недоступность байтов

Хотя C++ считает, что каждый байт доступен по отдельности, компьютеры часто компенсируют медленность физической памяти путем выборки данных большими блоками. Самые мелкие процессоры могут выбирать из основной памяти отдельные байты, но процессоры настольных компьютеров могут выбирать за один раз по 64 байта. Некоторые суперкомпьютеры и графические процессоры выполняют выборку еще большего размера.

Когда C++ выбирает многобайтные данные, такие как тип `int`, `double` или указатель, может оказаться, что байты, составляющие данные, входят в два слова физической памяти. Это называется *невыровненным доступом к памяти*. Важным для оптимизации является то, что *невыровненный доступ требует в два раза больше времени*, чем если бы все байты были в одном и том же слове, поскольку при этом требуется чтение двух слов. Компилятор C++ выравнивает структуры таким образом, чтобы каждое поле начиналось с адреса байта, кратного размеру поля. Но это создает собственную проблему: "дыры" в структурах, содержащие неиспользуемые данные. Обращая внимание на размер полей данных и их порядок в структуре, можно сделать структуры максимально компактными при сохранении выровненности.

Одни обращения к памяти медленнее других

Для компенсации "медленности" основной памяти многие компьютеры содержат *кеш-память*, разновидность быстрой временной памяти, располагающейся близко к процессору, чтобы ускорить доступ к наиболее часто используемым словам памяти. Одни компьютеры обходятся без кеша; другие имеют один или несколько уровней кеша, каждый из которых меньше, быстрее и дороже предыдущего. Когда процессору нужны байты из слова кешированной памяти, они могут быть быстро извлечены без повторного обращения к основной памяти. Насколько кеш-память быстрее? Эмпирическое правило гласит, что каждый уровень кеш-памяти примерно в 10 раз быстрее, чем предыдущий уровень в иерархии памяти. На процессорах настольных компьютеров время доступа к памяти может меняться на пять порядков в зависимости от того, к чему осуществляется доступ: к кеш-памяти первого, второго или третьего уровня, к основной памяти или к странице виртуальной памяти на диске. Это одна из причин, по которым увлеченность тактами выполнения команд процессором и другими подобными тайнами так часто оказывается глупой и бесполезной — состояние кеша делает время выполнения команды весьма неопределенным.

Когда процессор нуждается в выборке данных, находящихся не в кеше, другие данные, расположенные в кеше, могут быть удалены из него для того, чтобы освободить место. Данные, выбираемые для удаления, как правило, являются наиболее давно использовавшимися. Это важно для оптимизации, поскольку означает, что *доступ к интенсивно используемым ячейкам памяти можно получить быстрее, чем к используемым реже*.

Чтение даже одного байта данных, который не находится в кеше, приводит к кешированию множества близлежащих байтов (как следствие это означает, что множество байтов, находящихся в настоящее время в кеше, удаляются из него).

Эти близлежащие байты будут готовы для быстрого доступа. Это важно для оптимизации, поскольку означает, что *обращение к соседним ячейкам памяти (в среднем) быстрее, чем к памяти в отдаленных местах.*

В терминах C++ это означает, что блок кода, содержащий цикл, может выполняться быстрее, поскольку инструкции, составляющие цикл, активно используются, располагаются близко одна к другой и, таким образом, вероятно, будут оставаться в кеше. Блок кода, содержащий вызовы функций или инструкции `if`, которые приводят к дальним переходам, могут выполняться более медленно, потому что используются части кода, далеко отстоящие одна от другой. Такой код использует больше пространства кеша, чем цикл. Если программа большая, а кеш конечен, часть кода будет удаляться из кеша, чтобы освободить место для других вещей, что приведет к замедлению доступа в следующий раз, когда потребуется этот код. Аналогично доступ к структуре данных, состоящей из последовательных местоположений в памяти, такой как массив или вектор, может быть быстрее, чем к структуре данных, состоящих из узлов, связанных указателями, поскольку данные в последовательных местоположениях с большей вероятностью будут оставаться в кеш-памяти. Доступ к структуре данных, состоящих из записей, связанных с помощью указателей (например, к списку или дереву), может быть медленнее из-за необходимости считывания данных каждого узла из основной памяти в новые строки кеша.

Остроконечные и тупоконечные слова

Из памяти может быть выбран один байт данных, но зачастую одновременно извлекается несколько последовательных байтов, образующих число. Например, в Microsoft Visual C++ значение типа `int` образуют четыре байта, считываемые из памяти вместе. Так как к памяти можно обращаться двумя способами, люди, которые проектируют компьютеры, должны ответить на важный вопрос: что содержится в первом байте (адрес которого наименьший) — старшие или младшие биты значения `int`?

На первый взгляд кажется, что это не должно иметь значения. Конечно, важно, чтобы все части компьютера одинаково считали, с какого конца `int` адрес меньше, иначе воцарится хаос. Вот в чем заключается разница между этими способами хранения. Если значение `int`, равное `0x01234567`, хранится по адресам 1000–1003 и первым хранятся старшие биты, то по адресу 1000 содержится байт `0x01`, а по адресу 1003 — байт `0x67`, в то время как если сначала хранится младший байт, то по адресу 1000 содержится `0x67`, а по адресу 1003 — `0x01`. Компьютеры, которые хранят старшие биты в байте с младшим адресом, называются компьютерами с *обратным порядком байтов* ("тупоконечниками", big-endian). Компьютеры с *прямым порядком байтов* ("остроконечники", little-endian) сначала читают младшие биты. Итак, имеется два способа хранения целого числа (или указателя), и нет никаких причин предпочесть один другому, так что разные команды, работающие на разных процессорах для разных компаний, могут делать разный выбор.

Проблемы начинаются, когда данные, записанные на диск или отправленные по сети одним компьютером, должны быть прочитаны другим компьютером. Диски и сети пересылают информацию побайтно, а не весь `int` одновременно. Поэтому

оказывается важно, какой конец числа сохраняется (или отправляется) первым. Если отправляющий и принимающий компьютеры не согласованы, то значение, отправляемое как `0x01234567`, может быть получено как `0x67452301`.

Порядок байтов является лишь одной из причин, по которым C++ не определяет, как биты располагаются в `int` или как значение одного поля в объединении влияет на другие поля. Это одна из причин, по которым программу можно написать так, что она будет успешно работать на компьютере одного вида, но приводить к аварийному завершению на другом.

Количество памяти ограничено

Память компьютера не бесконечна. Чтобы сохранить иллюзию бесконечной памяти, операционная система может использовать физическую память наподобие кеш-памяти и хранить данные, которые не помещаются в физическую память, в виде файла на диске. Эта схема называется *виртуальной памятью*. Виртуальная память создает иллюзию большего количества физической памяти. Однако получение блока памяти с диска занимает десятки миллисекунд — вечность для современного компьютера.

Быстрая кеш-память весьма дорогостоящая. В настольном компьютере или смартфоне может быть гигабайт памяти, но кеш размером лишь в несколько мегабайтов. Программы и их данные обычно в кеш не помещаются.

Результатом кеширования и применения виртуальной памяти может быть то, что *из-за кеширования определенная функция, работающая в контексте всей программы, может выполняться медленнее, чем та же функция в тестовой программе*, в 10 тысяч раз. В контексте всей программы функции и данные не могут все время оставаться в кеше, в то время как в контексте теста это именно так и происходит. Этот эффект усиливает преимущества оптимизаций, которые снижают использование памяти или диска, в то время как преимущества оптимизаций, которые уменьшают размер кода, остаются небольшими.

Вторым результатом кеширования является то, что если большая программа выполняет рассеянный доступ ко многим участкам памяти, то кеш-памяти может оказаться недостаточно для хранения данных, непосредственно используемых программой. Это приводит к снижению производительности, которое называют *пробуксовкой страницы*. Когда пробуксовка страницы происходит во внутреннем кеше микропроцессора, результатом является снижение производительности. Когда это происходит в файле виртуальной памяти операционной системы, производительность падает тысячекратно. Эта проблема возникала чаще, когда физическая память была дороже и меньше по размеру, но она встречается и сейчас.

Медленное выполнение команд

Простые микропроцессоры наподобие встроенных в кофеварки и микроволновые печи предназначены для выполнения команд с той же скоростью, с которой они могут извлекаться из памяти. Микропроцессоры настольного компьютера имеют дополнительные ресурсы для параллельной обработки нескольких команд, поэтому они способны выполнять команды во много раз быстрее, чем те могут быть извлечены из

основной памяти, так что большую часть времени для хранения команд и их передачи процессору используется быстрая кеш-память. Важность этого для оптимизации заключается в том, что *время доступа к памяти превышает время вычислений*.

Современные настольные компьютеры выполняют команды с удивительной скоростью, *если* им ничто не мешает. Они могут завершать команды каждые несколько сотен пикосекунд (пикосекунда представляет собой 10^{-12} с, до смешного короткое время). Но это не значит, что каждая команда выполняется так быстро. Процессор содержит "конвейер" одновременно выполняемых команд. Команды проходят через конвейер, дешифруются, получают свои аргументы, выполняют вычисления и сохраняют результаты. Чем более мощный процессор, тем более сложен его конвейер, разбивающий выполнение команды на десяток этапов так, чтобы как можно больше команд могли быть обработаны одновременно.

Если команда А вычисляет значение, которое требуется команде Б, то команда Б не может выполнить свое вычисление до тех пор, пока команда А не даст необходимый результат. Это приводит к *остановке конвейера*, короткой паузе в выполнении команд, которая возникает, когда выполнение двух команд не может полностью перекрываться. Остановка конвейера в особенности долгая, если команда А извлекает значение из памяти, а затем выполняет вычисление, которое дает значение, необходимое команде Б. Остановке конвейера подвержены все современные микропроцессоры, что делает их время от времени почти такими же медленными, как процессор в вашем тостере.

Трудное принятие решений

Еще одна вещь, которая может вызвать остановку конвейера, — это принятие решения компьютером. После большинства команд выполнение продолжается с команды, находящейся в памяти по следующему адресу. Большую часть времени эта следующая команда уже находится в кеше. Последовательные команды можно загружать в конвейер, как только на первом этапе конвейера для этого появляется место.

Но не таковы команды передачи управления. Команда перехода или вызова подпрограммы заменяет адрес выполнения произвольным новым значением. "Следующая" команда не может быть считанной из памяти и попасть в конвейер до тех пор, пока в некоторый момент в процессе обработки команды перехода не будет обновлен адрес выполнения. У слова памяти по новому адресу выполнения меньше шансов находиться в кеше. Конвейер останавливается на время обновления адреса выполнения и загрузки в конвейер новой "следующей" команды.

После команды условного ветвления выполнение продолжается в одном из двух разных мест: выполняется либо следующая команда, либо команда по адресу, который является целевым для команды ветвления, в зависимости от результатов некоторых предыдущих вычислений. Конвейер останавливается, пока не будут завершены все команды, участвующие в предыдущих вычислениях, и остается в этом состоянии до тех пор, пока не будет определен следующий выполняемый адрес и не будет прочитана соответствующая команда по этому адресу.

Важность этого явления для оптимизации заключается в том, что *вычисление быстрее принятия решения*.

Множественные потоки выполнения

Любая программа, работающая в современной операционной системе, разделяет компьютер с другими программами, выполняемыми в то же время, с периодическими обслуживающими процессами типа проверки диска или поиска обновлений Java или Flash, и с различными частями операционной системы, управляющими сетевым интерфейсом, дисками, звуковыми устройствами и другими периферийными устройствами. *Каждая программа конкурирует с другими программами за ресурсы компьютера.*

Программа обычно не слишком об этом осведомлена. Она просто работает немного медленнее. Исключением является ситуация, когда одновременно запускается много программ, и все они конкурируют за память и диск. Для настройки производительности, *если программа должна запускаться при запуске системы или в периоды пиковой нагрузки, ее производительность должна измеряться под нагрузкой.*

По состоянию на начало 2016 года настольные компьютеры имеют до 16 ядер процессора, а микропроцессоры, используемые в телефонах и планшетах, — до восьми. Беглый взгляд на диспетчер задач Windows, вывод состояния процесса Linux или список задач Android обычно показывает гораздо больше процессов, чем ядер, и большинство процессов имеют несколько потоков выполнения. Операционная система выполняет каждый поток в течение короткого времени, а затем переключает контекст на другой поток или процесс. С точки зрения программы это выглядит, как если бы одна команда выполнялась наносекунду, а следующая — 60 мс.

Что означает переключение контекстов? Если операционная система переключается от одного потока к другому в одной программе, это означает сохранение регистров процессора для приостановки потока и загрузки сохраненных регистров для возобновления другого потока. Регистры современного процессора содержат сотни байтов данных. Когда новый поток возобновляет выполнение, данные могут не быть в кеше, так что имеется начальный период медленного выполнения, пока новый контекст загружается в кеш. Таким образом, при переключении контекстов потоков имеются значительные затраты.

Процедура переключения операционной системой контекста от одной программы к другой еще более дорогостоящая. Все "грязные" страницы кеша (с записанными данными, которые еще не внесены в основную память) должны быть сброшены в физическую память, а все регистры процессора сохранены. Затем сохраняются регистры страниц отображения физической памяти на виртуальную в диспетчере памяти. Далее для нового процесса загружаются соответствующие регистры памяти и регистры процессора. И наконец выполнение программы может возобновиться. Но кеш в этот момент пуст, так что начальный период характеризуется низкой производительностью и конфликтами памяти.

Когда программа должна ожидать некоторое событие, это ожидание может продолжаться даже после того, как это событие произойдет, пока операционная система не освободит процессор для продолжения программы. При выполнении программы в контексте работы других программ, конкурирующих за ресурсы компьютера, это может увеличить время выполнения программы и сделать его более неопределенным.

Исполнительные устройства многоядерных процессоров и связанная с ними кеш-память для достижения лучшей производительности работают более или менее независимо друг от друга. Однако все исполнительные устройства имеют одну и ту же основную память. Они вынуждены конкурировать за доступ к оборудованию, связывая его с основной памятью, что делает узкое место фон Неймана в компьютере с несколькими исполнительными устройствами еще более ограничивающим.

Когда исполнительное устройство записывает значение, оно сначала попадает в кеш-память. В конечном итоге оно должно быть записано из кеша в основную память, так что это значение станет видимым для других исполнительных устройств. Однако из-за конфликтов доступа к основной памяти среди исполнительных устройств основная память может не обновляться в течение сотен команд после того, как значение было изменено.

Если на компьютере имеется несколько исполнительных устройств, то одно из них может, таким образом, на протяжении длительного периода времени не увидеть данные, записанные другим устройством в основной памяти, а изменения в основной памяти могут произойти не в том же порядке, что и порядок выполнения команд. В зависимости от непредсказуемых временных факторов исполнительное устройство может увидеть как старое значение общей памяти, так и обновленное значение. Для того чтобы различные исполнительные устройства видели согласованные представления памяти, должны использоваться специальные команды синхронизации. Значение этого эффекта для оптимизации состоит в том, что *обращение к совместно используемым потоками выполнения данным гораздо медленнее, чем к не разделяемым данным.*

Вызовы операционной системы являются дорогостоящими

Все процессоры, кроме самых мелких, имеют аппаратное обеспечение для обеспечения изоляции между программами, так что программа А не может читать, писать или выполнять команды в физической памяти, принадлежащей программе Б. То же самое оборудование защищает ядро операционной системы от перезаписи программами. С другой стороны, ядру операционной системы требуется доступ к памяти, принадлежащей каждой программе, чтобы эти программы могли делать системные вызовы операционной системы. Некоторые операционные системы также позволяют программам делать запросы для совместного использования памяти. Способы организации системных вызовов и совместно используемой памяти разнообразны и полны тайной магии. С точки зрения оптимизации важным является то, что *системные вызовы дороги*, в сотни раз дороже, чем вызовы функций внутри одного потока одной программы.

C++ тоже лжет

Самая большая ложь C++ в том, что он рассказывает своим пользователям, что компьютер, который выполняет программу, представляет собой простую последовательную структуру. В обмен на то, что разработчики делают вид, что верят в эту ложь, C++ позволяет разработчикам программировать без знаний интимных подробностей каждого микропроцессорного устройства, которые совершенно необходимы при использовании языка ассемблера.

Не все инструкции одинаково дорогие

В мирные, давно минувшие дни программирования на языке C Кернигана и Ритчи каждая инструкция была примерно такой же дорогой, как и любая другая. Вызов функции может содержать вычисления произвольной сложности. Однако оператор присваивания в общем случае копирует нечто, помещающееся в машинном регистре, во что-то другое, что может хранить содержимое регистра компьютера. Таким образом, инструкция

```
int i,j;
...
i = j;
```

копирует 2 или 4 байта из `j` в `i`. Объявление может быть `int`, `float` или `struct big struct*`, но инструкция присваивания все равно выполняет одно и то же количество работы.

В настоящее время это не так. В C++ присвоение одного `int` другому представляет собой точно такой же объем работы, как и в случае соответствующей инструкции C. Но инструкция наподобие `BigInstance i = OtherObject;` может копировать целые структуры. Более того, этот вид присваивания вызывает конструктор `BigInstance`, который может скрывать произвольно сложный механизм. Конструктор вызывается также для каждого выражения, переданного функции в качестве формального аргумента, и вновь, когда функция возвращает значение. Арифметические операторы и операторы сравнения также могут быть перегружены, так что выражение `A = B*C;` может умножать *n*-мерные матрицы, а `if (x<y) ...` может сравнивать два пути через ориентированный граф произвольной сложности. Значение этого для оптимизации заключается в том, что *некоторые инструкции скрывают большие количества вычислений. Вид инструкции ничего не говорит об ее стоимости.*

Разработчики, которые начинали изучение языков программирования с C++, могут не увидеть в этом ничего удивительного; но для тех, кто начинал с изучения C, их инстинкты могут привести к катастрофическим заблуждениям.

Инструкции выполняются не по порядку

Программы на C++ ведут себя так, как если бы они выполнялись в порядке, указанном инструкциями управления потоком выполнения C++. Хитрая оговорка "как если бы" в предыдущем предложении является главной, на которой построено множество оптимизаций и трюков современного компьютерного оборудования.

За кулисами компилятор может — и зачастую так и делает — переупорядочивать инструкции для повышения производительности. Но компилятор знает, что переменная должна содержать последний результат вычисления, присвоенный до проверки или присваивания его другой переменной. Современные микропроцессоры также могут выбрать команды для выполнения не по порядку их следования, но они содержат логические схемы, которые гарантируют выполнение записи в память до последующего чтения того же места. Логика управления памятью микропроцессора может даже выбрать задержку записи в память для оптимального использования шины памяти. Однако контроллер памяти знает, что именно в настоящее время находится в

процессе записи от исполняющего устройства через кеш-память в основную память, и гарантирует, что если будет считываться тот же адрес, то будет получено корректное значение, даже если его запись выполнена еще не до конца.

Параллелизм усложняет эту картину. Программы на C++ компилируются без знания о других потоках, которые могут выполняться одновременно с данным. Компилятор C++ не знает, какие переменные, если таковые имеются, совместно используются потоками. Совокупный эффект изменения порядка инструкций компилятором и компьютером и задержка записи в основную память разрушают иллюзию выполнения инструкций в указываемом программой порядке, если программа содержит параллельные потоки, которые совместно используют данные. Разработчик должен добавить явную синхронизацию кода многопоточных программ для гарантии получения согласованного предсказуемого поведения. *Синхронизация кода снижает степень параллелизма, получаемого при совместном использовании данных параллельными потоками.*

Резюме

- *Доступ к памяти доминирует над другими действиями процессора.*
- *Невыровненный доступ требует в два раза больше времени, чем когда все байты находятся в одном и том же слове.*
- *Доступ к интенсивно используемым ячейкам памяти можно получить быстрее, чем к используемым реже.*
- *Обращение к соседним ячейкам памяти выполняется быстрее, чем к памяти в отдаленных местах.*
- *Из-за кеширования определенная функция, работающая в контексте всей программы, может выполняться медленнее, чем та же функция в тестовой программе.*
- *Обращение к совместно используемым потоками выполнения данным выполняется гораздо медленнее, чем к не разделяемым данным.*
- *Вычисления осуществляются быстрее принятий решений.*
- *Каждая программа конкурирует с другими программами за ресурсы компьютера.*
- *Если программа должна запускаться при запуске системы или в периоды пиковой нагрузки, ее производительность должна измеряться под нагрузкой.*
- *Каждое присваивание, инициализация аргумента функции и возврат из функции вызывает конструктор — функцию, которая может скрывать произвольно большое количество кода.*
- *Некоторые инструкции скрывают большие количества вычислений. Вид инструкции ничего не говорит об ее стоимости.*
- *Синхронизация кода снижает степень параллелизма, получаемого при совместном использовании данных параллельными потоками.*

ГЛАВА 3

Измерение производительности

Измеряй все измеримое и делай измеримым не являющееся таковым.
— Галилео Галилей (Galileo Galilei) (1564–1642)

Измерения и эксперименты являются основой любой серьезной попытки улучшить производительность программы. В этой главе представлены два программных инструмента, которые измеряют производительность: профайлер и программный таймер. Рассмотрим, как спроектировать эксперименты по измерению производительности так, чтобы получить результаты значимые, а не вводящие в заблуждение.

Самые главные и наиболее часто выполняемые измерения производительности программного обеспечения отвечают на вопрос "Как долго?" Сколько времени выполняется функция? Как долго считывается конфигурация с диска? Сколько времени выполняется запуск или завершение программы?

На эти вопросы можно попытаться ответить (пусть и неуклюже) с помощью до смешного простых инструментов. Исаак Ньютон пытался измерять гравитационную постоянную по времени падения объектов, измеряемому с помощью сердцебиения. Я уверен, что каждый разработчик время от времени начинал считать (вслух или про себя) "двадцать один, двадцать два...", чтобы получить приблизительное количество прошедших секунд. Цифровые наручные часы с секундомером — не веяние моды, а *обязательный* атрибут программиста. В мире встроенных устройств разработчики имеют в своем распоряжении отличные инструменты, включая частотомеры и осциллографы, которые могут точно измерять время выполнения даже очень коротких процедур. Производители программного обеспечения выпускают и продают различные специализированные инструменты, которых слишком много даже для беглого обзора в этой книге.

Эта глава сосредоточена на двух широко доступных инструментах, в общем случае полезных и недорогих. Первый инструмент, *профайлер*, обычно поставляется вместе с компилятором. Профайлер создает табличный отчет по совокупному времени, затраченному на выполнение каждой функции, вызываемой во время работы программы. Это важный инструмент оптимизации программного обеспечения, который создает список узких мест в программе.

Второй инструмент — *программный таймер* — представляет собой инструмент, который могут создавать сами разработчики, как рыцари-джедаи создают свои световые мечи (если вы простите ссылку на "Звездные войны"). Если цена компилятора с профайлером слишком велика или если поставщик компилятора не предлагает такового, разработчик все равно может проводить эксперименты по выяснению времени выполнения достаточно длительных действий. Программный таймер полезен также для выяснения продолжительности задач, которые не являются вычислениями.

Третий инструмент, лабораторный журнал, имеет столь почтенный возраст, что многие разработчики полагают, что он полностью вышел из употребления. Однако лабораторный журнал (или эквивалентный текстовый файл) является незаменимым инструментом в наборе инструментов оптимизации.

Оптимизирующее мышление

Прежде чем погрузиться в измерения и эксперименты, я хотел бы немного поговорить о практикуемой мною философии оптимизации, которой я надеюсь обучить вас в этой книге.

Производительность должна быть измерена

Чувства человека, как правило, недостаточно точны для того, чтобы обнаружить нарастающие изменения в производительности. Ваша память может не позволить вам точно вспомнить результаты многих экспериментов. Учебник может обмануть вас, заставив поверить не всегда верным утверждениям. Интуиция часто подводит разработчиков, когда они решают, следует ли оптимизировать определенную часть кода. Они пишут функции, зная, что те будут использованы, но мало заботясь о том, как часто или каким кодом они будут использованы. Затем неэффективный фрагмент кода попадает в критический компонент, где он вызывается миллиард раз. Опыт также может вас подвести. Языки программирования, компиляторы, библиотеки и процессоры — все это находится в постоянном развитии. Функции, которые ранее были узкими местами, могут стать более эффективными (но может случиться и обратное). Только измерение скажет вам, выиграли вы или проиграли в игре оптимизации.

Разработчики, навыки которых в области оптимизации я уважаю более других, подходят к задаче оптимизации систематически.

- Они высказывают проверяемые предположения и записывают свои прогнозы.
- Они ведут учет изменений кода.
- Они выполняют измерения с наилучшими доступными инструментами.
- Они подробно описывают полученные экспериментальные результаты.

Остановись и подумай

Вернитесь назад и прочтите предыдущий раздел еще раз. Он содержит наиболее важные советы в этой книге. Большинство разработчиков (включая автора) не ведут себя столь методически — это навык, который необходимо практиковать.

Оптимизация — большая игра

Я требую взлететь и взорвать всю станцию с орбиты. Это единственный надежный способ.

— Элен Рипли (Сигурни Уивер (Sigourney Weaver)), *к/ф "Чужие"*, 1986

Оптимизация — игра с большими ставками. Стоит ли ускорение программы на 1% риска внести ошибки при изменении рабочей программы? Изменения должны быть существенными по крайней мере локально, чтобы считать их полезными. Кроме того, ускорение в 1% может оказаться всего лишь ошибкой измерения, которая будет принята за улучшение. Любое ускорение должно быть *доказано* с применением рандомизации, статистических методов и определением доверительного интервала. Не слишком ли много работы для слишком малого эффекта? Это не наш путь, не то, ради чего я взялся за написание данной книги.

Улучшение на 20% — дело совсем иное. Оно развеивает все возражения по поводу методологии. В книге не так уж много статистических данных, и я не прошу за это прощения. Смысл книги в том, чтобы помочь разработчику найти возможности улучшения производительности, достаточно существенные для того, чтобы перевешивать любой вопрос их стоимости. Это улучшение по-прежнему может зависеть от множества факторов, таких как операционная система и компилятор, поэтому оно может не оказать большого влияния в другой системе или в другое время. Но такие большие изменения почти никогда не приводят к снижению производительности при переносе кода на новую систему.

Правило 90/10

Фундаментальным правилом оптимизации является правило 90/10: 90% *времени работы программа затрачивает на выполнение 10% кода*. Это правило эвристическое; это не закон природы, а скорее полезное обобщение для размышлений и планирования. Это правило иногда называют правилом 80/20; впрочем, идея остается той же. Интуитивно правило 90/10 означает, что определенные блоки кода представляют собой "горячие точки", которые выполняются очень часто, в то время как другие части кода не выполняются практически никогда. Эти горячие точки и являются целями для приложения усилий по оптимизации.

Из истории оптимизационных войн

С правилом 90/10 как профессиональный разработчик я впервые столкнулся в одном из моих первых проектов — встроенного устройства с клавиатурой, которое совершенно случайно называлось 9010A (рис. 3.1).

Рис. 3.1. Fluke 9010A (British Computer History Museum)

В нем имелась функция опроса клавиатуры, предназначенная для того, чтобы увидеть, не нажата ли клавиша STOP. Эта функция часто вызывалась каждой подпрограммой. Ручная оптимизация кода одной лишь этой функции на языке ассемблера Z80, сгенерированного компилятором C (я затратил на это 45 минут), повысила общую производительность на 7%, что для данного устройства было весьма существенно.

Эта ситуация в общем случае типична для проблемы оптимизации. В начале процесса оптимизации очень много времени выполнения тратилось в одном месте программы. Это место было довольно очевидным: служебные действия выполнялись многократно, на каждой итерации каждого цикла. Оптимизация требовала работы с кодом на языке ассемблера вместо C. Однако выполненная работа на языке ассемблера была очень ограниченной, чтобы уменьшить риск, который влечет за собой выбор этого языка.

Ситуация была типична и в том, что горячей точкой был только один блок кода. После того, как был улучшен этот код, первое место занял другой блок кода, но его вклад в общее время выполнения был гораздо меньшим. Фактически он был настолько мал, что мы прекратили оптимизацию после внесения всего лишь одного описанного выше изменения. Мы не могли обнаружить никаких других возможностей оптимизации, которые дали бы нам хотя бы 1% ускорения.

Из правила 90/10 следует, что оптимизация каждой процедуры в программе бессмысленна. Оптимизация малой части кода дает практически все улучшения производительности, которые можно получить. Выявление "горячих" 10% кода — с пользой проведенное время. Выбор кода для оптимизации на основании предположений, скорее всего, будет временем, потраченным впустую.

Я хотел бы вернуться к цитате Кнута из главы 1, "Обзор оптимизации". Вот более длинная версия той же цитаты.

> *Программисты тратят огромное количество времени, размышляя (или беспокоясь) о скорости некритических частей их программ, и эти попытки повышения эффективности на самом деле имеют сильное отрицательное влияние на отладку и обслуживание. Мы не должны помнить о малой эффективности, скажем, около 97 процентов времени: преждевременная оптимизация является корнем всех зол.*
>
> — Дональд Кнут (Donald Knuth), *Structured Programming with go to Statements*, ACM Computing Surveys 6 (Dec 1974): 268. CiteSeerX: 10.1.1.103.6084 (`http://bit.ly/knuth-1974`)

Доктор Кнут не говорит, что оптимизация в общем случае есть зло, как считают некоторые. Он сказал лишь, что злом является тратить время на оптимизацию некритических 90% программы. Видимо, ему тоже было известно о правиле 90/10.

Закон Амдала

Закон Амдала, придуманный одним из пионеров вычислительной техники Джином Амдалом (Gene Amdahl) и названный в его честь, описывает, насколько повысится общая производительность при оптимизации части кода. Существует несколько способов выражения закона Амдала, но что касается оптимизации, он может быть выражен с помощью формулы

$$S_T = \frac{1}{(1-P) + \frac{P}{S_P}}$$

Здесь S_T — улучшение времени выполнения программы в целом в результате оптимизации, P — доля оптимизированного общего времени выполнения, а S_P — показатель улучшения в оптимизированной части P.

Предположим, например, что выполнение программы занимает 100 с. Посредством профилирования (см. раздел "Профилирование выполнения программы" далее в главе) вы обнаружили, что программа тратит 80 с на выполнение нескольких вызовов одной функции `f`. Теперь предположим, что вы переписали функцию `f`, сделав ее на 30% быстрее. Насколько это улучшит общее время выполнения программы?

P, часть исходного времени выполнения функции `f`, равно 0,8. S_P равно 1,3. Подстановка этих величин в закон Амдала дает

$$S_T = \frac{1}{(1-0.8) + \frac{0.8}{1.3}} \approx 1.22$$

Повышение производительности одной этой функции на 30% увеличивает производительность всей программы на 22%. В этом случае закон Амдала проиллюстрировал правило 90/10 и предоставил пример того, насколько существенным может быть улучшение в 10% кода.

Давайте рассмотрим еще один пример. Вновь будем считать, что общее время выполнения программы равно 100 с и профилирование выяснило, что программа тратит 10 с на выполнение нескольких вызовов одной функции `g`. Мы переписали ее так, что теперь она работает в 100 раз быстрее. Насколько повысится производительность программы в этом случае?

Часть общего времени выполнения, которое программа затрачивает на функцию `g`, составляет $P = 0.1$, а $S_p = 100$. Подстановка этих значений в закон Амдала дает

$$S_T = \frac{1}{(1-0.1)+\frac{0.1}{100}} \approx 1.11$$

В этом случае закон Амдала, по сути, предостерегает нас. Даже если героические усилия программиста или применение черной магии сведет время работы функции `g` к нулю, она все равно останется в неважных 90% программы. Общее улучшение производительности все равно составит 11%, с точностью до двух знаков после запятой. Закон Амдала гласит, что даже самая успешная оптимизация не является существенной, если оптимизированный код не составляет бо́льшую часть времени выполнения программы в целом. Уроком закона Амдала является, что если ваш коллега приходит на рабочую встречу команды и сообщает, что он ускорил некоторые вычисления в 10 раз, это вовсе *не обязательно* значит, что все ваши проблемы производительности решены.

Проведение экспериментов

Разработка программного обеспечения — это всегда эксперимент, в том смысле, что вы начинаете писать программу, думая, что она будет делать некоторые конкретные вещи, а затем смотрите, что же она делает на самом деле. Настройка производительности является экспериментом в более формальном смысле этого понятия. Вы должны начинать работу с правильного кода, в том смысле, что это код, который делает то, что вы ожидаете и что от него требуется. Вы должны посмотреть на этот код новым взглядом и задаться вопросом "Где в этом коде узкое место?" Почему эта конкретная функция из многих сотен функций вашей программы появляется в верхней части списка профайлера? Эта функция тратит время на выполнение чего-то лишнего? Есть ли более быстрый способ выполнить те же вычисления? Данная функция использует дорогостоящие ресурсы? Эта функция на самом деле предельно эффективна, насколько это вообще возможно, но просто вызывается слишком часто, что не оставляет никакой возможности для повышения ее производительности?

Ваш ответ на вопрос "Где в этом коде узкое место?" формирует гипотезу, которую вы будете проверять. Эксперимент принимает форму двух измерений времени выполнения программы: одно — до внесения изменений в программу, а второе — после.

Если второе измерение дает значение, меньшее первого, эксперимент *подтверждает* гипотезу.

Обратите внимание на примененную терминологию. Эксперимент не обязательно *доказывает* что-либо вообще. Измененный код может выполняться быстрее или медленнее по разным причинам, не имеющим ничего общего с внесенными вами изменениями, например:

- в момент проведения вами измерений компьютер может получать почту или проверять наличие обновлений Java;
- коллега мог установить обновленную библиотеку как раз перед тем, как вы перекомпилировали программу с изменениями;
- ваши изменения дают код, работающий более быстро, но с неверными результатами.

Хорошие ученые всегда скептики. Они всегда подозрительны. Если ожидаемое улучшение не наступает или проявляется слишком хорошо, чтобы это было правдой, скептик ставит новые эксперименты, пересматривает свои исходные предположения или ищет ошибки.

Хорошие ученые всегда открыты для новых знаний, даже если они противоречат знаниям, уже имеющимся в их головах. В процессе написания этой книги я узнал несколько неожиданных для себя вещей о оптимизации. Технические рецензенты тоже узнали много нового. Хорошие ученые никогда не прекращают учиться!

Из истории оптимизационных войн

В главе 5, "Оптимизация алгоритмов", имеется пример функции для поиска ключевых слов. Я написал несколько версий примера. Один использовал линейный поиск, другой — бинарный. Когда я измерил производительность этих двух функций, линейный поиск все время оказывался на несколько процентов *быстрее* бинарного поиска. Это показалось мне нонсенсом. Бинарный поиск просто *обязан* быть быстрее. Но цифры говорили совсем о другом.

Я читал, как кто-то в Интернете говорил, что линейный поиск часто оказывается быстрее потому, что лучше использует локальность кеша, чем бинарный поиск. Действительно, моя реализация линейного поиска должна была иметь отличную локальность кеша. Но этот результат противоречил всему опыту и всем почтенным книгам, посвященным производительности поиска.

При более глубоком изучении вопроса я понял, что тестировал поиск на таблицах только с несколькими ключевыми словами и использовал в качестве искомых слова, которые в моих экспериментах гарантированно имелись в таблице. Если таблица содержит восемь записей, линейный поиск в среднем просматривает половину из них (четыре) перед возвратом результата.

Бинарный поиск будет делить таблицу пополам три раза при каждом вызове. Так что эти два алгоритма имеют одинаковую среднюю производительность на малых наборах ключевых слов. Эта реальность противоречила моей интуиции, которая искренне считала, что бинарный поиск будет лучшим "всегда".

Но это был не тот результат, который я хотел продемонстрировать! Так что я сделал таблицу побольше, рассчитывая, что должен быть некоторый размер, при котором бинарный поиск будет проходить быстрее. Я также добавил к тесту несколько слов, отсутствующих в таблице. Результаты, как и раньше, были в пользу линейного поиска. В этот раз мне пришлось отложить эту задачу на пару дней, но результат не давал мне покоя.

Я все еще был уверен, что бинарный поиск должен быть быстрее. Я пересмотрел модульные тесты для обоих поисков, как оказалось, для того, чтобы обнаружить, что линейный поиск всегда возвращает успешный результат после первого же сравнения. Мой тест проверял не корректный возврат, а просто ненулевое значение. Обзывая себя последними словами, я исправил код линейного поиска и тестов. Эксперимент немедленно подтвердил ожидаемый результат: бинарный поиск быстрее линейного.

В этом случае экспериментальные результаты сначала отвергли, но позже подтвердили мою гипотезу, хотя и бросили попутно вызов моим представлениям о поиске.

Ведение лабораторного журнала

Хорошие оптимизаторы (как и все хорошие ученые) заинтересованы в повторяемости результатов. Здесь в игру вступает такая вещь, как лабораторный журнал. Каждый хронометраж начинается с гипотезы, одной или нескольких настроек кода и набора входных данных, а заканчивается ничем не примечательным числом миллисекунд. Не так уж трудно запомнить время, которое показал предыдущий запуск, чтобы сравнить его со следующим. Пока каждое изменение кода оказывается успешным, этого достаточно.

Но в конце концов разработчик где-то поступит неправильно, и время последнего запуска оказывается хуже времени предыдущего запуска. Перед разработчиком будет стоять масса вопросов. Был ли запуск № 5 быстрее запуска № 3, несмотря на то что он оказался медленнее запуска № 4? Какое изменение кода было выполнено перед запуском № 3? Разница в скорости в пределах погрешности или программа действительно работает быстрее?

Если каждый эксперимент задокументирован, ответить на такие вопросы несложно. Документирование делает ответы на эти вопросы тривиальными. В противном случае разработчик должен вернуться и повторно выполнить предыдущий эксперимент, чтобы получить интересующее время работы, т.е., конечно, *если* он может

точно вспомнить, какие изменения кода были сделаны. Если выполнение программы — несложная задача, а память у разработчика хорошая, то ему повезло и он может затратить лишь небольшое время. Но так может и не повезти, и будет потеряно перспективное направление работы или просто рабочий день.

Когда я даю этот совет, всегда находится кто-то, кто высокомерно заявляет "Я могу сделать это без бумаги! Я могу написать сценарий Perl, который будет изменять команду foo с помощью инструмента Smart-foo так, чтобы сохранять результаты каждого хронометража с набором выполненных изменений. Если я сохраню тестовые результаты в файл... Если я выполню тест в каталоге, то..."

Я не хочу препятствовать инновациям разработчиков программного обеспечения. Если вы старший менеджер, который полагает этот способ наилучшим, поступайте как знаете. Замечу, однако, что письмо на бумаге является надежным и простым в использовании методом, проверенным тысячелетиями. Он будет работать, если команда обновит систему управления версиями или систему тестирования. Он будет работать и при выполнении нового задания разработчика. Это древнее решение может работать и сегодня и, вероятно, будет так же хорошо служить разработчикам завтра.

Измерение базовой производительности и постановка целей

Для разработчиков, работающих в одиночку в собственное рабочее время, процесс оптимизации может быть случайным, выполняемым до тех пор, пока производительность не станет "выглядеть достаточно хорошей". Однако разработчики, работающие в группах, имеют руководство и должны удовлетворять заинтересованные стороны. Усилия по оптимизации руководствуются двумя показателями: базовой производительностью до оптимизации и целевой производительностью. Базовая производительность важна не только для измерения успеха отдельных улучшений, но и для обоснования стоимости усилий по оптимизации для заинтересованных сторон.

Целевая производительность важна, поскольку оптимизация представляет собой процесс с убывающей отдачей. Изначально есть, так сказать, "низко висящие плоды", которые легко достать — просто протянув руку: отдельные процессы или наивно закодированные функции, которые обеспечивают большие возможности повышения производительности при оптимизации. Но как только эти улучшения выполнены, для каждого последующего шага оптимизации требуется все больше и больше усилий.

Многие команды изначально не думают об установке целевых производительности и гибкости просто потому, что не привыкли делать это. К счастью, низкая производительность обычно очевидна (пользовательский интерфейс с длинными периодами "зависания", сервер, который не в состоянии справляться с высокими нагрузками, чрезмерные расходы процессорного времени и др.). После того как команда обратит свое внимание на производительность приложения, будет не так уж сложно определить количественные цели. Имеется целая наука о том, как пользователи воспринимают время ожидания. Для начала взгляните на следующий список часто встречающихся аспектов производительности, а также на критерии, которые говорят о наличии проблем.

Время запуска

Время, которое проходит от нажатия клавиши <Enter> до момента, когда программа входит в свой основной цикл обработки сообщений. Часто, хотя и не всегда, разработчик может просто измерить время, проходящее между входом в функцию `main()` и входом в основной цикл. Производители операционных систем, которые предлагают выполнение сертификации программ, имеют строгие требования в отношении программ, которые запускаются при запуске компьютера или при каждом входе пользователя в систему. Например, Microsoft требует от поставщиков аппаратного обеспечения, чтобы оболочка Windows входила в свой основной цикл менее чем за 10 секунд после запуска. Это ограничивает количество программ, которые производителю позволено предварительно загружать и запускать в занятой запуском среде. Корпорация Майкрософт предлагает специализированные инструменты для измерения времени запуска.

Время выключения

Время, которое проходит с момента, когда пользователь щелкает на пиктограмме Close или вводит команду выхода, до фактического завершения работы процесса. Часто, хотя и не всегда, можно просто измерить время, проходящее между получением главным окном команды завершения и фактическим выходом из `main()`. Время завершения работы включает также время, необходимое для остановки всех потоков и зависимых процессов. Производители операционных систем, которые предлагают выполнение сертификации программ, имеют строгие требования в отношении времени выключения программ. Время завершения работы имеет также важное значение потому, что время, необходимое для перезапуска службы или длительно работающей программы, равно суммарному времени ее выключения и запуска.

Время отклика

Среднее или наихудшее время, необходимое для выполнения команды. Для веб-сайтов среднее и наихудшее время отклика вносит существенный вклад в удовлетворенность пользователей сайтом. Время отклика можно грубо разделить на следующие диапазоны.

Менее 0,1 с: непосредственное управление со стороны пользователя

Если время отклика составляет менее 0,1 с, пользователи чувствуют себя так, как будто они непосредственно управляют пользовательским интерфейсом, а изменения пользовательского интерфейса вызываются непосредственно их действиями. Это максимальная задержка между моментом, когда пользователь начинает перетаскивать объект, и моментом, когда этот объект приходит в движение, или между щелчком пользователя на поле и его подсветкой. Пользователь чувствует себя так, как будто он выдает команду, которую компьютер выполняет мгновенно.

От 0,1 до 1 с: пользователь управляет командами

Если время отклика составляет от 0,1 до 1 с, пользователи чувствуют, что они руководят компьютером, рассматривая краткую задержку как выполнение компьютером команды, приводящей к изменению пользовательского интерфейса. Пользователи могут терпеть такую задержку без рассеяния своего внимания и отвлечения мыслей от происходящего в компьютере.

От 1 до 10 с: потеря управления

Если время отклика — от 1 до 10 с, то пользователям кажется, что, выполнив команду, они потеряли управление над компьютером, в то время как он обрабатывает команду. Пользователи могут потерять концентрацию и забыть находящуюся в краткосрочной памяти информацию, которая необходима им для выполнения задачи; 10 с — максимальное время, которое пользователь еще может удерживать концентрацию. Удовлетворенность пользователей интерфейсом с таким временем отклика очень быстро сходит на нет.

Более 10 с: пора попить кофе

Если время отклика превышает 10 с, пользователи начинают считать, что у них достаточно времени, чтобы заняться некоторыми другими задачами. Если их работа требует использовать пользовательский интерфейс, они станут пить кофе, пока компьютер перемалывает данные. Если пользователь может себе это позволить, он просто закроет программу и пойдет искать удовлетворение в другом месте.

Якоб Нильсен (Jakob Nielsen) написал интересную статью (`http://bit.ly/powers-10`) о масштабах времени в пользовательском интерфейсе, которая указывает на крайне любопытные научные исследования.

Пропускная способность

Значение, обратное времени отклика. Пропускная способность в общем случае выражается как среднее число операций за единицу времени при некоторой тестовой рабочей нагрузке. Пропускная способность, по сути, измеряет то же, что и время отклика, но является более подходящей для программ, ориентированных на пакетную работу, таких как базы данных и веб-службы. Как правило, желательно, чтобы это число было настолько большим, насколько это возможно.

С оптимизацией можно и переборщить. Например, во многих случаях пользователи рассматривают время отклика менее 0,1 с как мгновенный отклик. В такой ситуации уменьшение времени отклика от 0,1 с до 1 мс практически не имеет никакого значения, пусть это время отклика и в 100 раз меньше.

Улучшить можно только измеряемое

Оптимизация одной функции, подсистемы, задачи или контрольного примера никогда не является такой же, как повышение производительности всей программы. Настройка тестовой версии во многих отношениях отличается от работы над окончательной версией программы, работающей с данными пользователя, так что повышение производительности, измеренное при тестировании, редко соответствует улучшению производительности в реальных условиях. Ускорение выполнения одной задачи может не повлиять на скорость работы всей программы в целом, даже если задача составляет большую часть логики программы.

Например, если разработчик базы данных профилирует выполнение конкретного запроса выборки из базы данных 1000 раз и выполняет оптимизацию на основе этих данных, это ускоряет не всю базу данных, а только выполнение ею конкретного запроса. Такое ускорение может также повысить производительность других запросов. Но оно будет иметь менее предсказуемое влияние на запросы удаления или обновления, индексирование и прочие действия базы данных.

Профилирование выполнения программы

Профайлер — это программа, которая генерирует статистические данные о том, как и на что другая программа тратит свое время работы. Профайлер создает отчет, показывающий частоту выполнения каждой инструкции или функции и суммарное время выполнения каждой функции.

Многие компиляторы, включая Visual Studio в Windows и GCC в Linux, поставляются с профайлером, который помогает обнаружить узкие места в программе. Исторически Майкрософт предлагает профайлер только в дорогостоящих версиях Visual Studio, но Visual Studio 2015 Community Edition поставляется с профайлером. Имеются и иные профайлеры с открытым исходным кодом для Windows, для более ранних версий Visual Studio.

Имеется несколько способов реализации профайлера. Один из методов, используемый и в Windows, и в Linux, работает следующим образом.

1. Программист перекомпилирует профилируемую программу со специальными параметрами компиляции, которые оснащают каждую функцию программы специальными программными средствами для профилирования. Это оснащение добавляет некоторые команды на языке ассемблера в начале и конце каждой функции.
2. Программист компонует оснащенную измерителями программу с профилирующей библиотекой.
3. При каждом запуске такой программы на диске создается файл с таблицей с соответствующей информацией.
4. Профайлер получает эту таблицу в качестве своих входных данных и генерирует ряд текстовых и графических отчетов.

Другой метод профилирования работает следующим образом.

1. Оснащение программы измерительным кодом происходит при компоновке немодифицированной программы с профилирующей библиотекой. Эта библиотека содержит подпрограммы, которые с большой частотой прерывают выполнение программы и записывают значение указателя команд.
2. При каждом запуске такой программы ею на диске создается файл с таблицей с соответствующей информацией.
3. Профайлер получает эту таблицу в качестве своих входных данных и генерирует ряд текстовых и графических отчетов.

Выходные данные профайлера могут принимать различные формы. Одной из них является листинг исходного текста, аннотированный количеством выполнений каждой строки. Другой является список имен функций вместе с количеством вызовов каждой из них. Еще один вариант представляет собой такой же список функций с общим временем работы каждой функции и всех функций, вызванных из нее. Возможен вариант списка функций со временем работы каждой функции минус время работы вызванных из них функций, системного кода или ожидания событий.

Средства профилирования тщательно разработаны, чтобы быть настолько недорогими, насколько это возможно. Влияние профайлера на общее время выполнения программы невелико и обычно составляет несколько процентов замедления для каждой операции. Первый метод дает точные цифры за счет более высоких накладных расходов и отключения некоторых оптимизаций. Второй метод дает приблизительные результаты и может пропустить несколько редко вызываемых функций, но имеет преимущество возможности работы с окончательной версией программы.

Наиболее важным преимуществом профайлера является то, что он непосредственно выводит список "горячих функций" кода. Процесс оптимизации сводится к получению списка функций, его исследованию и проверке возможности оптимизации каждой функции, внесению изменений и повторному выполнению кода для получения новых выходных данных профайлера, пока в программе не останется откровенно узких мест или пока вы не исчерпаете все идеи по оптимизации кода. Поскольку обнаруживаемые профайлером узкие места — это по определению места, где происходит много вычислений, такой процесс в целом прост.

Мой опыт работы с профилированием говорит о том, что профилирование отладочной версии программы дает результаты, которые оказываются такими же актуальными, как и полученные для окончательной версии программы. В некотором смысле отладочную версию проще профилировать, потому что она включает все функции, в том числе и встраиваемые, в то время как окончательная версия скрывает от профилирования очень часто вызываемые встроенные функции.

Совет от профессионала

Одна из проблем, связанных с профилированием отладочной версии в Windows, заключается в том, что отладочная сборка компонуется с отладочной версией библиотеки времени выполнения. Отладочная версия функций диспетчера памяти выполняет множество дополнительных проверок, чтобы позволять сообщать о таких событиях, как повторное освобождение памяти или утечки памяти. Стоимость этих дополнительных проверок может значительно увеличить стоимость некоторых функций. Имеется переменная окружения, которая требует от отладчика не использовать отладочную версию диспетчера памяти. Воспользуйтесь управляющим элементом Панель управления⇨Система⇨Дополнительные параметры системы⇨Переменные среды...⇨Системные переменные (Control Panel⇨System Properties⇨Advanced System Settings⇨Environment Variables⇨System Variables в англоязычной версии Windows) и добавьте новую переменную с именем `_NO_DEBUG_HEAP` со значением `1`.

Применение профайлера — отличный способ найти кандидатов для оптимизации, но не идеальный.

- Профайлер не в состоянии посоветовать вам более эффективный алгоритм для решения вашей задачи. А "вылизывание" плохого алгоритма — пустая трата времени.
- Профайлер не может дать ясные и четкие результаты для входных данных, которые выполняют много различных задач. Например, база данных SQL может выполнять совершенно иной код при выполнении запроса вставки, чем при выполнении запроса выборки. Таким образом, код, который оказывается горячей точкой при заполнении базы данных запросами вставки, может вообще не выполняться при чтении из базы данных с помощью запросов выборки. Выполнение теста, который представляет собой смесь загрузки базы данных и запросов к ней, делает код вставки менее заметным в отчете профайлера, если только профилируемая программа не выполняет много вычислений.

  Таким образом, чтобы определить самые горячие функции было легче, попробуйте оптимизировать по одной задаче за раз. Этому может способствовать выполнение теста для конкретного оптимизируемого кода. Оптимизируя несколько узких мест одновременно, вы тем самым вносите еще один источник неопределенности, поскольку оптимизация конкретного кода не обязательно улучшает производительность всей системы. При запуске программы со входными данными, которые ведут к выполнению множества задач, результаты оптимизации конкретной функции могут выглядеть менее заметными.
- Профайлер предоставляет искаженную информацию для программ с интенсивным вводом и выводом, как и в многопоточных программах, поскольку

вычитает время, проведенное в системных вызовах или в ожидании событий. Удаление этого времени теоретически является разумным, потому что программа не единственная, кто ответственен за эти ожидания. Но в результате профайлер рассказывает о том, сколько работы выполняет программа, а не сколько времени на это уходит. Некоторые профайлеры дают подсчет количества вызовов функций, а также время, затраченное на выполнение каждой функции. Если профайлер скрывает время работы, ключом может оказаться очень высокое количество вызовов.

Профайлер — рабочая лошадка с ограниченными умениями; есть возможности оптимизации, которые он не в состоянии указать, а при интерпретации полученной от профайлера информации возникают свои проблемы. Тем не менее для многих программ профилировщик выдает достаточно хорошие результаты, так что никакой другой метод и не требуется.

Длительно работающий код

Если программа выполняет одну задачу, которая в основном сосредоточена на вычислениях, профилирование автоматически покажет вам ее узкие места. Но если программа делает много разных вещей, никакая функция не будет резко выделяться профайлером. Программа может также проводить много времени в ожидании ввода-вывода или внешних событий, что снижает ее общую производительность, измеряемую как общее время работы. В этих случаях нужно хронометрировать отдельные части программы, а затем пытаться сокращать время выполнения медленных частей.

Разработчик использует хронометраж для поиска горячих точек программы, последовательно сужая исследуемую часть продолжительной задачи до тех пор, пока один из разделов не станет отнимать слишком большое время. Выявив подозрительный раздел кода, разработчик проводит эксперименты с небольшими подсистемами или отдельными функциями.

Хронометраж выполнения — эффективный способ проверки гипотез о том, как уменьшить стоимость конкретной функции.

Не так сложно сообразить, что компьютер легко может быть запрограммирован для работы в качестве секундомера. Вероятно, ваш телефон или ноутбук и так будит вас по утрам в будние дни или напоминает за 5 минут до назначенного важного звонка. Измерение субмикросекундных интервалов времени выполнения функций в современных компьютерах оказывается более сложным, в частности, потому, что распространенные платформы Windows/PC исторически имели проблемы с предоставлением часов с высоким разрешением, которые вели бы себя согласованно в различных аппаратных моделях и версиях программного обеспечения.

Как разработчик вы должны быть готовы к созданию собственного программного секундомера, понимая, что он может измениться в будущем. Для этого рассмотрим, как измеряется время и какие инструменты поддерживают измерения времени на компьютерах.

"Полузнайство" об измерении времени

И полузнайство ложь в себе таит[1].

— Александр Поуп (Alexander Pope), "Опыт о критике", 1774

Идеальное измерение каждый раз должно точно указывать размер, вес или, в случае этой книги, продолжительность измеренного явления. Сделать идеальное измерение — это все равно что лучнику раз за разом попадать точно в центр мишени, раскалывая своей стрелой попавшую туда ранее. Такая стрельба из лука происходит только в легендах, и то же самое можно сказать и о точном измерении.

Реальные измерения (как и реальные лучники) должны бороться с *погрешностями*: источниками ошибок, которые не позволяют достичь совершенства. Погрешность может быть *случайной* и *систематической*. Случайные погрешности влияют на каждое измерение по-разному, как порыв ветра, который заставляет конкретную стрелу немного отклониться в полете. Систематические погрешности влияют на все измерения аналогичным образом, как, например, поза лучника может влиять на каждый его выстрел, заставляя стрелу немного отклоняться, скажем, влево от предполагаемой цели.

Сами погрешности тоже могут быть измерены. Сводные показатели погрешности являются свойствами измерения под названием *прецизионность* (precision) и *истинность* (trueness). Вместе эти свойства формируют одно интуитивное свойство, называемое *точностью* (accuracy).

Прецизионность, истинность и точность

Ученые, которых интересуют измерения, часто спорят об используемой терминологии — достаточно посмотреть на слово "точность" в Википедии. Я выбрал для разъяснений термины из стандарта ISO 5725-1 1994 года (`http://bit.ly/iso-57251`), "*Accuracy (trueness and precision) of measurement methods and results — Part 1: General principles and definitions*".

Измерение является *прецизионным*, если оно свободно от случайной погрешности. То есть, если многократно измеряется одно и то же явление и измеренные значения находятся близко одно к другому, такое измерение является прецизионным. Однако прецизионные измерения по-прежнему могут содержать систематические погрешности. Лучник, который попадает несколькими выстрелами в одно и то же место, очень прецизионный[2], даже если и не очень точный. Его мишень может выглядеть так, как показано на рис. 3.2.

Если я измеряю некоторое значение (скажем, время выполнения функции) 10 раз и получаю один и тот же результат все 10 раз, можно предположить, что мое измерение является прецизионным. (Как и при любом эксперименте, я должен оставаться скептически настроенным до тех пор, пока у меня не накопится много доказательств моей гипотезы.) Если же я получил шесть раз один результат, а три раза — несколько отличный от него и еще раз — совсем иной результат, измерение является менее прецизионным.

[1] Перевод А. Субботина.

[2] Что касается стрельбы, то в русском языке для этого явления имеется свой термин — "кучность". — *Примеч. пер.*

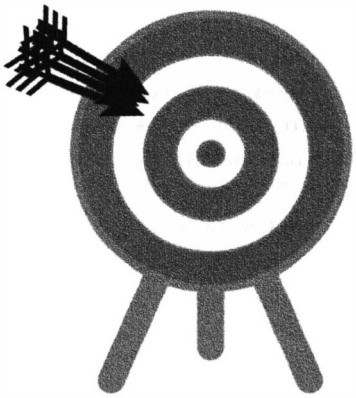


Рис. 3.2. Кучная (хотя и неточная) стрельба

Измерение является *истинным*, если оно свободно от систематической погрешности. То есть, если многократно измерять одно и то же явление и среднее всех результатов множества измерений оказывается близким к фактической измеряемой величине, то можно верить, что измерение является истинным. Отдельные измерения могут быть затронуты случайными погрешностями и оказываться ближе или дальше от фактического значения. Правильность не является вознаграждаемым навыком при стрельбе из лука. На рис. 3.3 среднее четырех выстрелов было бы в яблочко, если бы была такая стрела. Кроме того, все эти выстрелы имеют одну и ту же точность (расстояние от центра) в единицах колец.

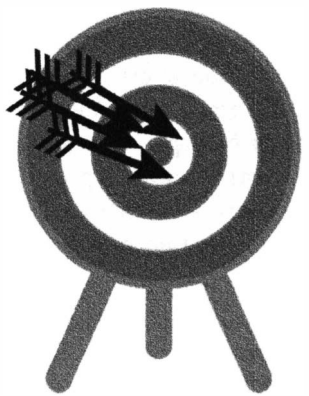


Рис. 3.3. Истинная стрельба

Точность измерения является неофициальной концепцией, которая зависит от того, насколько каждое индивидуальное измерение оказывается близким к фактическому измеряемому значению. Расстояние от фактического значения включает как случайную, так и систематическую погрешность. Чтобы быть точным, измерение должно быть и прецизионным, и истинным.

Измерение времени

Измерения производительности программного обеспечения, рассматриваемые в этой книге, представляют собой либо *продолжительность* (время, которое проходит между двумя событиями), либо *скорость* (количество событий в единицу времени; величина, обратная продолжительности). Инструмент, используемый для измерения продолжительности, — *часы*.

Все часы работают путем подсчета некоторых периодических событий. В одних часах значение времени для представления пользователю делится на часы, минуты и секунды. В других часах время представлено непосредственно как количество тактов. Но часы (за исключением солнечных) не измеряют непосредственно часы и минуты. Они подсчитывают такты, и только путем сравнения тактов с другими эталонными часами часы могут быть откалиброваны для представления часов, минут и секунд.

Все источники периодических событий могут иметь отклонения, которые делают их несовершенными. Одни из этих отклонений случайные; другие — систематические.

- В солнечных часах используется периодическое вращение Земли. По определению один полный оборот Земли составляет один день. Земля является несовершенными часами потому, что их период очень продолжительный, а также потому, что скорость вращения Земли несколько варьируется (на уровне микросекунд) в связи с дрейфом континентов на всей ее поверхности. Эта вариация скорости является случайной. Приливные силы Солнца и Луны замедляют общую скорость вращения Земли. Эта вариация скорости является систематической.
- Дедушкины часы подсчитывают количество регулярных качаний маятника. Шестерни преобразуют качание маятника в движение стрелок, которые отображают время. Период маятника может быть скорректирован вручную таким образом, чтобы отображаемое время было синхронизировано с вращением Земли. Период колебаний маятника зависит от его длины, так что каждый маятник может оказаться быстрее или медленнее, чем требуется. Такое отклонение является систематическим. Трение, давление воздуха и накопление пыли — все может повлиять на маятник, даже если он изначально отлично работал. Эти факторы являются источниками случайных отклонений.
- В электрических часах используется периодическая, с частотой 50 Гц (60 Гц в США), синусоидальная волна переменного тока для привода синхронного двигателя. Шестерни переводят эти колебания в электросети в движение стрелок для отображения времени. Электрические часы не являются идеальными, поскольку 50 Гц — это только соглашение, а не фундаментальный закон природы. При интенсивном использовании электроэнергии в часы пиковых нагрузок частота может падать[3]. Это отклонение является случайным. Электри-

[3] Именно потому что на частоте 50 Гц работало очень много часов по стране, в СССР отслеживалось изменение частоты из-за изменения нагрузки, и в случае ее снижения на какое-то время затем частота специально увеличивалась, чтобы компенсировать временное уменьшение и обеспечить высокую точность электрических часов. — *Примеч. пер.*

ческие часы, созданные для работы в США, будут работать медленнее, будучи подключенными к розетке 50 Гц в Европе (такое отклонение носит систематический характер).

- В цифровых наручных часах используются колебания кристалла кварца. Логическая схема управляет дисплеем на основании подсчета этих колебаний. Частота колебаний кристалла зависит от его размера, температуры и приложенного напряжения. Влияние размера кристалла является систематическим отклонением, в то время как изменения температуры и напряжения являются случайными.

Количество тактов по своей природе является целым беззнаковым значением. Не существует такого понятия, как -5 тактов. Я упомянул этот, казалось бы, очевидный факт, потому что, как показано ниже, многие программисты реализуют функции хронометража со знаковым представлением длительности. Я не знаю, почему они так поступают.

Разрешение измерений

Разрешением измерения является размер единиц, в которых представлены результаты измерений.

Лучник, стреляющий в цель, получает одинаковое количество очков при попадании в любую точку одного круга. Центр мишени является не бесконечно малой точкой, а кругом определенного диаметра (рис. 3.4). Стрела попадает в центр, первое кольцо, второе кольцо и т.д. Ширина кольца и является разрешением оценки попадания.

Рис. 3.4. Разрешение: попадание в любую точку круга мишени дает одинаковое количество очков

Разрешение измерения времени ограничено продолжительностью лежащих в основе измерения периодических событий. Измерение времени может дать один такт или два такта, но не некоторое промежуточное значение. Таким образом, разрешением часов является период между их тактами.

Наблюдатель может *различать* события, происходящие между двумя тактами медленных часов, таких как маятниковые. Это означает, что у людей в головах имеются более быстрые (но менее точные) часы, которые они неформально сравнивают

с маятниковыми. Но если наблюдатель планирует измерять такие малые и незаметные для человеческого восприятия длительности, как миллисекунды, необходимо пользоваться только тактами часов с высоким разрешением.

Нет явно выраженной связи между точностью измерения и разрешением, с которым представлены его результаты. В моей повседневной деятельности я могу сказать, что мне потребовалось два дня, чтобы написать этот раздел книги. Полезным разрешением данного измерения является один день. Я мог бы преобразовать это время в секунды и сказать, что мне потребовалось 172 800 секунд. Но если только у меня в руках не было реально отсчитывающего время секундомера, приведенное значение в секундах может дать ложное чувство того, что измерение было более точным, чем на самом деле, или ложное впечатление, что я все это время не ел и не спал, а только работал за клавиатурой.

Результаты измерения могут быть приведены в единицах, меньших значения полезного разрешения, поскольку эти единицы являются стандартными. У меня есть духовка, которая отображает температуру в ней в градусах Цельсия. Однако разрешение термостата, который управляет духовкой, имеет полезное разрешение 5°C, так что при нагреве духовки на дисплее появляются значения 150, 155, 160°C и т.д. Имеет смысл отображать значение температуры в знакомых общепринятых единицах вместо того, чтобы вводить единицы для данного термостата. Это просто означает, что наименьшая значащая цифра измерения может принимать значения только 0 и 5.

Читатель может быть удивлен и разочарован, узнав реальные разрешения многих недорогих термометров, весов и других измерительных приборов, имеющих на экране разрешение в одну стандартную единицу или даже в одну десятую ее часть.

Измерение с помощью нескольких часов

> *Человек с одними часами всегда знает, который час. Человек с двумя часами никогда не уверен полностью.*
>
> — Чаще всего приписывают Ли Сегаллу (Lee Segall)

Когда два события происходят в одном месте, время между ними легко измерить в тактах одних часов. Когда два события происходят на большом расстоянии одно от другого, может потребоваться использовать двое часов. Количество тактов между различными часами нельзя сравнивать непосредственно.

Человечество подошло к решению этой проблемы путем синхронизации часов с помощью универсального глобального времени (Coordinated Universal Time). Всеобщее скоординированное время синхронизируется с астрономической полночью на долготе 0 — произвольной линии, которая проходит через Королевскую обсерваторию в Гринвиче, Англия (рис. 3.5). Это позволяет время в тактах преобразовывать во время в часах, минутах и секундах после полуночи UTC (Universal Time Coordinated — всеобщее скоординированное время; неуклюжее сокращение, являющееся результатом переговоров между французскими и английскими учеными).

Если двое часов идеально синхронизированы с UTC, то UTC-время одних часов можно непосредственно сравнивать с показаниями других. Но, конечно, идеальная

синхронизация не представляется возможной. Любые часы содержат независимые источники отклонений, которые приведут к расхождению их показаний с UTC и одних с другими.

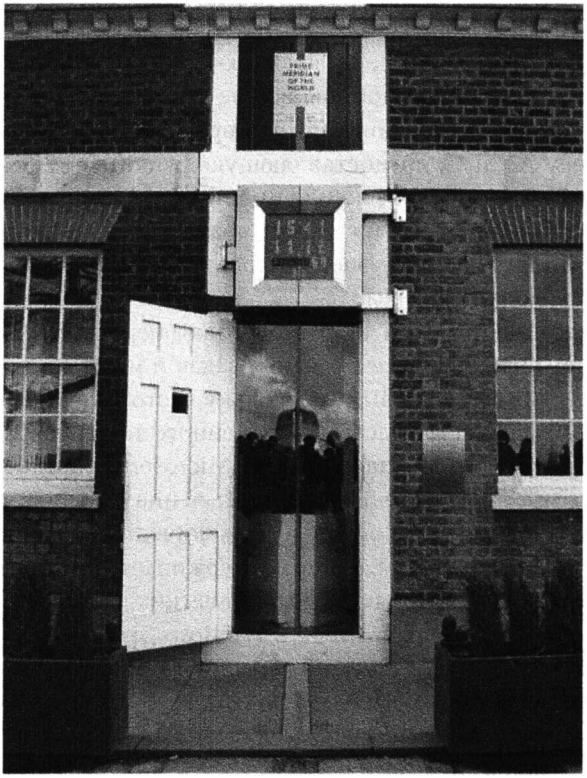


Рис. 3.5. Нулевой меридиан в Королевской обсерватории в Гринвиче

Измерение времени с помощью компьютеров

Для создания часов на компьютере необходимы источник периодических сигналов — предпочтительно с хорошей прецизионностью и истинностью — и способ программного получения тактов из этого источника. Можно легко разработать специализированный компьютер, предназначенный для сообщения информации о времени. Однако при разработке наиболее популярных современных компьютерных архитектур никто особо не задумывался о том, как обеспечить хорошие часы. Я проиллюстрирую имеющиеся проблемы с использованием архитектуры PC и Microsoft Windows. Проблемы у Linux и встраиваемых платформ аналогичны.

Кристаллический генератор в сердце компьютерных часов имеет типичную базовую точность около 0,01%, или около 8 с в день. Хотя эта точность только немногим лучше точности дешевых цифровых наручных часов, ее более чем достаточно для выполнения измерений производительности, для которых вполне годятся результаты с точностью до нескольких процентов. Хотя схемы часов в дешевых

встраиваемых процессорах менее точны, наиболее серьезная проблема не с источником периодических колебаний, а с надежным получением тактов для использования программами.

Аппаратная эволюция счетчиков тактов

Первоначально в IBM PC вовсе не было каких-либо аппаратных счетчиков тактов. В них имелись обычные часы, предоставляющие текущее время дня, которое могло быть считано программно. Ранние библиотеки времени выполнения Microsoft C копировали библиотеку ANSI C, предоставляющую функцию `time_t time(time_t*)`, которая возвращает количество секунд, прошедших с момента 00:00 1 января 1970 года UTC. Оригинальная версия `time()` возвращала 32-битное целое число со знаком, но во время подготовки к решению проблемы 2000 года оно было изменено на 64-разрядное знаковое целое число.

Оригинальный компьютер IBM PC использовал периодическое прерывание от блока питания для выполнения переключения задач и иных операций ядра. Период этих тактов составлял 16,67 мс в США, поскольку частота переменного тока в США составляет 60 Гц, и 10 мс там, где частота переменного тока равна 50 Гц.

Начиная с Windows 98 (возможно, и ранее) Microsoft C предоставляет функцию ANSI C `clock_t clock()`, которая возвращает значение счетчика тактов в знаковом формате. Константа `CLOCKS_PER_SEC` указывает количество тактов в секунду. Ее значение `-1` означает, что функция `clock()` не поддерживается. Первоначально эта функция сообщала о тактах, основанных на периодических прерываниях от переменного тока. Функция `clock()`, реализованная в Windows, отличается от спецификации ANSI, измеряя прошедшее время, а не затраченное процессорное время. Функция `clock()` была недавно реализована заново на основе `GetSystemTimeAsFileTime()`, а в 2015 году она возвращала миллисекундные такты с разрешением 1 мс, что делает ее хорошими миллисекундными часами в Windows.

Начиная с Windows 2000 программный счетчик тактов, основанный на прерываниях электропитания, был сделан доступным с помощью вызова `DWORD GetTickCount()`. Такты, подсчитываемые с помощью `GetTickCount()`, зависят от аппаратного обеспечения компьютера и могут быть значительно длиннее одной миллисекунды. Функция `GetTickCount()` выполняет вычисление для преобразования тактов в миллисекунды, чтобы частично устранить эту неопределенность. Обновленный вариант этого вызова `ULONGLONG GetTickCount64()`, возвращает тот же счетчик тактов как 64-битное беззнаковое целое число, позволяющее измерять более длительные интервалы. Хотя нет никакого способа для получения текущего периода прерываний, есть пара функций, которые уменьшают период, а затем восстанавливают его:

```
MMRESULT timeBeginPeriod(UINT)
MMRESULT timeEndPeriod(UINT)
```

Эти функции воздействуют на глобальную переменную, влияющую на все процессы и многие другие функции, такие как `Sleep()`, зависящие от прерывания переменного тока. Другой вызов, `DWORD timeGetTime()`, как представляется, получает значение того же счетчика тактов другим методом.

Начиная с архитектуры Pentium, компания Intel предоставила аппаратный регистр под названием "Счетчик штампа времени" (Time Stamp Counter — TSC). Это 64-битный регистр, который подсчитывает такты часов процессора. Этот счетчик можно очень быстро получить с помощью машинной команды `RDTSC`. Начиная с Windows 2000, счетчик можно прочитать путем вызова функции `BOOL QueryPerformanceCounter(LARGE_INTEGER*)`, которая дает значение счетчика тактов без конкретного разрешения. Значение разрешения может быть получено с помощью вызова `BOOL QueryPerformanceFrequency(LARGE_INTEGER*)`, который возвращает частоту в виде количества тактов в секунду. `LARGE_INTEGER` представляет собой структуру, которая хранит 64-битное целочисленное значение в знаковом формате, поскольку в момент введения этой функции Visual Studio еще не имела типа для 64-битных знаковых целочисленных значений.

Проблема с первоначальной версией `QueryPerformanceCounter()` заключалась в том, что частота тактов зависит от часов процессора, а они разные у разных процессоров и материнских плат. Старые компьютеры, особенно с процессорами от AMD, в то время не имели TSC. При отсутствии TSC функция `QueryPerformanceCounter()` фактически возвращалась к низкому разрешению `GetTickCount()`.

В Windows 2000 была также добавлена функция `void GetSystemTimeAsFileTime(FILETIME*)`, которая возвращает количество 100-наносекундных тактов, прошедших с 00:00 1 января 1601 года UTC. `FILETIME` представляет собой структуру, которая хранит 64 бита целого числа, на этот раз в беззнаковом формате. Несмотря на кажущееся очень высокое разрешение счетчика тактов, некоторые реализации используют тот же медленный счетчик, что и в `GetTickCount()`.

Вскоре выявились и другие проблемы, связанные с `QueryPerformanceCounter()`. Некоторые процессоры реализованы с переменной тактовой частотой для снижения энергопотребления. Это привело к изменению периода между тактами. В многопроцессорных системах с использованием нескольких дискретных процессоров значение, возвращаемое `QueryPerformanceCounter()`, зависит от того, на каком процессоре выполняется данный поток. Процессоры также стали реализовывать возможности переупорядочения команд, так что команда `RDTSC` может оказаться отложенной, тем самым уменьшая точность программного обеспечения, использующего TSC.

Для решения этих проблем Windows Vista использует для `QueryPerformanceCounter()` другой счетчик, обычно таймер ACPI. С помощью этого счетчика решается проблема хронометража в многопроцессорных системах, но значительно увеличивается задержка. Тем временем Intel переделала TSC так, чтобы получать максимальную, неизменяемую частоту часов. Intel также добавила непереупорядочиваемую команду `RDTSCP`.

Начиная с Windows 8 становится доступным надежный аппаратный счетчик тактов с высоким разрешением на основе TSC. Функция `void GetSystemTimePreciseAsFileTime(FILETIME*)` обеспечивает такты с высокой разрешающей способностью с фиксированной частотой и субмикросекундной точностью на любой системе под управлением Windows 8 или более поздней версии.

Подытоживая этот урок по истории, можно резюмировать, что компьютеры никогда не разрабатывались для функционирования в качестве часов, поэтому счетчики

тактов, которые они предоставляют, являются ненадежными. Если экстраполировать последние 35 лет развития, то будущие процессоры и будущие операционные системы по-прежнему могут не позволять получить надежные, с высоким разрешением счетчики тактов.

Единственный счетчик тактов, доступный в персональных компьютерах всех поколений, — это `GetTickCount()` со всеми его недостатками. Лучше использовать такты с периодом 1 мс, возвращаемые функцией `clock()`; они должны быть доступны во всех компьютерах, производимых в последние 10 лет или около того. Если ограничиться только Windows 8 и более поздними версиями и новыми процессорами, то очень точным является счетчик 100-наносекундных тактов от `GetSystemTimePreciseAsFileTime()`. Однако мой опыт утверждает, что миллисекундной точности для проведения экспериментов вполне достаточно.

Циклический возврат

Циклический возврат (*wraparound*) — это то, что происходит, когда счетчик тактов часов достигает своего максимального значения, а затем переходит к нулевому значению. Точно то же происходит в полдень и в полночь с двенадцатичасовыми аналоговыми часами. Windows 98 "зависала" при непрерывной работе в течение 49 дней (см. Q216641 (`http://bit.ly/windows-49`)) из-за циклического возврата 32-битного счетчика миллисекундных тактов. Ошибка 2000 года связана с циклическим возвратом времени, представленного в виде двух цифр года. Циклический возврат календаря майя в 2012 году был многими воспринят как сигнал о наступающем конце света. В январе 2038 года произойдет циклический возврат эпохи Unix (знаковое 32-битное значение количества секунд, прошедших с 00:00 1 января 1970 года по Гринвичу), возможно, приведя к фактическому концу света для некоторых долгоживущих встраиваемых систем. Проблема циклического возврата заключается в отсутствии дополнительных битов для записи следующего значения, так что в результате очередное более позднее значение времени численно оказывается более ранним. Часы с проблемой циклического возврата хорошо работают только при измерении длительности, которая меньше соответствующего интервала.

Например, в Windows функция `GetTickCount()` возвращает счетчик тактов с разрешением 1 мс в виде 32-битного беззнакового целого числа. Циклический возврат значения функции `GetTickCount()` происходит примерно каждые 49 дней. Функция `GetTickCount()` может беспроблемно использоваться для хронометража операций, продолжительность которых гораздо меньше, чем 49 дней. Если программа вызывает `GetTickCount()` в начале и в конце операции, разница между возвращаемыми значениями может трактоваться как число миллисекунд, прошедших между вызовами, например:

```
DWORD start = GetTickCount();
DoBigTask();
DWORD end = GetTickCount();
cout << "Выполнение продолжается " << end-start << " мс" << endl;
```

Способ реализации беззнаковой арифметики в C++ дает корректный результат даже при наличии циклического возврата.

Функция `GetTickCount()` менее эффективна для запоминания времени после запуска. Многие серверы с продолжительным временем работы могут продолжать работать месяцы или даже годы. Проблема с циклическими возвратами заключается в отсутствии битов для записи их количества, `end-start` может давать одно и то же значение без циклических возвратов, с одним или с несколькими.

Начиная с Vista корпорация Майкрософт добавила функцию `GetTickCount64()`, которая возвращает беззнаковое 64-битное количество тактов с разрешением 1 мс. До циклического возврата `GetTickCount64()` должны пройти миллионы лет, что значительно снижает вероятность возникновения такой проблемы.

Разрешение — не то же, что и точность

В Windows функция `GetTickCount()` возвращает 32-битное беззнаковое целое число. Если программа вызывает `GetTickCount()` в начале и в конце операции, разница между возвращаемыми значениями может быть истолкована как число миллисекунд, прошедших между вызовами. Таким образом, выводимое разрешение `GetTickCount()` составляет 1 мс.

Например, следующий блок кода измеряет относительную производительность произвольной функции под названием `Foo()` в Windows путем вызова `Foo()` в цикле. Количества тактов, полученные в начале и конце цикла, дают время работы цикла в миллисекундах:

```
DWORD start = GetTickCount();
for(unsigned i = 0; i < 1000; ++i) {
    Foo();
}
DWORD end = GetTickCount();
cout << "1000 вызовов Foo() занимают " << end-start << " мс" << endl;
```

Если функция `Foo()` выполняет некоторые существенные вычисления, этот блок кода может сгенерировать следующий вывод:

```
1000 вызовов Foo() занимают 16 мс
```

К сожалению, точность вызова `GetTickCount()` может быть 10 или 15,67 мс, как описано на веб-странице Microsoft, посвященной функции `GetTickCount()` (`http://bit.ly/gettickcount`). То есть, если вы вызываете `GetTickCount()` два раза подряд, разница может быть 0 или 1 мс, или 10, 15 или 16 мс. Таким образом, фундаментальная точность измерения составляет 15 мс. Так что фактическим временем выполнения предыдущего блока могут быть 10, 20 или ровно 16 мс.

Особенно разочаровывает в функции `GetTickCount()` то, что, помимо разрешения в 1 мс, не гарантируется реализация этой функции каким-либо определенным образом или хотя бы одним и тем же образом на двух разных компьютерах под управлением Windows.

Я проверял различные функции времени в Windows, чтобы выяснить, каково их разрешение на определенном компьютере (Surface 3 на базе i7) под управлением определенной операционной системы (Windows 8.1). Тест, показанный в примере 3.1, вызывает функцию хронометража и рассматривает разницу между значениями, возвращаемыми последовательными вызовами. Если разрешение тактов больше, чем

задержка при вызове функции, последовательные вызовы будут возвращать одинаковые значения или значения, различающиеся на размер базового такта, в единицах разрешения функции. Я усреднял ненулевые различия просто на случай, если операционная система забирала время для работы некоторых других задач.

Пример 3.1. Измерение длительности `GetTickCount()`

```
unsigned nz_count = 0, nz_sum = 0;
ULONG last, next;
for (last = GetTickCount(); nz_count < 100; last = next) {
    next = GetTickCount();
    if (next != last) {
        nz_count += 1;
        nz_sum += (next - last);
    }
}
std::cout << "Среднее разрешение GetTickCount() "
          << (double)nz_sum / nz_count
          << " тактов" << std::endl;
```

Результаты этого теста подытожены в табл. 3.1.

Таблица 3.1. Измерение длительности такта (i7 Surface Pro 3, Windows 8.1)

Функция	Длительность такта
`time()`	1 с
`GetTickCount()`	15,6 мс
`GetTickCount64()`	15,6 мс
`timeGetTime()`	15,6 мс
`clock()`	1,0 мс
`GetSystemTimeAsFileTime()`	0,9 мс
`GetSystemTimePreciseAsFileTime()`	~450 нс
`QueryPerformanceCounter()`	~450 нс

Особо следует отметить, что функция `GetSystemTimeAsFileTime()`, которая имеет выводимое разрешение 100 нс, похоже, основана на тех же тактах с частотой 1 мс, что и функция `clock()`, и что, похоже, функция `GetSystemTimePreciseAsFileTime()` реализована с использованием `QueryPerformanceCounter()`.

Современные компьютеры имеют базовые часы с периодом, который измеряется в сотнях пикосекунд (100 пс составляют 10^{-10} с!). Они выполняют команды за несколько наносекунд. Но на персональном компьютере нет доступных счетчиков тактов с пикосекундным или наносекундным разрешением. Самый быстрый доступный счетчик тактов имеет 100-наносекундное разрешение, но его точность может быть гораздо хуже, чем разрешение. Как следствие для многих функций невозможно измерить продолжительность одного вызова. Чтобы узнать, как преодолеть эту проблему, читайте раздел “Преодоление проблем измерений” далее в этой главе.

Задержка

Задержка — это время, которое проходит между командой на выполнение некоторого действия и его фактическим началом. Например, задержка — это время между падением монетки в колодец и моментом, когда вы услышите всплеск (рис. 3.6). Это время между выстрелом стартового пистолета и моментом начала движения бегуна.

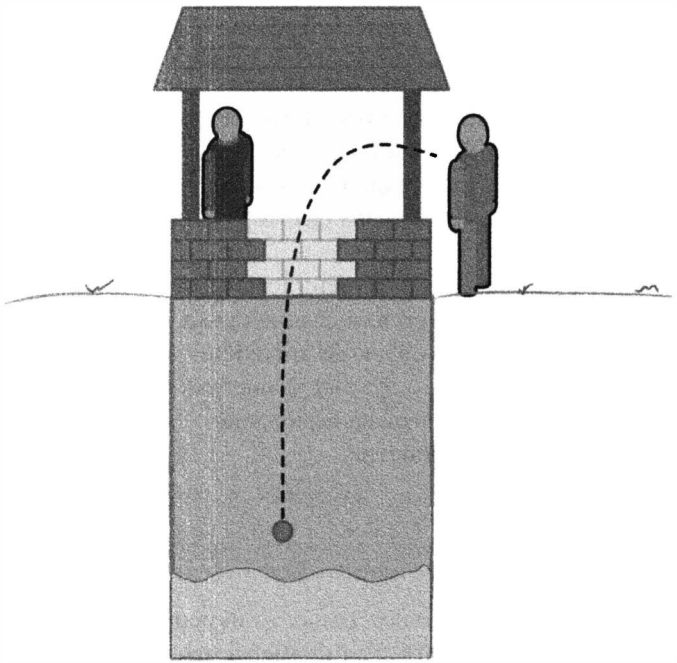


Рис. 3.6. Задержка: время между броском монеты и услышанным всплеском

Что касается измерения времени на компьютерах, то задержка вызывается тем, что запуск часов, запуск эксперимента и остановка часов являются операциями, выполняемыми поочередно. Выполнение измерения может быть разбито на пять этапов.

1. "Запуск часов" включает вызов функции, которая получает от операционной системы количество тактов. Этот вызов занимает ненулевое количество времени. Где-то в средине функции считывается фактическое значение регистра процессора. Это значение и становится начальным временем. Назовем этот интервал t_1.
2. После того как значение счетчика тактов считано, его нужно вернуть и присвоить переменной. Эти действия также занимают время. Часы идут вперед, но начальное количество тактов не увеличивается. Назовем этот интервал t_2.
3. Начинается и заканчивается измеряемый эксперимент. Назовем этот интервал t_3.

4. "Остановка часов" включает еще один вызов функции для получения значения счетчика тактов. Во время этого вызова до момента фактического чтения значения счетчика часы продолжают идти, хотя сам эксперимент уже завершен. Назовем этот интервал t_4.
5. После того как значение счетчика тактов считано, его нужно вернуть и присвоить переменной. Но поскольку значение счетчика уже считано, никаких отклонений больше не накапливается, несмотря на то, что часы продолжают идти. Назовем этот интервал t_5.

Итак, в то время как фактическая продолжительность эксперимента равна t_3, измеренная продолжительность оказывается более длительной — $t_2+t_3+t_4$. Таким образом, задержка составляет t_2+t_4. Если задержка составляет существенную долю времени выполнения эксперимента, экспериментатор должен вычесть ее значение из общего времени.

Предположим, например, что для получения значения счетчика тактов требуется 1 мкс и что счетчик тактов считывается последней командой среди выполняемых за это время. В приведенном далее псевдокоде измерение времени начинается с последней команды первого вызова `get_tick()`, так что перед началом измеряемого действия нет никакой задержки. Задержка вызова `get_tick()` в конце теста добавляется к измеренной продолжительности:

```
start = get_tick()      // 1 мкс задержки до начала значения не имеет
do_activity()
stop = get_tick()       // 1 мкс задержки после окончания добавляется
duration = stop-start   // к измеренному интервалу
```

Если измеряемые действия занимают 1 мкс, то измеренное значение будет равно 2 мкс, т.е. давать ошибку, равную 100%. Если измеряемые действия занимают 1 мс, то измеренное значение будет равно 1,001 мс, с ошибкой 0,1%.

Если до и после эксперимента вызывается одна и та же функция, то $t_1=t_4$ и $t_2=t_5$. Задержка представляет собой просто время вызова функции таймера.

Я измерял задержку вызова хронометрирующих функций Windows, которая просто равна их продолжительности. В примере 3.2 представлен типичный тест функции `GetSystemTimeAsFileTime()`.

Пример 3.2. Задержка функции хронометража Windows

```
ULONG start = GetTickCount();
LARGE_INTEGER count;
for (counter_t i = 0; i < nCalls; ++i)
    QueryPerformanceCounter(&count);
ULONG stop = GetTickCount();
std::cout << stop - start
          << " мс для 100000000 вызовов QueryPerformanceCounter()"
          << std::endl;
```

Результаты этого теста подытожены в табл. 3.2.

Таблица 3.2. Задержка функций хронометража Windows; VS 2013, i7, Win 8.1

Функция	Длительность вызова, нс
`GetSystemTimeAsFileTime()`	2,8
`GetTickCount()`	3,8
`GetTickCount64()`	6,7
`QueryPerformanceCounter()`	8,0
`clock()`	13
`time()`	15
`TimeGetTime()`	17
`GetSystemTimePreciseAsFileTime()`	22

Следует отметить, что все задержки находятся в диапазоне в несколько наносекунд на моем планшете i7, так что все эти вызовы достаточно эффективны. Это означает, что задержка не влияет на точность измерения, когда вызовы функций измеряются в цикле продолжительностью около одной секунды. Тем не менее времена выполнения функций, которые считывают одно и то же значение с низким разрешением, могут существенно различаться — до 10 раз. Наибольшую задержку имеет функция `GetSystemTimePreciseAsFileTime()`, и эта задержка наибольшая по отношению к длительности такта функции — около 5%. На медленных процессорах задержка становится гораздо более серьезной проблемой.

Недетерминистическое поведение

Компьютеры представляют собой невероятно сложные устройства с массой внутренних устройств и состояний, никоим образом не видимых для разработчиков. Выполнение функции изменяет состояние компьютера (например, содержимое кеш-памяти) таким образом, что каждое повторное выполнение происходит в условиях, отличных от предыдущего. Таким образом, неконтролируемые изменения внутреннего состояния являются источником случайных отклонений при измерениях.

Кроме того, операционная система планирует задачи непредсказуемым образом, так что другая деятельность процессоров и шины памяти варьируется во время выполнения измерений. Это делает измерения менее точными.

Операционная система может даже приостановить хронометрируемый код, чтобы предоставить время для выполнения другой программе. Во время такой паузы счетчик тактов продолжает работать, так что измерение покажет бо́льшую продолжительность, чем если бы операционная система не передавала часть процессорного времени другой программе. Это приводит к еще бо́льшим случайным отклонениям, которые могут влиять на результаты измерений.

Преодоление проблем измерений

Итак, насколько плохо обстоят дела в действительности? Можем ли мы в принципе использовать компьютер для измерения времени? И что нам нужно сделать, чтобы все заработало? В этом разделе кратко подытожен мой личный опыт использования описанного в следующем разделе класса для тестирования функций при написании этой книги.

Не беспокойтесь о мелочах

Действительно хорошая новость заключается в том, что измерений с точностью в пару процентов будет вполне достаточно для наших целей оптимизации. Говоря иначе, для обычных линейных улучшений, которые ожидаются от оптимизации, при измерениях достаточно иметь около двух значащих цифр (так, для эксперимента, который в цикле выполняет некоторые функции около 1000 мс, 10 мс — допустимая погрешность). Если мы взглянем на наиболее вероятные источники погрешностей, показанные в табл. 3.3, все они дают существенно меньшие отклонения.

Таблица 3.3. Вклад в отклонения 1-секундного измерения времени в Windows

Отклонение	Вклад, %
Задержка функции счетчика тактов	< 0,00001
Стабильность часов	< 0,01
Реальное разрешение счетчика тактов	< 0,1

Относительная производительность

Отношение времени выполнения оптимизированного кода ко времени выполнения исходного кода называется *относительной производительностью*. Среди прочего относительные измерения обладают тем хорошим свойством, что они сводят на нет систематические отклонения, так как одно и то же отклонение применяется к обоим измерениям. Кроме того, относительное значение в процентном выражении проще для интуитивного понимания, чем число миллисекунд.

Повышение повторяемости с помощью измеряемых модульных тестов

Модульные тесты, которые выполняют тестирование подсистем с использованием заранее подготовленных входных данных, часто оказываются хорошими, повторяемыми действиями для профилирования или измерения производительности. Многие организации имеют обширную библиотеку модульных тестов, и новые тесты могут быть в нее добавлены специально для настройки производительности.

Общая озабоченность по поводу настройки производительности выглядит следующим образом: "Мой код — запутанный клубок, и у меня пока что нет каких-либо контрольных примеров. Я должен протестировать производительность реальных входных данных (или с реальной базой данных), которые постоянно изменяются. В результате я не получаю согласованных или повторяющихся результатов. Что же делать?"

У меня нет магических пуль для этого конкретного зверя. Если я тестирую модули или подсистемы с помощью некоторого искусственного повторяемого набора входных данных, то, как правило, улучшение производительности, полученное в этих тестах, дает повышение производительности и для реальных данных. Если определить горячие функции в большом, но не повторяемом тесте, то улучшение этих функций с помощью модульных контрольных примеров обычно улучшает производительность и для реальных данных. Каждый разработчик знает, почему следует

собирать программные системы из слабо связанных модулей. Каждый разработчик знает, почему следует поддерживать библиотеку хороших контрольных примеров. Оптимизация представляет собой еще одну причину для этого.

Настройка производительности с использованием метрик

Для разработчиков, вынужденных поднимать производительность в непредсказуемой живой системе, все же есть луч надежды. Вместо измерения, скажем, значения критического времени отклика разработчик может собирать статистические данные, такие как среднее и дисперсия, или экспоненциально сглаженное среднее время отклика. Поскольку значения статистических данных получаются на основе большого количества отдельных событий, устойчивое улучшение этих статистических данных указывает на успешное изменение кода.

Вот лишь несколько из вопросов, которые могут возникать при оптимизации с использованием метрик.

- Чтобы быть корректными, статистические данные должны быть основаны на большом количестве событий. При использовании метрик цикл "изменение/проверка/оценка" оказывается потребляющим существенно большее количество времени по сравнению с непосредственным измерением для фиксированных входных данных.
- Для сбора метрик необходима гораздо большая инфраструктура, чем для профилирования или хронометража выполнения. Обычно для хранения статистических данных требуется постоянное хранилище. Его наполнение информацией может оказаться достаточно дорогим с точки зрения производительности. Система для сбора метрик должна быть тщательно разработана, чтобы обладать достаточной гибкостью для поддержания множества экспериментов.
- Хотя имеются хорошо разработанные методики проверки или опровержения статистических гипотез, для корректной работы они требуют от разработчиков определенной изощренности и знаний.

Повышение точности путем усреднения множества итераций

Эксперимент может повысить точность одного измерения путем усреднения нескольких измерений. Это происходит, когда разработчик многократно вызывает функции в цикле или выполняет программу со входными данными, которые приводят к многократному вызову определенной функции.

Одним из преимуществ измерения вызова функции с помощью многих итераций является то, что случайные вариации, как правило, взаимно компенсируются. Состояния кеша обычно сходятся к одному значению, так что все итерации оказываются "одна в одну". На достаточно долгом интервале случайности поведения планировщика имеют примерно одно и то же влияние на первоначальную и оптимизированную функции. Хотя абсолютное время может не быть типичным для тех же функций в более крупной программе, относительные показатели по-прежнему достаточно точно измеряют степень улучшения кода.

Еще одним преимуществом является то, что можно использовать более грубый и простой в работе счетчик тактов. Компьютеры в настоящее время достаточно быстрые, чтобы одной секунды хватило для выполнения тысяч или даже миллионов итераций.

Уменьшение недетерминированности поведения операционной системы путем повышения приоритета

Уменьшить вероятность передачи фрагментов времени операционной системой другим процессам можно путем увеличения приоритета процесса измерения. Приоритетом процесса в Windows управляет функция `SetPriorityClass()`. Функция `SetThreadPriority()` делает то же самое для приоритета потока. Следующий код увеличивает приоритет текущего процесса и потока:

```
SetPriorityClass(GetCurrentProcess(), ABOVE_NORMAL_PRIORITY_CLASS);
SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST);
```

После выполнения измерений приоритеты процесса и потока должны быть восстановлены до нормальных:

```
SetPriorityClass(GetCurrentProcess(), NORMAL_PRIORITY_CLASS);
SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_NORMAL);
```

Как справиться с недетерминированностью

Мой способ измерения производительности для оптимизации весьма неформальный. Он не основывается на больших знаниях статистики. Мои тесты выполняются в течение секунд, но никак не часов. Я не чувствую необходимости извиняться за такой подход к делу, так как он приводит к видимым улучшениям производительности программы, так что я считаю, что я на правильном пути.

Если запустить один и тот же эксперимент в разные дни, то будут получены результаты, которые могут отличаться на величину примерно от 0,1 до 1%. Это, несомненно, связано с различиями в исходной ситуации на компьютере. У меня нет возможности контролировать эти различия, поэтому я о них и не беспокоюсь. Если я увижу большие отклонения, я могу сделать время хронометража более продолжительным. Так как это приводит к удлинению цикла "тестирование/отладка", я не делаю этого без особой необходимости.

Даже если я вижу разницу между выполнениями в несколько процентов, относительные отклонения измерений времени в рамках одного запуска должны быть менее 1%. То есть я могу видеть даже относительно небольшие изменения времени работы двух вариантов функции при одном и том же тестовом запуске.

Я стараюсь выполнять хронометраж на "спокойном" компьютере, который не воспроизводит потоковое видео, не занимается обновлением Java или распаковкой большого архива. Я стараюсь не двигать мышь и не переключать окна во время хронометража. Если процессор одноядерный, это действительно весьма важно. На современных многоядерных процессорах я не замечаю больших отклонений, даже если забываю выключить воспроизведение фильма.

Если тест вызывает функцию 10 тысяч раз, то код и данные будут оставаться в кеше. Если я делаю проверку наихудшего абсолютного времени выполнения для

системы реального времени с жесткими ограничениями, это будет иметь значение. Но я выполняю измерения относительного времени выполнения в системе, ядро которой является недетерминированным. Кроме того, я тестирую только те функции, которые мой профайлер уже определил как часто выполняемое узкое место. Таким образом, во время реальной работы программы они будут находиться в кеше, поэтому условия итеративного теста достаточно реалистичны.

Если измененная функция выглядит на 1% быстрее, такое изменение обычно не стоит затраченных усилий. Закон Амдала сократит этот небольшой вклад в общее время выполнения до ничтожного значения. Минимальное значимое улучшение — это улучшение на 10%. Изменение в два раза гораздо более интересно. *Принятие только больших изменений производительности освобождает разработчика от беспокойства о методологии.*

Создание класса-секундомера

Для измерения времени выполнения части программы оснастим код классом секундомера. Этот класс работает точно так же, как и его механический аналог. Создайте экземпляр этого класса и вызовите его функцию-член `start()`, и секундомер начнет отсчитывать время. Вызовите функцию-член `stop()` секундомера или уничтожьте экземпляр класса, и секундомер остановится и отобразит истекшее время.

Создать такой класс (или найти готовый в Интернете) несложно. В примере 3.3 показан секундомер, который использовал я.

Пример 3.3. Класс секундомера

```
template <typename T> class basic_stopwatch : T {
    typedef typename T BaseTimer;
public:
    // Создание и необязательный запуск
    explicit basic_stopwatch(bool start);
    explicit basic_stopwatch(char const* activity = "Stopwatch",
                             bool start=true);
    basic_stopwatch(std::ostream& log,
                    char const* activity="Stopwatch",
                    bool start=true);

    // Остановка и уничтожение секундомера
    ~basic_stopwatch();

    // Получение времени последнего этапа (время последней остановки)
    unsigned LapGet() const;

    // Предикат: возвращает true, если секундомер работает
    bool IsStarted() const;

    // Показывает накопленное время, продолжая работать
    unsigned Show(char const* event="show");

    // (Пере)запуск секундомера, установка/возврат времени этапа
    unsigned Start(char const* event_name="start");
```

```
    // Остановка работающего секундомера, установка/возврат времени этапа
    unsigned Stop(char const* event_name="stop");

private: // Члены
    char const*   m_activity; // Строка "деятельности"
    unsigned      m_lap;      // Время последнего останова
    std::ostream& m_log;      // Поток для записи журнала
};
```

Этот код просто воспроизводит определение класса. Функции-члены для максимальной производительности являются встраиваемыми.

Класс, который является значением параметра шаблона `T`, представляет собой еще более простой вид таймера, который зависит от операционной системы и используемого стандарта C++, для доступа к счетчику тактов. Я написал несколько версий класса `TimerBase`, чтобы проверить различные реализации счетчика тактов. В зависимости от того, насколько современный компилятор C++ используется, класс `T` может использовать библиотеку C++ `<chrono>` или получать такты непосредственно от операционной системы. В примере 3.4 показан класс `TimerBase`, который использует библиотеку `<chrono>`, доступную в C++11 и более поздних стандартах.

Пример 3.4. Класс `TimerBase` с использованием `<chrono>`

```
#include <chrono>
using namespace std::chrono;
class TimerBase {
public:
    // Сброс таймера
    TimerBase() : m_start(system_clock::time_point::min()) {}

    // Сброс таймера
    void Clear() {
        m_start = system_clock::time_point::min();
    }

    // Возвращает true, если таймер работает
    bool IsStarted() const {
        return (m_start.time_since_epoch() !=
                system_clock::duration(0));
    }

    // Запуск таймера
    void Start() {
        m_start = system_clock::now();
    }

    // Получение количества миллисекунд с момента запуска таймера
    unsigned long GetMs() {
        if (IsStarted()) {
            system_clock::duration diff;
            diff = system_clock::now() - m_start;
            return (unsigned)(
                    duration_cast<milliseconds>(diff).count());
        }
        return 0;
    }
```

```
private:
    system_clock::time_point    m_start;
};
```

Этот класс обладает преимуществом переносимости между операционными системами, но требует поддержки C++11.

В примере 3.5 показан класс `TimerBase` с той же функциональностью, но использующий функцию `clock()`, доступную как в Linux, так и в Windows.

Пример 3.5. Класс `TimerBase` с использованием `clock()`

```
class TimerBaseClock {
public:
    // Сбрасывает таймер
    TimerBaseClock() { m_start = -1; }

    // Сбрасывает таймер
    void Clear() { m_start = -1; }

    // Возвращает true, если таймер работает
    bool IsStarted() const { return (m_start != -1); }

    // Запуск таймера
    void Start() { m_start = clock(); }

    // Получение количества миллисекунд с момента запуска таймера
    unsigned long GetMs() {
        clock_t now;
        if (IsStarted()) {
            now = clock();
            clock_t dt = (now - m_start);
            return (unsigned long)(dt * 1000 / CLOCKS_PER_SEC);
        }
        return 0;
    }
private:
    clock_t    m_start;
};
```

Этот класс имеет преимущество переносимости между разными версиями C++ и операционными системами, но обладает тем недостатком, что функция `clock()` измеряет несколько разные вещи в Linux и Windows.

В примере 3.6 показан класс `TimerBase`, который работает для более старых версий Windows и Linux. В Windows должна быть явно предоставлена функция `gettimeofday()`, поскольку она не является частью стандартной библиотеки C или Windows API.

Пример 3.6. Класс `TimerBase` с использованием `gettimeofday()`

```
#include <chrono>
using namespace std::chrono;
class TimerBaseChrono {
public:
    // Сброс таймера
    TimerBaseChrono() :
        m_start(system_clock::time_point::min()) {
```

```
    }

    // Сброс таймера
    void Clear() {
        m_start = system_clock::time_point::min();
    }

    // Возвращает true, если таймер работает
    bool IsStarted() const {
        return (m_start != system_clock::time_point::min());
    }

    // Запуск таймера
    void Start() {
        m_start = std::chrono::system_clock::now();
    }

    // Получение количества миллисекунд с момента запуска таймера
    unsigned long GetMs() {
        if (IsStarted()) {
            system_clock::duration diff;
            diff = system_clock::now() - m_start;
            return (unsigned)
                   (duration_cast<milliseconds>(diff).count());
        }
        return 0;
    }
private:
    std::chrono::system_clock::time_point    m_start;
};
```

Этот класс переносим между разными версиями C++ и операционными системами, но требует реализации функции `gettimeofday()` при работе в Windows.

Простейший способ использования класса секундомера — идиома RAII (захват ресурса при инициализации). Программа создает экземпляр класса в начале любого набора операторов, заключенного в фигурные скобки. При создании экземпляра секундомера его действие по умолчанию — запуск. Когда секундомер будет уничтожен при достижении закрывающей фигурной скобки (выход из области видимости), он выводит накопленное время. Программа может вывести его и при выполнении программы, с помощью вызова функции-члена `show()`. Это позволяет разработчику работать с несколькими связанными областями кода с помощью одного таймера, например:

```
{
    Stopwatch sw("activity");
    DoActivity();
}
```

Этот код выведет на стандартный вывод следующие строки:

```
activity: start
activity: stop 1234 мс
```

Секундомер не привносит никаких накладных расходов во время работы. Задержка при запуске и остановке состоит из стоимости системного вызова получения текущего времени, а также стоимости вывода, если вызывается функция-член `show()`. Эта задержка не является значительной, если используется для задач, выполняемых десятки или более миллисекунд. Накладные расходы становятся значительными, а точность, таким образом, пониженной, если разработчик пытается хронометрировать микросекундные интервалы.

Вероятно, наибольшей слабостью хронометража времени выполнения является необходимость в интуиции (а значит, в опыте) при интерпретации результатов. После того как многократные выполнения программы сужают поиск интересующей области, разработчик должен исследовать код или выполнить эксперименты по выявлению и удалению узких мест. Исследование кода опирается на опыт разработчика или эвристические правила, изложенные в этой книге. Эти правила обладают тем преимуществом, что помогают идентифицировать код, который имеет высокую стоимость выполнения. Недостатком их является то, что они не указывают на наиболее узкое место кода однозначно.

> **Совет от профессионала**
>
> Разработчик не всегда может хронометрировать действия, выполняющиеся до того, как функция `main()` получает управление, или после выхода из функции `main()`. Если в глобальной области видимости создается много экземпляров классов, это может стать проблемой. Время, затраченное программой за пределами функции `main()`, может быть источником значительных накладных расходов в больших программах со многими глобальными переменными. Дополнительную информацию о запуске программы можно найти в разделе "Удаление кода запуска и завершения" главы 12, "Оптимизация параллельности".

Хронометраж функции в тесте

После того как профайлер или анализ выполнения обнаруживает функцию-претендент для оптимизации, простейший способ работы с ней — создание теста, который вызывает данную функцию много раз. Он позволяет увеличить время выполнения функции до измеримого значения. Многократный вызов выравнивает также отклонения времени выполнения, вызванные наличием фоновых задач, переключения контекста и т.д. Цикл "редактирование/компиляция/запуск" для изолированной функции более быстрый, чем тот же цикл, за которым следует запуск профайлера и изучение его результатов. Во многих примерах в этой книге использована эта методика.

Такой хронометраж (пример 3.7) представляет собой просто вызов функции в цикле, скажем, из 10 тысяч итераций, находящемся в блоке с секундомером.

Пример 3.7. Хронометраж функции

```
typedef unsigned counter_t;
counter_t const iterations = 10000;



{
    Stopwatch sw("function_to_be_timed()");
    for (counter_t i = 0; i < iterations; ++i) {
        result = function_to_be_timed();
    }
}
```

Количество итераций приходится угадывать. Общее время выполнения должно быть от нескольких сотен до нескольких тысяч миллисекунд на настольных компьютерах, если счетчик тактов, используемый классом `Stopwatch`, имеет полезное разрешение около 10 мс.

Я использую тип `counter_t`, а не `unsigned` или `unsigned long`, так как для некоторых достаточно кратких функций может потребоваться 64-битная переменная типа `unsigned long long`. Легче привыкнуть к использованию `typedef`, чем потом возвращаться и повторно вводить все имена типов. Это — оптимизация самого процесса оптимизации.

Важное значение имеет внешний набор фигурных скобок. Он определяет область, внутри которой существует экземпляр `sw` класса `Stopwatch`. Поскольку `Stopwatch` использует идиому RAII, конструктор `sw` получает начальное значение счетчика тактов, а его деструктор получает значение счетчика тактов по окончании вычислений и записывает результаты в стандартный поток вывода.

Оценка стоимости кода для поиска узких мест

Опыт научил меня, что профилирование и измерения времени выполнения являются эффективными методами поиска кандидатов для оптимизации. Профайлер может указать часто вызываемые функции или функции, на выполнение которых уходит значительная часть общего времени работы. Маловероятно ткнуть пальцем в определенную строку кода и сразу попасть на самое узкое место в программе. Могут быть случаи, когда подготовка кода для профилирования оказывается очень дорогой. Хронометраж также может указать на большой блок кода без уточнения конкретной проблемы.

Так что следующий шаг разработчика заключается в том, чтобы оценить стоимость каждой инструкции в выявленном блоке кода. Это не такая строго поставленная задача, как доказательство теоремы. Бо́льшую часть информации можно получить, бегло просмотрев исходный текст в поисках дорогостоящих инструкций и структур, которые увеличивают стоимость кода.

Оценка стоимости отдельных инструкций C++

Как указано в разделе "Медленная память" в главе 2, "Оптимизация, влияющая на поведение компьютера", стоимость времени доступа к памяти доминирует над стоимостью выполнения других инструкций. На простых микропроцессорах, используемых в тостерах и кофеварках, время выполнения команды буквально равно времени, необходимому для чтения каждого байта команды из памяти, плюс время, необходимое для считывания данных, необходимых в качестве входных данных для этой команды, а также плюс время, необходимое для записи результата выполнения команды. Само декодирование и выполнение команды занимает сравнительно незначительное количество времени, которое скрывает время доступа к памяти.

На микропроцессорах класса настольных компьютеров ситуация оказывается более сложной. Одновременно на различных стадиях выполнения находится много команд. Стоимость чтения потока команд является скрытой. Однако стоимость доступа к данным, с которыми работают команды, скрыть труднее. По этой причине для оценки относительной стоимости выполнения команд в микропроцессорах всех классов могут использоваться стоимость чтения и записи данных.

Для оценки стоимости инструкций C++ полезным эмпирическим правилом является *подсчет количества операций чтения и записи памяти*, выполняемых инструкцией. Например, в инструкции `a = b + c;`, где `a`, `b` и `c` являются целыми числами, значения `b` и `c` должны быть считаны из памяти, а сумма должна быть записана в память в местоположение переменной `a`. Таким образом, эта инструкция требует трех обращений к памяти. Такой подсчет не зависит от набора команд микропроцессора. Это неизбежная стоимость инструкций.

Инструкция `r = *p + a[i];` может быть вычислена следующим образом: одно обращение — для чтения `i`, одно — для `a[i]`, одно — для `p`, еще одно — для чтения данных `*p`; для записи результата в `r` требуется еще одно обращение; итого — всего пять обращений. Стоимость вызовов функций в обращениях к памяти описана в разделе "Стоимость вызовов функций" главы 7, "Оптимизация инструкций".

Важно понимать, что это правило эвристическое, приближенное. В реальном аппаратном обеспечении требуются дополнительные обращения к памяти для выборки команд. Однако эти обращения последовательны, поэтому, скорее всего, они очень эффективны. И эти дополнительные расходы пропорциональны стоимости доступа к данным. Компилятор может оптимизировать некоторые из обращений путем повторного использования предыдущих вычислений или использования статического анализа кода. Стоимость в единицах времени зависит также от того, находится ли память, к которой обращается инструкция C++, в кеше.

При равенстве всех прочих факторов имеет значение то, сколько чтений и записей в память необходимо для доступа к данным, используемым инструкцией. Эта эвристика не идеальна, но это все, что можно получить, не обращаясь к промежуточному выводу компилятора на языке ассемблера, что является утомительной и обычно неблагодарной работой.

Оценка стоимости циклов

Поскольку каждая инструкция C++ обычно выполняет только несколько обращений к памяти, маловероятно, что одна инструкция окажется "узким местом", если только какой-то иной фактор не подействует на нее таким образом, что она будет выполняться очень часто. Одним из таких факторов является нахождение инструкции в теле цикла. В таком случае стоимость инструкции умножается на количество ее выполнений.

Если вам очень повезет, вы можете наткнуться на код, который просто кричит о том, насколько здесь "горячо". Профайлер может указать на функцию, которая выполняется миллион раз, а некоторые горячие функции могут содержать цикл наподобие следующего:

```
for (int i=1; i<1000000; ++i) {
    do_something_expensive();
    if (mostly_true) {
        do_more_stuff();
        even_more();
    }
}
```

Очевидно, что этот цикл выполняется миллион раз и, таким образом, является очень горячей точкой. Вероятно, придется немного поработать и оптимизировать его.

Оценка количества повторений вложенных циклов

Когда один цикл вложен в другой, количество повторений блока кода равно количеству повторений внутреннего цикла, умноженному на количество повторений внешнего цикла, например:

```
for (int i=0; i<100; ++i) {
    for (int j=0; j<50; ++j) {
        fiddle(a[i][j]);
    }
}
```

Здесь количество повторений — 100*50 = 5 000.

Этот конкретный блок чрезвычайно прост, но есть бесконечные вариации вложенных циклов. Например, в математике есть важные случаи, когда код выполняет итерации по треугольной части матрицы. А иногда плохой стиль кодирования затрудняет определение границ вложенных циклов.

Вложенные циклы могут не быть очевидными. Если цикл вызывает функцию, а функция содержит другой цикл, этот внутренний цикл является вложенным. Как мы увидим в разделе "Перемещение циклов в функции для снижения накладных расходов при вызовах" главы 7, "Оптимизация инструкций", иногда можно устранить стоимость повторяемого вызова функции во внешнем цикле.

Внутренний цикл может быть встроен в функции стандартной библиотеки, в особенности при обработке строк или при вводе-выводе символов. Если количество повторений достаточно велико, может иметь смысл переписывание функций стандартной библиотеки, чтобы избежать стоимости их вызовов.

Оценка циклов с переменным количеством повторений

Не каждый цикл обеспечивает простой и точный подсчет количества его итераций. Многие циклы повторяются до тех пор, пока не выполнится некоторое условие, например цикл, который обрабатывает символы до тех пор, пока не находит пробел, или цифры, пока не обнаружит символ, не являющийся цифрой. Количество повторений таких циклов можно оценить. Необходима лишь примерная оценка; скажем, пять цифр на число или шесть букв на слово. Цель заключается в том, чтобы обнаружить возможных кандидатов для оптимизации.

Распознавание неявных циклов

Программы, которые обрабатывают события (например, программы с пользовательским интерфейсом в Windows), содержат неявные циклы на верхнем уровне. Такой цикл даже никак не отображается в программе, потому что скрыт в используемом каркасе. Если каркас получает события с максимально возможной скоростью, любой код, который выполняется при каждом получении управления обработчиком событий и до диспетчеризации события (или во время таковой), может быть кандидатом на оптимизацию, как и код обработки наиболее часто диспетчеризуемых событий.

Распознавание ложных циклов

Не каждая инструкция `while` или `do` является циклом. Я встречал код, в котором инструкция `do` использовалась для облегчения потока управления. Есть и другие способы кодирования приведенного ниже примера, но при более сложной логике `if-then-else` такая идиома может иметь смысл. Этот "цикл" выполняется ровно один раз. Если управление достигает инструкции `while(0)`, выполняется выход из цикла:

```
do {
    if (!operation1())
        break;
    if (!operation2(x,y,z))
        break;
} while(0);
```

Эта идиома иногда используется для упаковки нескольких инструкций в макросе C.

Другие пути поиска узких мест

Разработчик, знакомый с кодом, может решить положиться на одну лишь интуицию, строя гипотезы о том, какие области кода могут существенно влиять на общее время выполнения программы, и проводя эксперименты, позволяющие увидеть влияние изменений в одной из этих областей на общую производительность.

Я не рекомендую этот путь, если только вы не работаете в одиночку. Исследование кода с помощью профайлера или программного таймера позволяет разработчикам продемонстрировать коллегам и руководителям достигнутый прогресс в деле оптимизации. Если же вы полагаетесь на интуицию и не получаете результатов, а иногда даже если вы их получаете, ваши товарищи по команде будут ставить ваши методы

под вопрос и как минимум отвлекать вас от работы, требуя поделиться информацией о них. И они правы. С их точки зрения, нет иного способа узнать, используете ли вы профессиональную интуицию высококвалифицированного программиста или просто тычетесь в код в случайном порядке.

Из истории оптимизационных войн

В своем личном подходе я использую и опыт, и интуицию. Однажды я сократил время безответного запуска интерактивной игры с неприемлемых 16 с до около 4 с. К сожалению, я не сохранил начальные измерения, так что мой руководитель поверил только в то, что я сократил время запуска с восьми секунд до четырех; это было единственное, что я мог ему показать. Затем этот руководитель занялся проверкой старого исходного кода с помощью профайлера и выдал мне свои результаты. Интересно, что его поиск привел почти к такому же списку функций, как и моя интуиция. Однако мой авторитет в качестве эксперта по оптимизации был подорван, потому что я не подошел к решению задачи методически.

Резюме

- *Производительность должна быть измерена.*
- *Делайте проверяемые предсказания и записывайте их.*
- *Записывайте вносимые в код изменения.*
- *Если каждый экспериментальный запуск документирован, его можно быстро повторить.*
- *Программа тратит 90% времени работы на выполнение 10% кода.*
- *Чтобы быть точными, измерения должны быть истинными и прецизионными.*
- *Разрешение и точность — разные понятия.*
- *В Windows функция* `clock()` *обеспечивает надежное 1-миллисекундное разрешение. В Windows 8 и более поздних функция* `GetSystemTimePreciseAsFileTime()` *предоставляет субмикросекундные такты.*
- *Принятие только больших изменений производительности освобождает разработчика от беспокойства о методологии.*
- *Для оценки стоимости инструкций C++ подсчитайте количество чтений и записей памяти, выполняемых ею.*

Оптимизация использования строк

Коснувшись волшебной строки,
Обрящешь ты громкую славу.
А если она не по нраву —
Умри со строкою в душе.

— Оливер Уинделл Холмс (Oliver Wendell Holmes),
Безгласие (*The Voiceless*) (1858)

Шаблонный класс C++ `std::string` является одним из наиболее активно используемых классов стандартной библиотеки C++. Например, в статье на форуме разработчиков Google Chromium (`http://bit.ly/chromium-dev`) утверждается, что `std::string` ответственен за половину всех обращений к диспетчеру памяти в Chromium. Любой часто выполняемый код, работающий со строками, оказывается благодатной почвой для оптимизации. В этой главе мы используем обсуждение оптимизации работы со строками в качестве иллюстрации ряда периодически возникающих тем в области оптимизации.

Почему строки представляют собой проблему

Концептуально строки довольно просты, но с точки зрения эффективной реализации обладают рядом тонкостей. В классе `std::string` его возможности сочетаются таким образом, что делают эффективную реализацию практически невозможной. На момент написания этой книги несколько популярных компиляторов предоставляли реализации `std::string`, которые тем или иным образом не соответствовали стандарту.

Кроме того, поведение `std::string` с годами изменилось, чтобы идти в ногу с новейшими изменениями стандарта C++. Это означает, что соответствующая стандарту реализация `std::string` в компиляторе C++98 может вести себя не так же, как реализация `std::string` после появления стандарта C++11.

Строки обладают рядом возможностей, которые делают их применение дорогостоящим независимо от реализации. *Они используют динамическое выделение памяти, ведут себя в выражениях, как значения, а их реализация требует большого количества копирований.*

Строки используют динамическое выделение памяти

Строки удобны, потому что автоматически увеличиваются по мере увеличения хранимого содержимого. В отличие от них, функции библиотеки C (`strcat()`, `strcpy()` и др.) работают с массивами символов фиксированного размера. Для реализации этой гибкости строки используют динамическое выделение памяти. Динамическое распределение по сравнению с большинством других функций C++ является дорогостоящим, поэтому строки автоматически оказываются горячими точками для оптимизации. Динамически выделяемая память автоматически освобождается, когда строковая переменная выходит из области видимости или переменная получает новое значение. Это удобнее, чем освобождение памяти вручную, требуемое при работе с динамически выделяемым массивом символов в стиле C, как показано в следующем коде:

```
char* p = (char*) malloc(7);
strcpy(p, "string");
...
free(p);
```

Внутренний буфер символов строки, тем не менее, по-прежнему имеет фиксированный размер. Любая операция, которая делает строку длиннее, такая как добавление символа или строки, может привести к тому, что размер хранимой строки станет превышать размер внутреннего буфера. Когда это происходит, строка получает новый буфер от диспетчера памяти и копирует свое содержимое в новый буфер.

Реализации `std::string` используют трюк для уменьшения амортизированной стоимости перераспределения памяти для буфера символов при росте строки. Вместо запроса у диспетчера памяти блока точного размера, соответствующего количеству символов, реализация строки выполняет округление запроса вверх, запрашивая большее количество памяти. Например, некоторые реализации округляют запрос до следующей степени 2. Таким образом, строка может увеличиваться в два раза относительно текущего размера без необходимости нового вызова диспетчера памяти. Если очередная операция, удлиняющая строку, видит, что в выделенном буфере для этого достаточно места, новый буфер не выделяется. Преимущество этого трюка заключается в том, что стоимость добавления в пересчете на один символ строки при постоянном росте строки асимптотически приближается к константе. Стоимостью этого трюка является повышенный расход памяти. Если строка реализует политику округления запросов к степени 2, то до половины памяти в строке может быть неиспользованной.

Строки как значения

Строки ведут себя, как *значения* (см. раздел "Объекты-значения и объекты-сущности" главы 6, "Оптимизация переменных в динамической памяти") в присваиваниях и выражениях. Числовые константы наподобие `2` и `3.14159` являются значениями. Вы можете присвоить новое значение переменной, но изменение переменной не изменяет значения, например:

```
int i,j;
i = 3;  // i имеет значение 3
```

```
j = i;  // j также имеет значение 3
i = 5;  // i теперь имеет значение 5, j имеет значение 3
```

Присваивание одной строки другой работает так, как если бы каждая строковая переменная имела собственную копию содержимого:

```
std::string s1, s2;
s1 = "hot";   // s1 == "hot"
s2 = s1;      // s2 == "hot"
s1[0] = 'n';  // s2 все еще "hot", но s1 == "not"
```

Поскольку строки являются значениями, результаты строковых выражений также являются ими. При конкатенации строк, как в инструкции `s1 = s2 + s3 + s4;`, результатом `s2 + s3` является временное строковое значение с отдельно выделенной памятью. Результатом конкатенации `s4` с этой временной строкой является *другое* временное строковое значение. Это значение заменяет предыдущее значение `s1`. Затем освобождается динамически выделенная память для хранения первой временной строки и предыдущего значения `s1`. Это приводит ко множеству вызовов диспетчера памяти.

Строки выполняют массу копирований

Поскольку строки ведут себя, как значения, изменение одной строки не должно влиять на любые другие строки. Но строки имеют и операции, которые изменяют их содержимое. Из-за этих трансформирующих операций каждая строковая переменная должна вести себя так, как будто она имеет собственную копию своей строки. Простейший способ реализовать это поведение — копировать содержимое строки при ее создании, присваивании или передаче в качестве аргумента функции. Если строки будут реализованы таким образом, то присваивание и передача аргумента будут дорогостоящими операциями, но трансформирующие функции и неконстантные ссылки окажутся дешевыми.

Существует известная идиома программирования для объектов, которые ведут себя, как значения, но имеют дорогое копирование. Она называется "копирование при записи" и часто записывается сокращенно как COW (от "copy on write") в литературе по C++ (см. раздел "Реализация идиомы "копирования при записи"" в главе 6, "Оптимизация переменных в динамической памяти"). В COW-строке одна динамически выделенная память может использоваться несколькими строками. Счетчик ссылок позволяет каждой строке знать, использует ли он разделяемую память. Когда выполняется присваивание одной строки другой, копируется только указатель и увеличивается счетчик ссылок. Любая операция, которая изменяет значение строки, сначала убеждается, что существует только один указатель на эту память. Если же таких указателей несколько, то трансформирующая операция (любая операция, которая может изменить содержимое строки) выделяет новую память и делает копию строки перед ее изменением:

```
COWstring s1, s2;
s1 = "hot";   // s1 == "hot"
s2 = s1;      // s2 == "hot" (s1 и s2 указывают на одну память)
s1[0] = 'n';  // s1 делает новую копию содержимого перед изменением
              // s2 все еще "hot", но s1 == "not"
```

Копирование при записи является столь известной идиомой, что разработчики могут легко предположить, что `std::string` реализуется именно таким образом. Но *копирование при записи просто не разрешается для соответствующих стандарту C++11 реализаций* и в любом случае является проблемным.

Если строки будут реализованы с помощью копирования при записи, то присваивание и передача аргументов будут дешевыми, но неконстантные ссылки плюс любые вызовы трансформирующих функций потребуют дорогостоящей операции выделения памяти и копирования, если содержимое строки является совместно используемым. COW-строки являются дорогостоящими и в случае параллельного кода. Каждая трансформирующая функция и неконстантная ссылка обращаются к счетчику ссылок. Когда такое обращение осуществляется из нескольких потоков, каждый поток должен использовать специальные инструкции для получения копии счетчика ссылок из основной памяти, гарантирующие, что никакой другой поток не может изменить его значение (см. раздел "Барьеры памяти" главы 12, "Оптимизация параллельности").

В C++11 и более поздних бремя копирования несколько снижается благодаря наличию rvalue-ссылок и связанной с ними семантики перемещения (см. раздел "Реализация семантики перемещения" главы 6, "Оптимизация переменных в динамической памяти"). Если функция принимает rvalue-ссылку в качестве аргумента, то, когда фактический аргумент является rvalue-выражением, строка может выполнить недорогое копирование указателя, экономя одно копирование содержимого.

Первая попытка оптимизации строк

Предположим, что профилирование большой программы показывает, что функция `remove_ctrl()`, приведенная в примере 4.1, потребляет в программе значительное количество времени. Эта функция удаляет управляющие символы из строки ASCII-символов. Она выглядит достаточно невинно, но производительность этой функции по многим причинам оказывается ужасной. На самом деле эта функция является компактной демонстрацией опасности кодирования без учета производительности.

Пример 4.1. `remove_ctrl()`: код, готовый к оптимизации

```
std::string remove_ctrl(std::string s) {
    std::string result;
    for(int i=0; i<s.length(); ++i) {
        if (s[i] >= 0x20)
            result = result + s[i];
    }
    return result;
}
```

Функция `remove_ctrl()` в цикле обрабатывает каждый символ строки-аргумента `s`. Код цикла и есть то, что делает данную функцию узким местом. Условие `if` получает символ строки и сравнивает его с литеральной константой. Вряд ли здесь кроется какая-то проблема. Но инструкция в пятой строке — история совсем другая.

Как уже отмечалось, оператор конкатенации строк достаточно дорогой. Он вызывает диспетчер памяти для создания нового временного строкового объекта для получающейся в результате строки. Если аргумент `remove_ctrl()` обычно является строкой символов, то `remove_ctrl()` создает новый временный строковый объект почти для каждого символа параметра `s`. Для строки из 100 символов это 100 вызовов диспетчера памяти для создания хранилища для временной строки и еще 100 вызовов, чтобы освободить выделенную память.

Помимо временных строк, выделяемых для хранения результата операции конкатенации, могут выделяться дополнительные строки, когда строковое выражение присваивается переменной `result`, — в зависимости от того, как именно реализованы строки.

- Если строки реализованы с использованием идиомы копирования при записи, то оператор присваивания выполняет эффективное копирование указателя и увеличение счетчика ссылок.
- Если строки имеют реализацию с не разделяемыми буферами, то оператор присваивания должен копировать содержимое временных строк. При наивной реализации, или если буфер строки `result` имеет недостаточный размер, оператор присваивания выполняет выделение нового буфера для копирования в него. В результате выполняется 100 операций копирования и до 100 выделений памяти.
- Если компилятор реализует rvalue-ссылки C++11 и семантику перемещения, то тот факт, что выражение конкатенации является rvalue, позволяет компилятору вызывать перемещающее присваивание переменной `result` вместо копирующего. В результате программа выполняет эффективное копирование указателя.

Операция конкатенации при каждом выполнении копирует также все ранее обработанные символы во временную строку. Если в строке аргумента `n` символов, `remove_ctrl()` копирует $O(n^2)$ символов. В результате этого выделения памяти и копирования получается низкая производительность.

Поскольку `remove_ctrl()` представляет собой небольшую автономную функцию, можно построить тестовую программу, которая вызывает ее несколько раз для точного измерения производительности и выяснения, насколько ее можно повысить за счет оптимизации. Вопросы создания тестов и измерения производительности рассматриваются в главе 3, "Измерение производительности". Коды тестов, как и другой код из данной книги, можно загрузить с моего сайта (`http://www.guntheroth.com`).

Я написал хронометрирующий тест, который многократно вызывает `remove_ctrl()` с 222-символьной строкой в качестве аргумента, содержащей несколько управляющих символов. В среднем каждый вызов занимает 24,8 мкс. Это число важно не само по себе, так как оно зависит от используемого компьютера (планшет Intel i7), операционной системы (Windows 8.1) и компилятора (Visual Studio 2010, 32-битная рабочая версия). Это просто исходная точка для измерения повышения производительности. В следующих разделах описаны набор шагов оптимизации и обусловленное ими повышение производительности функции `remove_ctrl()`.

Использование модифицирующих операций для устранения временных значений

Я начал оптимизацию функции `remove_ctrl()` с устранения выделений памяти и операций копирования. Пример 4.2 содержит усовершенствованную версию `remove_ctrl()`, в которой выражение конкатенации в строке 5, которое создает такое большое количество временных объектов, заменено модифицирующим присваивающим оператором конкатенации `+=`.

Пример 4.2. `remove_ctrl_mutating()`: модифицирующие операторы

```
std::string remove_ctrl_mutating(std::string s) {
    std::string result;
    for (int i=0; i<s.length(); ++i) {
        if (s[i] >= 0x20)
            result += s[i];
    }
    return result;
}
```

Это маленькое изменение оказывает большое влияние на производительность. Хронометраж при тех же исходных условиях дает среднее время одного вызова 1,72 мкс, или ускорение в 13 раз. Источник этого улучшения — ликвидация всех вызовов выделения памяти для временных строковых объектов, хранящих результат конкатенации, и связанных с ними копирований и удалений временных значений. В зависимости от реализации строки устраняются также выделение памяти и копирование для выполнения присваивания.

Уменьшение работы с памятью с помощью резервирования

Функция `remove_ctrl_mutating()` по-прежнему выполняет операцию удлинения строки `result`. Это означает, что `result` периодически выполняет копирование во внутренний динамический буфер большего размера. Как уже отмечалось, одна возможная реализация `std::string` удваивает объем памяти буфера при каждом выделении. При такой реализации `std::string` перераспределение может выполниться для 100 символов 8 раз.

Если исходить из строк, в основном состоящих из печатных символов, с малым количеством удаляемых управляющих символов, то длина строки-аргумента `s` обеспечивает отличную оценку возможной длины строки результата. В примере 4.3 выполняется улучшение `remove_ctrl_mutating()` — заранее выделяется место в памяти с помощью функции-члена `reserve()` класса `std::string`. Это не только устраняет необходимость перераспределения буфера строки, но и улучшает локальность кеша данных, к которым обращается функция, так что мы получаем еще больший эффект от такого изменения.

Пример 4.3. `remove_ctrl_reserve()`: резервирование памяти

```
std::string remove_ctrl_reserve(std::string s) {
    std::string result;
    result.reserve(s.length());
```

```
    for (int i=0; i<s.length(); ++i) {
        if (s[i] >= 0x20)
            result += s[i];
    }
    return result;
}
```

Устранение ряда выделений памяти приводит к значительному повышению производительности. Тестовое выполнение дает время выполнения вызова `remove_ctrl_reserve()`, равное 1,47 мкс, т.е. повышение производительности на 15% по сравнению с `remove_ctrl_mutating()`.

Устранение копирования строкового аргумента

До сих пор я успешно оптимизировал `remove_ctrl()`, удаляя вызовы диспетчера памяти. Поэтому имеет смысл продолжить искать очередные распределения памяти для удаления.

Если строковое выражение передается в функцию по значению, формальный аргумент (в данном случае `s`) конструируется с помощью копирования. В зависимости от реализации строки это может привести к следующему копированию.

- Если строка реализована с использованием идиомы копирования при записи, то компилятор генерирует вызов копирующего конструктора, который выполняет копирование указателя и увеличивает счетчик ссылок.
- Если строка реализована с использованием собственного, не разделяемого буфера, то копирующий конструктор должен выделить новый буфер и скопировать в него содержимое фактического аргумента.
- Если компилятор реализует rvalue-ссылки в стиле C++11 и семантику перемещения, то, если фактический аргумент является выражением, он будет rvalue, так что компилятор сгенерирует вызов перемещающего конструктора, копируя в результате указатель. Если фактическим аргументом является переменная, то вызывается копирующий конструктор формального аргумента, что приводит к выделению памяти и копированию. Семантика перемещения и rvalue-ссылки более подробно описаны в разделе "Реализация семантики перемещения" главы 6, "Оптимизация переменных в динамической памяти".

Функция `remove_ctrl_ref_args()` в примере 4.4 представляет собой усовершенствованную функцию, которая никогда не копирует `s` при вызове. Поскольку функция не изменяет `s`, нет причины делать отдельную копию `s`. Вместо этого `remove_ctrl_ref_args()` получает в качестве аргумента константную ссылку на `s`. Это избавляет нас от еще одного выделения памяти. Поскольку выделение памяти — операция дорогостоящая, может иметь смысл отказ даже от одной такой операции.

Пример 4.4. `remove_ctrl_ref_args()`: устранение копирования аргумента

```
std::string remove_ctrl_ref_args(std::string const& s) {
    std::string result;
    result.reserve(s.length());
    for (int i=0; i<s.length(); ++i) {
```

```
        if (s[i] >= 0x20)
            result += s[i];
    }
    return result;
}
```

Результат удивителен — хронометраж показывает, что `remove_ctrl_ref_args()` требует 1,60 мкс на вызов, что на 8% хуже, чем у функции `remove_ctrl_reserve()`.

Что же происходит? Visual Studio 2010 копирует строковые значения при вызове, поэтому данное изменение должно сэкономить выделение памяти. Либо эта экономия осталась не реализованной, либо что-то иное, связанное с превращением `s` из строки в ссылку на строку, "съело" всю эту экономию.

Ссылки на переменные реализованы как указатели. Так что везде, где в `remove_ctrl_ref_args()` встречается `s`, программа выполняет разыменование указателя, которое не требуется делать в `remove_ctrl_reserve()`. Я предполагаю, что этой дополнительной работы может быть достаточно для снижения производительности.

Устранение разыменований с помощью итераторов

Решение заключается в использовании итераторов строки, как показано в примере 4.5. Итератор строки представляет собой простой указатель в буфер символов. По сравнению с применением в цикле ссылки, использование итераторов должно экономить две операции разыменования.

Пример 4.5. `remove_ctrl_ref_args_it()`: версия `remove_ctrl_ref_args()` с итераторами

```
std::string remove_ctrl_ref_args_it(std::string const& s) {
    std::string result;
    result.reserve(s.length());
    for(auto it=s.begin(), end=s.end(); it != end; ++it) {
        if (*it >= 0x20)
            result += *it;
    }
    return result;
}
```

Хронометраж `remove_ctrl_ref_args_it()` дает удовлетворительный результат в 1,04 мкс на вызов. Это, определенно, лучше, чем в версии без применения итераторов. Но что можно сказать о превращении `s` в ссылку на строку? Чтобы убедиться, что эта оптимизация действительно что-то дает, я написал версию функции `remove_ctrl_reserve()` с использованием итераторов. Хронометраж `remove_ctrl_reserve_it()` дал 1,26 мкс на вызов, что явно меньше 1,47 мкс. Использование ссылки вместо значения, определенно, улучшило производительность.

На самом деле я использовал версии с итераторами для всех функций, производных от `remove_ctrl()`. Итераторы во всех версиях давали выигрыш по сравнению с применением индексов (однако в разделе "Вторая попытка оптимизации строк" далее в главе вы увидите, что это не всегда так).

Функция `remove_ctrl_ref_args_it()` содержит еще одну оптимизацию, о которой следует сказать. Значение `s.end()`, используемое для управления циклом `for`, кешируется при инициализации цикла. Это сохраняет еще *2n* косвенных обращений, где *n* — длина строки-аргумента.

Устранение копирования возвращаемого значения

Исходная функция `remove_ctrl()` возвращает результат по значению. C++ создает результат с помощью копирующего конструирования в вызывающем контексте, хотя компилятор имеет право удалить это копирующее конструирование, если может обойтись без него. Если мы хотим быть *уверенными*, что копирования нет, примем к сведению, что есть несколько вариантов, как поступить. Один из вариантов, который работает на всех версиях C++ и со всеми реализациями строк, должен возвращать строку в качестве выходного параметра. Собственно, именно так и поступает компилятор при упомянутом выше устранении копирования. Улучшенная версия `remove_ctrl_ref_args_it()` приведена в примере 4.6.

Пример 4.6. `remove_ctrl_ref_result_it()`: устранение копирования возвращаемого значения

```
void remove_ctrl_ref_result_it(std::string& result,
                               std::string const& s)
{
    result.clear();
    result.reserve(s.length());
    for(auto it=s.begin(), end=s.end(); it != end; ++it) {
        if (*it >= 0x20)
            result += *it;
    }
}
```

Когда программа вызывает `remove_ctrl_ref_result_it()`, в качестве первого аргумента `result` передается ссылка на некоторую строковую переменную. Если строковая переменная, на которую ссылается `result`, пустая, вызов `reserve()` выделяет память для достаточного количества символов. Если эта переменная использовалась раньше (при вызове программой функции `remove_ctrl_ref_result_it()` в цикле), буфер уже может быть достаточно большим, и в этом случае новое выделение не происходит. При возврате из функции строковая переменная в вызывающем коде уже содержит возвращаемое значение, и копирование не требуется. Функция `remove_ctrl_ref_result_it()` красива тем, что во множестве случаев полностью устраняет все выделения памяти.

Измеренная производительность `remove_ctrl_ref_result_it()` составляет 1,02 мкс на вызов, т.е. примерно на 2% быстрее, чем предыдущая версия.

Функция `remove_ctrl_ref_result_it()` *очень* эффективна, но ее интерфейс легко использовать неверно, чего не позволял интерфейс исходной функции `remove_ctrl()`. Ссылка — даже константная — ведет себя не в точности так же,

как значение. Следующий вызов приведет к неожиданным результатам, возвращая пустую строку:

```
std::string foo("this is a string");
remove_ctrl_ref_result_it(foo, foo);
```

Использование массивов символов вместо строк

Когда от программы требуется максимальная производительность, можно отказаться от стандартной библиотеки C++ вообще и самому писать код на основе строковых функций в стиле C, как показано в примере 4.7. Строковые функции в стиле C сложнее в использовании, чем `std::string`, но выигрыш в производительности может оказаться впечатляющим. Чтобы использовать эти функции, программист должен либо вручную выделять и освобождать буфера для символов, либо использовать статические массивы с размерами, достаточными для наихудшего случая. Объявление статических массивов проблематично, в особенности при ограниченной памяти. Однако, как правило, имеется возможность статически объявить большие временные буфера в локальной памяти (т.е. в стеке функции). Эти буфера освобождаются очень дешево при выходе из выполняемой функции. За исключением наиболее ограниченных встраиваемых сред, не проблема выделить тысячу или даже 10 тысяч символов в стеке для такого буфера.

Пример 4.7. `remove_ctrl_cstrings()`: работа в C-стиле

```
void remove_ctrl_cstrings(char* destp, char const* srcp, size_t size)
{
    for (size_t i=0; i<size; ++i) {
        if (srcp[i] >= 0x20)
            *destp++ = srcp[i];
    }
    *destp = 0;
}
```

Функция `remove_ctrl_cstrings()` требует всего лишь 0,15 мкс на вызов. Это в 6 раз быстрее ее предшественницы и в невероятные 170 раз быстрее исходной функции. Одной из причин такого улучшения является ликвидация нескольких вызовов функций с соответствующим улучшением локальности кеша.

Однако отличная локальность кеша может способствовать заблуждениям при простых измерениях производительности. В общем случае другие операции между вызовами `remove_ctrl_cstrings()` сбрасывают кеш. Но при вызове в непрерывном цикле и инструкции, и данные остаются в кеше.

Другим фактором, влияющим на функцию `remove_ctrl_cstrings()`, является интерфейс, существенно отличный от интерфейса первоначальной функции. Если эта функция вызывается из многих мест, изменение всех вызовов может потребовать слишком значительных усилий. Тем не менее функция `remove_ctrl_cstrings()` является примером того, насколько может быть повышена производительность, если разработчик готов полностью перекодировать функцию и изменить ее интерфейс.

Остановись и подумай

Я думаю, по этому мосту мы можем зайти слишком далеко.

— Генерал-лейтенант Фридрих Браунинг (Frederick Browning) (1896–1965)

Замечание, сделанное 10 сентября 1944 года по поводу плана фельдмаршала Монтгомери по захвату союзниками моста в Арнеме. Замечание оказалось пророческим, так как операция закончилась катастрофой для войск союзников.

Как отмечалось ранее, усилия по оптимизации могут привести в точку, где дополнительная производительность добывается за счет простоты и безопасности. Функция `remove_ctrl_ref_result_it()` требует изменений в сигнатуре функции, предоставляющих возможность потенциального неправильного использования функции, которое не представляется возможным в случае функции `remove_ctrl()`. Оптимизация `remove_ctrl_cstrings()` получена ценой ручного управления временным хранилищем. Для некоторых команд программистов это может оказаться мостом, ведущим слишком далеко.

Разработчики имеют разные, иногда весьма упорно отстаиваемые, мнения о том, оправдывает ли конкретное повышение производительности дополнительную сложность интерфейса или необходимость пересмотра использования функции. Разработчики, которые выступают за оптимизацию посредством возвращаемого параметра, могут заявить, что опасность некорректного использования функции преувеличена, а изменение сигнатуры может быть хорошо документировано. Возврат строки через параметр, кроме того, оставляет возвращаемое значение функции доступным для использования, например для возврата кодов ошибок. Те, кто выступает против этой оптимизации, могут сказать, что ничто не сможет защитить пользователей от неверного использования функции, ведущего к появлению трудно выявляемых ошибок, так что потери от проблем будут *б*ольшими, чем выгода от оптимизации. В конце концов, команда должна ответить на вопрос "Насколько нам нужно это улучшение производительности?"

Я не могу предложить никаких советов по обнаружению ситуации, когда усилия по оптимизации зашли слишком далеко. Это зависит от того, насколько важно повышение производительности. Но разработчикам нельзя упустить этот момент, чтобы остановиться и немного подумать.

C++ предлагает разработчикам широкий выбор между простым и безопасным, но медленно работающим кодом и радикально быстрым кодом, который следует использовать с предельной осторожностью. Сторонники других языков могут назвать это слабостью, но с точки зрения оптимизации это одна из самых сильных сторон C++.

Итоги первой попытки оптимизации

В табл. 4.1 кратко подытожены результаты усилий по оптимизации функции `remove_ctrl()`. Эти результаты получены с помощью одного простого правила: удалять распределения памяти и связанные с ними операции копирования. Первая оптимизация дала самое значительное ускорение.

Таблица 4.1. Итоги повышения производительности; VS 2010, i7

Функция	Отладочная версия, мкс	Δ скорости, %	Производственная версия, мкс	Δ скорости, %	Соотношение времени отладочной и производственной версий
`remove_ctrl()`	967		24,80		39,00
`remove_ctrl_mutating()`	104	83,0	1,72	1343	60,47
`remove_crtl_reserve()`	102	84,8	1,47	1587	69,39
`remove_ctrl_ref_args_it()`	215	35,0	1,04	2285	206,73
`remove_ctrl_ref_result_it()`	215	35,0	1,02	2331	210,78
`remove_ctrl_cstrings()`	1	966,0	0,15	16433	6,67

На значения абсолютного времени выполнения влияет множество факторов, включая процессор, базовую тактовую частоту, частоту шины памяти, используемые компилятор и оптимизатор. Я добавил результаты испытаний для отладочной и производственной (оптимизированной) версий, чтобы это продемонстрировать. В то время как код производственной версии *намного* быстрее, чем отладочной, улучшения хорошо видны в обоих вариантах кода.

Процентное улучшение представляется гораздо более существенным в производственных версиях. Вероятно, это проявление закона Амдала. В отладочной версии отключено встраивание функций, что увеличивает стоимость каждого вызова функции и уменьшает долю времени выполнения, приходящуюся на выделение памяти.

Вторая попытка оптимизации строк

Есть и другие пути, по которым в поисках лучшей производительности может пойти разработчик. Несколько из них мы рассмотрим в этом разделе.

Использование лучшего алгоритма

Один из путей заключается в попытке улучшить используемый алгоритм. Оригинальная функция `remove_ctrl()` использует простой алгоритм, который копирует в результирующую строку один символ за раз. Этот неудачный выбор дает наихудшее возможное поведение выделения. Пример 4.8 улучшает исходный дизайн путем перемещения в результирующую строку целых подстрок. Это изменение приводит к сокращению числа операций выделения памяти и копирования. Другая оптимизация,

введенная в remove_ctrl_block(), — кеширование длины строки аргумента для снижения стоимости проверки завершения внешнего цикла for.

Пример 4.8. remove_ctrl_block(): более быстрый алгоритм

```
std::string remove_ctrl_block(std::string s) {
    std::string result;
    for(size_t b=0, i = b, e = s.length(); b < e; b = i+1) {
        for(i = b; i < e; ++i) {
            if (s[i] < 0x20)
                break;
        }
        result = result + s.substr(b,i-b);
    }
    return result;
}
```

Функция remove_ctrl_block() проходит тест за 2,91 мкс, примерно в 7 раз быстрее исходной remove_ctrl().

Эта функция, в свою очередь, может быть улучшена так же, как и раньше, путем замены конкатенации трансформирующими операциями (remove_ctrl_block_mutate(), 1,27 мкс на вызов), но substr() по-прежнему создает временную строку. Поскольку функция добавляет символы в result, разработчик может использовать одну из перегрузок функции-члена append() класса std::string для копирования подстроки без создания временной строки. Результирующая функция remove_ctrl_block_append() (показанная в примере 4.9), проходит хронометраж с результатом 0,65 мкс на вызов. Этот результат превосходит лучшее время в 1,02 мкс для функции remove_ctrl_ref_result_it() и оказывается в 36 раз лучше оригинальной функции remove_ctrl(). Это простая демонстрация мощи выбора хорошего алгоритма.

Пример 4.9. remove_ctrl_block_append(): более быстрый алгоритм

```
std::string remove_ctrl_block_append(std::string s) {
    std::string result;
    result.reserve(s.length());
    for (size_t b=0,i=b; b < s.length(); b = i+1) {
        for (i=b; i<s.length(); ++i) {
            if (s[i] < 0x20) break;
        }
        result.append(s, b, i-b);
    }
    return result;
}
```

Эти результаты, в свою очередь, могут быть улучшены путем резервирования памяти для result и удаления копирования аргумента (remove_ctrl_block_args(), 0,55 мкс на вызов) и путем удаления копирования возвращаемого значения (remove_ctrl_block_ret(), 0,51 мкс на вызов).

Одна вещь, которая не улучшает результаты, по крайней мере на первый взгляд, — это переписывание remove_ctrl_block() с использованием итераторов. Однако после того, как и аргумент, и возвращаемое значение передаются как ссылки, версия с

итераторами вместо того, чтобы оставаться в 10 раз дороже, оказывается на 20% дешевле, как показано в табл. 4.2.

Таблица 4.2. Производительность при изменении используемого алгоритма

Функция	Время одного вызова, мкс	Δ скорости по отношению к предыдущей, %
`remove_ctrl()`	24,80	
`remove_ctrl_block()`	2,91	752
`remove_ctrl_block_mutate()`	1,27	129
`remove_ctrl_block_append()`	0,65	95
`remove_ctrl_block_args()`	0,55	18
`remove_ctrl_block_ret()`	0,51	8
`remove_ctrl_block_ret_it()`	0,43	19

Еще один путь повышения производительности — изменение строки-аргумента путем удаления из нее управляющих символов с помощью функции-члена `erase()` класса `std::string`. Этот подход показан в примере 4.10.

Пример 4.10. `remove_ctrl_erase()`: изменение аргумента вместо создания результата

```
std::string remove_ctrl_erase(std::string s) {
    for (size_t i = 0; i < s.length(); )
        if (s[i] < 0x20)
            s.erase(i,1);
        else ++i;
    return s;
}
```

Преимуществом этого алгоритма является то, что, поскольку `s` становится короче, не потребуется никакое перераспределение памяти, за исключением, возможно, распределения для возвращаемого значения. Производительность этой функции очень высока; хронометраж показал 0,81 мкс на вызов, что в 30 раз быстрее, чем для оригинальной функции `remove_ctrl()`. Разработчик, достигший этого превосходного результата с первой попытки, может быть объявлен победителем и покинуть поле битвы без каких-либо дальнейших усилий по оптимизации. *Иногда другой алгоритм оказывается проще для оптимизации или по самой своей природе является более эффективным.*

Использование лучшего компилятора

Я выполнил те же хронометрирующие тесты с помощью компилятора Visual Studio 2013. Visual Studio 2013 реализует семантику перемещения, которая сделала некоторые из функций значительно более быстрыми. Однако результаты оказались неоднозначными. Работа в отладчике Visual Studio 2013 была на 5–15% быстрее, чем для Visual Studio 2010. При запуске из командной строки VS2013 оказался на 5–20% медленнее. Пробная версия Visual Studio 2015 оказалась еще медленнее. Это могло быть

вызвано изменениями в классах контейнеров. Новый компилятор может повысить производительность, но это необходимо проверять, а не принимать на веру.

Использование лучшей библиотеки для работы со строками

Определение `std::string` первоначально было довольно расплывчатым, что обеспечило наличие широкого диапазона реализаций. Требования эффективности и предсказуемости в конечном итоге привели к добавлениям в стандарт C++, которые сделали нестандартными самые новые реализации. Таким образом, поведение, определенное для `std::string`, является компромиссом, который постепенно эволюционировал из конкурирующих конструктивных соображений в течение длительного периода времени.

- Подобно прочим библиотечным контейнерам, `std::string` предоставляет итераторы для доступа к отдельным символам строки.
- Подобно символьным строкам в C, `std::string` предоставляет возможность индексирования в духе массивов с помощью `operator[]` для доступа к его элементам. `std::string` обеспечивает также механизм получения указателя на строку в стиле C с завершающим нулевым символом.
- `std::string` имеет оператор конкатенации и возвращающие значения функции, которые придают строкам семантику значений, подобную строкам BASIC.
- `std::string` предоставляет множество операций, которое некоторые программисты считают слишком ограниченным.

Желание сделать `std::string` таким же эффективным, как символьные массивы C, подталкивает реализации к представлениям строк в виде непрерывного блока памяти. Стандарт C++ требует, чтобы итераторы были итераторами произвольного доступа, и запрещает семантику копирования при записи. Это упрощает определение, какие действия делают итераторы `std::string` недействительными и почему, но ограничивает возможность более интеллектуальных реализаций.

Кроме того, реализация `std::string`, поставляемая с коммерческим компилятором C++, должна быть достаточно прямолинейной, чтобы ее можно было протестировать для гарантии соответствия стандарту и приемлемой эффективности в любой мыслимой ситуации. Стоимость ошибки для производителя компилятора слишком высока. Это подталкивает к простым реализациям.

Определенное в стандарте поведение `std::string` приводит к ряду слабых мест. Вставка одного символа в строку из миллиона символов заставляет копировать все окончание строки и может привести к перераспределению памяти. Аналогично все операции, возвращающие значения подстрок, должны выделять память и копировать свои результаты. Некоторые разработчики ищут возможности оптимизации путем отмены одного или нескольких описанных выше ограничений (итераторы, индексирование, доступ к C-строке, семантика значений, простота).

Применение иной библиотеки для std::string

Иногда использование лучшей библиотеки означает не более чем предоставление дополнительных строковых функций. Вот некоторые из множества библиотек, работающих с `std::string`.

Boost string library (`http://bit.ly/boost-string`)

Библиотека Boost string library предоставляет функции для лексического анализа, форматирования и иных экзотических манипуляций `std::string`. Она доставит немало удовольствия тем, кто любит копаться в заголовочном файле `<algorithm>` стандартной библиотеки.

C++ String Toolkit Library (`http://bit.ly/string-kit-lib`)

Еще одним вариантом является C++ String Toolkit Library (StrTk). Эта библиотека особенно полезна для анализа строк и их разделения на лексемы и совместима с `std::string`.

Применение std::stringstream во избежание семантики значений

C++ содержит несколько различных реализаций строк: шаблонные, с доступом через итераторы, переменной длины строки `std::string`; с простым итераторным интерфейсом `std::vector<char>`; более старые в стиле C строки с завершающими нулевыми символами в массивах фиксированного размера.

Хотя и сложно хорошо использовать символьные строки C, мы уже провели эксперимент, который показал, что замена строк C++ `std::string` строками символов C в массивах резко повышает производительность. Ни одна из этих реализаций не подходит идеально для любой ситуации.

C++ содержит еще один вид строк. `std::stringstream` делает для строк то, что `std::ostream` делает для выходных файлов. Класс `std::stringstream` инкапсулирует буфер с динамически изменяемым размером (в действительности обычно это `std::string`) иначе, как *субъект* (см. раздел "Объекты-значения и объекты-сущности" главы 6, "Оптимизация переменных в динамической памяти"), к которому могут быть добавлены данные. `std::stringstream` является примером того, как наложение другого API поверх аналогичной реализации может стимулировать более эффективное кодирование. Пример 4.11 иллюстрирует его использование.

Пример 4.11. std::stringstream: похож на строку, но является объектом

```
std::stringstream s;
for (int i=0; i<10; ++i) {
    s.clear();
    s << "Квадрат " << i << " равен " << i*i << std::endl;
    log(s.str());
}
```

В этом фрагменте показано несколько методов оптимального кодирования. Поскольку `s` изменяется как субъект, длинное выражение вставки не создает каких-либо временных переменных, для которых должна быть выделена память и которые должны копироваться. Другой метод состоит в том, что переменная `s` объявлена вне

цикла. Таким образом, внутренний буфер `s` используется повторно. При первой итерации цикла, возможно, придется выполнить несколько перераспределений памяти при добавлениях символов к буферу, но последующие итерации вряд ли потребуют перераспределения памяти. Если же переменная `s` была бы объявлена внутри цикла, то при каждом выполнении цикла итерация начиналась бы с пустого буфера и потенциально при каждом операторе вставки требовалось бы выполнение перераспределения памяти.

Если класс `std::stringstream` реализован с использованием `std::string`, он может так никогда по-настоящему и не превзойти `std::string`. Его преимущество заключается в предотвращении использования некоторых практик программирования, ведущих к неэффективному коду.

Применение более новой реализации строки

Разработчик может счесть саму абстракцию строки неадекватной. Одной из наиболее важных функций C++ является то, что абстракции наподобие строк не встроены в язык, а предоставлены в качестве библиотек шаблонов или функций. Альтернативные реализации имеют доступ к тем же возможностям языка, что и `std::string`, так что производительность может быть улучшена достаточно умным и опытным программистом. Отказ от одного или нескольких ограничений, перечисленных в начале этого раздела (итераторы, индексирование, доступ к строке в стиле C, семантика значений, простота) предоставляет возможности создания пользовательского класса строки с оптимизацией, которой не может воспользоваться `std::string`.

В течение длительного времени развития C++ для строк предлагались многие сложные структуры данных, которые обещают значительно снизить затраты на перераспределение памяти и копирование содержимого строки. По ряду причин это может быть не более чем красивое, но невыполнимое обещание.

- Любой претендент на трон `std::string` должен в самых разнообразных ситуациях быть более выразительным и более эффективным, чем `std::string`. Большинство предлагаемых альтернативных реализаций не имеет никаких гарантий повышения общей производительности.
- Замена каждого вхождения `std::string` в большой программе некоторой другой строкой представляет собой большую работу, не гарантирующую, что она даст разницу в производительности.
- Хотя были предложены многие альтернативные концепции строк, и некоторые из них были реализованы, найти полную, хорошо протестированную и понятную реализацию строки — далеко не минутное дело.

Замена `std::string` может быть более практична при рассмотрении дизайна, чем при его оптимизации. Это вполне возможно для большой команды при наличии времени и ресурсов. Но выигрыш от такой замены слишком неопределенный и может сделать такую оптимизацию мостом, который способен завести слишком далеко. Тем не менее для храбрых и отчаянных парней имеется много мест, где они могут себя проявить.

`std::string_view`

`string_view` решает некоторые проблемы `std::string`. Этот класс содержит указатель (которым не владеет) на строковые данные и их длину, так что он представляет подстроку `std::string` или литеральную строку символов. Операции наподобие получения подстроки и отсечения начальных и конечных подстрок более эффективны, чем соответствующие функции-члены `std::string`, возвращающие значения. Некоторые компиляторы уже имеют реализацию этого класса в пространстве имен `std::experimental`. Класс `string_view` имеет практически тот же интерфейс, что и `std::string`.

Проблемой `string_view` является то, что он не владеет указателем. Программист должен гарантировать, что время жизни любого объекта `string_view` не превышает время жизни объекта `std::string`, в который он указывает.

`folly::fbstring` (`http://bit.ly/folly-lib`)

Folly — большая библиотека кода, используемого Facebook на своих серверах. Она включает в себя очень высокооптимизированную замену для `std::string`, которая реализует невыделяемый буфер для оптимизации коротких строк. Проектировщики `fbstring` заявляют об измеряемом повышении производительности.

Folly — необычно надежная и полная библиотека. Она поддерживается в Linux.

Набор строковых классов (`http://bit.ly/toolbox-strings`)

Эта статья и код 2000 года описывают шаблонный строковый тип с таким же интерфейсом, как и реализация SGI класса `std::string`. Он обеспечивает строки с фиксированным максимальным размером и тип строк переменной длины. Это проявление мощи магии метапрограммирования шаблонов, из-за которой многим программистам очень трудно понять данную реализацию. Для тех, кто заинтересован в разработке лучшего строкового класса, этот класс является реальным кандидатом.

"C++03 Expression Templates" (`http://craighenderson.co.uk/papers/exptempl/`)

Это статья 2005 года, содержащая шаблонный код для решения конкретной проблемы конкатенации. Шаблоны выражений переопределяют оператор `+` так, что он создает промежуточный тип, символически представляющий результат конкатенации двух строк или строки и строкового выражения. Шаблоны выражений откладывают выделение памяти и копирование до конца выражения, выполняя только одно выделение памяти — при присваивании шаблона выражения строке. Шаблоны выражений совместимы с `std::string`. Существующий код может значительно повысить производительность, если строковое выражение представляет собой длинный список сцепленных подстрок. Та же концепция может быть расширена на всю строковую библиотеку.

The Better String Library (`http://bstring.sourceforge.net/`)

Этот архив кода содержит реализацию строк общего назначения, которая отличается от `std::string`, но содержит некоторые мощные возможности. Если многие строки построены из частей других строк, `bstring` позволяет одной строке формироваться из смещения и длины в другой строке. Я работал с собственной реализацией этой идеи, которая оказалась очень эффективной. Для библиотеки `bstring` имеется класс-обертка C++ с именем `CBString`.

`rope<T,alloc>` (`https://www.sgi.com/tech/stl/Rope.html`)

Эта строковая библиотека подходит для вставок и удалений в очень длинных строках. Она не совместима с `std::string`.

Boost String Algorithms (`http://bit.ly/booststring`)

Эта библиотека строковых алгоритмов дополняет функции-члены `std::string`. Они строятся вокруг концепции поиска и замены.

Использование лучшего менеджера памяти

Внутри каждого объекта `std::string` имеется динамически выделенный массив `char`. `std::string` является специализацией более общего шаблона, который выглядит следующим образом:

```
namespace std {
    template < class charT,
               class traits = char_traits<charT>,
               class Alloc = allocator<charT>
             > class basic_string;

    typedef basic_string<char> string;
    ...
};
```

Третий параметр шаблона, `Alloc`, определяет распределитель памяти, или *аллокатор*, который представляет собой специализированный интерфейс к диспетчеру памяти C++. Значением `Alloc` по умолчанию является `std::allocator`, который вызывает `::operator new()` и `::operator delete()`, глобальные функции распределения памяти C++.

Поведение операторов `::operator new()` и `::operator delete()`, и объектов аллокатора более подробно рассматривается в главе 13, "Оптимизация управления памятью". Пока что я скажу только то, что `::operator new()` и `::operator delete()` выполняют очень сложную и большую работу, выделяя память для множества типов динамических переменных. Они должны работать с большими и малыми объектами, а также в одно- и многопоточных программах. Их дизайн представляет собой компромисс, направленный на достижение такой обобщенности. Иногда более специализированный аллокатор может работать лучше. Таким образом, `Alloc` может быть определен иначе, чем по умолчанию, чтобы обеспечить специализированный механизм распределения памяти для `std::string`.

Я написал очень простой аллокатор, чтобы продемонстрировать, какого повышения производительности можно достичь таким образом. Этот механизм распределения управляет несколькими блоками памяти фиксированного размера. Я создал новый typedef для типа строки, которая использует этот аллокатор. Затем я заменил исходную, очень неэффективную, версию функции remove_ctrl() версией с использованием специальных строк, как показано в примере 4.12.

Пример 4.12. Исходная версия remove_ctrl() с простым аллокатором блоков фиксированного размера

```
typedef std::basic_string<
    char,
    std::char_traits<char>,
    block_allocator<char, 10>> fixed_block_string;

fixed_block_string remove_ctrl_fixed_block(std::string s) {
    fixed_block_string result;
    for (size_t i=0; i<s.length(); ++i) {
        if (s[i] >= 0x20)
            result = result + s[i];
    }
    return result;
}
```

Результат был потрясающим. Функция remove_ctrl_fixed_block() выполнила тот же тест примерно в 7,7 раза быстрее исходной версии.

Изменение аллокаторов — занятие не для слабонервных. Нельзя присваивать строки с различными аллокаторами одна другой. Показанный здесь пример работает только потому, что s[i] представляет собой char, а не односимвольную строку std::string. Можно скопировать содержимое одной строки в другую путем его преобразования в строку C, например, с помощью result = s.c_str();.

Изменение всех вхождений std::string на fixed_block_string существенно влияет на всю базу исходных текстов. По этой причине, если команда полагает, что в процессе работы может поменять представление строк, то хорошей идеей является применение typedef для всего проекта на раннем этапе проектирования:

```
typedef std::string MyProjString;
```

Тогда эксперименты, включающие глобальные изменения, могут быть проведены в одном месте. Этот способ будет работать, если новая строка будет иметь те же функции-члены, что и та, которую она заменяет. Строки std::basic_string с разными аллокаторами удовлетворяют этому условию.

Устранение преобразования строк

К сложностям современного мира можно отнести и то, что имеется несколько видов символьных строк. Как правило, строковые функции позволяют сравнивать, присваивать или использовать в качестве операндов или аргументов только строки одной разновидности, так что программист вынужден постоянно преобразовывать строки одного вида в другие. Любое преобразование, которое включает копирование

символов или выделение динамической памяти, является возможностью повышения производительности.

Хотя сама библиотека функций преобразования также может подвергнуться повышению производительности, важнее то, что дизайн большой программы может ограничить применение преобразований.

Преобразование строк в стиле C в `std::string`

Распространенным источником расходования впустую процессорного времени являются ненужные преобразования из строк в стиле C с завершающим нулевым символом в `std::string`, например:

```
std::string MyClass::Name() const {
    return "MyClass";
}
```

Эта функция должна преобразовать строковую константу `"MyClass"` в `std::string`, выделяя память для хранения и копирования символов в `std::string`. C++ делает это преобразование автоматически, так как `std::string` имеет конструктор, который принимает аргумент типа `const char*`.

Преобразование в `std::string` является ненужным. `std::string` имеет конструктор, который принимает аргумент типа `const char*`, поэтому, когда значение, возвращаемое `Name()`, присваивается строке или передается функции, которая принимает строковый аргумент, преобразование выполняется автоматически. Предыдущая функция может так же легко быть написана следующим образом:

```
char const* MyClass::Name() const {
    return "MyClass";
}
```

Так преобразование возвращаемого значения откладывается до момента, когда оно на самом деле используется. С точки зрения использования преобразование часто оказывается излишним:

```
char const* p = myInstance->Name(); // Преобразование не требуется
std::string s = myInstance->Name(); // Преобразование в std::string
std::cout << myInstance->Name();    // Преобразование не требуется
```

Большой проблемой преобразование строк делает то, что большая программная система может иметь несколько слоев. Если один слой принимает `std::string`, а слой ниже принимает `char*`, то может иметься код, который обращает преобразование в `std::string`:

```
void HighLevelFunc(std::string s) {
    LowLevelFunc(s.c_str());
}
```

Преобразование между кодировками

Современные программы C++ вынуждены иметь дело со сравнениями (например) литеральной строки C (ASCII, со знаковыми байтами) и строки UTF-8 (беззнаковые байты с переменным количеством на один символ) из веб-браузера или при

преобразовании выходных символьных строк из XML-анализатора, который производит потоки слов UTF-16 (с прямым или обратным порядком байтов) в UTF-8. Число возможных комбинаций неимоверно.

Наилучший способ устранения преобразований состоит в том, чтобы выбрать единый формат для всех строк и хранить все строки в этом формате. Вы можете захотеть предоставить специализированные функции для сравнения между выбранным вами форматом и строками в стиле C с завершающим нулевым символом, так что их можно будет не преобразовывать. Я предпочитаю UTF-8, потому что он может представлять все коды символов Unicode, непосредственно сопоставим (для проверки равенства) со строками C и используется большинством браузеров.

В больших и торопливо написанных программах можно найти строку, преобразованную из исходного формата в новый, а затем обратно в исходный формат, при ее прохождении через слои программного обеспечения. Исправление ситуации заключается в том, чтобы переписать функции-члены интерфейсов классов так, чтобы они принимали один и тот же тип строк. К сожалению, эта задача подобна добавлению константной корректности в программу на языке C++. Изменение, как правило, распространяется по всей программе таким образом, что делает трудной задачей контроль области видимости изменения.

Резюме

- *Использование строк дорогостоящее из-за применения динамического распределения памяти, их поведения в выражениях как значений и требования большого количества копирований при реализации.*
- *Рассматривая строки как объекты, а не как значения, можно существенно снизить частоту распределения памяти и копирований.*
- *Резервирование памяти в строках снижает накладные расходы на ее распределение.*
- *Передача константной ссылки на строку в функцию представляет собой почти то же самое, что и передача значения, но может быть более эффективной.*
- *При передаче результата из функции через передаваемую ссылку используется память фактического аргумента, что потенциально более эффективно, чем выделение новой памяти.*
- *Оптимизация, которая только иногда удаляет накладные расходы на распределение памяти, все равно остается оптимизацией.*
- *Иногда другой алгоритм оказывается проще для оптимизации или по самой своей природе является более эффективным.*
- *Реализации классов стандартной библиотеки являются обобщенными и простыми. Они не обязательно обладают высокой производительностью или оптимальностью для любого конкретного применения.*

ГЛАВА 5

Оптимизация алгоритмов

Время лечит то, что не может вылечить разум.

— Луций Анней Сенека (4 г. до н.э.–65 г. н.э.)

Если программа выполняется на протяжении часов там, где необходимы секунды, вероятно, единственным успешным способом оптимизации является выбор более эффективного алгоритма. Большинство оптимизаций повышают производительность только на постоянный множитель. Замена неэффективного алгоритма более эффективным является единственно верным способом улучшения производительности на порядки.

Разработка эффективных алгоритмов является темой многих книг по компьютерным наукам и не менее многочисленных диссертаций. Многие ученые посвятили свою карьеру анализу алгоритмов. Такую обширную тему решительно невозможно охватить одной короткой главой. Поэтому в данной главе лишь бегло рассматривается временная стоимость алгоритмов, которая поможет вам понять, когда у вас начинаются проблемы.

Я рассмотрю распространенные алгоритмы поиска и сортировки и представлю инструментарий для оптимизации поиска и сортировки в существующих программах. Помимо выбора алгоритма, который является оптимальным для неизвестных данных, существуют алгоритмы с исключительной производительностью для отсортированных или почти отсортированных данных, или иных данных, обладающих некоторыми особыми свойствами.

Ученые изучают важные алгоритмы и структуры данных, поскольку они являются примерами того, как оптимизировать код. Я собрал некоторые важные методы оптимизации в надежде, что читатель сможет распознать места, где они могут быть применены.

Из истории оптимизационных войн

Многие проблемы программирования имеют простые решения, которые слишком медленны, и более тонкие и сложные, но относительно более эффективные решения. Наилучшим вариантом действий команды может быть поиск эксперта в области анализа алгоритмов для выяснения, имеется ли эффективное решение конкретной проблемы. Наем такого консультанта может быть эффективным вложением средств.

Однажды я был в команде разработки тестеров (включая Fluke 9010A, показанный на фото в главе 2, "Оптимизация, влияющая на поведение компьютера"). Наш тестер имел встроенную проверку памяти для диагностики дефектов тестируемого прибора. Как-то я решил проверить охват этого теста путем подключения тестера к компьютеру Commodore PET и запуска проверки его видеопамяти так, чтобы шаблоны проверки были видны на встроенном экране компьютера. Я внес ошибки в схему памяти, просто воткнув отвертку между соседними выводами чипа видеопамяти. Мы были очень удивлены, узнав, что явно видимые на экране ошибки часто оказывались не обнаруженными нашими проверками, которые разрабатывали очень умные инженеры. Кроме того, согласно закону Мура, каждые 18 месяцев удваивался объем памяти, который мы должны были тестировать. Нам был нужен новый алгоритм тестирования памяти, который был бы гораздо быстрее существующего и с лучшим выявлением ошибок.

Исчерпывающая проверка всей памяти была неприемлема, так как требовала $O(2^n)$ обращений к памяти (где n — количество адресов памяти (о записи с помощью "больших О" рассказывается ниже, в разделе "Временная стоимость алгоритмов")). Опубликованные алгоритмы тестирования памяти в то время были в основном слишком медленными, со временем работы $O(n^2)$ или $O(n^3)$, разработанные тогда, когда устройства памяти имели лишь несколько сотен слов, и совершенно не учитывавшие тенденции развития вычислительной техники. Опубликованные тесты были теоретически надежными, но требовали 30 обращений к каждой ячейке для обеспечения охвата основных сбоев. Я выдвинул идею гораздо более привлекательного теста с помощью псевдослучайных последовательностей, но мне не хватало математических знаний для демонстрации его обоснованности. Увы, интуиция не гарантирует успех, так что нам был нужен эксперт в области алгоритмов.

Звонки к моим старым знакомым профессорам в Университете Вашингтона вывели нас на аспиранта Дэвида Якобсона (David Jacobson), который был счастлив получить небольшую зарплату в дополнение к своей стипендии. Наше сотрудничество привело к лучшим тестам памяти, которые требовали только пяти проходов памяти, к нескольким другим новым алгоритмам функциональных испытаний и к полудюжине патентов.

Временная стоимость алгоритмов

Временная стоимость алгоритма представляет собой абстрактную математическую функцию, описывающую, как быстро растет стоимость алгоритма как функция от размера входных данных. На время выполнения программы на конкретном компьютере влияют многие факторы. В результате время выполнения оказывается не слишком хорошим критерием при выяснении производительности алгоритма. Временная стоимость абстрагируется от деталей, оставляя простую связь между стоимостью вычислений и входным размером. Алгоритмы могут быть разделены на ряд семейств с аналогичными затратами времени, и для каждого семейства можно изучить его общие черты. Временная стоимость хорошо описана в любом учебнике по алгоритмам и структурам данных (лично я предпочитаю книгу *The Algorithm Design Manual* Стивена С. Скина (Steven S. Skiena)), так что этот раздел будет кратким.

Стоимость времени обычно записывается с помощью "больших О", как O(*f(n)*), где *n* — некоторый важный аспект размера входных данных, а *f(n)* — функция, описывающая, сколько некоторых крупных операций выполняет алгоритм при обработке входных данных размера *n*. Функция *f(n)* обычно упрощена так, что показывает только наиболее быстро растущий член, потому что этот член доминирует в значении *f(n)* для больших значений *n*.

Используя алгоритмы поиска и сортировки в качестве примера, если *n* — количество элементов, для которых выполняется поиск или сортировка, то *f(n)* часто представляет собой количество сравнений элементов, выполняемых для того, чтобы разместить эти элементы в отсортированном порядке.

Вот очень грубое руководство по временной сложности некоторых распространенных алгоритмов и следствия этих стоимостей для времени работы программы.

O(1), или константное время

Наиболее быстрые алгоритмы из всех возможных алгоритмов имеют константное время работы. Они имеют фиксированную стоимость, которая не зависит от размера входных данных. Такие алгоритмы — как Святой Грааль: невероятно ценные, если будут найдены, но их можно искать всю жизнь и никогда не найти. *Остерегайтесь незнакомцев, которые пытаются продать вам алгоритмы, выполняющиеся за константное время.* Константа пропорциональности может быть чрезмерно высокой. Стоимость может выглядеть как одна операция, но это может быть ложная одна операция. На самом деле она может быть замаскированной стоимостью O(*n*), а то и хуже.

O($log_2 n$)

Временная стоимость может быть меньше линейной. Например, алгоритм поиска, который может разделить входные данные на две равные части на каждом шаге, требует времени O($log_2 n$). Алгоритмы с менее чем линейной временной стоимостью имеют стоимость, которая растет медленнее, чем входные данные. Они являются достаточно эффективными, так что при их наличии часто (но не всегда) нет оснований искать более быстрый алгоритм. Код реализации алгоритма просто оказывается недостаточно "горячим", чтобы появиться

в списке дорогостоящих функций, генерируемом профайлером. Алгоритмы с временной стоимостью $O(log_2 n)$ можно вызывать в программе многократно, не боясь сделать ее неприемлемо медленной. Распространенным алгоритмом с данной временной стоимостью является бинарный поиск.

$O(n)$, или линейное время

Если временная стоимость алгоритма составляет $O(n)$, то время работы такого алгоритма пропорционально размеру входных данных. Такой алгоритм называется алгоритмом с *линейным временем*. Стоимость $O(n)$ типична для алгоритмов, которые сканируют входные данные от одного конца до другого, чтобы найти наименьшее или наибольшее значение в списке. Стоимость алгоритма с линейным временем работы растет с той же скоростью, что и входные данные. Такие алгоритмы достаточно недороги, так что программа может работать со все большими и большими входными данными без неожиданно больших требований к вычислительным ресурсам. Однако *несколько алгоритмов с линейным временем можно объединить так, что их общее время выполнения будет $O(n^2)$ или еще хуже*, так что окажется, что общее время работы программы слишком велико для больших входных данных.

$O(n \cdot log_2 n)$

Алгоритм может иметь сверхлинейную временную стоимость. Например, многие алгоритмы сортировки сравнивают пары входных данных и разделяют сортируемую память на каждом шагу на две части. Такие алгоритмы имеют время выполнения $O(n \cdot log_2 n)$. Алгоритмы с временной стоимостью $O(n \cdot log_2 n)$ с ростом n становятся относительно все более и более дорогостоящими, но скорость относительного роста настолько медленная, что эти алгоритмы обычно в состоянии справиться даже с большими значениями n. Тем не менее не стоит вызывать такой алгоритм во время выполнения программы без особой необходимости.

$O(n^2)$, $O(n^3)$ и т.п.

Некоторые алгоритмы, включая менее эффективные алгоритмы сортировки, должны сравнивать каждое входное значение с каждым другим входным значением. Такие алгоритмы имеют временную стоимость $O(n^2)$. Стоимость таких алгоритмов растет настолько быстро, что становится узким местом для больших значений n. Существует целый ряд задач, которые имеют простые решения со стоимостью $O(n^2)$ или $O(n^3)$, и более интеллектуальные сложные решения, которые работают быстрее.

$O(2^n)$

Временная стоимость таких алгоритмов растет настолько быстро, что их можно рассматривать только для малых значений n. Иногда этого хватает. Алгоритмы, которые требуют проверки всех комбинаций n входных данных, являются алгоритмами с временной стоимостью $O(2^n)$. Расписания и планирования маршрутов, включая знаменитую задачу коммивояжера, являются

алгоритмами O(2). Если основная проблема решается алгоритмом со стоимостью O(2), то у разработчика остается мало вариантов действий: использование эвристического алгоритма (что не гарантирует оптимальное решение), ограничение только небольшими значениями *n* или поиск иного способа добавления входных значений, который не предполагает решения данной задачи.

В табл. 5.1 приведены оценки времени работы алгоритмов с различной временной стоимостью, если каждая операция для каждого из *n* входных значений занимает одну наносекунду. Более полную версию этой таблицы можно найти на с. 38 книги *The Algorithm Design Manual* Стивена С. Скина (Steven S. Skiena).

Таблица 5.1. Стоимость времени выполнения алгоритма, одна операция которого выполняется за 1 нс

	$\log_2 n$	n	$n \cdot \log_2 n$	n^2	2^n
10	< 1 мкс	< 1 мкс	< 1 мкс	< 1 мкс	1 мкс
20	< 1 мкс	< 1 мкс	< 1 мкс	< 1 мкс	1 мс
30	< 1 мкс	< 1 мкс	< 1 мкс	< 1 мкс	1 с
40	< 1 мкс	< 1 мкс	< 1 мкс	1,6 мкс	18 мин
50	< 1 мкс	< 1 мкс	< 1 мкс	2,5 мкс	13 дней
100	< 1 мкс	< 1 мкс	< 1 мкс	10 мкс	∞
1000	< 1 мкс	1 мкс	10 мкс	1 мс	∞
10 000	< 1 мкс	10 мкс	130 мкс	100 мс	∞
100 000	< 1 мкс	100 мкс	2 мс	10 с	∞
1 000 000	< 1 мкс	1 мс	> 20 мс	17 мин	∞

Временная стоимость в наилучшем, среднем и наихудшем случаях

Обычная запись с "большим O" предполагает, что алгоритм имеет одно и то же время для любых входных данных. Однако некоторые алгоритмы чувствительны к характеристикам входных данных, для одних последовательностей данных работая быстрее, чем для других последовательностей того же размера. При выяснении, какой алгоритм следует использовать в критичном с точки зрения производительности коде, важно знать, насколько плох наихудший случай алгоритма. С некоторыми примерами мы познакомимся в разделе "Временная стоимость алгоритмов сортировки" далее в этой главе.

Аналогично некоторые алгоритмы имеют очень высокую производительность в лучшем случае, например если входные данные уже отсортированы или почти отсортированы. Когда высока вероятность того, что входные данные могут обладать такими полезными свойствами, как, например, будучи почти отсортированными, выбор алгоритма с высокой производительностью в наилучшем случае может улучшить время выполнения программы.

Амортизированная временная стоимость

Амортизированная временная стоимость означает затраты времени в среднем для большого количества входных данных. Например, вставка элемента в пирамиду выполняется за время $O(log_2 n)$. Таким образом, построение пирамиды по одному элементу требует времени $O(n \cdot log_2 n)$. Однако построение пирамиды наиболее эффективным методом требует времени $O(n)$, а это означает, что вставка каждого элемента этим методом имеет амортизированную временную стоимость $O(1)$. Однако наиболее эффективный алгоритм не вставляет элементы по одному. Он обрабатывает все элементы в подпирамиде с помощью подхода "разделяй и властвуй".

Важность амортизированной временной стоимости заключается в том, что некоторые отдельные операции выполняются быстро, в то время как другие являются медленными. Например, амортизированная временная стоимость добавления символа к `std::string` является константной, но включает вызов диспетчера памяти при выполнении некоторой части вставок. Если строка короткая, диспетчер памяти может вызываться почти каждый раз, когда добавляется символ. Амортизированная стоимость станет малой только после того, как программа вставит тысячи или миллионы символов.

Прочие стоимости

Иногда алгоритм можно ускорить путем сохранения промежуточных результатов. Такой алгоритм имеет не только временную стоимость, но и стоимость вспомогательной памяти. Например, знакомый рекурсивный алгоритм обхода бинарного дерева выполняется за линейное время, но при этом требует $log_2 n$ стековой памяти во время рекурсии. Алгоритмы, которые имеют высокую стоимость вспомогательной памяти, могут не подходить для ограниченных сред.

Существуют алгоритмы, которые работают быстрее при распараллеливании, но имеют свою стоимость, выражающуюся в количестве параллельно работающих процессоров, необходимых для получения теоретически обещанного увеличения скорости. Алгоритмы, требующие более чем $log_2 n$ процессоров, как правило, невозможно использовать на компьютерах общего назначения, которые имеют небольшое, фиксированное число процессоров. Такие алгоритмы можно использовать при наличии специализированных аппаратных или графических процессоров. Я хотел бы, чтобы в этой книге можно было отвести место для параллельных алгоритмов, но, увы, это оказалось невозможным.

Оптимизации сортировки и поиска

Оптимизации сортировки и поиска включают три средства.

- Замена алгоритмов с плохой временной стоимостью в среднем алгоритмами, имеющими лучшую временную стоимость в среднем.
- Использование дополнительной информации о данных (например, данные обычно отсортированы или почти отсортированы) для выбора алгоритмов,

имеющих особенно хорошее поведение для данных с этими свойствами, и отказ от алгоритмов с особенно плохим поведением для таких данных.

- Настройка алгоритма для повышения его производительности на постоянный коэффициент.

Применение этих средств будет рассматриваться в главе 9, "Оптимизация сортировки и поиска".

Эффективные алгоритмы поиска

Временны*е* стоимости наиболее важных алгоритмов поиска и сортировки представлены в каждой учебной программе по информатике и прекрасно известны всем разработчикам еще со времени обучения в колледже. Основная проблема, связанная с алгоритмами и структурами данных, заключается в том, что соответствующие курсы слишком краткие. Преподаватель может всесторонне осветить лишь несколько алгоритмов, только вскользь упомянув о существовании остальных и их временно*й* стоимости. Вероятно, тот же преподаватель одновременно учит и программированию. В результате студенты покидают учебные классы, гордясь новыми знаниями и собой и даже не подозревают, какое множество нюансов они пропустили. Эта неполнота знаний приводит к возможности оптимизации кода даже при использовании номинально оптимальных алгоритмов.

Временн*ая* стоимость алгоритмов поиска

Насколько быстр в терминах "большого О" самый быстрый метод поиска записи в таблице? Подсказка: бинарный поиск с O($log_2 n$) является полезным ориентиром, но не самым быстрым алгоритмом.

Прямо сейчас многие читатели недоуменно говорят: "Подождите... *ЧТО*?!" Увы, они не учили ничего, кроме линейного и бинарного поиска. Но алгоритмов поиска куда больше.

- *Линейный поиск* с O(*n*) имеет высокую временную стоимость, но он крайне универсален. Он может использоваться для поиска в неотсортированной таблице. Все, чего он требует, — чтобы ключи можно было сравнивать на равенство, и поэтому он работает, даже если ключи нельзя разместить в определенном порядке. Конечно же, линейный поиск может завершиться до просмотра всех элементов таблицы. Это все равно дает временную стоимость O(*n*), но в среднем линейный поиск требует выполнения примерно *n/2* сравнений.

  Если мы можем изменять таблицу, то в ряде ситуаций может очень хорошо работать версия линейного поиска, которая перемещает каждый результат поиска в начало таблицы. Например, поиск в таблице символов в компиляторе языка программирования выполняется всякий раз при использовании идентификатора в выражении. Обычно имеется так много выражений, которые выглядят как `i = i + 1;`, что эта оптимизация может сделать применение линейного поиска достаточно полезным.

- *Бинарный поиск* с $O(log_2 n)$ имеет хорошую производительность, но это не лучший возможный поиск. Бинарный поиск требует отсортированных по ключу поиска входных данных, и ключи, которые можно сравнивать не только на равенство, но и для такого отношения упорядочения, как "меньше чем".

  Бинарный поиск является рабочей лошадкой в мире сортировки и поиска. Это алгоритм "разделяй и властвуй", который многократно делит отсортированную таблицу пополам, сравнивая ключ со средним элементом той части таблицы, в которой выполняется поиск, для определения, где находится искомый элемент — до или после среднего.
- *Интерполяционный поиск* делит отсортированный таблицу на две части, как и бинарный поиск, но использует дополнительные знания о ключах для улучшения разбиения. Интерполяционный поиск при равномерном распределении ключей достигает очень респектабельной производительности $O(log\ log\ n)$. Это улучшение может быть весьма заметным, если таблица очень велика или если имеет важное значение стоимость проверки записи таблицы (например, когда она находится на диске). Однако и интерполяционный поиск не является наиболее быстрым.
- Можно найти запись в среднем за время $O(1)$, если использовать *хеширование*: преобразовать ключ в индекс массива в хеш-таблице. Хеширование не работает с произвольным списком пар "ключ/значение". Оно требует специально сконструированной таблицы. Записи в хеш-таблице сравниваются только на равенство. Хеширование имеет наихудшую производительность $O(n)$ и может потребовать большего количества записей хеш-таблицы, чем имеется искомых записей. Однако, если таблица имеет фиксированное содержимое (например, названия месяцев или ключевые слова языка программирования), эти патологические случаи могут быть устранены.

Все поиски равноценны при малых *n*

Какова стоимость поиска в таблице с одной записью? Играет ли при этом роль, какой алгоритм поиска используется? В табл. 5.2 показана стоимость поиска в отсортированной таблице с использованием лучших возможных версий линейного, бинарного поиска и поиска в хеш-таблице. Ответ заключается в том, что *для небольших таблиц все методы исследуют одинаковое количество записей*. Однако для таблицы со 100 тысячами записей будет доминировать член функции временной стоимости, и все станет выглядеть иначе.

Таблица 5.2. Количество обращений при поиске в таблицах разного размера

Размер таблицы	Линейный поиск	Бинарный поиск	Хеш-таблица
1	1	1	1
2	1	2	1
4	2	3	1
8	4	4	1

Окончание табл. 5.2

Размер таблицы	Линейный поиск	Бинарный поиск	Хеш-таблица
16	8	5	1
26	13	6	1
32	16	6	1

Эффективные алгоритмы сортировки

Существует удивительный зверинец алгоритмов сортировки. Многие из них были предложены только в последние 10 лет или около того. Многие из новейших сортировок являются гибридными, такими, которые улучшают производительность наилучшего или наихудшего случая. Если вы завершили образование до 2000 года, вам, определенно, стоит почитать современную литературу. Доступные резюме алгоритмов сортировки можно найти в Википедии. Вот несколько забавных фактов (о которых не всегда рассказывают на лекциях), доказывающих, что более глубокие знания имеют преимущества.

- "Все знают", что оптимальная сортировка выполняется за время $O(n \cdot log_2 n)$, правильно? Неправильно. На самом деле неправильно дважды. Такое поведение имеют только те виды сортировки, которые работают путем сравнения пар входных значений. Поразрядные сортировки (которые многократно помещают входные данные в одну из r корзин) имеют временную стоимость $O(n \cdot log_r n)$, где r — это основание системы счисления, или количество корзин сортировки. Это значительно лучше, чем сортировка сравнением больших наборов данных. Кроме того, когда ключи сортировки выбираются из определенных множеств, например если это последовательные целые числа от 1 до n, мигающая сортировка (flashsort) может отсортировать эти данные за время $O(n)$.
- Часто реализуемая и в целом неплохая быстрая сортировка имеет в худшем случае производительность $O(n^2)$. Достоверно избежать худшего случая невозможно, а наивные реализации этой сортировки часто работают достаточно плохо.
- Некоторые сортировки, в том числе сортировка вставками, имеют отличную (линейную) производительность, если данные почти отсортированы, хотя для случайных данных она работает не так хорошо. Другие разновидности (такие, как наивная реализация быстрой сортировки, о которой я только что упоминал) имеют для почти отсортированных данных наихудший случай. Если обычно приходится сортировать почти отсортированные данные, знание этого дополнительного свойства можно использовать при выборе сортировки, имеющей хорошую производительность для отсортированных данных.

Временная стоимость алгоритмов сортировки

В табл. 5.3 приведена временная сложность нескольких сортировок для наилучшего случая, в среднем и для наихудшего случая. Хотя большинство из этих сортировок имеют одну и ту же среднюю производительность $O(n \cdot log_2 n)$, их наилучшая и

наихудшая производительности различаются, как и размер дополнительной памяти, необходимый для их работы.

Таблица 5.3. Временная стоимость некоторых алгоритмов сортировки

Сортировка	Наилучший случай	В среднем	Наихудший случай	Требуемая память	Примечания к наилучшему/наихудшему случаям
Сортировка вставками	n	n^2	n^2	1	Наилучший случай — почти отсортированные данные
Быстрая сортировка	$n \cdot log_2 n$	$n \cdot log_2 n$	n^2	$log_2 n$	Наихудший случай — отсортированные данные и наивный выбор опорного элемента (первый/последний)
Сортировка слиянием	$n \cdot log_2 n$	$n \cdot log_2 n$	$n \cdot log_2 n$	1	
Сортировка бинарным деревом	$n \cdot log_2 n$	$n \cdot log_2 n$	$n \cdot log_2 n$	n	
Пирамидальная сортировка	$n \cdot log_2 n$	$n \cdot log_2 n$	$n \cdot log_2 n$	1	
Тимсорт	n	$n \cdot log_2 n$	$n \cdot log_2 n$	n	Наилучший случай — отсортированные данные
Интроспективная сортировка	$n \cdot log_2 n$	$n \cdot log_2 n$	$n \cdot log_2 n$	1	

Замена сортировки с плохой производительностью в наихудшем случае

Быстрая сортировка очень популярна. Ее внутренние накладные расходы довольно низки, а средняя производительность оптимальна для сортировки на основе сравнения двух ключей. Но у нее есть и недостаток. Если вы запустите быструю сортировку для массива, который уже отсортирован (или почти отсортирован), и в качестве опорного используете первый или последний элемент, то производительность окажется очень плохой. Сложные реализации быстрой сортировки, чтобы избавиться от этого недостатка, выбирают опорный элемент случайным образом или тратят время на вычисление медианы и используют ее в качестве опорного элемента. Таким образом, наивно полагать, что быстрая сортировка всегда будет иметь хорошую производительность. Вы либо должны иметь дополнительную информацию о входных данных — в частности, что они далеки от отсортированного состояния, — либо знать, что реализация алгоритма тщательно выбирает опорный элемент.

Если о входных данных ничего не известно, то сортировка слиянием, деревом или пирамидальная сортировка обеспечивает уверенность в том, что патологических случаев, снижающих производительность до неприемлемого уровня, не будет.

Использование информации о входных данных

Если входные данные отсортированы или почти отсортированы, то обычно неприемлемая сортировка вставками обеспечивает для них отличную производительность $O(n)$.

Сравнительно новая гибридная сортировка Тимсорт (Timsort) также имеет отличную производительность $O(n)$, если данные отсортированы или почти отсортированы, и оптимальную производительность $O(n \cdot log_2 n)$ в остальных случаях. Эта сортировка в настоящее время является стандартной сортировкой в языке Python.

Последняя сортировка, которая называется интроспективной, — это гибрид быстрой и пирамидальной сортировок. Интроспективная сортировка начинает выполнять быструю сортировку, но переключается на пирамидальную сортировку, если плохие входные данные приводят к слишком большой глубине рекурсии быстрой сортировки. Данная сортировка гарантирует разумное наихудшее время $O(n \cdot log_2 n)$, используя преимущества эффективной реализации быстрой сортировки для незначительного сокращения времени выполнения в среднем случае. Начиная с C++11, интроспективная сортировка является предпочтительной реализацией `std::sort()`.

Еще одна недавно предложенная сортировка — мигающая (flashsort) — имеет отличную производительность $O(n)$ для данных, полученных на основе определенного распределения вероятностей. Она связана с поразрядной сортировкой; данные сортируются в корзины на основе процентилей распределения вероятностей. Простой случай мигающей сортировки осуществляется для равномерного распределения элементов данных.

Шаблоны оптимизации

Опытные разработчики в оптимизации не полагаются исключительно на головокружительные интуитивные прозрения в поисках возможностей для улучшения производительности кода. Есть ряд шаблонов, которые повторяются в оптимизированном коде. Разработчики изучают алгоритмы и структуры данных отчасти потому, что в них содержится библиотека идей для повышения производительности.

В этом разделе собрано несколько общих методов улучшения производительности, которые настолько полезны, что заслуживают особого упоминания. Читатель может узнать некоторые из них как старых знакомых по структурам данных, функциям языка C++ или аппаратным новшествам.

Предвычисления

Устранение вычислений из горячей части программы путем их выполнения до горячего кода — ранее в программе или во время компоновки, компиляции или проектирования.

Отложенные вычисления

Устранение вычислений из некоторых путей в коде путем их выполнения ближе к точке, в которой потребуются их результаты.

Пакетирование

Выполнение вычислений для нескольких элементов вместе, а не по одному элементу за раз.

Кеширование

Уменьшение количества вычислений путем сохранения и повторного использования результатов дорогостоящих вычислений без вычисления их заново.

Специализация

Уменьшение количества вычислений путем устранения неиспользуемой общности.

Группировка

Снижение стоимости повторяемых операций путем работы с бо́льшими группами входных данных за раз.

Подсказки

Уменьшение количества вычислений путем предоставления подсказок, которые могут повысить производительность.

Оптимизация ожидаемого пути

Тестирование входных данных или событий во время выполнения в порядке убывания ожидаемой частоты.

Хеширование

Вычисление компактного числового значения (хеша) для больших структур данных, таких как строка символов переменной длины. Хеш может представлять структуры данных в сравнениях для повышения производительности.

Двойная проверка

Уменьшение количества вычислений путем выполнения недорогой проверки, за которой дорогостоящая проверка следует только в случае необходимости.

Предвычисления

Предвычисления — весьма общий метод, целью которого является удаление вычислений из горячего кода путем выполнения его ранее, до выполнения кода в горячей точке. Имеются различные варианты перемещения вычислений из горячей точки: к менее горячей части кода, во время компоновки, во время компиляции или даже во время проектирования. Словом, чем раньше можно выполнить вычисления, тем лучше.

Предвычисления возможны только постольку, поскольку вычисляемое значение не зависит от контекста. Вычисления, такие как показанное ниже, могут вычисляться компилятором заранее потому, что они не зависят от чего-либо в программе:

```
int sec_per_day = 60 * 60 * 24;
```

Связанные вычисления наподобие следующего зависят от значений переменных при выполнении программы:

```
int sec_per_weekend = (date_end - date_beginning + 1) * 60 * 60 * 24;
```

Трюк с предвычислением состоит в том, что, например, выясняется, что в программе `(date_end-date_beginning+1)` всегда равно 2, так что можно заменить это выражение конкретным значением или найти часть выражения, которая может быть вычислена заранее.

Вот несколько примеров предвычислений.

- Компилятор C++ автоматически выполняет предварительное вычисление значения константных выражений с помощью встроенных правил приоритетов и ассоциативности операторов. В приведенном выше примере компилятор без проблем выполняет предварительное вычисление значения `sec_per_day`.
- Вызов шаблонной функции с аргументами шаблона, являющимися конкретными значениями, вычисляется во время компиляции. Компилятор генерирует эффективный код, когда аргументы являются константами.
- Проектировщик может заметить, например, что в рамках конкретной программы понятие "выходные" всегда составляет два дня, так что он предварительно вычисляет данную константу для ее использования в программе.

Отложенные вычисления

Целью отложенных вычислений является задержка вычисления до точки, более близкой к тому месту кода, где требуется результат этого вычисления. У отложенных вычислений есть несколько преимуществ. Если вычисление не требуется на всех путях выполнения (всех ветвях `if-then-else`) функции, оно выполняется только на том пути, где необходим его результат. Вот пара примеров.

Построение в два этапа

Часто информация, необходимая для создания объекта, недоступна в то время, когда объект может быть создан статически. Вместо создания всего объекта сразу конструктор содержит минимальный код создания пустого объекта. Позже работающая программа вызывает инициализирующую функцию-член для завершения построения объекта. Задержка инициализации до получения дополнительных данных означает, что созданный объект часто может быть эффективной "плоской" структурой данных (см. раздел "Плоские структуры данных" главы 6, "Оптимизация переменных в динамической памяти").

В некоторых случаях проверка того, не было ли уже проведено отложенное вычисление, имеет определенную стоимость. Эта стоимость сопоставима со

стоимостью выяснения, корректен ли указатель на объект динамически созданного класса.

Копирование при записи

Вместо копирования динамического члена-данных при копировании объекта два экземпляра совместно используют одну и ту же копию динамической переменной. Копирование откладывается до тех пор, пока один из экземпляров не захочет изменить эти данные.

Пакетирование

Целью пакетирования является сбор несколько рабочих элементов и их совместная обработка. Пакетная обработка может использоваться для удаления повторных вызовов функций или других вычислений, которые будут происходить при обработке элементов по одному. Этот метод может также использоваться из-за наличия более эффективного алгоритма для обработки всех входных данных вместе или для того, чтобы отложить вычисление на то время, когда будет доступно больше вычислительных ресурсов.

- Буферизованный вывод является примером пакетирования, когда выводимые символы добавляются в буфер до тех пор, пока буфер не будет заполнен или программа не достигнет конца строки или конца файла. В процедуру вывода передается весь буфер, экономя тем самым стоимость вызова процедуры вывода для каждого символа.
- Оптимальный метод преобразования неотсортированного массива в пирамиду является примером пакетирования для использования более эффективного алгоритма. Затраты времени на вставку n элементов в пирамиду по одному составляют $O(n \cdot log_2 n)$. Стоимость создания пирамиды сразу, за один этап равна $O(n)$.
- Многопоточная очередь задач представляет собой пример пакетирования для эффективного использования компьютерных ресурсов.
- Сохранение или обновление в фоновом режиме также является примером пакетирования.

Кеширование

Кеширование представляет собой любой из множества методов снижения количества вычислений путем сохранения и повторного использования результата дорогостоящих вычислений вместо того, чтобы проводить эти вычисления всякий раз, когда они нужны, заново.

- Компилятор кеширует результат малых повторяющихся фрагментов кода наподобие вычислений, необходимых для разыменования элемента массива. Он может встретить инструкцию наподобие `a[i][j] = a[i][j] + c;` и сохранить выражение для элемента массива, производя код, который выглядит скорее как `auto p = &a[i][j]; *p = *p + c;`.

- Кеш-память представляет собой специальные схемы в компьютерах, которые обеспечивают для часто необходимых значений в памяти более быстрый доступ со стороны процессора. Кеширование является очень важной концепцией в разработке компьютерного оборудования. Есть много уровней кеширования в аппаратном и программном обеспечении персональных компьютеров с архитектурой x86.
- Длина символьной строки в стиле C должна вычисляться каждый раз, когда она нужна, с помощью подсчета символов. `std::string` кеширует длину строки, а не вычисляет ее каждый раз, когда она необходима.
- Пул потоков представляет собой кеш потоков с дорогостоящим созданием.
- Динамическое программирование представляет собой алгоритмический метод, в котором вычисление, обладающее рекуррентным характером, ускоряется путем решения подзадач и кеширования полученных при этом результатов.

Специализация

Специализация является противоположностью обобщения. Цель специализации — ликвидировать часть дорогостоящих вычислений, которые не нужны в конкретной ситуации.

Можно упростить действие (или структуру данных), удалив возможность, которая делает его дорогостоящим, но не нужна в данной конкретной ситуации. Этого можно достичь путем устранения ограничения из задачи или добавления ограничения к реализации, например делая динамическую память статической, накладывая ограничения на в общем случае неограниченную величину и т.д.

- Шаблонная функция `std::swap()` имеет реализацию по умолчанию, которая может копировать свои аргументы. Однако разработчик может предоставить специализацию, которая оказывается более эффективной из-за учета информации о внутреннем устройстве структуры данных. (Версия C++11 функции `std::swap()` использует для достижения эффективного результата семантику перемещения, если тип аргумента реализует перемещающий конструктор.)
- Строка `std::string` может динамически менять размер для хранения символьных строк переменной длины. Этот класс предоставляет множество операций для работы со строками. Если же нам нужно только сравнивать фиксированные строки, то массив в стиле C или указатель на литерал строки и применение функции сравнения может дать лучшую производительность.

Группировка

Целью группировки является сокращение числа итераций повторяющихся операций и сокращение накладных расходов, связанных с повторением. Стратегии группировки таковы.

- Запрос больших блоков входных данных от операционной системы или отправка больших блоков на вывод для снижения накладных расходов, связанных с

вызовами функций ядра для малых блоков или отдельных элементов. Обратной стороной медали является то, что большее количество байтов, особенно при записи, может привести к потере данных при аварийном завершении работы программы. Это может быть проблемой, например, для файлов журнала.

- Перемещение или очистка буферов словами или двойными словами вместо побайтного. Такая оптимизация улучшает производительность только тогда, когда оба диапазона выровнены на границы одинакового размера.
- Сравнение строк по словам или двойным словам. Этот способ работает только на машинах с обратным порядком байтов, но не на архитектуре x86 с прямым порядком байтов (см. раздел "Остроконечные и тупоконечные слова" главы 2, "Оптимизация, влияющая на поведение компьютера"). Такие хакерские штучки, зависящие от типа используемой машины, могут быть опасны из-за непереносимости.
- Выполнение большего количества работы при активизации потока. Чем большую работу выполнит поток перед тем, как передаст управление другому потоку, тем меньшими будут накладные расходы на повторяющиеся пробуждения потоков из приостановленного состояния.
- Обслуживающие действия можно выполнять не на каждой итерации цикла, а на каждой, скажем, 8- или 128-й итерации.

Подсказки

Будучи предоставленной, подсказка может уменьшить необходимое количество вычислений и тем самым снизить стоимость операции.

Например, одна из перегрузок функции-члена `insert()` класса `std::map` принимает необязательный аргумент точки вставки. Оптимальная подсказка может сделать вставку операцией со стоимостью O(*1*); в противном случае вставка в `std::map` имеет стоимость O($log_2 n$).

Оптимизация ожидаемого пути

В коде `if-then-else` с несколькими `else-if`, если проверки указаны в случайном порядке, при прохождении блока `if-then-else` будет вычисляться в среднем около половины всех условий. Если же некоторый случай происходит с вероятностью 95% и соответствующий ему тест выполняется первым, то в 95% случаев будет выполняться только один этот тест.

Хеширование

В случае хеширования большая входная структура данных или длинная символьная строка превращается алгоритмом в некоторое целочисленное значение, которое называется "хеш". Хеши двух входных данных могут эффективно исключить равенство самих входных данных. Если хеши разные, то, безусловно, входные данные тоже разные. Если же хеши совпадают, то входные данные, скорее всего, равны. Хеширование может использоваться с двойной проверкой для оптимизации

детерминистического сравнения на равенство. Обычно хеш для входных данных кешируется, чтобы его не нужно было пересчитывать.

Двойная проверка

При двойной проверке для исключения некоторых случаев используется недорогая проверка, а затем при необходимости для исключения остальных случаев используется дорогостоящая проверка.

- Двойная проверка часто используется с кешированием. Чтобы узнать, есть ли в кеше нужное значение, выполняется быстрая проверка, и если его там нет, то выполняется его выборка или вычисление с помощью более дорогостоящего процесса.
- Сравнение на равенство двух объектов `std::string` обычно требует их посимвольного сравнения. Однако предварительное сравнение длин двух строк может быстро исключать равенство.
- Двойная проверка может использоваться с хешированием. Хеши двух входных данных можно эффективно сравнить на равенство. Если хеши разные, входные данные не равны; и только если хеши совпадают, необходимо сравнить данные побайтно.

Резюме

- *Остерегайтесь незнакомцев, предлагающих алгоритмы с константной временной стоимостью. На самом деле они могут быть алгоритмами O(n).*
- *Несколько эффективных алгоритмов могут быть скомбинированы таким образом, что общая производительность упадет до $O(n^2)$ или до еще худшего значения.*
- *Бинарный поиск с временной стоимостью $O(\log_2 n)$ не является самым быстрым. Интерполяционный поиск имеет стоимость O(log log n), а хеширование — константную временную стоимость.*
- *Для малых таблиц менее чем с четырьмя значениями все алгоритмы поиска просматривают примерно одинаковое количество записей.*

ГЛАВА 6

Оптимизация переменных в динамической памяти

Потому что там есть деньги.

— Грабитель банков Вилли Саттон (Willie Sutton) (1901–1980)

Приписывается Саттону в качестве ответа на вопрос репортера "Почему вы грабите банки?" Позже Саттон отрицал, что когда-либо говорил эту фразу.

За исключением использования неоптимальных алгоритмов, наивное использование динамически выделяемых переменных является наибольшим грабителем производительности в программах C++. Использование динамически выделяемых переменных настолько часто является местом, "где собака зарыта" (или, для упомянутого в эпиграфе Саттона, где лежат деньги), что *разработчик может эффективно оптимизировать код, не делая ничего, кроме уменьшения количества вызовов менеджера памяти.*

Возможности C++, использующие динамически выделяемые переменные, такие как контейнеры стандартной библиотеки, интеллектуальные указатели и строки, повышают продуктивность написания приложений на C++. Но эта выразительность имеет свою темную сторону. Когда производительность имеет значение, `new` из друга превращается во врага[1].

Чтобы не начинать панику, позвольте мне сразу же сказать, что цель оптимизации управления памятью не в том, чтобы жить жизнью аскета, не зная радостей применения многих полезных возможностей C++, использующих динамически выделяемые переменные. Скорее цель заключается в том, чтобы *устранить грабеж производительности, ненужные вызовы диспетчера памяти* путем умелого использования этих же возможностей.

Мой опыт говорит, что удаления даже одного вызова диспетчера памяти из цикла или часто вызываемой функции достаточно, чтобы значительно повысить производительность; и, как правило, имеются возможности для удаления куда более одного вызова.

---

[1] Используя "грабительскую" аналогию автора, можно добавить: "дружба дружбой, а денежки врозь". — *Примеч. пер.*

Прежде чем говорить о том, как оптимизировать использование динамически выделяемых переменных, в этой главе речь пойдет о том, как C++ работает с переменными. Здесь также рассматривается набор инструментов API динамической памяти.

Переменные C++

Каждая переменная C++ (каждая переменная простого типа данных; каждый массив, структура или экземпляр класса) имеет фиксированное размещение в памяти, а ее размер известен во время компиляции. C++ позволяет программе получить как размер переменной в байтах, так и указатель на эту переменную, но не побитовую схему ее размещения в памяти. Правила C++ позволяют разработчику знать порядок и размещение переменных-членов в структурах. C++ также предоставляет типы объединений, в которых несколько переменных разных типов используют один блок памяти, но что именно программа видит при обращении к объединению — зависит от конкретной реализации.

Длительность хранения переменной

Каждая переменная имеет свое *время (длительность) хранения*, или время жизни, которое описывает период времени, в течение которого *хранилище*, или байты памяти, занятые переменной, имеет содержательное значение. Стоимость выделения памяти для переменной зависит от времени хранения.

Синтаксис C++ объявления переменных может выглядеть запутанным из-за необходимости поддерживать совместимость с синтаксисом C при добавлении нескольких новых концепций. В C++ длительность хранения не указывается непосредственно, но выводится из объявления переменной.

Статическая длительность хранения

Переменные со *статической длительностью хранения* располагаются в памяти, зарезервированной компилятором. Каждая статическая переменная занимает фиксированное количество памяти по фиксированному адресу, определяемому во время компиляции. Память для статических переменных резервируется на время работы программы. Каждая глобальная статическая переменная создается до выполнения функции `main()` и уничтожается после ее завершения. Статические переменные, объявленные в области видимости функции, создаются "до первого входа в функцию во время выполнения", т.е. такая переменная может быть создана как одновременно с глобальными статическими переменными, так и позже, при первом вызове этой функции. C++ определяет порядок создания и уничтожения глобальных статических переменных, так что разработчик может точно говорить о продолжительности их жизни, но соответствующие правила настолько сложны, что служат скорее предупреждением, чем описанием того, как их использовать.

Каждая статическая переменная имеет собственное имя, по которому к ней можно обратиться; доступ к статической переменной возможен также с помощью указателей и ссылок. Имя, указывающее статическую переменную, а

также указатели и ссылки на нее, могут быть видимыми коду в конструкторах и деструкторах других статических переменных, прежде чем эта переменная получит содержательное значение, и после того, как ее значение будет уничтожено.

Нет никакой стоимости времени выполнения создания хранилища статической переменной. Однако это хранилище не доступно для повторного использования. Таким образом, статические переменные подходят для данных, которые будут использоваться на протяжении всей жизни программы.

Переменные, объявленные в области видимости пространства имен, и переменные, объявленные как `static` или `extern`, имеют статическое время хранения.

Длительность хранения, локальная по отношению к потоку

Начиная с C++11, программа может объявлять переменные с *длительностью хранения, локальной по отношению к потоку*. До C++11 некоторые компиляторы и каркасы предоставляли аналогичные возможности в качестве нестандартных расширений.

Переменные, локальные по отношению к потоку, создаются при входе в поток и уничтожаются при выходе из потока; их время жизни совпадает со временем существования потока. Каждый поток имеет собственную отдельную копию этих переменных.

В зависимости от операционной системы и компилятора доступ к переменным, локальным по отношению к потоку, может быть более дорогостоящим, чем доступ к статическим переменным. В некоторых системах память для хранения таких переменных выделяется потоком, так что стоимость доступа к переменной, локальной по отношению к потоку, на одно косвенное обращения больше, чем стоимость доступа к статической переменной. В других системах переменные, локальные по отношению к потоку, должны быть доступны посредством глобальной таблицы, индексированной идентификатором потока. Хотя эта операция может быть выполнена за константное время, она требует вызова функции и некоторых вычислений, делающих стоимость доступа значительно большей.

В C++11 переменные, объявленные с ключевым словом `thread_local` в качестве спецификатора класса памяти, имеют длительность хранения, локальную по отношению к потоку.

Автоматическая длительность хранения

Переменные с *автоматической длительностью хранения* существуют в памяти, зарезервированной компилятором в стеке вызовов функции. Каждая автоматическая переменная занимает фиксированный объем памяти с фиксированным смещением от указателя стека, которое определяется во время компиляции, но абсолютный адрес автоматической переменной не известен до тех пор, пока выполнение не входит в область видимости объявления.

Автоматические переменные существуют во время выполнения в области видимости окружающего блока кода, ограниченного фигурными скобками. Они создаются в точке, где они объявлены, и уничтожаются, когда выполнение покидает область видимости.

Подобно статическим переменным, автоматические переменные также доступны по имени. В отличие от статических переменных, имя доступно только после создания переменной и до того, как она будет уничтожена. Ссылки и указатели на автоматические переменные могут существовать и после окончания времени жизни переменной, вызывая хаос неопределенного поведения при попытке их разыменования.

Как и в случае статических переменных, нет никакой стоимости выделения памяти для хранилища автоматических переменных. Но, в отличие от статических переменных, в любой момент времени имеется ограничение на общий объем памяти, который может быть занят автоматическими переменными. Превышение этого максимального значения из-за частых рекурсий или вызовов очень глубоко вложенных функций вызывает *переполнение стека* и аварийное завершение программы. Автоматические переменные являются подходящими для объектов, используемых окружающим их кодом.

Переменные формальных аргументов функции имеют автоматическую длительность хранения. То же самое относится и к переменным, объявленным внутри выполнимых блоков и не имеющих иных модификаторов.

Динамическая длительность хранения

Переменные с *динамической* длительностью хранения располагаются в памяти, запрошенной работающей программой. Программа выполняет вызов функции *диспетчера памяти*, который представляет собой набор системных функций и структур данных среды выполнения C++, которые администрируют пул памяти от имени программы. Программа явно запрашивает память и создает динамическую переменную с помощью *выражения* `new` (более подробно рассматривается в разделе "Построение динамических переменных с помощью выражений new" главы 13, "Оптимизация управления памятью"), которое может находиться в любом месте программы. Позже программа явно удаляет переменную и возвращает выделенную для нее память диспетчеру памяти в *выражении* `delete` (см. раздел "Уничтожение динамических переменных с помощью выражения delete" главы 13, "Оптимизация управления памятью"). Это может произойти в программе в любой момент, когда переменная больше не нужна.

Как и для автоматических переменных (но не для статических), адрес динамической переменной определяется во время выполнения.

В отличие от статических, локальных по отношению к потоку и автоматических переменных, синтаксис объявления массивов расширен так, что наивысшая размерность динамического массива может быть указана во время выполнения с помощью (неконстантного) выражения. Это единственный случай в C++, когда размер переменной не фиксирован во время компиляции.

Динамическая переменная не имеет собственного имени. Вместо этого, когда она будет создана, диспетчер памяти C++ возвращает указатель на динамическую переменную. Программа должна присвоить этот указатель переменной, чтобы иметь возможность вернуть динамически выделенную память диспетчеру памяти перед уничтожением последнего указателя на эту переменную (в противном случае имеется риск исчерпания памяти, используемой для создания динамических переменных). Если динамические переменные не возвращаются должным образом, современный процессор может исчерпать гигабайты памяти за минуты.

В отличие от статических переменных и переменных, локальных по отношению к потоку, количество и тип динамических переменных могут изменяться со временем без каких-либо фиксированных ограничений на объем потребляемой памяти. Кроме того, в отличие от статических и автоматических переменных, имеется значительная стоимость времени выполнения, связанная с управлением памятью, используемой динамическими переменными.

Переменные, возвращаемые *выражениями* `new`, имеют динамическую длительность хранения.

Владение переменными

Еще одной важной концепцией, связанной с переменными C++, является концепция *владения*. Владелец переменной определяет, когда переменная создается и когда она будет уничтожена. Иногда длительность хранения говорит о том, когда создается и уничтожается та или иная переменная, но владение — это отдельная концепция, имеющая важное значение для оптимизации динамических переменных.

Глобальное владение

Переменными со статической длительностью хранения владеет вся программа в целом. Они создаются до выполнения функции `main()` и уничтожаются после того, как программа возвращается из `main()`.

Владение лексической области видимости

Переменными с автоматической длительностью хранения владеет лексическая область видимости, состоящая из блока кода, окруженного фигурными скобками. Это может быть тело функции; управляемый блок инструкции `if`, `while`, `for` или `do`; конструкция `try` или `catch`; составной оператор, отделенный фигурными скобками. Переменные создаются при входе выполнения в лексическую область видимости и уничтожаются, когда выполнение покидает лексическую область видимости.

Внешняя лексическая область видимости — вход в которую выполняется первым и которая покидается последней — это тело функции `main()`. Для всех применений автоматическая переменная, объявленная в `main()`, имеет то же время жизни, что и статическая переменная.

Владение членом

Переменные-члены классов и структур принадлежат экземпляру класса, который их содержит. Они создаются конструктором класса при создании экземпляра класса и уничтожаются при уничтожении соответствующего экземпляра класса.

Владение динамическими переменными

Динамические переменные не имеют предопределенного владельца. Вместо этого *выражение* `new`, которое создает динамическую переменную, возвращает указатель, которым должна явно управлять программа. Память, выделенная для динамической переменной, должна быть возвращена диспетчеру памяти с помощью *выражения* `delete` для освобождения, прежде чем будет уничтожен последний указатель на динамическую переменную. Таким образом, время жизни динамической переменной является полностью программируемым; это мощный, но опасный инструмент. Если память динамической переменной не возвращается диспетчеру памяти через *выражение* `delete` перед уничтожением последнего указателя на эту переменную, память этой переменной и она сама на все оставшееся время выполнения программы становится потерянной.

Владение динамическими переменными должно обеспечиваться разработчиком и быть закодировано в логике программы. Оно не контролируется компилятором и не определяется языком C++. Владение динамическими переменными имеет важное значение для оптимизации. Программы со строго определенным владением можно сделать более эффективными, чем программы с рассеянным владением.

Объекты-значения и объекты-сущности

Смысл некоторых переменных определяется в программе их содержимым. Такие переменные называются *объектами-значениями* (value object). Смысл других переменных определяется ролью, которую они играют в программе. Эти переменные называются сущностями или *объектами-сущностями* (entity object).

В C++ нет различия, как себя ведет переменная: как значение или как сущность, — роль переменной закодирована программистом в логике программы. C++ позволяет разработчику определить конструктор копирования и `operator==()` для любого класса. Роль переменной определяет, *должен* ли разработчик определить эти функции-члены. Если разработчик не предпринимает шаги для запрета операций, которые не имеют смысла, компилятор не станет на это жаловаться.

Объекты-сущности можно идентифицировать по их общим свойствам.

Сущности уникальны

Некоторые объекты в программе концептуально уникальны. Мьютекс, который охраняет конкретную критическую область или таблица символов со множеством записей являются примерами объектов программы с уникальной идентичностью.

Сущности изменяемы

Программа может заблокировать или разблокировать мьютекс, но это все еще один и тот же мьютекс. Программа может добавить символы в таблицу символов, но это по-прежнему та же таблица символов. Вы можете завести свой автомобиль и ехать в нем на работу, и он останется вашим автомобилем. Сущности имеют смысл как единое целое. Изменение состояния сущности не изменяет ее основной смысл в программе.

Сущности не копируемы

Сущности не копируются. Их природа исходит от того, как они используются, а не из составляющих их битов. Вы можете скопировать все биты системной таблицы символов в другую структуру данных, но это не будет таблица символов. Программа будет по-прежнему искать таблицу символов в старом месте, и копия использоваться не будет. Если программа иногда будет изменять копию вместо оригинала, таблица символов перестанет быть корректной. Вы можете копировать биты мьютекса, охраняющего критический раздел, но блокировка копии не приведет к взаимоисключению. Взаимоисключение является свойством, которое возникает из согласия двух потоков сигнализировать один другому с помощью определенного набора битов.

Сущности не сравниваемы

Операция сравнения равенства сущностей не имеет смысла. Сущности по своей природе являются индивидуалистами. Сравнение всегда должно возвращать значение `false`.

Аналогично объекты-значения имеют общие свойства, которые являются противоположностью свойств объектов-сущностей.

Значения взаимозаменяемы и сравниваемы

Целое число `4` и строка `"Hello, World!"` являются значениями. Выражение `2 + 2` равно значению `4` и демонстрирует неравенство значению `5`. Значение выражения `string("Hello") + string("!")` равно строке `"Hello!"` и не равно строке `"Hi"`. Смысл значения определяется его битами, а не тем, как оно используется в программе.

Значения неизменяемы

Не существует операции, которая может превратить `4` в `5` так, чтобы было справедливо равенство `2 + 2 = 5`. Вы можете так изменить целочисленную переменную, которая содержит `4`, что она получает значение `5`. Это изменяет переменную, которая является сущностью с уникальным именем, но не меняет значение `4`.

Значения копируемы

Копирование значения имеет смысл. Две строковые переменные могут иметь значение `"foo"`, и программа по-прежнему будет правильной.

То, чем является переменная, сущностью или значением, определяет, имеет ли смысл копирование и сравнение на равенство. Сущности не должны копироваться или сравниваться. То, является ли переменная-член класса сущностью или значением, определяет, как должен быть написан копирующий конструктор класса. Экземпляры класса могут владеть сущностями, но не могут их копировать. Понимание разницы объектов-сущностей и объектов-значений имеет важное значение, поскольку переменные, которые ведут себя как сущности, часто являются структурами со многими динамически выделенными переменными, копирование которых, имей оно смысл, было бы слишком дорогим.

API динамических переменных C++

C++ содержит набор средств для управления динамическими переменными, варьирующихся от автоматизации до тонкого контроля над управлением памятью и создания динамически выделяемых переменных C++. Даже опытные разработчики в своей практике могли использовать только основные из этих инструментов. Мы начнем с беглого обзора, прежде чем перейти к глубокому рассмотрению функций, которые особенно важны для оптимизации.

Указатели и ссылки

Динамические переменные в C++ не имеют имен. Они доступны с помощью указателей в стиле C или ссылок на переменную. Указатели абстрагируют аппаратные адреса, чтобы скрыть сложность и разнообразие компьютерных архитектур. Не каждое значение, которое может содержать переменная указателя, соответствует используемому местоположению в памяти, но в составляющих указатель битах нет никакой информации, которые могли бы об этом сказать. Одно конкретное значение, которое в C++11 называется `nullptr`, никогда не указывает на допустимое местоположение в памяти, что гарантируется стандартом C++. До C++11 для этой цели использовалось целочисленное значение `0`, которое в C++11 преобразуемо в `nullptr`, однако все нули в битовом шаблоне переменной-указателя не обязательно означают `nullptr`. Указатели в стиле C, объявленные без инициализатора, не имеют предопределенных значений (для достижения большей эффективности). Ссылочные переменные не могут быть объявлены без инициализатора, поэтому они всегда указывают на допустимое местоположение в памяти[2].

Выражения `new` и `delete`

Динамические переменные в C++ создаются с помощью *выражения* `new` (см. раздел "Построение динамических переменных с помощью выражений

[2] Разработчик может выполнить преобразование числового значения машинного адреса в ссылку с целью инициализации ссылочной переменной. Хотя это может показаться глупым действием, оно может быть вполне полезным в программировании встраиваемых устройств, где архитектура целевого компьютера фиксирована и известна заранее. Для задания адреса переменной, объявленной как `extern`, более эффективно использовать компоновщик, чем прибегать к приведению числовой константы к ссылке или указателю с помощью компилятора. Однако мой совет — "Отвернись, сюда не стоит смотреть".

new" главы 13, "Оптимизация управления памятью"). Это выражение получает память, необходимую для хранения переменной, конструирует переменную указанного типа в памяти и возвращает типизированный указатель на вновь созданную переменную. Выражение для создания массивов отличается от выражения для создания одного экземпляра, но оба они возвращают один и тот же тип указателя.

Динамические переменные удаляются с помощью *выражения* `delete` (см. раздел "Уничтожение динамических переменных с помощью выражения delete" главы 13, "Оптимизация управления памятью"). Это выражение удаляет переменную и освобождает ее память. Выражение `delete` для удаления массива отличается от выражения для удаления отдельных экземпляров. Оба работают с указателями одного и того же типа, однако по значению указателя нельзя сказать, на что он указывает — на массив или на скаляр. Разработчик должен помнить тип выражения `new`, использовавшийся для создания динамической переменной, и уничтожать ее с соответствующим типом выражения `delete`, иначе воцарится хаос. Если указатель на массив удаляется как указатель на экземпляр или наоборот, в результате получается неопределенное поведение.

Выражения `new` и `delete` встроены в синтаксис языка C++. Ниже приведены краткие примеры основных видов этих выражений, с которыми знакомы все разработчики.

```
{
    int n = 100;
    char* cp;           // cp не имеет определенного значения
    Book* bp = nullptr; // bp указывает на некорректный адрес

    // ...

    cp = new char[n];               // cp указывает на новый массив
    bp = new Book("Optimized C++"); // Динамический экземпляр класса

    // ...

    char array[sizeof(Book)];
    Book* bp2 = new(array) Book("Moby Dick"); // Размещающий new

    // ...

    delete[] cp;   // Удаление динамического массива перед
                   // изменением указателя
    cp = new char; // cp указывает на отдельный динамический символ

    // ...

    delete bp;     // Для экземпляра класса
    delete cp;     // Для динамически выделенного символа
    bp2->~Book(); // Уничтожение размещенного экземпляра класса
}
// Удалите все динамические переменные
// перед выходом из области видимости!
```

Функции управления памятью

Выражения `new` и `delete` вызывают функции управления памятью стандартной библиотеки C++ для выделения памяти и ее возврата в пул, который стандарт C++ называет "свободной памятью". Эти функции являются перегрузками `operator new()`, `operator new[]()` для массивов, `operator delete()` и `operator delete[]()` для массивов. C++ предоставляет также классические функции управления памятью из стандартной библиотеки C, например `malloc()` и `free()`, которые выделяют и освобождают нетипизированные блоки памяти.

Конструкторы и деструкторы классов

C++ позволяет каждому классу определить функцию-член конструктора, которая вызывается, чтобы установить начальное значение объекта при создании экземпляра этого класса. Еще одна функция-член, деструктор, вызывается всякий раз, когда экземпляр класса уничтожается. Среди прочих преимуществ эти специальные функции-члены предоставляют место для размещения выражений `new` и `delete` таким образом, чтобы любые динамические переменные автоматически управлялись экземпляром класса, который их содержит.

Интеллектуальные указатели

Стандартная библиотека C++ предоставляет шаблонные классы "интеллектуальных указателей", которые ведут себя подобно простым указателям, но удаляют переменные, на которые указывают, при выходе из области видимости. Интеллектуальные указатели помнят, был ли выделен один экземпляр или массив, и вызывают соответствующее выражение `delete` для типа интеллектуального указателя. Интеллектуальные указатели рассматриваются далее, в следующем разделе.

Шаблоны аллокаторов

Стандартная библиотека C++ предоставляет шаблоны аллокаторов, которые обобщают выражения `new` и `delete` для использования со стандартными контейнерами. Аллокаторы подробно рассматриваются в разделе "Пользовательские аллокаторы стандартной библиотеки" главы 13, "Оптимизация управления памятью".

Автоматизация владения интеллектуальными указателями

Владение динамическими переменными не контролируется компилятором и не определяется языком C++. Программа может объявить переменную-указатель в одном месте, присвоить динамическую переменную указателю с помощью выражения `new` в другом месте, скопировать указатель в другой указатель в третьем месте и уничтожить динамическую переменную, на которую указывает этот второй указатель, с помощью выражения `delete` еще в одном месте. Однако программу, которая ведет себя таким образом, может быть очень трудно тестировать и отлаживать из-за такого разброса владения динамической переменной. Владение динамическими переменными обеспечивается разработчиком и кодируется в логике программы. Когда

владение распылено, каждая строка программы может потенциально создать динамическую переменную, добавить или удалить ссылку на нее или уничтожить переменную. Разработчик вынужден отслеживать все потенциальные пути выполнения, чтобы убедиться, что динамические переменные всегда правильно возвращены диспетчеру памяти.

Эта сложность может быть уменьшена с помощью идиом программирования. Одной общепринятой практикой является объявление переменной-указателя в качестве закрытого члена класса. Конструктор класса может установить этот указатель равным `nullptr`, скопировать в него указатель-аргумент или передать результат выражения `new` для создания динамической переменной. Поскольку указатель является закрытым членом, любое его изменение должно происходить только в функциях-членах класса. Это ограничивает число строк кода, которые могут повлиять на значение указателя, и облегчает программирование и отладку. Деструктор класса может содержать выражение `delete` для динамической переменной. Когда экземпляр класса ведет себя таким образом, говорят, что он владеет динамической переменной.

Простой класс может быть разработан с единственной целью — владения динамической переменной. Помимо создания и уничтожения динамической переменной, класс реализует `operator->()` и `operator*()`. Такой простой класс называется *интеллектуальным указателем*, потому что главным образом ведет себя, как указатель в стиле C, но при этом уничтожает динамический объект, на который указывает, при уничтожении объекта самого интеллектуального указателя.

C++ предоставляет шаблонный класс интеллектуального указателя с именем `std::unique_ptr<T>` для поддержания владения динамической переменной типа `T`. Работа с `unique_ptr<>` компилируется в код, сравнимый по эффективности с рукописным кодом для работы с обычным указателем.

Автоматизация владения динамическими переменными

Интеллектуальные указатели автоматизируют владение динамическими переменными путем соединения времени жизни динамической переменной со временем жизни интеллектуального указателя, который ею владеет. Динамическая переменная должным образом уничтожается, а ее память освобождается автоматически, в зависимости от объявления указателя.

- Экземпляр интеллектуального указателя, объявленный с автоматической длительностью хранения, удаляет динамическую переменную, которой он владеет, когда выполнение выходит из области видимости, включающей его объявление, причем не важно, как именно это происходит — при выполнении инструкций `break` или `continue`, возврате из функции или генерации исключения.
- Экземпляр интеллектуального указателя, объявленный как член класса, удаляет динамическую переменную, которой владеет, при удалении содержащего его экземпляра класса. Кроме того, поскольку правила уничтожения класса требуют уничтожения каждого члена после выполнения деструктора класса, нет необходимости писать в деструкторе явный код удаления динамической переменной. Интеллектуальный указатель удаляется с помощью встроенного механизма C++.

- Экземпляр интеллектуального указателя, объявленный как локальный по отношению к потоку, удаляет динамическую переменную, которой он владеет, при нормальном завершении потока (но в общем случае не тогда, когда поток завершается операционной системой).
- Экземпляр интеллектуального указателя, объявленный со статической длительностью хранения, удаляет динамическую переменную, которой владеет, при завершении программы.

Поддержка единственного владельца в общем случае и использование `std::unique_ptr` для поддержки владения в частности значительно упрощают выяснение, указывает ли указатель на корректное местоположение в памяти и возвращается ли память диспетчеру, когда она больше не нужна. Использование `unique_ptr` требует малых накладных расходов, так что этот интеллектуальный указатель должен быть основным выбором разработчиков, заинтересованных в оптимизации.

Общее владение динамической переменной более дорогое

C++ позволяет указывать на динамическую переменную нескольким указателям и ссылкам одновременно. Несколько указателей могут указывать на динамическую переменную, если несколько структур данных указывают на динамическую переменную или если указатель на динамическую переменную передается в качестве аргумента функции, так что один указатель находится в области видимости вызывающей функции, а другой — в области видимости вызываемой функции. Недолгое время несколько указателей на динамическую переменную существует в ходе присваивания или создания объекта, назначение которого — владеть динамической переменной.

В любой момент существования нескольких указателей, указывающих на динамическую переменную, разработчик должен знать, какой из них является ее владельцем. Разработчик не должен явным образом удалять динамическую переменную через указатель, не являющийся владельцем, разыменовывать любой указатель после удаления динамической переменной или допускать наличие двух указателей-владельцев для одного объекта, так что он будет удален дважды. Этот анализ становится особенно важным в случае ошибки или исключения.

Иногда владение динамической переменной *должно* быть разделено между двумя или несколькими указателями. Владение должно быть общим, когда времена жизни двух указателей на переменную перекрываются, но время жизни ни одного из них не содержит полностью время жизни другого.

При необходимости разделения владения можно использовать класс стандартной библиотеки шаблонов C++ `std::shared_ptr<T>`, который предоставляет интеллектуальный указатель, способный управлять общим владением. Экземпляры `shared_ptr` содержат один указатель на динамическую переменную, а другой указатель на динамический объект, содержащий счетчик ссылок. Когда динамическая переменная присваивается `shared_ptr`, оператор присваивания создает объект счетчика ссылок и устанавливает счетчик ссылок равным 1. Когда один `shared_ptr` присваивается другому, счетчик ссылок увеличивается. Когда `shared_ptr` уничтожается, деструктор уменьшает счетчик ссылок и удаляет динамическую переменную только в том

случае, если значение счетчика ссылок становится равным 0. Над счетчиком ссылок выполняются дорогостоящие атомарные операции инкремента и декремента, так что shared_ptr работает и в многопоточных программах. Класс std::shared_ptr поэтому существенно дороже, чем указатель C или класс std::unique_ptr.

Разработчик должен следить за тем, чтобы не присваивать указатель в стиле C (такой, как указатель, возвращаемый выражением new) нескольким интеллектуальным указателям одновременно. Если такой указатель окажется присвоенным нескольким интеллектуальным указателям, он будет удален несколько раз, так что в результате получится то, что стандарт C++ называет "неопределенным поведением". Это звучит очевидно, но поскольку интеллектуальный указатель можно создать из простого старого указателя в стиле C, преобразование типов во время передачи аргумента может молча привести к этому.

std::auto_ptr и классы контейнеров

До стандарта C++11 имелся интеллектуальный указатель std::auto_ptr<T>, который управлял неразделяемым владением динамической переменной. Во многих отношениях auto_ptr ведет себя подобно unique_ptr. Однако auto_ptr не реализует семантику перемещения (обсуждаемую ниже, в разделе "Реализация семантики перемещения" данной главы) и не имеет копирующего конструктора.

Большинство контейнеров стандартной библиотеки до C++11 выполняли копирующее конструирование значений элементов во внутренней памяти контейнера, поэтому auto_ptr не мог использоваться в качестве типа элемента. До введения unique_ptr контейнеры стандартной библиотеки должны были работать с использованием указателей в стиле C, глубокого копирования объектов или shared_ptr. Каждое из этих решений имеет свои проблемы. Указатели C создавали риск ошибок или утечки памяти, глубокое копирование объектов было неэффективным для объектов большого размера, а интеллектуальный указатель shared_ptr дорогостоящий по своей природе. Некоторые проекты реализовывали специальные небезопасные интеллектуальные указатели для классов контейнеров, копирующие конструкторы которых выполняли операцию перемещения, такую как использование std::swap() для передачи в конструктор владеющего указателя. Это позволяло многим (но не всем) функциям-членам класса контейнера работать так, как ожидалось. Это было эффективное, но небезопасное и трудное в отладке решение.

До C++11 было обычной практикой использование класса std::shared_ptr для типа элемента в экземплярах классов контейнеров стандартной библиотеки. Эта практика приводила к корректному и легче отлаживаемому коду, но ценой неэффективности shared_ptr.

Динамические переменные имеют стоимость времени выполнения

Поскольку C++ компилируется в машинный код, непосредственно выполняемый компьютером, стоимость большинства инструкций C++ — максимум несколько обращений к памяти. Однако стоимость выделения памяти для динамической переменной измеряется в тысячах обращений к памяти. Эта стоимость в среднем настолько высока, что устранение даже одного вызова диспетчера памяти может заметно

улучшить производительность функции, как неоднократно было продемонстрировано в главе 4, "Оптимизация использования строк".

Концептуально функция выделения памяти выполняет в коллекции блоков свободной памяти поиск блока, который может удовлетворить запрос. Функция, найдя блок подходящего размера, удаляет его из коллекции свободных блоков и возвращает его адрес. Если функция находит блок, который намного больше запрашиваемого, она может предпочесть разделить блок и вернуть адрес его части. Очевидно, что это общее описание оставляет простор для множества вариантов реализации.

Если нет блоков свободной памяти, которые могли бы быть возвращены в ответ на запрос, функция распределения делает дорогостоящий вызов ядра операционной системы для получения блока дополнительной памяти из пула памяти операционной системы. Память, возвращенная ядром, может быть не кеширована в физической оперативной памяти (см. раздел "Количество памяти ограничено" главы 2, "Оптимизация, влияющая на поведение компьютера"), что приводит к еще большему времени задержки при первом обращении к ней. Проход по списку блоков свободной памяти является дорогостоящей операцией сам по себе. Блоки разбросаны в памяти и имеют меньше шансов находиться в кеше, чем блоки памяти, видимые в данный момент выполняющейся программе.

Коллекция блоков свободной памяти является ресурсом, используемым совместно всеми потоками программы. Любые изменения коллекции свободных блоков должны быть безопасны с точки зрения потоков. Если несколько потоков делают частые вызовы функций распределения или освобождения памяти, они конкурируют за работу с диспетчером памяти как за ресурс, что приводит к работе только одного потока, в то время как остальные находятся в состоянии ожидания.

Программа на C++ должна освободить память, выделенную динамической переменной, которая больше не нужна. Концептуально функция освобождения памяти помещает возвращаемый блок памяти в коллекцию свободных блоков. В реальных реализациях поведение свободной функции значительно сложнее. Большинство реализаций пытаются объединить освобожденный блок памяти с соседними свободными блоками. Это предохраняет коллекцию блоков свободной памяти от заполнения блоками размера, слишком малого для того, чтобы быть полезными. С вызовами функции освобождения памяти связаны все те же вопросы снижения производительности кеша и конкуренция при многопоточном доступе к коллекции свободных блоков, что и с вызовами для выделения памяти.

Уменьшение использования динамических переменных

Динамические переменные являются надежным средством решения многих проблем. Однако их слишком дорого использовать для решения *каждой* проблемы. Статически создаваемые переменные часто могут выполнить работу динамических переменных.

Статическое создание экземпляров класса

Экземпляры класса можно создавать динамически. Однако большинство экземпляров класса, которые не являются частью контейнеров, можно (и в общем случае должно) создавать статически (т.е. без использования `new`). Иногда экземпляры классов создаются динамически, потому что разработчик не понимает, что есть и иная возможность. Например, неопытного разработчика C++, который сначала работал на Java, легко узнать по попыткам применить типичный синтаксис Java для создания экземпляра класса следующим образом:

```
MyClass myInstance = new MyClass("hello", 123);
```

Если наивный программист использует синтаксис Java в программе на языке C++, компилятор C++ заставит его сделать `myInstance` указателем, сообщая, что "нельзя преобразовать `"MyClass*"` в `"MyClass"`." Более опытный разработчик может сделать `myInstance` интеллектуальным указателем, чтобы избежать необходимости явного удаления экземпляра динамически созданного класса.

```
MyClass* myInstance = new MyClass("hello", 123);
```

Однако оба решения неэкономны. Вместо этого класс следует, как правило, создавать статически:

```
MyClass myInstance("hello", 123);
```

или:

```
MyClass anotherMC = MyClass("hello",123); // Может быть менее
                                          // эффективно
```

Если переменная `myInstance` объявлена в блоке выполнимого кода, она имеет автоматическую длительность хранения. Экземпляр класса будет уничтожен, когда выполнение покидает блок, содержащий его объявление. Если программе требуется хранить `myInstance` дольше, чем сказано, этот экземпляр можно объявить во внешней области или в долгоживущем объекте и передавать функциям, которые должны использовать `myInstance`, указатель на него. Объявление можно переместить в область видимости файла, если `myInstance` требуется все время работы программы.

Статическое создание переменных-членов класса

Когда переменные-члены класса сами являются экземплярами классов, эти члены могут быть созданы статически при построении содержащего их класса. Это экономит затраты на выделение памяти для членов.

Иногда кажется, что экземпляр класса должен создаваться динамически потому, что он является членом другого класса, и некоторые ресурсы, необходимые для создания члена, недоступны в момент построения включающего класса. Альтернативный вариант заключается в том, чтобы сделать проблемный класс (а не указатель на него) членом содержащего его класса и просто не полностью инициализировать проблемный класс при построении. Создайте функцию-член в проблемном классе для полной инициализации переменной, когда все ресурсы будут доступны.

Затем вставьте вызов функции-члена инициализации в точке, где был бы создан динамический экземпляр с использованием `new`. Такой шаблон называется *двухэтапной инициализацией*.

Такая двухэтапная инициализация не имеет никакой лишней стоимости. Член все равно не может использоваться до момента, где он был бы построен. Стоимость любых проверок, полностью ли инициализирован экземпляр члена, такая же, как и расходы на проверку, не равен ли указатель на него значению `nullptr`. Дополнительным преимуществом описанного здесь метода является то, что функция-член инициализации может возвращать коды ошибок или другую информация, в то время как конструктор делать это не в состоянии.

Двухэтапная инициализация особенно полезна, когда класс должен выполнять во время инициализации какие-то действия, отнимающие много времени, как, например, чтение файла, или действия, которые могут завершиться ошибкой, как запрос веб-страницы из Интернета. Предоставление отдельной функции инициализации позволяет выполнять такую инициализацию параллельно с другой деятельностью программы (см. главу 12, "Оптимизация параллельности") и облегчает ее повторение в случае сбоя.

Использование статических структур данных

`std::string`, `std::vector`, `std::map` и `std::list` — вот те контейнеры, которые большинство разработчиков на C++ используют ежедневно. При осторожном и внимательном применении они оказываются достаточно эффективными. Но это не единственно возможный выбор. `std::string` и `std::vector` время от времени прибегают к перераспределению памяти, чтобы иметь возможность добавлять элементы в контейнер. `std::map` и `std::list` выделяют память для каждого добавляемого элемента. Иногда это оказывается слишком дорогостоящим. Здесь мы рассмотрим альтернативные варианты.

Применение `std::array` вместо `std::vector`

`std::vector` позволяет программе определить массивы любого типа с динамически изменяемыми размерами. Если размер массива или его максимальный размер известен во время компиляции, аналогичный интерфейс предлагает и `std::array`, но для массива фиксированного размера и без вызовов диспетчера памяти.

`std::array` создает элементы с помощью копирования и предлагает итераторы произвольного доступа в стиле стандартной библиотеки, а также индексирование с помощью `operator[]`. Функция `size()` возвращает фиксированный размер массива.

С точки зрения оптимизации `std::array` почти ничем не отличается от массива в стиле C. С точки зрения программирования `std::array` имеет удобные сходства с контейнерами стандартной библиотеки.

Создание больших буферов в стеке

В главе 4, "Оптимизация использования строк", было показано, что вставка в строку может быть дорогой из-за необходимости перераспределения памяти для строки с ее ростом. Если известно, каким может быть максимально возможный размер строки,

или хотя бы максимальный разумный размер, можно использовать большие массивы в стиле C с автоматической длительностью хранения в качестве временного буфера, выполнить изменения строки в нем, а затем скопировать результат.

Шаблон проектирования представляет собой вызов функции для создания или изменения данных, которая объявляет очень большой автоматический массив. Функция копирует данные своего аргумента в локальный массив и выполняет вставки, удаления или другие перестановки статического массива.

Хотя имеется ограничение на общее количество памяти в стеке, этот предел часто бывает очень большим. В настольных системах, например, если только алгоритм не использует очень глубокую рекурсию, в стеке достаточно места для массивов из 10 000 символов.

Осторожный разработчик, беспокоящийся о переполнении локального массива, может проверять длину аргумента строки или массива и, если аргумент слишком велик для локального массива, использовать динамически сконструированный массив.

Почему при этом копирование будет более быстрым, чем при использовании динамических структур данных, таких как `std::string`? Отчасти потому, что трансформирующие операции зачастую копируют входные данные в любом случае, а отчасти — потому, что копирование блоков даже из нескольких тысяч байтов на оборудовании класса настольных компьютеров по сравнению со стоимостью выделения динамической памяти для хранения промежуточных результатов обходится недорого.

Статическое создание связанных структур данных

Связанные структуры данных могут быть созданы как статически инициализированные данные. Например, показанная на рис. 6.1 структура данных может быть воспроизведена статически.

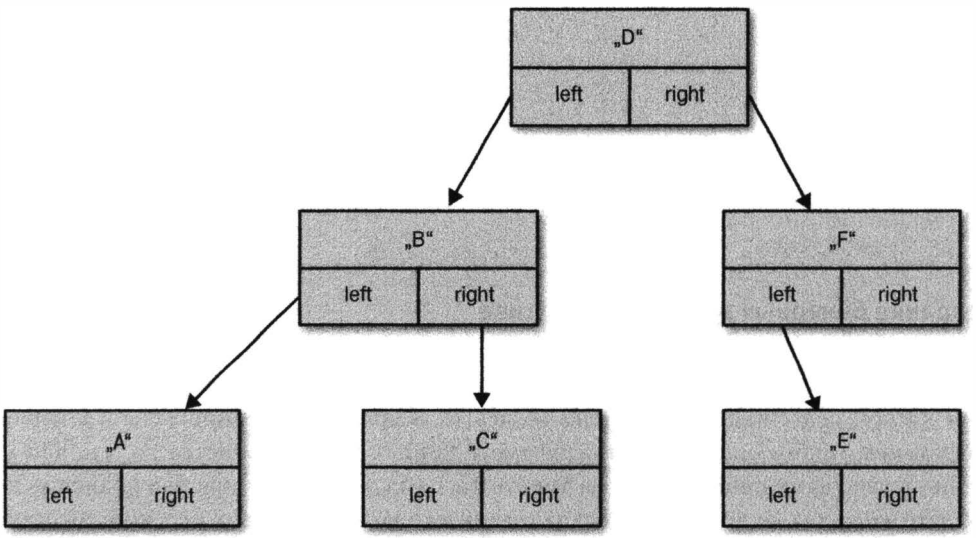


Рис. 6.1. Простая структура дерева

В данном конкретном примере дерево представляет собой дерево бинарного поиска, узлы которого размещаются в массиве в порядке поиска в ширину, и первый элемент является корнем:

```
struct treenode {
    char const* name;
    treenode*   left;
    treenode*   right;
} tree[] = {
    { "D", &tree[1], &tree[2] }
    { "B", &tree[3], &tree[4] },
    { "F", &tree[5], nullptr },
    { "A", nullptr, nullptr },
    { "C", nullptr, nullptr },
    { "E", nullptr, nullptr },
};
```

Этот код работает, поскольку адреса элементов массива являются константными выражениями. Эта запись позволяет определить любую связанную структуру. Однако инициализаторы оказываются не слишком мнемоничными, так что при построении такой структуры очень легко сделать опечатку.

Другой способ создания статически связанных структур заключается в инициализации переменных для каждого элемента структуры. Этот механизм достаточно мнемоничен, но имеет тот недостаток, что должны быть объявлены опережающие ссылки. Рассмотрим, например, приведенную ниже циклическую структуру. Самый естественный способ объявления этой структуры (в порядке от `first` до `fourth`) требует объявления как `extern` всех четырех переменных. Если использовать обратный порядок объявления (в порядке от `fourth` до `first`), то большинство ссылок будут на уже объявленные переменные, и можно обойтись только одной опережающей ссылкой.

```
struct cyclenode {
    char const* name;
    cyclenode* next;
}
extern cyclenode first; // forward reference
cyclenode fourth = { "4", &first  };
cyclenode third  = { "3", &fourth };
cyclenode second = { "2", &third  };
cyclenode first  = { "1", &second };
```

Создание бинарных деревьев в массиве

Большинство разработчиков изучали бинарные деревья в качестве примера связанной структуры данных, в которой каждый узел является отдельным экземпляром класса, содержащим указатели на левый и правый дочерние узлы. Печальным следствием такого определения структуры данных дерева является то, что для хранения в узле даже одного символа требуется память, достаточная для хранения двух указателей, а также возникают накладные расходы, связанные с работой диспетчера динамической памяти.

Альтернативный подход заключается в создании бинарного дерева в массиве. Вместо хранения явных ссылок на дочерние узлы индексы дочерних узлов в массиве вычисляются на основе индекса родительского узла. Если индекс родительского узла — i, два дочерних узла имеют индексы $2i$ и $2i + 1$. В качестве дополнительного бонуса родительский узел имеет индекс $i/2$[3]. Поскольку такие умножения и деления могут быть реализованы как сдвиги, вычисление оказывается недорогим даже для очень скромных встраиваемых процессоров.

Для реализации узлов дерева в массиве нужен механизм определения того, являются ли их левые и правые дочерние узлы корректными узлами дерева или они являются эквивалентом нулевого указателя. Если дерево сбалансировано влево, то достаточно одного целочисленного значения — индекса первого недопустимого узла в массиве.

Эти свойства (возможность вычисления дочерних и родительского узлов и эффективность представления левосбалансированного дерева) делают дерево в массиве эффективной реализацией пирамидальных структур данных.

Представление дерева в массиве может оказаться для несбалансированных бинарных деревьев менее эффективным, чем представление с использованием указателей. Для представления несбалансированного бинарного дерева из n узлов может потребоваться до $2n$ позиций в массиве, в зависимости от степени сбалансированности. Кроме того, операции балансировки потребуют копирования узлов в другие местоположения в массиве, а не просто обновления указателей. На слабых процессорах и для узлов больших размеров операция копирования может оказаться слишком дорогой. Но если размер узла меньше трех указателей, представление дерева в массиве может дать выигрыш в производительности.

Использование циклического буфера вместо дека

`std::deque` и `std::list` так часто используются для буфера FIFO (first-in-first-out, первым вошел — первым вышел), что стандартная библиотека имеет адаптер контейнера `std::queue`.

Двусторонняя очередь, или дек (deque), может быть реализована, в частности, с использованием циклического буфера. Циклический буфер представляет собой массив, в котором начало и конец очереди задаются двумя индексами массива, по модулю длины массива. Циклический буфер имеет свойства, аналогичные свойствам дека, включая константное время операций `push_back()` и `pop_front()`, и итераторы произвольного доступа. Однако циклический буфер не требует перераспределения памяти до тех пор, пока потребитель работает с той же скоростью, что и производитель. Размер циклического буфера не определяет, сколько входных данных он может обработать; вместо этого он определяет, насколько далеко вперед может зайти производитель.

Разница между циклическим буфером и списком или деком заключается в ограничении числа элементов в буфере, очевидного при использовании циклического буфера. Специализация обобщенной структуры данных очереди путем наложения указанного ограничения делает возможной значительно более эффективную реализацию.

[3] Если считать, что корень дерева находится в элементе массива с индексом 1. Если использовать привычную нумерацию с нуля, то для узла с индексом i дочерними являются узлы с индексами $2i+1$ и $2i+2$, а родительским — $(i–1)/2$. — *Примеч. ред.*

Boost предлагает циклический буфер (http://bit.ly/b-buffer), реализованный в виде контейнера стандартной библиотеки. В вебе есть множество других реализаций. Циклический буфер может быть спроектирован со статическим буфером, размер которого определяется параметром шаблона, с начальным размером, требующим одного выделения памяти, или с динамически изменяемым размером буфера, как у std::vector. Производительность циклического буфера аналогична производительности std::vector.

Использование std::make_shared вместо new

Интеллектуальный указатель с совместным владением, такой как класс std::shared_ptr, в действительности хранит два указателя. Один указывает на объект, на который ссылается интеллектуальный указатель. Второй указывает на другую динамическую переменную, содержащую счетчик ссылок, который совместно используется всеми std::shared_ptr, указывающими на объект. Таким образом, инструкция

```
std::shared_ptr<MyClass> p(new MyClass("hello", 123));
```

вызывает диспетчер памяти *дважды*: один раз — для создания экземпляра MyClass и второй — для создания скрытого счетчика ссылок.

Обходной путь для устранения выделения памяти для счетчика ссылок до принятия стандарта C++11 состоял в добавлении назойливого счетчика ссылок в виде базового класса для MyClass и создании пользовательского интеллектуального указателя, который использует этот счетчик ссылок:

```
custom_shared_ptr<RCClass> p(new RCClass("Goodbye", 999));
```

Авторы стандарта поняли всю боль такого решения и добавили в стандарт шаблонную функцию с именем std::make_shared(), которая выделяет единый блок памяти для хранения и счетчика ссылок, и экземпляра MyClass. Класс std::shared_ptr имеет функцию удаления, которая знает, каким из способов был создан интеллектуальный указатель, и содержится ли счетчик ссылок в отдельном блоке памяти. Теперь вы понимаете, почему класс std::shared_ptr кажется настолько сложным при проходе с помощью отладчика.

Использовать функцию make_shared() очень просто:

```
std::shared_ptr<MyClass> p = std::make_shared<MyClass>("hello", 123);
```

Или даже еще проще при использовании объявления в стиле C++11:

```
auto p = std::make_shared<MyClass>("hello", 123);
```

Не следует разделять владение без необходимости

Несколько экземпляров класса std::shared_ptr могут совместно владеть динамической переменной. Интеллектуальный указатель shared_ptr полезен, когда время существования различных указателей перекрывается непредсказуемо, но обходится это дорого. Увеличение и уменьшение счетчика ссылок в shared_ptr осуществляется не с помощью простой инструкции инкремента, а путем дорогостоящего атомарного

инкремента с полным ограждением памяти (см. раздел "Атомарные операции над общими переменными" главы 12, "Оптимизация параллельности"), так что shared_ptr корректно работает в многопоточных программах.

Если время жизни одного объекта shared_ptr полностью охватывает время жизни другого, расходы на второй shared_ptr не нужны. Следующий код иллюстрирует распространенную ситуацию:

```
void fiddle(std::shared_ptr<Foo> f);
...
shared_ptr<Foo> myFoo = make_shared<Foo>();
...
fiddle(myFoo);
```

myFoo владеет динамическим экземпляром Foo. Когда программа вызывает fiddle(), вызов создает вторую ссылку на динамический экземпляр Foo, увеличивая счетчик ссылок shared_ptr. Когда выполняется возврат из fiddle(), аргумент shared_ptr освобождает свое владение динамическим экземпляром Foo. Вызывающий объект по-прежнему владеет указателем. Минимальная стоимость этого вызова — излишние атомарные инкремент и декремент, каждый с полным ограждением памяти. В одной функции эта дополнительная плата является незначительной. Но в качестве практики программирования передача во всей программе всем функциям, которым требуется указатель на Foo, аргумента shared_ptr может привести к значительной стоимости.

Передача функции fiddle() обычного указателя устраняет эту стоимость:

```
void fiddle(Foo* f);
...
shared_ptr<Foo> myFoo = make_shared<Foo>();
...
fiddle(myFoo.get());
```

Имеется общепринятое мнение, что обычные указатели в стиле C никогда не должны появляться в программе, за исключением реализации интеллектуальных указателей. Но есть и другое мнение, что простые указатели вполне допустимы до тех пор, пока используются для представления не владеющих указателей. В команде, в которой разработчики уверены, что простые указатели — это ни что иное, как игрушки дьявола, тот же результат можно получить с помощью ссылок. Делая аргумент функции имеющим тип Foo&, мы сообщаем о том, что вызывающий объект отвечает за корректность ссылки на время вызова и что указатель отличен от nullptr:

```
void fiddle(Foo& f);
...
shared_ptr<Foo> myFoo = make_shared<Foo>();
...
if (myFoo)
    fiddle(*myFoo.get());
```

Оператор разыменования * преобразует указатель на Foo, который возвращает get(), в ссылку на Foo. Таким образом, код при этом не генерируется вовсе; это всего лишь примечание для компилятора. Ссылки представляют собой соглашение, гласящее: "ненулевой указатель, которым я не владею".

Использование "главного указателя" для владения динамическими переменными

Интеллектуальный указатель `std::shared_ptr` прост. Он автоматизирует управление динамическими переменными. Но, как отмечалось ранее, такие интеллектуальные указатели дорогостоящи. Во многих случаях они излишни.

Очень часто одна структура данных владеет динамической переменной на продолжительности всей ее жизни. Ссылки или указатели на динамическую переменную могут быть переданы и возвращены функциями, присвоены переменным и так далее, но ни одна из этих ссылок не переживет "главную" ссылку.

Если существует такая главная ссылка, то она может быть эффективно реализована с помощью `std::unique_ptr`. Для обращения к объекту во время вызова функции может использоваться обычный указатель в стиле C или ссылка C++. При таком последовательном применении указателей и ссылок следует документировать, что они являются "не владеющими" указателями.

Разработчики начинают чувствовать себя очень неуютно при любом отходе от использования класса `std::shared_ptr`. Однако те же разработчики каждый день используют итераторы, возможно, не понимая, что они ведут себя, как не владеющие указатели, которые могут быть недействительными. Мой опыт использования главных указателей в нескольких крупных проектах показал, что на практике проблем утечек памяти или двойного освобождения не возникает. При наличии очевидного владельца главные указатели представляют собой большую победу оптимизации. Если вы сомневаетесь, класс `std::shared_ptr` всегда к вашим услугам.

Уменьшение количества перераспределений динамических переменных

Часто удобство динамической переменной просто слишком велико, чтобы его пропустить. На ум сразу приходит `std::string`. Но это не извиняет неосторожность разработчика. Существуют методы для сокращения количества выделений памяти при использовании контейнеров стандартной библиотеки. Эти методы могут быть обобщены и для разработчиков собственных структур данных.

Предварительное выделение памяти для динамических переменных для предотвращения перераспределений

Когда к объекту `std::string` или `std::vector` добавляются данные, его динамически выделенная внутренняя память в конечном итоге становится заполненной. Очередная операция добавления заставляет строку или вектор выделять больший блок памяти и копировать старое содержимое в новое хранилище. И вызов диспетчера памяти, и копирование выполняют много обращений к памяти и состоят из многих инструкций. Да, действительно, добавление данных имеет временную стоимость $O(1)$, но скрытая в ней константа (которая говорит о конкретном времени работы в миллисекундах) может оказаться весьма значительной.

И `string`, и `vector` имеют функцию-член `reserve(size_t n)`, которая просит строку или вектор убедиться, что в них имеется достаточно места минимум для `n` записей. Если размер можно вычислить или оценить, то вызов `reserve()` предохраняет строку или вектор внутренней памяти от необходимости перераспределения, пока данные растут до указанного предела:

```
std::string errmsg;
errmsg.reserve(100);  // Одно выделение памяти для всех добавлений
errmsg += "Ошибка 1234: переменная ";
errmsg += varname;
errmsg += " использована до присваивания. Неопределенное поведение.";
```

Вызов `reserve()` действует для строки или вектора в качестве подсказки. В отличие от выделения статического буфера для наихудшего случая, единственным штрафом за слишком малое предположение является возможность дополнительных перераспределений. Даже расточительное завышение ожидаемого размера не является проблемой, если строка или вектор будет использоваться недолго, а затем будет уничтожен. Для возврата неиспользуемой памяти диспетчеру памяти после применения `reserve()` к строке или вектору можно воспользоваться функцией-членом `shrink_to_fit()` класса `std::string` или `std::vector`.

Тип хеш-таблицы `std::unordered_map` стандартной библиотеки (см. раздел "std::unordered_map и std::unordered_multimap" главы 10, "Оптимизация структур данных") содержит базовый массив (список корзин), содержащий ссылки на остальные структуры данных. Он также имеет функцию-член `reserve()`. К сожалению, в классе `std::deque`, который также имеет базовый массив, функция-член `reserve()` отсутствует.

Разработчики собственных структур данных, содержащих базовый массив, облегчат жизнь пользователям, если включат в класс такую функцию, как `reserve()`, для того чтобы заранее выделить память для этого массива.

Создание динамических переменных вне циклов

В приведенном далее маленьком цикле есть большая проблема. Он добавляет строки каждого файла из списка `namelist` в переменную `config` типа `std::string`, а затем извлекает небольшое количество данных из `config`. Проблема заключается в том, что `config` создается при каждой итерации цикла и каждый раз при увеличении размера перераспределяется. Затем в конце цикла `config` он выходит из области видимости и уничтожается, возвращая память диспетчеру памяти:

```
for (auto& filename : namelist) {
    std::string config;
    ReadFileXML(filename, config);
    ProcessXML(config);
}
```

Одним из способов поднять эффективность этого цикла является перемещение объявления переменной `config` вне цикла. Внутри цикла `config` очищается, однако функция `clear()` не освобождает динамический буфер внутри `config`. Она просто устанавливает длину содержимого равной нулю. После первой итерации

перераспределение config происходить не будет, если только следующий файл не окажется значительно больше первого:

```
std::string config;
for (auto& filename : namelist) {
    config.clear();
    ReadFileXML(filename, config);
    ProcessXML(config);
}
```

Вариации на эту тему включают превращение config в переменную-член класса. От этого совета некоторым разработчикам может пахнуть серой пропаганды использования глобальных переменных. В определенном смысле так и есть. Однако превращение динамически выделяемых переменных в долгожители может иметь огромное влияние на производительность.

Этот трюк работает не только с std::string, но и с std::vector и любыми другими структурами данных, которые имеет динамически изменяемый базовый размер.

Из истории оптимизационных войн

Однажды я писал многопоточную программу, в которой каждый поток входил в экземпляр класса. Каждый из этих потоков записывал свои действия в журнал. Функция ведения журнала первоначально определяла локальную временную строку, в которой форматировались выводимые в журнал данные. Я обнаружил, что при наличии более полудесятка потоков, каждый из которых обращался к диспетчеру памяти при создании строк для журнала, конкуренция за диспетчер памяти настолько выросла, что производительность резко упала. Решение заключалось в том, чтобы сделать временную строку, хранящую каждую строку журнала, членом класса, который повторно использовался при выводе каждой строки журнала. После того как строка использовалась несколько раз, она стала достаточно длинной, чтобы ее перераспределение больше не требовалось. Применение одного этого трюка во всей большой программе дало прирост производительности, больший, чем все остальные вносимые мною изменения, так что данная программа (телефонный сервер) легко выдерживала удесятеренную нагрузку одновременных вызовов.

Устранение излишнего копирования

В классическом определении C от Кернигана (Kernighan) и Ритчи (Ritchie) (K&R) все сущности, которые могут быть непосредственно присвоены, были примитивными типами, такими как char, int, float и указатели, которые вписывались в один регистр процессора. Таким образом, любой оператор присваивания типа a = b; был очень эффективным, генерируя только одну или две команды для выборки значения b и его сохранения в a. В C++ присваивание базовых типов наподобие char, int или float такое же эффективное.

Но имеются присваивания простого вида, которые далеко не столь эффективны в C++. Если `a` и `b` являются экземплярами класса `BigClass`, то присваивание `a = b;` вызывает функцию-член `BigClass`, которая называется *оператором присваивания*. Оператор присваивания *может* быть таким же простым, как и копирование полей `b` в `a`. Но дело в том, что ему *позволено* делать абсолютно все, что может делать функция C++. Класс `BigClass` может иметь десятки копируемых полей. Если большой класс владеет динамическими переменными, которые могут копироваться, в результате выполняются вызовы диспетчера памяти. Если `BigClass` владеет отображением `std::map` с миллионом записей или даже массивом `char` с миллионом символов, стоимость оператора присваивания может быть значительной.

В C++, если `Foo` — имя класса, инициализирующее объявление `Foo = b;` может вызывать другую функцию-член, именуемую *копирующим конструктором*. Копирующий конструктор и оператор присваивания являются тесно связанными функциями-членами, которые в основном делают одну и ту же работу: копируют поля одного экземпляра класса в другой. И, как и в случае с оператором присваивания, верхнего предела стоимости копирующего конструктора нет.

Разработчик, ищущий узкие места в коде, должен уделить особое внимание присваиваниям и объявлениям, поскольку это места, в которых может скрываться дорогостоящее копирование. Фактически копирование может выполняться в любом из следующих мест.

- Инициализация (вызов конструктора).
- Присваивание (вызов оператора присваивания).
- Аргументы функции (каждое выражение аргумента создается путем перемещающего или конструирующего копирования в формальный аргумент).
- Возврат значения из функции (вызов перемещающего или копирующего конструктора, возможно, дважды).
- Вставка элементов в контейнер стандартной библиотеки (элементы создаются путем перемещающего или конструирующего копирования).
- Вставка элементов в вектор (все элементы пересоздаются путем перемещающего или конструирующего копирования при перераспределении вектора)

Скотт Мейерс (Scott Meyers) среди прочих премудростей работы с C++ основательно осветил вопрос копирующего конструирования в своей книге *Effective C++*. Приведенные здесь краткие примечания представляют собой не более чем набросок его описания.

Устранение нежелательного копирования в определении класса

Не каждый объект в программе необходимо копировать. Например, объекты, которые ведут себя, как сущности (см. раздел "Объекты-значения и объекты-сущности" данной главы), не должны копироваться, иначе они теряют свой смысл.

Многие объекты, которые ведут себя как сущности, имеют значительное количество состояний (например, вектор из 1000 строк или таблица из 1000 символов).

Да, программа, которая копирует сущность в функцию для изучения состояния этой сущности, будет корректно работать, но стоимость копирований времени выполнения может быть очень большой.

При дорогостоящем или нежелательном копировании экземпляра класса один из способов избежать его является запрет копирования. Объявление копирующего конструктора и оператора присваивания закрытыми предотвращает их вызов. Поскольку их нельзя вызвать, их определение не является необходимым; вполне достаточно только лишь объявления.

```
// Способ запрета копирования до появления стандарта C++11
class BigClass {
private:
    BigClass(BigClass const&);
    BigClass& operator=(BigClass const&);
public:
    ...
};
```

В C++11 для достижения того же эффекта к объявлению копирующего конструктора и оператора присваивания можно добавить ключевое слово `delete`. В этом случае удаленные конструкторы лучше делать открытыми, так как тогда компилятор дает более понятные сообщения об ошибках:

```
// Запрет копирования в C++11
class BigClass {
public:
    BigClass(BigClass const&)            = delete;
    BigClass& operator=(BigClass const&) = delete;
...
};
```

Любая попытка присвоения экземпляра класса, объявленного таким образом, передачи его в функцию или возврат из нее по значению, или использование в качестве значения элемента контейнера стандартной библиотеки приведет к ошибкам компиляции.

По-прежнему разрешается присваивать или инициализировать переменные указателями или ссылками на класс, или передавать в функцию и возвращать из нее ссылки или указатели на экземпляр. С точки зрения оптимизации присваивание, передача и возврат указателя или ссылки являются гораздо более эффективными, поскольку указатель или ссылка помещается в регистр процессора.

Устранение копирования при вызове функции

Я начну этот раздел с подробного описания накладных расходов во время вызова функции, когда C++-программа вычисляет аргументы. Прочитайте это описание внимательно, поскольку это очень важно с точки зрения оптимизации. Когда программа вызывает функцию, вычисляется каждое выражение аргумента и выполняется построение каждого формального аргумента со значением соответствующего выражения аргумента в качестве его инициализатора.

"Построение" означает, что вызывается конструктор формального аргумента. Если формальный аргумент имеет базовый тип, такой как `int`, `double` или `char*`, конструктор этого типа является концептуальной, а не фактической функцией. Программа просто копирует значение в память, отведенную формальному аргументу.

Однако если формальный аргумент является экземпляром некоторого класса, то для инициализации экземпляра вызывается один из копирующих конструкторов класса. Во всех, кроме тривиальных, случаях копирующий конструктор является фактической функцией. Выполняется код вызова этой функции и то, что делает сам копирующий конструктор. Если аргумент представляет собой экземпляр `std::list` с миллионом записей, то его копирующий конструктор миллион раз вызывает диспетчер памяти для создания новых записей. Если аргумент представляет собой список отображений или строк, то будет копироваться вся структура данных, узел за узлом. Для очень больших и сложных аргументов копирование может занять достаточно длительное время, чтобы привлечь внимание разработчиков. Но если аргумент во время тестирования содержит лишь несколько записей, существует риск, что ошибка будет оставаться нераспознанной до тех пор, пока данный дизайн не закрепится и не станет препятствием для масштабирования программы. Рассмотрим следующий пример:

```
int Sum(std::list<int> v) {
    int sum = 0;
    for (auto it : v)
        sum += *it;
    return sum;
}
```

Когда вызывается показанная здесь функция `Sum()`, фактическим аргументом является список, например

```
int total = Sum(MyList);
```

Формальный аргумент `v` также представляет собой список. `v` конструируется с помощью конструктора, который принимает ссылку на список. Это копирующий конструктор. Копирующий конструктор `std::list` создает копию каждого элемента списка. Если `MyList` всегда длиной только несколько элементов, излишняя стоимость оказывается сносной. Однако с ростом списка накладные расходы могут стать обременительными. Если список содержит 1000 элементов, диспетчер памяти вызывается 1000 раз. В конце функции формальный аргумент `v` выходит из области видимости, так что эти 1000 элементов по одному возвращаются в список свободной памяти.

Чтобы избежать этой цены, формальные аргументы могут быть определены как типы, имеющие тривиальные копирующие конструкторы. При передаче экземпляров класса в функцию указатели или ссылки имеют тривиальные конструкторы. В предыдущем примере `v` может быть `std::list<int> const&`. Тогда вместо копирующего конструирования экземпляра класса ссылка инициализируется ссылкой на фактический аргумент. Ссылка обычно реализуется с помощью указателя.

Передача ссылки на экземпляр класса может повысить производительность, если копирующее конструирование экземпляра вызывает диспетчер памяти для копирования внутренних данных (как в случае с контейнерами стандартной библиотеки),

класс содержит массив, который должен быть скопирован, или есть много локальных переменных. Имеется определенная стоимость доступа к экземпляру через ссылку: указатель, реализующий ссылку, должен разыменовываться всякий раз при обращении к экземпляру. Если функция большая и ее тело использует значение аргумента много раз, теоретически стоимость постоянного разыменования ссылки может превысить экономию, достигнутую отказом от копирования. Но для небольшой функции передача аргументов по ссылке повышает производительность — кроме разве что уж очень маленьких классов.

Ссылочные аргументы не ведут себя так же, как аргументы-значения. Ссылочный аргумент, изменяемый в функции, приводит к изменению экземпляра, на который он указывает, в то время как изменение аргумента-значения не оказывает никакого влияния за пределами функции. Объявление ссылочных аргументов как `const` предотвращает их случайное изменение.

Ссылочные аргументы могут также создавать псевдонимы, вызывающие непредвиденные последствия. Например, если функция имеет сигнатуру

```
void func(Foo& a, Foo& b);
```

то вызов `func(x,x);` вводит псевдоним. Если `func()` обновляет `a`, то и `b` окажется неожиданно обновленной.

Устранение копирования при возврате из функции

Когда функция возвращает значение, это значение создается копированием в неименованной временной переменной типа, возвращаемого функцией. Копирующее конструирование тривиально для фундаментальных типов, таких как `long`, `double` или указатели, но когда переменные являются экземплярами классов, копирующий конструктор обычно представляет собой реальный вызов функции. Чем больше и сложнее класс, тем более длительным является его копирующий конструктор. Вот простой пример:

```
std::vector<int> scalar_product(std::vector<int> const& v, int c) {
    std::vector<int> result;
    result.reserve(v.size());
    for (auto val : v)
        result.push_back(val * c);
    return result;
}
```

Стоимости копирующего конструирования уже созданного возвращаемого значения в некоторых случаях можно избежать, возвращая ссылку, как описано в предыдущем разделе. К сожалению, этот трюк не работает, если возвращаемое значение вычисляется в функции и присваивается переменной с автоматической длительностью хранения. Такая переменная выходит из области видимости при возврате из функции, оставляя висящие ссылки на неизвестные байты в конце стека, которые будут почти сразу же перезаписаны. Еще хуже то, что вычисления результата внутри функции — очень распространенная ситуация, так что большинство функций возвращают значения, а не ссылки.

Мало того что сама по себе стоимость копирования в возвращаемое значение достаточно плоха, так еще и возвращаемое из функции значение часто присваивается переменной в вызывающем коде, как, например, в инструкции

```
auto res = scalar_product(argarray, 10);
```

Таким образом, поверх одного копирующего конструктора, вызываемого внутри функции, в вызывающей функции используется другой копирующий конструктор или оператор присваивания.

Такое двойное копирование было сущим убийцей производительности в ранних программах C++. К счастью, головастые разработчики стандарта C++ и многих замечательных компиляторов C++ нашли способ устранения дополнительного вызова копирующего конструктора. Такая оптимизация носит разные имена — *устранение копирования* или *оптимизация возвращаемого значения* (return value optimization — RVO). Разработчики, возможно, слышали о RVO и полагают, что это означает, что они могут возвращать объекты по значению, не беспокоясь о стоимости. К сожалению, это не так. Условия, при которых компилятор может выполнять RVO, очень специфичны. Функция должна возвращать локально созданный объект. Компилятор должен иметь возможность определить, что один и тот же объект возвращается на всех путях выполнения. Объект должен иметь тип, совпадающий с объявленным возвращаемым типом функции. В приближении первого порядка, если функция невелика и имеет единственный путь выполнения, то вполне вероятно, что компилятор выполнит RVO. Если функция побольше или путь выполнения ветвится, компилятору становится сложнее определить, что RVO возможна. Качество анализа возможных оптимизаций у разных компиляторов весьма различно.

Существует способ устранения конструирования экземпляра класса внутри функции, а также *обоих* копирований (или эквивалентного копирующего конструктора с последующим оператором присваивания), которые происходят при возврате из функции. Он включает в себя прямые действия разработчика, поэтому результат оказывается более определенным, чем надежда на то, что компилятор прибегнет к RVO. Значение может быть возвращено не только с помощью оператора `return`, но и в виде *выходного параметра*, который представляет собой ссылочный аргумент, модифицируемый возвращаемым значением внутри функции:

```
void scalar_product(std::vector<int> const& v,
                    int c,
                    vector<int>& result) {
    result.clear();
    result.reserve(v.size());
    for (auto val : v)
        result.push_back(val * c);
}
```

Здесь выходной параметр `result` добавлен в список аргументов функции. У такого механизма имеется ряд преимуществ.

- Объект уже построен при вызове функции. Иногда объект должен быть очищен или повторно инициализирован, но это вряд ли будет дороже, чем конструирование.

- Объект, обновляемый внутри функции, не нужно копировать в неименованную временную переменную в операторе `return`.
- Поскольку фактические данные возвращаются в аргумент, возвращаемый тип функции может быть `void`; возвращаемое значение может также использоваться для указания ошибки или информации о состоянии выполнения функции.
- Поскольку обновленный объект уже связан с именем в вызывающей функции, его не нужно копировать или присваивать после возврата из функции.

Но это еще не все. Многие структуры данных (строки, векторы, хеш-таблицы) имеют динамически выделяемый базовый массив, который часто может быть использован повторно, если функция в программе вызывается несколько раз. Иногда результат вызываемой функции должен быть сохранен в вызывающей функции, но стоимость этого сохранения не больше, чем стоимость копирующего конструктора, который вызывался бы, если бы функция возвращала экземпляр класса по значению.

Есть ли у такого механизма какие-то дополнительные расходы времени выполнения, такие как стоимость дополнительного аргумента? Не совсем. На самом деле компилятор преобразует функцию, возвращающую значение, в функцию с дополнительным аргументом — ссылкой на неинициализированную память для неименованной временной переменной, возвращаемой функцией.

Есть одно место в C++, где нет иного выбора, кроме возвращения объектов по значению: это функции операторов. Программисты, пишущие математические функции, работающие с матрицами и желающие использовать удобочитаемый операторный синтаксис `A = B*C;`, не могут использовать ссылочные аргументы. Вместо этого они должны сосредоточиться на тщательной реализации функций операторов — таким образом, чтобы они могли максимально эффективно использовать RVO и семантику перемещения.

Библиотеки без копирования

Когда буфер, `struct` или некоторые другие структуры данных, которые должны быть заполнены информацией, являются аргументами функций, ссылка может дешево передаваться через несколько слоев вызовов библиотеки. Я слышал, как библиотеки, реализующие такое поведение, называют библиотеками, "свободными от копирования". Этот метод встречается во многих библиотеках функций, для которых производительность критически важна и стоит того, чтобы познакомиться с ним поближе.

Например, функция-член `istream::read()` стандартной библиотеки C++ имеет следующую сигнатуру:

```
istream& read(char* s, streamsize n);
```

Эта функция читает `n` байтов в хранилище, на которое указывает `s`. Буфер `s` является выходным параметром, поэтому считанные данные не должны копироваться во вновь выделяемую память. Поскольку `s` является аргументом, `istream::read()` может использовать возвращаемое значение для чего-то иного; в данном случае возвращается разыменованный указатель `*this`, который представляет собой ссылку на объект потока.

Сама функция-член `istream::read()` не выбирает данные из ядра операционной системы. Она вызывает другую функцию. Реализации могут быть различными, и она может вызвать, например, библиотечную функцию C `fread()`, которая имеет следующую сигнатуру:

```
size_t fread(void* ptr, size_t size, size_t nmemb, FILE* stream);
```

Функция `fread()` читает `size*nmemb` байтов данных и сохраняет их в памяти, на которую указывает `ptr`. Аргумент `ptr` функции `fread()` совпадает с аргументом `s` функции `istream::read()`.

Но и `fread()` — не конец цепи вызовов. В Linux `fread()` вызывает стандартную функцию Unix `read()`. В Windows `fread()` вызывает функцию `ReadFile()` из Win32. Эти две функции имеют похожие сигнатуры:

```
ssize_t read(int fd, void *buf, size_t count);

BOOL ReadFile(HANDLE hFile, void* buf, DWORD n, DWORD* bytesread,
              OVERLAPPED* pOverlapped);
```

Обе функции принимают указатель `void*` на буфер для заполнения и максимальное число читаемых байтов. Хотя его тип был приведен из `char*` к `void*` на пути вниз по цепи вызовов, указатель указывает на ту же самую память.

Существует альтернативная точка зрения на дизайн, согласно которой эти структуры и буфера должны возвращаться по значению. Они создаются в функции, поэтому не обязаны существовать до вызова функции. Такая функция оказывается на один аргумент "проще". C++ позволяет разработчику возвращать структуры по значению, поэтому этот путь должен быть "естественным" в C++. При том что разработчики Unix, Windows, C и C++ выступают за стиль без копирования, я крайне удивлен, что у этой альтернативной точки зрения имеются сторонники, несмотря на ее высокую стоимость: многократное копирование структуры или буфера при проходе по слоям библиотеки. Если возвращаемое значение имеет динамические переменные, стоимость может включать многократные вызовы менеджера памяти для создания копий. Попытка выделить структуру один раз и возвращать указатель на нее требует неоднократной передачи владения этим указателем. RVO и семантика перемещения лишь частично решают проблему расходов и требуют пристального внимания разработчика для хорошей реализации. С точки зрения производительности дизайн без копирования является гораздо более эффективным.

Реализация идиомы "копирования при записи"

Копирование при записи (copy on write — COW) представляет собой идиому программирования, которая используется для эффективного копирования экземпляров класса, содержащих динамические переменные с дорогостоящими операциями копирования. COW является хорошо известной оптимизацией с долгой историей. Она давно и долго использовалась, в частности — в реализации классов символьных строк C++. Символьные строки `CString` в Windows используют COW, как и некоторые старые реализации `std::string`. Однако стандарт C++11 явно запрещает его применение в реализации `std::string`. COW не всегда дает выигрыш

в оптимизации, поэтому его необходимо использовать разумно, несмотря на его долгую историю.

Обычно, если копируется объект, который владеет динамической переменной, должна быть создана новая копия динамической переменной. Это называется операцией *глубокого копирования*. Объект, содержащий невладеющие указатели, может обойтись копированием указателей, а не переменных, на которые они указывают. Это называется *поверхностным копированием*.

Идея копирования при записи заключается в том, что две копии объекта совпадают, пока один из них не изменяется. Таким образом, до тех пор, пока один из экземпляров не будет модифицирован, они могут совместно использовать указатели на любые части с дорогостоящим копированием. Копирование при записи сначала выполняет операцию поверхностного копирования и задерживает выполнение глубокого копирования до изменения элемента объекта.

В реализации COW в современном C++ любой член класса, который ссылается на динамическую переменную, реализуется с помощью интеллектуального указателя с разделяемым владением, такого как `std::shared_ptr`. Копирующий конструктор класса копирует указатель с разделяемым владением, задерживая создание новой копии динамической переменной до тех пор, пока любая из копий не захочет изменить динамическую переменную.

Любая трансформирующая операция класса проверяет счетчик ссылок общего указателя перед продолжением работы. Если значение счетчика ссылок больше 1, что указывает на общее владение, операция создает новую копию объекта, обменивает член с общим указателем на новую копию и освобождает старую копию, уменьшая счетчик ее ссылок. Теперь трансформирующая операция может свободно продолжаться, поскольку динамическая переменная теперь всецело принадлежит текущему объекту.

Важно создавать динамическую переменную в COW-классе, используя `std::make_shared()`. В противном случае использование общего указателя требует дополнительного обращения к диспетчеру памяти, чтобы получить объект счетчика ссылок. Для многих динамических переменных это та же стоимость, что и стоимость простого копирования динамической переменной в новое хранилище и его присвоение (не разделяемому) интеллектуальному указателю. Так что, если сделано много копий или если изменяющие содержимое динамических переменных операции обычно не вызываются, идиома копирования при записи может не давать никакого выигрыша.

Срезы

Срез (slice) представляет собой идиому программирования, в которой одна переменная ссылается на часть другой. Например, экспериментальный тип `string_view`, предложенный для стандарта C++17, ссылается на подстроку другой строки и содержит указатель `char*` на начало подстроки и длину, которая определяет конец подстроки в строке, на которую ссылается.

Срезы представляют собой небольшие, легко копируемые объекты без высокой стоимости выделения памяти для элементов и копирования содержимого в

подмассив или подстроку. Если структурой данных среза владеют интеллектуальные указатели с общим владением, то такие срезы могут быть полностью безопасными. Но опыт учит, что срезы достаточно эфемерны. Они обычно недолго служат определенной цели и выходят из области видимости прежде, чем можно будет удалять структуру данных, на которую они ссылаются. `string_view`, например, использует не владеющий указатель в строку.

Реализация семантики перемещения

Что касается оптимизации, то семантика перемещения, добавленная в C++11, — пожалуй, самое ценное из того, что когда-либо происходило с C++. Семантика перемещения решает ряд проблем из предыдущих версий C++.

- Объект присваивается переменной, что приводит к большим затратам времени на копирование внутреннего содержимого объекта, после чего исходный объект немедленно уничтожается. По большому счету копирование выполнено напрасно, потому что можно было бы повторно использовать содержимое исходного объекта.
- Разработчик хочет присвоить переменной сущность (см. раздел "Объекты-значения и объекты-сущности" выше в данной главе), например `auto_ptr` или дескриптор ресурса. Операция копирования при присваивании для такого объекта не определена, поскольку такой объект уникален.

Обе эти проблемы трудно решить, работая с динамическими контейнерами, такими как `std::vector`, в которых внутренняя память контейнера должна перераспределяться по мере увеличения количества элементов в контейнере. Первый случай делает перераспределение контейнера более дорогим, чем это необходимо. Второй случай препятствует хранению в контейнерах таких сущностей, как `auto_ptr`.

Проблема возникает из-за того, что операция копирования, выполняемая копирующими конструкторами и операторами присваивания, отлично работает для фундаментальных типов и невладеющих указателей, но не имеет смысла для сущностей. Объекты с членами таких типов могут быть помещены в массивы в стиле C, но не в динамические контейнеры наподобие `std::vector`.

До появления стандарта C++11 не было стандартного способа эффективного перемещения содержимого переменной в другую переменную в случае, когда дорогостоящее копирование не требуется.

Нестандартная семантика копирования: болезненный хак

Когда переменная ведет себя, как сущность, создание копии обычно представляет собой билет в один конец в страну неопределенного поведения. Для такой переменной рекомендуется отключить копирующий конструктор и оператор присваивания. Но такие контейнеры, как `std::vector`, требуют от своих элементов возможности копирования при перераспределении памяти контейнера, поэтому отключение копирования означает, что такой тип в качестве элемента контейнера использовать нельзя.

Для отчаянных разработчиков, желающих во что бы то ни стало разместить сущности в контейнерах стандартной библиотеки, до появления семантики перемещения обходной путь заключался в реализации присваивания нестандартным образом. Например, можно было создать разновидность интеллектуального указателя, который реализовывал присваивание так, как показано в примере 6.1.

Пример 6.1. Хакерский интеллектуальный указатель с некопирующим присваиванием

```
hacky_ptr& hacky_ptr::operator=(hacky_ptr& rhs) {
    if (*this != rhs) {
        this->ptr_ = rhs.ptr_;
        rhs.ptr_ = nullptr;
    }
    return *this;
}
```

Этот оператор присваивания компилируется и выполняется. Такая инструкция, как `q = p;`, передает владение указателем переменной `q`, устанавливая указатель в `p` равным `nullptr`. Владение при таком определении сохраняется. Указатель, определенный таким образом, работает в `std::vector`.

Хотя сигнатура оператора присваивания тонко намекает на то, что `rhs` изменяется, что является необычным поведением для присваивания, само присваивание не предлагает никакого ключа к своему девиантному поведению (пример 6.2).

Пример 6.2. Сюрпризы `hacky_ptr`

```
hacky_ptr p, q;
p = new Foo;
q = p;
...
p->foo_func();     // Сюрприз! Разыменование nullptr
```

Новый разработчик, который ожидает, что этот код будет нормально работать, оказывается очень разочарованным, возможно, после длительного сеанса отладки. Такое понимание "копирования" ужасно, его применение трудно обосновать, даже когда оно кажется необходимым.

`std::swap()`: семантика перемещения для бедных

Еще одной возможной операцией с двумя переменными является обмен содержимым двух переменных. Такой обмен хорошо определен, даже когда переменные являются сущностями, поскольку в конце операции каждая сущность содержится только в одной переменной. C++ предоставляет шаблон функции `std::swap()` для обмена содержимым двух переменных:

```
std::vector<int> a(1000000,0);
...
std::vector<int> b;  // b пуст
std::swap(v,w);      // Теперь в b миллион элементов
```

Инстанцирование `std::swap()` по умолчанию до появления семантики перемещения было эквивалентным следующему коду:

```
template <typename T> void std::swap(T& a, T& b) {
    T tmp = a;   // Создание новой копии a
    a     = b;   // Копируем b в a, уничтожая старое значение a
    b     = tmp; // Копируем tmp в b, уничтожая старое значение b
}
```

Такое инстанцирование по умолчанию работает только для объектов, для которых определена операция копирования. Оно также является потенциально неэффективным: исходное значение `a` копируется два раза, а исходное значение `b` — один раз. Если тип `T` содержит динамически выделяемые члены, выполняется три копирования и три уничтожения (с учетом уничтожения переменной `tmp` при выходе из области видимости). Это дороже, чем концептуальная операция копирования, которая делает только одно копирование и одно удаление.

Мощь операции обмена заключается в том, что она может быть рекурсивно применена к членам класса. Вместо того чтобы копировать объекты, на которые указывают указатели, поменяться местами могут сами указатели. Для классов, которые указывают на большие динамически выделенные структуры данных, обмен является гораздо более эффективным, чем копирование. На практике функции `std::swap()` могут быть специализированы для любого требуемого класса. Стандартные контейнеры предоставляют функции-члены `swap()` для обмена местами указателей на их динамические члены. Контейнеры также специализируют `std::swap()`, позволяя выполнять эффективный обмен без вызовов диспетчера памяти. Определяемые пользователем типы также могут предоставлять специализации для `std::swap()`.

`std::vector` не использует обмен для копирования своего содержимого при росте его базового массива, но аналогичным структурам данных ничто не препятствует поступать таким образом.

Проблема с обменом заключается в том, что, хотя обмен и является более эффективным, чем копирование классов с динамическими переменными, требующими глубокого копирования, он менее эффективен, чем копирование для других классов. Но как минимум обмен имеет смысл и для простых типов, и для владеющих указателей, что делает его шагом в правильном направлении.

Разделяемое владение сущностями

Копировать сущности нельзя. Однако можно копировать разделяемый указатель на сущность. Таким образом, в то время как невозможно было создать, например, `std::vector<std::mutex>` до появления семантики перемещения, вполне можно было определить `std::vector<std::shared_ptr<std::mutex>>`. Копирование `shared_ptr` имеет точно определенный смысл: создание дополнительной ссылки на уникальный объект.

Конечно, создание `shared_ptr` для сущности представляет собой обходной путь. Хотя он и имеет то преимущество, что использует инструменты стандартной библиотеки C++, в нем много ненужной сложности и он обладает повышенными накладными расходами времени выполнения.

Перемещающая часть семантики перемещения

Создатели стандарта поняли, что им необходимо закрепить операцию "перемещения" как фундаментальную концепцию в C++. Перемещение должно передавать владение. Оно более эффективно, чем копирование, и точно определено и для значений, и для сущностей. Результат получил название *семантика перемещения*. Я собираюсь осветить здесь основные моменты семантики перемещения, но есть много деталей, которые я никак не могу охватить в столь кратком объеме. Крайне рекомендую обратить внимание на книгу Скотта Мейерса (Scott Meyers) *Effective Modern C++*[4]; из 42 разделов книги автор посвящает семантике перемещения 10 разделов. Статья Томаса Беккера (Thomas Becker) *C++ Rvalue References Explained* (`http://bit.ly/becker-rvalue`) представляет собой доступное введение в семантику перемещения, свободно доступное в Интернете и содержащее гораздо больше информации, чем приведено здесь.

Для облегчения семантики перемещения в компиляторы C++ внесено изменение, чтобы они могли распознавать, когда переменная существует только как временная. Такой экземпляр не имеет имени. Например, объект, возвращаемый функцией или образующийся в результате выполнения выражения `new`, не имеет имени. На такой объект не может быть других ссылок. Объект доступен для инициализации, для присваивания переменной или использования в качестве аргумента выражения или вызова функции, но будет уничтожен в следующей точке следования. Неименованные значения называются *rvalue*, потому что они похожи на результат выражения справа от оператора присваивания. *Lvalue*, напротив, представляет собой значение, которое является именем переменной. В инструкции `y = 2*x+1;` результат выражения `2*x+1` представляет собой rvalue; это временное значение без имени. Переменная слева от знака равенства является lvalue, а ее именем является `y`.

Когда объект является rvalue, его содержимое может быть уничтожено, чтобы стать значением lvalue. Единственное налагаемое требование — rvalue должно остаться в допустимом состоянии, чтобы его деструктор вел себя корректно.

Система типов C++ была расширена так, чтобы иметь возможность отличить rvalue от lvalue при вызове функции. Если `T` — произвольный тип, то объявление `T&&` представляет собой rvalue-ссылку на `T`, т.е. ссылку на rvalue типа `T`. Правила разрешения перегрузки функции были расширены так, что, когда фактическим аргументом функции является rvalue, перегрузка rvalue-ссылки является предпочтительной, а когда аргументом является lvalue, требуется перегрузка ссылки lvalue.

Список специальных функций-членов был расширен, и теперь он включает перемещающий конструктор и оператор перемещающего присваивания. Эти функции являются перегрузками копирующего конструктора и копирующего оператора присваивания, которые принимают rvalue-ссылку. Если класс реализует перемещающий конструктор и оператор перемещающего присваивания, то инициализация или присваивание экземпляра может использовать эффективную семантику перемещения.

[4] Имеется русский перевод: Скотт Мейерс. *Эффективный и современный C++: 42 рекомендации по использованию C++11 и C++14*. — М.: ООО "И.Д. Вильямс", 2016. — 304 с.

В примере 6.3 представлен простой класс, содержащий уникальную сущность. Компилятор автоматически генерирует перемещающие конструкторы и перемещающие операторы присваивания для простых классов. Эти перемещающие операторы выполняют операцию перемещения для каждого элемента, для которого определена операция перемещения, и копирование — для других членов. Это эквивалентно выполнению для каждого члена кода `this->member = std::move(rhs.member)`.

Пример 6.3. Класс с семантикой перемещения

```
class Foo {
    std::unique_ptr<int> value_;
public:
    ...
    Foo(Foo&& rhs) {
        value_ = rhs.release();
    }

    Foo(Foo const& rhs) : value_(nullptr) {
        if (rhs.value_)
            value_ = std::make_unique<int*>(*rhs.value_);
    }
};
```

В действительности компилятор автоматически генерирует перемещающий конструктор и перемещающий оператор присваивания только в простом случае, когда программа не определяет ни копирующего конструктора, ни оператора присваивания или деструктора[5] и когда оператор перемещения не запрещен ни в одном члене или базовом классе. Это правило имеет смысл, поскольку наличие определений этих специальных функций предполагает, что может потребоваться что-то особенное, более сложное, чем почленное перемещение.

Если перемещающий конструктор и перемещающее присваивание не предоставлены программистом и не сгенерированы автоматически компилятором, программа по-прежнему может компилироваться. Компилятор просто использует менее эффективный копирующий конструктор и оператор копирующего присваивания вместо перемещающих. Из-за сложности правил автоматической генерации на практике лучше всего объявлять явно, объявлять со значениями по умолчанию или отключать все специальные функции (конструктор по умолчанию, копирующий конструктор, оператор копирующего присваивания, перемещающий конструктор, оператор перемещающего присваивания и деструктор) вместе, чтобы явно указать намерения разработчика.

Изменения кода для использования семантики перемещения

Внесение изменений в код для использования семантики перемещения может быть выполнено поклассово. Вот небольшой контрольный список, который поможет осуществить этот процесс.

[5] Копирующий конструктор и оператор присваивания автоматически генерируются, даже если определен деструктор, хотя это правило и рассматривается как устаревшее, начиная с C++11. Лучше всего явно удалять копирующий конструктор и оператор присваивания, если таковые не должны быть определены.

- Определите проблемы, которые могут быть решены с помощью семантики перемещения. Например, большое количество времени, которое тратится в копирующих конструкторах и функции управления памятью, может указывать на активно используемые классы, которые могли бы получить выгоду от добавления перемещающего конструктора и оператора перемещающего присваивания.
- Обновите компилятор C++ (и стандартную библиотеку, если она не поставляется с компилятором) до версии, поддерживающей семантику перемещения. Повторно выполните тесты производительности после обновления, поскольку такое изменение компилятора может значительно повысить производительность кода, использующего компоненты стандартной библиотеки, такие как строки и векторы, и узким местом станут совсем иные части программы.
- Проверьте библиотеки сторонних производителей, чтобы выяснить, нет ли среди них более новой версии, которая поддерживает семантику перемещения. Простая поддержка семантики перемещения компилятором ничего не дает разработчику, если библиотеки не обновлены для ее использования.
- Определите перемещающий конструктор и перемещающий оператор присваивания для классов с выявленными проблемами производительности.

Тонкости семантики перемещения

Я не хочу сказать, что семантика перемещения является хаком. Нет, эта возможность имеет слишком важное значение, и авторы стандарта действительно выполнили очень большую работу, чтобы сделать ее семантически сходной с копирующим конструированием. Но я думаю, что вполне можно сказать, что семантика перемещения — вопрос *тонкий*. Это одна из тех возможностей C++, которая должна использоваться с осторожностью и серьезными знаниями и пониманием, чтобы можно было получить преимущества от ее использования.

Перемещение экземпляров в `std::vector`

Недостаточно просто написать перемещающий конструктор и оператор перемещающего присваивания, если вы хотите, чтобы объект эффективно перемещался, находясь в `std::vector`. Разработчик должен объявить перемещающий конструктор и оператор перемещающего присваивания как `noexcept`. Это необходимо, потому что `std::vector` обеспечивает *строгую гарантию безопасности исключений*: происшедшее во время операции исключение оставляет вектор в том же состоянии, что и до операции. Копирующий конструктор не изменяет исходный объект. Перемещающий конструктор его разрушает. Любое исключение в перемещающем конструкторе нарушает строгую гарантию безопасности исключений.

Если перемещающий конструктор и оператор перемещающего присваивания не объявлены как `noexcept`, `std::vector` вынужден использовать менее эффективные операции копирования. Компилятор может не выдавать никаких предупреждений о том, что это произошло, и код по-прежнему будет работать правильно, но медленно.

noexcept представляет собой очень сильное обещание. Обещание noexcept означает отсутствие вызовов диспетчера памяти, ввода-вывода или любых других функций, которые могут вызвать исключение. Или это может означать, что все исключения будут перехвачены без возможности сообщить, что программа так поступила. В Windows это означает, что преобразование структурированных исключений в исключения C++ сопряжено с риском, так как нарушение обещания noexcept означает внезапное невосстановимое прекращение работы программы. Это цена, которую приходится платить за эффективность.

Rvalue-ссылки в качестве аргументов являются lvalue

Когда функция принимает rvalue-ссылку в качестве аргумента, он использует ее для создания формального аргумента. Поскольку формальный аргумент имеет имя, он является lvalue, несмотря на то что был создан из rvalue-ссылки.

К счастью, разработчик может явно привести lvalue к rvalue-ссылке. Стандартная библиотека предоставляет отличный шаблон функции std::move() в заголовочном файле <utility> для выполнения этой работы, как показано в примере 6.4.

Пример 6.4. Явное перемещение

```
std::string MoveExample(std::string&& s) {
    std::string tmp(std::move(s));
    // Внимание! Сейчас s – пустая строка.
    return tmp;
}
...
std::string s1 = "hello";
std::string s2 = "everyone";
std::string s3 = MoveExample(s1 + s2);
```

В примере 6.4 вызов MoveExample(s1+s2) приводит к созданию s из rvalue-ссылки, а это означает, что фактический аргумент перемещается в s. Вызов std::move(s) создает rvalue-ссылку на содержимое s. Поскольку возвращаемым значением функции std::move() является rvalue-ссылка, оно не имеет имени. Rvalue-ссылка инициализирует tmp, вызывая перемещающий конструктор std::string. После этого s больше не указывает на фактический строковый аргумент для MoveExample(). Теперь это, вероятно, пустая строка. Когда возвращается tmp, концептуально происходит следующее: значение tmp копируется в анонимное возвращаемое значение, а затем tmp удаляется. Анонимное возвращаемое значение MoveExample() копируется в s3 с использованием копирующего конструктора. Однако в действительности в этом случае компилятор может выполнить RVO, так что аргумент s фактически будет перемещен непосредственно в память s3. В общем случае RVO более эффективна, чем перемещение.

Вот версия шаблона функции std::swap() с использованием семантики перемещения — функции std::move():

```
template <typename T> void std::swap(T& a, T& b) {
{
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

Эта функция выполняет три перемещения и ни одного перераспределения памяти при условии, что тип T реализует семантику перемещения. В противном случае будет использовано копирование.

Не возвращайте rvalue-ссылки

Еще одна тонкость семантики перемещений заключена в том, что функции обычно не должны быть определены как возвращающие rvalue-ссылки. Возвращение rvalue-ссылки имеет интуитивно понятный смысл. В вызове, таком как `x = foo(y)`, возвращение rvalue-ссылки будет приводить к эффективному перемещению возвращаемого значения из неименованной временной переменной в целевую переменную `x`.

Но в действительности возврат rvalue-ссылки препятствует оптимизации возвращаемого значения (см. раздел "Устранение копирования при возврате из функции" ранее в данной главе), которая позволяет компилятору устранить копирование неименованной временной переменной в целевой объект, передавая ссылку на целевой объект в функцию в качестве неявного аргумента. Если сделать возвращаемое значение rvalue-ссылкой, это приводит к двум операциям перемещения, в то время как возврат значения приводит к единственной операции перемещения с помощью RVO.

Таким образом, ни фактический аргумент оператора `return`, ни объявленный возвращаемый тип функции не должен быть rvalue-ссылкой, если возможно применение RVO.

Перемещение базовых классов и членов

Чтобы реализовать семантику перемещения для класса, необходимо реализовать семантику перемещения также для всех базовых классов и членов, как показано в примере 6.5. В противном случае базовые классы и члены будут не перемещены, а скопированы.

Пример 6.5. Перемещение базовых классов и членов

```
class Base {...};
class Derived : Base {
    ...
    std::unique_ptr<Foo> member_;
    Bar* barmember_;
};

Derived::Derived(Derived&& rhs)
  : Base(std::move(rhs)),
    member_(std::move(rhs.member_)),
    barmember_(nullptr) {

    std::swap(this->barmember_, rhs.barmember_);
}
```

В примере 6.5 показаны некоторые тонкости написания перемещающего конструктора. Если `Base` имеет перемещающий конструктор, то он вызывается, только если lvalue `rhs` приводится к rvalue-ссылке с помощью `std::move()`. Аналогично перемещающий конструктор `std::unique_ptr` вызывается, только если `rhs.member_` приведен к rvalue-ссылке. Что касается `barmember_`, который представляет собой

обычный указатель в стиле C, или любого объекта, в котором не определен перемещающий конструктор, операция в стиле перемещения реализуется с помощью `std::swap()`.

Функция `std::swap()` может привести к некоторым проблемам при реализации оператора перемещающего присваивания. Проблема заключается в том, что `this` может указывать на объект, который уже имеет выделенную память. `std::swap()` не освобождает ненужную память. Она сохраняет ее в `rhs`, и память не будет освобождена, пока не будет уничтожен объект `rhs`. Потенциально это может играть роль, если, скажем, член содержит строку из миллиона символов или таблицу с миллионом записей. В этом случае лучше явно скопировать указатель `barmember_`, а затем обнулить его в `rhs`, чтобы деструктору `rhs` не пришлось его освобождать:

```
void Derived::operator=(Derived&& rhs) {
    Base::operator=(std::move(rhs));
    delete(this->barmember_);
    this->barmember_ = rhs.barmember_;
    rhs.barmember_ = nullptr;
}
```

Плоские структуры данных

Структура данных может быть описана как *плоская*, если ее элементы хранятся в непрерывном блоке памяти. Плоские структуры данных имеют преимущества в производительности по сравнению со структурами данных, части которых связаны между собой с помощью указателей.

- Плоские структуры данных требуют меньше дорогостоящих вызовов диспетчера памяти для построения, чем структуры данных, части которых связаны между собой указателями. Одни структуры данных (список, дек, отображение, хеш-таблица) создают множество динамических объектов; другие же (вектор) существенно меньше. Как неоднократно показано в главе 10, "Оптимизация структур данных", даже если подобные операции имеют ту же производительность, выраженную через "большое O", плоские структуры данных, такие как `std::vector` и `std::array`, имеют при этом значительное преимущество.
- Плоские структуры данных, такие как `array` и `vector`, занимают меньше памяти, чем структуры данных на основе узлов, такие как список, отображение или хеш-таблица, из-за наличия связывающих указателей в структурах на основе узлов. Компактность улучшает локальность кеша, даже если общее количество байтов не является проблемой. Плоские структуры данных имеют преимущество в плане локальности кеша, что делает их более эффективными при обходе.
- Трюки наподобие создания векторов или отображений интеллектуальных указателей, которые были необходимы до появления семантики перемещения в C++11 для хранения некопируемых объектов, больше необходимыми не являются. Значительная стоимость времени выполнения, связанная с выделением памяти для интеллектуальных указателей и указываемых объектов, теперь может быть устранена.

Резюме

- *Наивное использование динамически выделяемых переменных является наибольшим врагом производительности в программах C++. Когда производительность важна,* `new` *вам не друг.*
- *Разработчик может существенно повысить производительность, не делая ничего, кроме уменьшения количества вызовов диспетчера памяти.*
- *Программа может глобально изменить принципы выделения и освобождения памяти, предоставляя определения операторов* `::operator new()` *и* `::operator delete()`.
- *Программа может глобально изменить принципы управления памятью, заменяя функции* `malloc()` *и* `free()`.
- *Интеллектуальные указатели автоматизируют владение динамическими переменными.*
- *Разделяемое владение динамическими переменными имеет большую стоимость.*
- *Создавайте экземпляры класса статически.*
- *Создавайте члены класса статически и при необходимости прибегайте к двухэтапной инициализации.*
- *Используйте главный указатель для владения динамическими переменными и невладеющие указатели вместо использования совместного владения.*
- *Создавайте функции без копирования, которые возвращают данные через выходные параметры.*
- *Реализуйте семантику перемещения.*
- *Предпочитайте плоские структуры данных.*

ГЛАВА 7

Оптимизация инструкций

Скульптура уже здесь, в глыбе мрамора. Все, что мне остается — убрать все лишнее.

— Микеланджело де Франческо де Нери де Миниато дель Сера и Лодовико ди Леонардо ди Буонарроти Симони (Michelangelo di Francesci di Neri di Miniato del Sera i Lodovico di Leonardo di Buonarroti Simoni) (1475–1564) в ответ на вопрос "Как вы создаете свои скульптуры?"

Оптимизация на уровне инструкций может быть смоделирована как процесс *удаления инструкций из потока выполнения*, так же, как Микеланджело описал процесс создания своих шедевров. Проблема в совете Микеланджело в том, что он не поясняет, какая часть глыбы лишняя, а какая является шедевром.

Проблема оптимизации на уровне инструкций заключается в том, что, помимо вызовов функций, никакие инструкции C++ не превращаются в большее, чем несколько машинных команд. Обычно такие мелкие оптимизации не дают улучшения, достаточного для того, чтобы оправдать приложенные усилия, если только разработчик не обнаружит факторы, которые увеличивают стоимость инструкции, делая ее достаточно "горячей", чтобы ее стоило оптимизировать. Эти факторы включают следующее.

Циклы

Стоимость операторов в цикле умножается на количество их повторений. Профайлер может указывать на функцию, содержащую горячий цикл, но не указать, какой именно цикл в функции самый горячий. Он может указать на функцию, которая является горячей потому, что вызывается в одном или нескольких циклах, но не укажет, какие именно места вызова являются горячими. Так как профайлер не указывает непосредственно на цикл, разработчик должен изучить код, чтобы найти конкретный цикл, используя вывод профайлера в качестве источника улик.

Часто вызываемые функции

Стоимость функции умножается на количество ее вызовов. Профайлер указывает непосредственно на горячие функции.

Идиомы, используемые в программе

Сюда относятся все категории инструкций и идиом C++, для которых имеются менее дорогостоящие альтернативы. Если эти идиомы широко используются в программе, их замена менее дорогими идиомами может повысить общую производительность.

Оптимизация кода на уровне инструкций может дать существенное увеличение производительности на небольших, более простых процессорах, которые встроены в инструменты, приборы, периферийные устройства и игрушки, потому что они получают команды непосредственно из памяти и выполняют их одну за другой. Однако процессоры уровня настольных компьютеров обеспечивают такой уровень параллелизма на уровне команд и кеширования, что оптимизация на уровне инструкций дает меньший эффект, чем оптимизация распределения памяти или копирования.

В программах, разработанных для класса настольных компьютеров, оптимизация на уровне инструкций может быть целесообразной только для часто вызываемых библиотечных функций или наиболее глубоко вложенных циклов программы, например в графических механизмах игр или при программирования переводчиков с других языков, которые работают все время.

Еще одна проблема оптимизации на уровне инструкций связана с тем, что эффективность оптимизации может зависеть от компилятора. Каждый компилятор имеет один или несколько планов, как генерировать код для определенного оператора C++. Идиома кодирования, которая улучшает производительность на одном компиляторе, может не давать результата на другом и даже замедлить код на третьем. Трюк, повышающий производительность при использовании GCC, может не работать с Visual C++. Более того, это означает, что, когда команда программистов обновляет свой компилятор до новой версии, новый компилятор может вызвать снижение оптимальности имеющегося тщательно настроенного кода. Это еще одна причина, по которой оптимизация на уровне инструкций может быть менее плодотворной, чем другие усилия по повышению производительности.

Удаление кода из циклов

Цикл состоит из двух частей: многократно выполняемого блока инструкций и управляющей конструкции, которая определяет, сколько раз повторяется цикл. Общие замечания по удалению вычислений из операторов C++ применимы и к инструкциям в теле цикла. Что касается управляющих конструкций циклов, то здесь имеются дополнительные возможности для оптимизации, потому что в определенном смысле они представляет собой накладные расходы.

Рассмотрим цикл `for` из примера 7.1, в котором выполняется обход строки с заменой пробела звездочкой.

Пример 7.1. Неоптимизированный цикл `for`

```
char s[] = "В этой строке немало символов пробела (0x20). ";
...
for (size_t i = 0; i < strlen(s); ++i)
    if (s[i] == ' ')
        s[i] = '*';
```

Проверка `i < strlen(s)` в цикле `for` выполняется для каждого символа строки[1]. Вызов `strlen()` — дорогостоящий, приводящий к обходу всей строки для подсчета ее символов и превращающий этот алгоритм из алгоритма O(n) в алгоритм O(n^2). Это пример внутреннего цикла, скрытого в библиотечной функции (см. раздел "Оценка стоимости циклов" главы 3, "Измерение производительности").

Десять миллионов итераций этого цикла занимают 13 238 мс при компиляции кода Visual Studio 2010 и 11 467 мс — при использовании Visual Studio 2015. Измерения показывают, что VS2015 генерирует на 15% более быстрый код, что говорит о том, что эти компиляторы генерируют для данного цикла разные коды.

Кеширование конечного значения цикла

Конечное значение цикла, возвращаемое дорогостоящей функцией `strlen()`, можно предварительно вычислить и кешировать в заголовке цикла для повышения производительности. Измененный цикл показан в примере 7.2.

Пример 7.2. Цикл `for` с кешированным конечным значением

```
for (size_t i = 0, len = strlen(s); i < len; ++i)
    if (s[i] == ' ')
        s[i] = '*';
```

Эффект от этого изменения потрясающий из-за высокой стоимости функции `strlen()`. Тест оптимизированного кода выполняется за 636 мс при компиляции VS2010 и 541 мс в случае VS2015 — почти в 20 раз быстрее, чем исходный код. VS2015 по-прежнему опережает VS2010, на этот раз — на 17%.

Применение более эффективных инструкций циклов

Вот как выглядит синтаксис цикла `for` в C++:

```
for(инициализация ; условие ; выражение_продолжения ) тело_цикла
```

Грубо говоря, этот цикл компилируется в код, выглядящий примерно таким образом:

```
    инициализация ;
L1: if ( ! условие ) goto L2;
    тело_цикла ;
    выражение_продолжения ;
    goto L1;
L2:
```

Цикл `for` должен выполнять переход дважды: один раз — при ложности *условия* и еще один раз — после вычисления *выражения_продолжения*. Эти переходы могут замедлить выполнение. C++ имеет более простой, но реже используемый цикл `do`, имеющий следующий синтаксис:

```
do тело_цикла while ( условие ) ;
```

[1] Некоторые читатели бурно отреагируют на этот код: "Ну сколько можно! Зачем писать такой код? Разве вы не в курсе, что `std::string` имеет функцию `length()` с константным временем работы?" Тем не менее такого рода код постоянно встречается в программах, нуждающихся в оптимизации. Кроме того, я хотел привести совершенно очевидный каждому пример.

Опять же, грубо говоря, этот цикл компилируется в код, выглядящий примерно таким образом:

```
L1: тело_цикла
    if ( условие ) goto L1;
```

Таким образом, цикл `for` часто может быть упрощен до цикла `do`, который может оказаться более быстрым. В примере 7.3 показан код сканирования строки с помощью цикла `do`.

Пример 7.3. Преобразование цикла `for` в цикл `do`

```
size_t i = 0, len = strlen(s); // Инициализация цикла for
do {
    if (s[i] == ' ')
        s[i] = '*';
    ++i;                       // Выражение_продолжения цикла for
} while (i < len);             // Условие цикла for
```

При использовании Visual Studio 2010 мы получаем выигрыш в производительности: тестовый цикл выполняется за 482 мс, т.е. быстрее на 12%. Однако в случае Visual Studio 2015 это изменение приводит к куда худшим результатам: тест выполняется за 674 мс, или на 25% *медленнее* цикла `for`.

Изменение направления цикла

Вариацией кеширования конечного значения является изменение направления отсчета; таким образом, кешируя конечное значение в переменной индекса цикла. Многие циклы имеют одно конечное условие, существенно менее дорогостоящее, чем другое. Например, в цикле в примере 7.3 одно конечное условие представляет собой константу 0, в то время как другое конечное условие является дорогостоящим вызовом `strlen()`. В примере 7.4 представлен цикл из примера 7.1, в котором изменено направление отсчета.

Пример 7.4. Оптимизация цикла путем изменения направления

```
for (int i = (int)strlen(s)-1; i >= 0; --i)
    if (s[i] == ' ')
        s[i] = '*';
```

Обратите внимание, что я изменил тип переменной `i` с беззнакового типа `size_t` на знаковый тип `int`. Условие завершения цикла имеет вид `i>=0`. Если бы `i` было беззнаковым, то оно, определенно, всегда было бы больше или равно нулю, так что такой цикл не мог бы завершиться. Это очень распространенная ошибка при обратном счете до нуля.

Тот же самый хронометрический тест показывает, что время выполнения цикла — 619 мс при компиляции Visual Studio 2010 и 571 мс — при компиляции Visual Studio 2015. Непонятно, почему наблюдается такое значительное отклонение от результатов кода из примера 7.2.

Устранение инвариантного кода из циклов

Пример 7.2, в котором конечное значение цикла кешируется для эффективного повторного использования, является примером более общей методики устранения *инвариантного* кода из цикла. Код является инвариантным относительно цикла, если он не зависит от индекса цикла. Например, в несколько надуманном цикле в примере 7.5 оператор присваивания `j = 100;` и подвыражение `j*x*x` являются инвариантными относительно цикла.

Пример 7.5. Цикл с инвариантным кодом

```
int i,j,x,a[10];
...
for (i=0; i<10; ++i) {
    j = 100;
    a[i] = i + j * x * x;
}
```

Этот цикл может быть переписан так, как показано в примере 7.6.

Пример 7.6. Цикл с удаленным инвариантным кодом

```
int i,j,x,a[10];
...
j = 100;
int tmp = j * x * x;
for (i=0; i<10; ++i) {
    a[i] = i + tmp;
}
```

Современные компиляторы хорошо находят фрагменты инвариантного кода (такие, как показанные здесь), которые можно вынести из тела цикла для повышения производительности. Разработчику обычно не нужно переписывать цикл, потому что компилятор сам находит инвариантный код и переписывает цикл при компиляции.

Когда инструкция внутри цикла вызывает функцию, компилятор может не быть в состоянии определить, зависит ли возвращаемое функцией значение от чего-то в цикле. Функция может быть сложной, тело функции может быть в другой единице компиляции и не видимым для компилятора. Разработчик должен сам определить вызовы функций, являющиеся инвариантными относительно цикла, и вручную удалить их из цикла.

Удаление ненужных вызовов функций из циклов

Вызов функции может выполнять сколь угодно большое количество инструкций. Если функция инвариантна относительно цикла, ее стоит убрать из цикла для повышения производительности. В примере 7.1, воспроизводимом здесь, вызов `strlen()` является инвариантным относительно цикла и может быть вынесен из него:

```
for (size_t i = 0; i < strlen(s); ++i)
    if (s[i] == ' ')
        s[i] = '*';      // Замена ' ' на '*'
```

В примере 7.7 показано, как может выглядеть исправленный цикл.

Пример 7.7. Цикл с инвариантом `strlen()`

```
size_t end = strlen(s);
for (size_t i = 0; i < end; ++i)
    if (s[i] == ' ')
        s[i] = '*';      // Замена ' ' на '*'
```

В примере 7.8 значение, возвращаемое `strlen()`, не является инвариантом относительно цикла, поскольку удаление пробела уменьшает длину строки. Конечное условие в данном случае нельзя выносить из цикла.

Пример 7.8. Цикл, в котором `strlen()` не является инвариантом цикла

```
for (size_t i = 0; i < strlen(s); ++i)
    if (s[i] == ' ')
        memmove(&s[i], &s[i+1],strlen(&s[i])); // Удаляем пробел
```

Не существует простого правила для определения, является ли функция инвариантной относительно цикла в конкретной ситуации. В примере 7.8 показано, что конкретная функция инвариантна для одного цикла, но не инвариантна для другого. Это то место, где человеческое мышление превосходит тщательный, но ограниченный анализ компилятора. (Вызов `strlen()` — не единственное неверное место в этой функции. Другие ошибки и возможности оптимизации остаются читателям в качестве упражнения.)

Одной из разновидностей функции, которая всегда может быть вынесена из цикла, является *чистая функция*, т.е. функция, возвращаемое значение которой зависит только от значений ее аргументов и которая не имеет побочных действий. Если такая функция появляется в цикле, а ее аргумент в цикле не изменяется, то это инвариантная относительно цикла функция, которая может быть вынесена из цикла. В примере 7.7 функция `strlen()` является чистой функцией. В этом цикле длина ее аргумента `s` никогда не изменяется, поэтому вызов `strlen()` инвариантен относительно цикла. В цикле в примере 7.8 вызов `memmove()` изменяет длину `s`, поэтому вызов `strlen(s)` не является инвариантным.

Вот еще один пример, включающий математические функции `sin()` и `cos()`, которые возвращают значения математических функций синуса и косинуса для значения в радианах. Многие математические функции являются чистыми, поэтому в численных расчетах такая ситуация возникает достаточно часто. В примере 7.9 графическое преобразование поворота применяется к фигуре с 16 вершинами. Хронометраж 100 миллионов выполнений этого преобразования занимает 7 502 мс для компилятора VS2010 и 6 864 мс для VS2015 (как видим, этот компилятор вновь демонстрирует преимущество в скорости генерируемого кода).

Пример 7.9. `rotate()` с циклом, содержащим инвариантные чистые функции

```
void rotate(std::vector<Point>& v, double theta) {
    for (size_t i = 0; i < v.size(); ++i) {
        double x = v[i].x_, y = v[i].y_;
        v[i].x_ = cos(theta)*x - sin(theta)*y;
        v[i].y_ = sin(theta)*x + cos(theta)*y;
    }
}
```

Функции `sin(theta)` и `cos(theta)` зависят только от аргумента функции `theta` и не зависят от переменной цикла. Поэтому их можно вынести из цикла, как показано в примере 7.10.

Пример 7.10. `rotate_invariant()` с инвариантными чистыми функциями, вынесенными из цикла

```
void rotate_invariant(std::vector<Point>& v, double theta) {
    double sin_theta = sin(theta);
    double cos_theta = cos(theta);
    for (size_t i = 0; i < v.size(); ++i) {
        double x = v[i].x_, y = v[i].y_;
        v[i].x_ = cos_theta*x - sin_theta*y;
        v[i].y_ = sin_theta*x + cos_theta*y;
    }
}
```

Эта функция работает примерно на 3% быстрее, со временем выполнения 7 382 мс (VS2010) и 6 620 мс (VS2015).

Экономия на настольном компьютере менее драматична, чем полученная путем отмены вызова `strlen()` в предыдущем разделе, так как математические функции, как правило, работают с одним или двумя числами в регистрах и не требуют доступа к памяти, как `strlen()`. На встроенном же процессоре без аппаратных команд с плавающей точкой или на древнем PC 1990-х годов без сопроцессора экономия может быть куда более существенной, поскольку вычисление синусов и косинусов там обходится гораздо дороже.

Иногда функция, вызываемая в цикле, вообще не совершает никакой работы или выполняет необязательную работу. Можно, конечно, удалять такие функции. Легко сказать "Только болван может вызывать функцию, которая не совершает никакой полезной работы!" Гораздо труднее запомнить все места, где вызывается функция, и проверить их все, когда поведение функции изменяется за несколько лет жизни проекта.

Приведенный далее код представляет собой идиому, с которой я не раз сталкивался на протяжении своей карьеры:

```
UsefulTool subsystem;
InputHandler input_getter;
...
while (input_getter.more_work_available()) {
    subsystem.initialize();
    subsystem.process_work(input_getter.get_work());
}
```

В этой повторяющейся схеме подсистема многократно инициализируется для работы, а затем запрашивает у процесса очередную порцию работы. Нет ли каких-то "некорректностей" в коде, можно определить только путем проверки `UsefulTool::initialize()`. Может оказаться, что `initialize()` нужно вызывать только перед *первой* порцией работы или, возможно, для первого блока данных и после ошибки. Часто `process_work()` на выходе устанавливает тот же инвариант класса, что и `initialize()`. Вызов `initialize()` в каждой итерации цикла просто повторяет тот же код, который выполняет `process_work()`. Если это так, вызов `initialize()` можно вынести из цикла:

```
UsefulTool subsystem;
InputHandler input_getter;
...
subsystem.initialize();
while (input_getter.more_work_available()) {
    subsystem.process_work(input_getter.get_work());
}
```

Не следует самонадеянно винить разработчика, который написал первоначальный код, в небрежности. Иногда поведение `initialize()` изменяется с перемещением кода в `process_work()`. Иногда проектной документации оказывается недостаточно или назначение кода `initialize()` оказывается неясным, и разработчик просто прибегает к оборонительному программированию. Я несколько раз встречался в реальных проектах с инициализацией, которая требовалась однократно, но выполнялась каждый раз перед очередной порцией работы.

Если экономия времени является достаточно важной, стоит взглянуть на каждый вызов функции в цикле, чтобы убедиться в том, что ее работа действительно необходима.

Удаление скрытых вызовов функций из циклов

Обычные вызовы функций имеют свой четко выраженный вид — с именем функции и списком выражений аргументов в скобках. Но код C++ может также вызывать функции неявно, без этого легко обнаруживаемого синтаксиса. Это может произойти, когда переменная класса появляется в любой из следующих ситуаций.

- Объявление экземпляра класса (вызывает конструктор).
- Инициализация экземпляра класса (вызывает конструктор).
- Присваивание экземпляру класса (вызывает оператор присваивания).
- Арифметическое выражение с участием экземпляров класса (вызывает функции операторов).
- Выход из области видимости (вызывает деструкторы экземпляров класса, объявленные в области видимости).
- Аргументы функции (каждое выражение аргумента, передаваемого по значению, конструирует формальный аргумент копированием).
- Функция, возвращающая экземпляр класса (вызывает копирующий конструктор, возможно, дважды).
- Вставка элемента в контейнер стандартной библиотеки (элементы создаются копирующим или перемещающим конструированием).
- Вставка элементов в вектор (все элементы создаются копирующим или перемещающим конструированием при перераспределении памяти).

Такие вызовы функций *скрытые*. Они не имеют знакомого внешнего вида вызова функции с именем и списком аргументов; они выглядят, как присваивания и объявления. Поэтому очень легко упустить тот факт, что это на самом деле вызов функции. Я говорил об этом раньше, в разделе "Устранение излишнего копирования" главы 6, "Оптимизация переменных в динамической памяти".

Вызовы скрытых функций, являющиеся результатом построения формальных аргументов функций, иногда могут быть устранены передачей ссылки или указателя на класс вместо передачи фактического аргумента по значению. Преимущество этого метода было продемонстрировано для строк в разделе "Устранение копирования строкового аргумента" главы 4, "Оптимизация использования строк", и для любого объекта с копированием данных в разделе "Устранение копирования при вызове функции" главы 6, "Оптимизация переменных в динамической памяти".

Вызовы скрытых функций в результате копирования возвращаемого функцией значения могут быть удалены, если изменить сигнатуру функции таким образом, чтобы возвращаемый экземпляр класса создавался по ссылке или указателю, используемому в качестве выходного параметра функции. Преимущество этого метода было продемонстрировано для строк в разделе "Устранение копирования возвращаемого значения" главы 4, "Оптимизация использования строк", и для любого объекта, который копирует данные в разделе "Устранение копирования при возврате из функции" главы 6, "Оптимизация переменных в динамической памяти".

Если присваивание или инициализированное объявление является инвариантом относительно цикла, его можно вынести из него. Иногда, даже если значение переменной необходимо устанавливать при каждой итерации цикла, можно вынести ее объявление из цикла и на каждой итерации выполнять менее дорогостоящую функцию. Например, `std::string` — это класс, который содержит динамически выделенный массив `char`. В коде

```
for (...) {
    std::string s("<p>");
    ...
    s += "</p>";
}
```

размещение объявления `s` в цикле `for` является дорогостоящим. По достижении закрывающей фигурной скобки блока инструкций вызывается деструктор `s`. Этот деструктор освобождает динамически выделенную для `s` память, так что при очередной итерации для этой строки вновь должна быть выделена память. Этот код можно переписать следующим образом:

```
std::string s;
for (...) {
    s.clear();
    s += "<p>";
    ...
    s += "</p>";
}
```

Теперь деструктор `s` не вызывается до более позднего момента. Это не только экономит стоимость вызова функции при каждой итерации, но и, как побочное действие, использует динамический массив внутри `s` повторно, тем самым, возможно, устраняя скрытый вызов диспетчера памяти при добавлении символов к `s`.

Это поведение применяется не только к строкам и не только к классам, которые содержат динамическую память. Экземпляр класса может содержать ресурс, полученный от операционной системы, например файл или дескриптор окна, или может выполнять иные дорогостоящие действия внутри конструкторов и деструкторов.

Удаление дорогих медленно меняющихся вызовов из циклов

Некоторые вызовы функций не являются инвариантными, но могут быть таковыми. Хорошим примером является вызов для получения текущего времени суток для использования в журнале. Он выполняет нетривиальное количество команд для получения времени суток от операционной системы и еще больше времени, чтобы отформатировать полученное время в виде текста. В примере 7.11 представлена функция форматирования текущего времени в массив символов с завершающим нулевым символом.

Пример 7.11. `timetoa()`: форматирование времени в массиве символов

```
#include <ctime>
#include <cstring>

char* timetoa(char *buf, size_t bufsz) {
    if (buf == 0 || bufsz < 9)
        return nullptr;                    // Неверный аргумент
    time_t t  = std::time(nullptr);        // Получение времени от
                                           // операционной системы
    tm tm     = *std::localtime(&t);       // Разделение на часы,
                                           // минуты, секунды
    size_t sz = std::strftime(buf, bufsz, // Форматирование в
                              "%c", &tm); // буфере buf
    if (sz == 0)                           // Ошибка
        std::strcpy(buf, "XX:XX:XX");
    return buf;
}
```

Хронометраж показывает, что `timetoa()` требует около 700 нс для получения значения времени и его форматирования. Это значительная стоимость, почти вдвое превышающая добавление двух текстовых строк в файл. В том же тестовом цикле инструкция

```
out << "Fri Jan 01 00:00:00 2016"
    << " Тестовая строка для вывода в журнальный файл\n";
```

требует только 372 нс, в то время как инструкция

```
out << timetoa(buf, sizeof(buf))
    << " Тестовая строка для вывода в журнальный файл\n";
```

выполняется за 1042 нс.

Ведение журнала должно быть максимально эффективным, иначе оно замедляет работу программы. Плохо, если это делает программу медленнее, но куда хуже, если такое замедление изменяет поведение программы так, что при включенном журнале ошибки исчезают. В приведенном примере записи в журнал доминирует стоимость получения текущего времени.

Время изменяется очень медленно по сравнению со скоростью выполнения команд современным компьютером. Между двумя тактами часов моя программа могла бы, пожалуй, записать в журнальный файл миллион строк. Таким образом, весьма вероятно, что два последовательных вызова получения текущего времени возвратят

одно и то же значение. Если несколько строк записываются в журнал в один присест, извлечение времени заново для каждой строки не имеет смысла.

Я выполнил еще один тест, имитирующий поведение ведения журнала, когда программа запрашивает время только один раз и выводит группу из 10 строк, используя одни и те же данные. Как и ожидалось, этот тест выводил строки со скоростью в среднем 376 нс на одну строку.

Перемещение циклов в функции для снижения накладных расходов при вызовах

Если программа выполняет проход по строке, массиву или другим структурам данных и вызывает функцию на каждой итерации, можно повысить производительность путем применения методики, именуемой *инверсией цикла*. Цикл, вызывающий функцию, инвертируется путем перемещения внутрь функции. Это изменяет интерфейс функции, которая теперь ссылается на всю структуру данных вместо ссылки на один элемент. Таким образом, если структура данных содержит n записей, стоимость $n-1$ вызовов функции можно устранить.

В качестве очень простого примера представьте библиотечную функцию, заменяющую непечатные символы точкой (`"."`):

```
# include <ctype>
void replace_nonprinting(char& c) {
    if (!isprint(c))
        c = '.';
}
```

Чтобы заменить все непечатные символы в строке, программа вызывает `replace_nonprinting()` в цикле:

```
for (unsigned i = 0, e = str.size(); i < e; ++i)
    replace_nonprinting(str[i]);
```

Если компилятор не может встроить вызов функции `replace_nonprinting()`, он будет вызывать функцию 35 раз при обработке строки `"Звякнула возвратившаяся каретка!!\x07\x07"`.

Разработчик библиотеки может добавить перегрузку `replace_nonprinting()` для обработки всей строки:

```
void replace_nonprinting(std::string& str) {
    for (unsigned i = 0, e = str.size(); i < e; ++i)
        if (!isprint(str[i]))
            c = '.';
}
```

Теперь цикл находится внутри функции, и устраняется необходимость в $n-1$ вызовах функции `replace_nonprinting()`.

Обратите внимание: код, реализующий поведение `replace_nonprinting()`, в новой перегрузке должен быть дублирован. Если просто переместить цикл в функцию, это не сработает. Следующая версия фактически добавляет лишний вызов функции ко всем вызовам, которые выполнялись ранее:

```
void replace_nonprinting(std::string& str) {
    for (unsigned i = 0, e = str.size(); i < e; ++i)
        replace_nonprinting(str[i]);
}
```

Выполняйте некоторые действия пореже

Вот мотивирующий вопрос: "Основной цикл программы обрабатывает около 1000 транзакций в секунду. Как часто нужно проверять команду завершения цикла?"

Конечно же, правильный ответ — "Зависит от обстоятельств". Ответ зависит от двух вещей: как быстро программа должна отреагировать на запрос завершения и как дорого стоит соответствующая проверка.

Если программа должна завершиться в пределах 1 с, чтобы отвечать поставленным требованиям, а проверка занимает в среднем 500±100 мс, то проверку надо выполнять раз в 400 миллисекунд (1000–(500+100) = 400 мс). Более частая проверка будет слишком расточительной.

Другим фактором является стоимость проверки завершения. Если цикл представляет собой цикл обработки сообщений Windows, а команда завершения получается как сообщение Windows `WM_CLOSE`, никаких дополнительных накладных расходов проверки нет. Стоимость проверки встроена в диспетчеризацию событий. Если же обработчик сигнала приводит к установке значения булева флага, стоимость проверки на каждой итерации составляет крошечную величину.

Но что делать, если цикл работает на встраиваемом устройстве, на котором для этого должен выполняться опрос клавиш, а "противодребезговая" защита[2] требует опроса в течение 50 мс? Тестирование на каждой итерации будет добавлять стоимость опроса клавиши, равную 50 мс, тем самым снижая скорость работы цикла с 1000 до $\frac{1}{0.051} \approx 20$ транзакций в секунду. Ой!

Если же программа будет выполнять опрос каждые 400 миллисекунд, его влияние на цикл окажется менее драматичным. Математика здесь немного утомительна, но все же давайте разберемся. Транзакции занимают время около 1 мс (исходя из 1000 транзакций в секунду). Для выполнения опроса длительностью 50 мс каждые 400 мс опрос должен выполняться через каждые 350 мс, или 2,5 раза за 1000 мс. Это приводит к скорости транзакций, равной 1000–(2,5*50) = 875 транзакций в секунду.

В примере 7.12 показан код, выполняющий такую проверку.

Пример 7.12. Более редкая проверка событий

```
void main_loop(Event evt) {
    static unsigned counter = 1;
    if ((counter % 350) == 0)
        if (poll_for_exit())
```

[2] Когда нажимается реальная механическая клавиша, в самом начале наблюдается прерывистость подключения (дребезг контактов). Если рассмотреть мгновенное состояние подключения, то можно оказаться в ситуации, когда опрос говорит, что клавиша не нажата, хотя на самом деле она нажата, но находится на первоначальном, прерывистом участке подключения. Противодребезговая защита задерживает сообщение о нажатии клавиши до тех пор, пока сигнал не станет непрерывным; типичное значение такой задержки — 50 мс.

```
            exit_program();     // Безвозвратный выход
        ++counter;

        switch (evt) {
        ...
        }
    }
```

Выполнение входит в `main_loop()` каждую миллисекунду (в предположении, что события происходят каждую миллисекунду). На каждой итерации счетчик `counter` увеличивается. Когда он достигает значения, кратного 350, программа вызывает функцию `poll_for_exit()`, которая выполняется в течение 50 мс. Если в `poll_for_exit()` обнаружен запрос на выход, код вызывает функцию `exit_program()`, которая за 400–600 мс выполняет выход из программы.

Этот метод редкого опроса показывает, как выполнять между опросами большее количество вычислений. Однако для применения рассмотренного метода нам требуется выполнение следующих предположений.

- Предполагается, что события происходят каждую миллисекунду, а не время от времени каждые 2 или каждые 5 миллисекунд, и что поток событий постоянен и не снижается, если никакой работы для программы нет.
- Предполагается, что опрос всегда занимает ровно 50 мс.
- Предполагается, что отладчик никогда не перехватывает управление, так что события обрабатываются по полминуты, пока программист изучает значения переменных.

Более аккуратный подход заключается в измерении прошедшего от события к событию времени и промежутка от начала до конца `poll_for_exit()`.

Разработчик, желающий достичь полной скорости в 1000 транзакций в секунду при перечисленных здесь ограничениях, должен найти некоторый источник истинного параллелизма для вынесения опроса клавиатуры из основного цикла. Обычно это реализуется через прерывания, несколько ядер или аппаратное сканирование клавиатуры.

И все остальное

В Интернете есть много источников информации по низкоуровневым методам оптимизации циклов. Например, некоторые источники отмечают, что `++i` в общем случае быстрее, чем `i++`, поскольку при этом отсутствует промежуточное значение, которое необходимо сохранять и возвращать. Некоторые источники рекомендуют развертывать циклы, чтобы уменьшить общее количество тестов в цикле и увеличить выполняемый код.

Основной проблемой, связанной с этими рекомендациями, является то, что они не всегда работают. Вы можете потратить время, чтобы провести эксперимент, и при этом не обнаружить никаких улучшений. Рекомендации могут быть основаны на предположениях, которые не выполнены при проведении эксперимента или могут применяться только к конкретной версии конкретного компилятора. Они могут

исходить из учебника по проектированию компиляторов и описывать оптимизации, которые компилятор C++ давно уже умеет делать сам. За тридцать лет современные компиляторы C++ научились *очень* хорошо находить код, который стоит вынести из цикла. Фактически в этом компилятор превосходит большинство программистов. Вот почему такая оптимизация так часто удручающе неэффективна и вот почему этот раздел такой короткий.

Удаление кода из функций

Функции, подобно циклам, состоят из двух частей: блока кода, который представляет собой тело функции, и заголовка функции, состоящего из списка аргументов и возвращаемого типа. Как и в случае циклов, эти две части могут быть оптимизированы по отдельности.

Хотя стоимость выполнения тела функции может быть сколь угодно велика, стоимость вызова функции, как и стоимость большинства инструкций C++, незначительна. Но когда функция вызывается много раз, стоимость умножается на количество вызовов и может стать значительной, так что сокращение этой стоимости становится достаточно важным.

Стоимость вызовов функций

Функции являются старейшей и наиболее важной абстракцией в программировании. Программист определяет функцию один раз, а затем вызывает ее имя из многих мест программы. Для каждого вызова компьютер сохраняет текущее место в выполняемом коде, передает управление телу функции, а затем возвращается к оператору, следующему после вызова функции, эффективно вставляя тело функции в поток выполнения инструкций.

Это удобство записи кода не дается бесплатно. Каждый раз, когда программа вызывает функцию, происходит примерно следующее (конечно, в зависимости от архитектуры процессора и настроек оптимизатора).

1. Выполняемый код помещает новый кадр в стек вызовов, в котором будут храниться аргументы и локальные переменные функции.
2. Каждое выражение аргумента вычисляется и копируется в кадр стека.
3. Адрес выполнения копируется в кадр стека для образования адреса возврата.
4. Выполняемый код обновляет адрес выполнения, так что вместо следующей инструкции после вызова функции он становится равным адресу первой инструкции тела функции.
5. Выполняются инструкции тела функции.
6. Из кадра стека в адрес выполнения заносится сохраненный перед вызовом адрес инструкции, следующей за вызовом функции, которой и передается управление.
7. Кадр стека снимается со стека.

Хорошей новостью о стоимости функций является то, что программа с функциями, как правило, более компактна, чем та же программа с большими функциями, но встраиваемыми в код. Это улучшает производительность кеша и виртуальной памяти. Тем не менее при прочих равных условиях повышение эффективности часто вызываемых функций может положительно сказаться на общей производительности.

Базовая стоимость вызовов функций

Есть ряд деталей, которые могут замедлить вызовы функций в C++ и которые, следовательно, образуют основу для оптимизации вызовов функций.

Аргументы функций

Помимо стоимости вычисления выражений аргументов, имеются затраты на копирование каждого значения аргумента в стек в памяти. Первые несколько аргументов (если они имеют небольшие размеры) могут быть эффективно переданы через регистры, но если аргументов много, по крайней мере некоторые из них передаются через стек.

Вызов функций-членов (в отличие от вызовов свободных функций)

Каждый вызов функции-члена имеет дополнительный скрытый аргумент — указатель `this` на экземпляр класса, через который вызывается функция-член, который должен быть записан в память в стеке вызовов или сохранен в регистре.

Вызов и возврат

Эти действия не добавляют ничего к полезной работе программы. Они представляют собой чисто накладные расходы, которых не было бы, если бы вызов функции был заменен ее телом. В действительности многие компиляторы вместо вызова встраивают в код тело функции, если функция невелика и если определение функции доступно в точке ее вызова. Если функция не может быть встроенной, расходы на ее вызов повышаются.

Вызов функции требует записи адреса выполнения в кадр стека в памяти для формирования адреса возврата из функции.

Возврат из функции требует чтения адреса возврата из стека и его загрузки в указатель выполняемой команды. При вызове функции и возврате из нее управление передается адресу в памяти, который может находиться далеко от текущего. Как отмечалось в разделе "Трудное принятие решений" главы 2, "Оптимизация, влияющая на поведение компьютера", компьютер выполняет последовательные инструкции очень эффективно. Однако после передачи управления в удаленное местоположение велики шансы замедления конвейера и/или отсутствия данных в кеше.

Стоимость виртуальных функций

В C++ программа может определить любую функцию-член как виртуальную. Производные классы могут определять функцию-член с такой же сигнатурой, которая *перекрывает* виртуальную функцию-член базового класса, предоставляя новое тело функции для использования при вызове виртуальной функции для экземпляра производного класса, даже если она вызывается посредством указателя или ссылки на тип базового класса. Программа выбирает функцию для вызова при разыменовании экземпляра класса. Таким образом, конкретная вызываемая перекрывающая функция определяется фактическим типом экземпляра класса во время выполнения.

Каждый экземпляр класса с виртуальными функциями-членами содержит неименованный указатель на *таблицу виртуальных функций* (которую часто сокращенно называют vtable), записи которой указывают на тела всех виртуальных функций, видимых в этом классе. Указатель на таблицу виртуальных функций обычно является первым полем экземпляра класса для снижения стоимости его разыменования.

Поскольку вызов виртуальной функции выбирает одно из нескольких тел функций, код вызова виртуальной функции разыменовывает указатель на экземпляр класса для получения указателя на таблицу виртуальных функций. Код для получения адреса выполняемой виртуальной функции индексирует таблицу (т.е. добавляет небольшое целочисленное смещение в таблице и разыменовывает соответствующий адрес). Таким образом, для каждого вызова виртуальной функции имеются две дополнительные загрузки из несмежной памяти, каждая с повышенной вероятностью отсутствия данных в кеше и замедления конвейера. Еще одной связанной с виртуальными функциями проблемой является то, что компилятору трудно их встроить. Компилятор может сделать это, только если он имеет доступ к телу функции и коду, который создает экземпляр (так что он может точно указать, какое тело виртуальной функции следует вызывать).

Вызовы функций-членов в производных классах

Когда один класс является производным от другого, от функции-члена производного класса может потребоваться выполнение дополнительной работы.

Виртуальные функции-члены, определенные в производном классе

Если первый базовый класс в корне иерархии наследования не имеет никаких виртуальных функций-членов, коду необходимо добавить смещение к указателю `this` на экземпляр класса, чтобы добраться до таблицы виртуальных функций производного класса, а затем индексировать запись в таблице, чтобы получить адрес выполняемой функции. Этот код содержит большее количество команд, и эти команды часто выполняются медленнее из-за дополнительных арифметических действий. Эта стоимость значительна для небольших встроенных процессоров, но на процессорах класса настольных компьютеров параллелизм на уровне инструкций скрывает бо́льшую часть стоимости этой дополнительной арифметики.

Вызовы функций-членов, определенных в производных классах с множественным наследованием

Коду необходимо добавить смещение к указателю `this` на экземпляр класса для того, чтобы получить указатель на экземпляр множественно унаследованного класса. Эта стоимость значительна для небольших встроенных процессоров, но на процессорах класса настольных компьютеров параллелизм на уровне инструкций скрывает бо́льшую часть стоимости этой дополнительной арифметики.

Вызовы виртуальных функций-членов, определенных в производных классах с множественным наследованием

Что касается вызовов виртуальных функций-членов в производном классе, если корневой базовый класс не имеет виртуальных функций-членов, коду необходимо добавить смещение к указателю `this` на экземпляр класса, чтобы добраться до таблицы виртуальных функций производного класса, а затем индексировать запись в таблице, чтобы получить адрес выполняемой функции. Коду также необходимо добавить потенциально различные смещения к указателю `this` на экземпляр класса для того, чтобы получить указатель на экземпляр производного класса. Эта стоимость значительна для небольших встроенных процессоров, но на процессорах класса настольных компьютеров параллелизм на уровне инструкций скрывает бо́льшую часть стоимости этой дополнительной арифметики.

Множественное виртуальное наследование

Код должен разыменовать таблицы в экземпляре класса, чтобы определить смещение, которое необходимо добавить к указателю на экземпляр класса для формирования указателя на экземпляр виртуально множественно унаследованного класса. Если вызываемая функция является виртуальной, такой вызов сопровождается описанными ранее дополнительными накладными расходами.

Стоимость указателей на функции

C++ предоставляет указатели на функции, так что код во время выполнения может явно выбрать, какое из нескольких тел функций будет выполнено в результате вызова функции с помощью такого указателя. Эти механизмы обладают, помимо базовой стоимости вызова функции и возврата из нее, дополнительной стоимостью.

Указатели на свободные функции и статические функции-члены

C++ позволяет программе определять переменные, которые являются указателями на функции. Указатели на функции позволяют программисту явным образом выбрать любую свободную функцию с определенной *сигнатурой* (состоящей из списка типов аргументов и возвращаемого типа) для вызова во время выполнения, при разыменовании указателя на функцию. Программа явно выбирает, какая функция будет вызвана посредством указателя на функцию, присваивая функцию указателю на функцию.

Код должен разыменовать указатель, чтобы получить выполняемый адрес функции. Маловероятно, чтобы компилятор мог встраивать такие функции.

Указатели на функции-члены

Объявление указателя на функцию-член идентифицирует как сигнатуру функции, так и класс в контексте, в котором интерпретируется вызов функции. Программа явно выбирает, какая функция будет вызвана посредством указателя на функцию-член, путем присвоения функции указателю на функцию.

Подведение итогов стоимости вызовов функций

Итак, самым дешевым оказывается вызов `void`-функции без аргументов в стиле C, не имеющий стоимости в случае встраивания, и требующий только два обращения к памяти плюс две нелокальные передачи управления в противном случае.

Наихудшим сценарием является вызов виртуальной функции, принадлежащей к производному классу, базовый класс которого не имеет виртуальных функций, и содержащейся в классе с виртуальным множественным наследованием. К счастью, такой вызов встречается крайне редко. В этом случае код должен разыменовать таблицу в экземпляре класса, чтобы определить смещение, которое надо добавить к указателю на экземпляр класса для создания указателя на виртуально множественно наследуемый экземпляр, а затем разыменовать этот экземпляр, чтобы получить адрес таблицы виртуальных функций и индексировать ее для получения адреса выполняемой функции.

На этом этапе читатель может быть шокирован как возможной дороговизной вызова функции, так и тем, насколько эффективно C++ реализует очень сложные возможности языка. Обе мысли правильно отражают реальность. Главное — понимать, что есть затраты на вызов функций и, таким образом, есть и возможности для оптимизации. Плохая новость заключается в том, что удаление одного доступа к непоследовательному местоположению в памяти дает не так уж много, если только эта функция не вызывается очень часто. Хорошей новостью является то, что профайлер будет указывать наиболее часто вызываемые функции, обеспечивая разработчика информацией о том, на какие функции следует обратить особое внимание.

Объявление коротких функций встраиваемыми

Эффективным способом устранения затрат на вызов функций является встраивание функций. Для встраивания функции компилятор должен иметь доступ к определению функции в точке, где ее вызов может быть заменен встраиванием тела функции. Функции, тела которых приводятся в теле определения класса, рассматриваются как неявно объявленные встраиваемыми. Функции, определенные вне определения класса, также могут быть объявлены встраиваемыми — явно, с помощью ключевого слова `inline`. Кроме того, компилятор может самостоятельно выбрать встраивание для коротких функций, если их определения находятся до первого использования в конкретной единице компиляции. Хотя стандарт C++ и говорит, что ключевое слово `inline` является только "подсказкой", а не обязательным требованием для компилятора, для коммерческого успеха компиляторы должны хорошо уметь встраивать функции.

Когда компилятор встраивает функцию, возникают дополнительные возможности улучшения кода, помимо удаления вызова и инструкции возврата. Некоторые арифметические операции могут быть выполнены во время компиляции; некоторые ветви могут быть устранены, если компилятор выясняет, что они не могут быть использованы при конкретных значениях аргументов. Таким образом, встраивание является средством удаления избыточных вычислений для повышения производительности путем выполнения вычислений во время компиляции.

Встраивание функций, вероятно, является самой мощной оптимизацией кода. Фактически основная разница в производительности между отладочной и производственной версиями в Visual Studio (или при использовании параметров командной строки `-d` или `-O` в GCC) главным образом обязана тому, что по умолчанию построение отладочной версии отключает встраивание функций.

Определение функций до их первого использования

Определение функции (предоставление тела функции) до точки первого вызова функции дает компилятору возможность оптимизировать вызов этой функции. Компилятор может самостоятельно прибегнуть к встраиванию функции, если при компиляции вызова функции ему доступно ее определение. Это возможно даже для виртуальных функций, если компилятор имеет доступ к телу функции и коду, который создает экземпляр переменной класса, указатель или ссылку, через которые вызывается эта виртуальная функция.

Устранение неиспользуемого полиморфизма

В C++ виртуальные функции-члены часто используются для реализации *полиморфизма времени выполнения*. Полиморфизм позволяет функции-члену выполнять одну из нескольких различных, но семантически связанных вещей, в зависимости от объекта, для которого вызывается эта функция-член.

Для реализации полиморфного поведения базовый класс определяет виртуальную функцию-член. Любые производные классы могут перекрыть поведение функции базового класса поведением, специализированным для производного класса. Все различные реализации связаны семантической концепцией, которая должна быть реализована по-разному для каждого в каждом производном классе.

Классическим примером полиморфизма является функция `Draw()`, определенная в иерархии классов, производных от `DrawableObject`, представляющих графические объекты для рисования на экране. Когда программа вызывает `drawObjPtr->Draw()`, реализация `Draw()` выбирается путем разыменования таблицы виртуальных функций в экземпляре, на который указывает `drawObjPtr`. Выбранная реализация `Draw()` рисует треугольник, если экземпляр класса является экземпляром `Triangle`, прямоугольник, если это экземпляр `Rectangle`, и т.д. Программа объявляет функцию `DrawableObject::Draw()` виртуальной, так что вызывается член `Draw()` соответствующего производного класса. Когда программа должна выбирать одну из нескольких реализаций во время выполнения, таблица виртуальных функций представляет собой очень быстрый механизм, несмотря на накладные расходы в виде двух дополнительных загрузок данных из памяти и связанное с ними замедление конвейера.

Тем не менее полиморфизм может иногда вызывать ненужные накладные расходы. Класс может первоначально быть разработан для облегчения создания иерархии производных классов, которая оказывается никогда не реализованной, или функции могут были объявлены виртуальными в ожидании полиморфного поведения, которое так и не реализуется. В предыдущем примере все рисуемые объекты могут в конечном итоге быть реализованы как список точек, связанных в определенном порядке, так что всегда можно использовать версию функции `Draw()` базового класса. Удаление спецификатора `virtual` в объявлении `Draw()` класса `DrawableObject` при отсутствии перекрытий ускоряет все вызовы `Draw()`.

> **Остановись и подумай**
>
> Существует определенная напряженность между желанием проектировщика сказать, что `DrawableObject` может быть корнем множества порожденных от `DrawableObject` классов, и желанием разработчика, занимающегося оптимизацией кода, повысить производительность на основе отсутствия перекрытий функции `Draw()`. Предположим, что эксперименты показали дороговизну функции-члена `Draw()`. В этом случае проектировщику, вероятно, придется уступить. Спецификатор `virtual` можно будет легко добавить позже, если это станет необходимо. Мудрые разработчики, занимающиеся оптимизацией кода, знают, что лучше не ослаблять дизайн без уважительной причины и не гоняться за каждой виртуальной функцией программы со скальпелем в руке.

Удаление неиспользуемых интерфейсов

В C++ виртуальные функции-члены могут использоваться для реализации *интерфейсов*. Интерфейс — это объявление общего набора функций, которые вместе описывают поведение объекта и которые могут быть реализованы по-разному при различных обстоятельствах. Базовый класс *определяет* интерфейс путем объявления набора чисто виртуальных функций (функция, объявленная с использованием модификатора `= 0`). C++ не позволяет создавать экземпляры такого базового класса. Производные же классы *реализуют* интерфейс путем перекрытия всех чисто виртуальных функций базового класса интерфейса. Одним из преимуществ идиомы интерфейса в C++ является то, что производный класс обязан реализовать все функции, объявленные в интерфейсе, иначе компилятор не позволит программе создавать экземпляры производного класса.

Например, разработчик может использовать класс интерфейса для отделения зависимости от операционной системы, в особенности если дизайн предусматривает реализацию программы для нескольких разных операционных систем. Класс для чтения и записи файлов может быть определен с помощью интерфейсного класса `File`,

который приведен ниже. `File` называется *абстрактным базовым классом*, поскольку его экземпляр не может быть создан:

```
// file.h - интерфейс
class File {
public:
    virtual ~File() {}
    virtual bool     Open(Path& p)  = 0;
    virtual bool     Close()        = 0;
    virtual int      GetChar()      = 0;
    virtual unsigned GetErrorCode() = 0;
};
```

Где-то еще в другом месте кода определен производный класс `WindowsFile`, который предоставляет перекрытия, реализующие эти функции в операционной системе Windows. Ключевое слово C++11 `override` является необязательным и сообщает компилятору, что это объявление предназначено для перекрытия объявления виртуальной функции базового класса. Если использован модификатор `override`, а в базовом классе нет объявления соответствующей виртуальной функции, компилятор выдаст предупреждающее сообщение:

```
// Windowsfile.h - интерфейс
# include "File.h"
class WindowsFile : public File { // Объявление в стиле C++11
public:
    ~File() {}
    bool     Open(Path& p)  override;
    bool     Close()        override;
    int      GetChar()      override;
    unsigned GetErrorCode() override;
};
```

В дополнение к заголовочному файлу имеется также файл `windowsfile.cpp`, содержащий реализации перекрытых функций для Windows:

```
// windowsfile.cpp - Реализация для Windows
# include "WindowsFile.h"
bool WindowsFile::Open(Path& p) {
    ...
}
bool WindowsFile::Close() {
    ...
}
...
```

Иногда программа определяет интерфейс, но предоставляет только одну реализацию. В этом случае стоимость вызовов виртуальных функций (особенно часто используется вызов `GetChar()`) может быть сэкономлена путем устранения интерфейса, удаления ключевого слова `virtual` из определения класса в `file.h` и обеспечения реализации функций-членов `File`.

Остановись и подумай

Как и в предыдущем разделе, имеется определенная напряженность между желанием проектировщика точно определить интерфейс (что является хорошим желанием) и желанием разработчика, занимающегося оптимизацией кода, повысить производительность (которая также имеет значение, если `GetChar()` была отмечена профайлером как горячая функция). Если программа достаточно зрелая, судить, может ли существовать иная реализация, должно быть проще. Это знание может предложить иное решение для оптимизации или сохранить оригинальный дизайн. Разработчик-оптимизатор, если он не является архитектором, первоначально определившим данный интерфейс, при внесении предложений об изменении должен быть готов к категорическому отказу. Со своей стороны могу посоветовать только одно — при таких кардинальных предложениях тщательно аргументировать их с результатами измерений в руках.

Выбор реализации интерфейса во время компоновки

Виртуальные функции позволяют работающей программе выбирать одну из нескольких реализаций. Интерфейсы позволяют разработчику указывать, какие функции должны быть закодированы в процессе разработки программы, чтобы сделать объект полезным. Проблемой реализации идиомы интерфейса с использованием виртуальных функций C++ является то, что виртуальные функции обеспечивают для задачи времени разработки решение времени выполнения, которое имеет соответствующую стоимость, влияющую на производительность программы.

В предыдущем разделе для изолирования зависимости от операционной системы был определен интерфейс, названный `File`. Этот интерфейс реализован в производном классе `WindowsFile`. Если программа перенесена на Linux, исходный код может в конечном итоге включать класс `LinuxFile`, производный от интерфейса `File`, но и `WindowsFile`, и `LinuxFile` никогда не будут создаваться в одной программе. Каждый из них выполняет низкоуровневые вызовы, которые осуществляются только в одной операционной системе. Накладные расходы вызовов виртуальных функций в этом случае не нужны. Кроме того, если программа читает большие файлы, функция-член `File::GetChar()` может оказаться достаточно горячей для оптимизации.

Если решение не должно приниматься во время выполнения, разработчик может использовать для выбора среди нескольких реализаций компоновщик. Вместо объявления интерфейса C++ заголовочный файл объявляет (но не реализует) функции-члены непосредственно, так же, как если бы они были функциями стандартной библиотеки:

```
// file.h - интерфейс
class File {
public:
    File();
    bool     Open(Path& p);
```

```
    bool     Close();
    int      GetChar();
    unsigned GetErrorCode();
};
```

Файл `windowsfile.cpp` содержит реализацию для Windows:

```
// windowsfile.cpp - реализация для Windows
# include "File.h"
bool File::Open(Path& p) {
    ...
}
bool File::Close() {
    ...
}
...
```

Аналогичный файл с именем `linuxfile.cpp` содержит реализацию для Linux. Файл проекта Visual Studio использует `windowsfile.cpp`. Файл `makefile` для Linux использует `linuxfile.cpp`. Решение реализуется с помощью списка аргументов, передаваемых компоновщику. Теперь вызов `GetChar()` будет настолько эффективен, насколько вообще может быть эффективен вызов функции. (Обратите внимание, что существуют и другие подходы к оптимизации вызовов наподобие `GetChar()`, включая инверсии цикла, описанные в разделе "Перемещение циклов в функции для снижения накладных расходов при вызовах" данной главы.)

Преимущество выбора реализации во время компоновки — в его обобщенности. Недостаток же этого метода заключается в том, что часть решения находится в `cpp`-файлах, а часть — в файле `makefile` или в файле проекта.

Выбор реализации интерфейса во время компиляции

В предыдущем разделе реализация абстракции `File` выбиралась компоновщиком. Это стало возможным потому, что реализация `File` зависела от операционной системы. Так как конкретный выполнимый файл программы может работать только в одной операционной системе, решение не обязано приниматься во время выполнения.

Если для двух различных реализаций `File` используется другой компилятор (скажем, Visual Studio для Windows и GCC для Linux), реализацию можно выбрать во время компиляции с помощью директив `#ifdef`. Заголовочный файл при этом остается прежним. На этот раз имеется единственный файл под названием `file.cpp`, а реализация выбирается препроцессором:

```
// file.cpp - реализация
# include "File.h"
# ifdef _WIN32
bool File::Open(Path& p) {
    ...
}
bool File::Close() {
    ...
}
...
# else // Linux
bool File::Open(Path& p) {
    ...
```

```
}
bool File::Close() {
    ...
}
...
# endif
```

Это решение требует макроса препроцессора, который может использоваться для выбора требуемой реализации. Некоторым разработчикам больше нравится данный подход, поскольку в нем принятие решения оказывается видимым в cpp-файле. Другие же находят наличие двух реализаций в одном файле слишком неорганизованным и не объектно-ориентированным.

Выбор реализации во время компиляции с помощью шаблонов

Специализации шаблонов C++ являются еще одним средством выбора реализации во время компиляции. Шаблоны позволяют разработчику создавать семейство классов с общим интерфейсом, но с поведением, которое зависит от параметра типа этого шаблона. Параметры шаблона могут быть любыми типами — классами, которые предоставляют собственный набор функций-членов, или фундаментальными типами с набором встроенных операторов. Таким образом, имеется два интерфейса: открытые члены класса шаблона и интерфейс, определяемый операторами и функциями параметров шаблона. Интерфейсы, определенные в абстрактных базовых классах, очень строгие, требующие от производных классов реализации всех чисто виртуальных функций абстрактного базового класса. Интерфейсы, определяемые шаблонами, менее строги. Должны быть определены только те функции в параметрах, которые фактически вызываются при конкретном использовании шаблона.

Это свойство шаблонов является "обоюдоострым": если разработчик забывает реализовать функцию интерфейса в конкретной специализации шаблона, компилятор не выдает сообщение об ошибке немедленно; но и разработчик может не реализовывать функции, которые не используются в данном контексте.

С точки зрения оптимизации наиболее важное различие между иерархиями полиморфных классов и экземплярами шаблонов заключается в том, что обычно во время компиляции доступен весь шаблон. В большинстве случаев использования C++ встраивает вызовы шаблонных функций, тем самым несколькими путями повышая производительность (о чем упоминалось в разделе "Объявление коротких функций встраиваемыми" данной главы).

Шаблонное программирование обеспечивает мощную оптимизацию. От разработчиков, не знакомых с шаблонами, требуются значительные умственные усилия, чтобы научиться эффективно использовать эту возможность C++.

Исключение применения идиомы PIMPL

Идиома *PIMPL* (Pointer to IMPLementation — указатель на реализацию) представляет собой идиому кодирования, используемую в качестве *брандмауэра компиляции*, механизма для предотвращения изменения в одном заголовочном файле от

инициированной перекомпиляции множества исходных файлов. Во времена отрочества C++ в 1990-е годы использование идиомы PIMPL было оправданным, поскольку время компиляции большой программы могло измеряться часами. Вот как это работает.

Представьте себе широко используемый класс `BigClass` (объявленный в примере 7.13), который имеет некоторые встраиваемые функции и реализуется с помощью классов `Foo`, `Bar` и `Baz`. Как правило, любые изменения в `bigclass.h`, `foo.h`, `bar.h` или `baz.h` (даже если таковые изменения составляют исправление одного символа в комментарии) будут вызывать перекомпиляцию множества файлов, включающих `bigclass.h`.

Пример 7.13. `bigclass.h` до реализации идиомы PIMPL

```
# include "foo.h"
# include "bar.h"
# include "baz.h"
class BigClass {
public:
    BigClass();
    void f1(int a)   { ... }
    void f2(float f) { ... }
    Foo foo_;
    Bar bar_;
    Baz baz_;
};
```

Для реализации идиомы PIMPL разработчик определяет новый класс (`Impl` в данном примере). Заголовочный файл `bigclass.h` изменяется так, как показано в примере 7.14.

Пример 7.14. `bigclass.h` после реализации идиомы PIMPL

```
class Impl;
class BigClass {
public:
    BigClass();
    void f1(int a);
    char f2(float f);
    Impl* impl;
};
```

C++ разрешает объявить указатель, который указывает на *неполный тип*, т.е. на объект, для которого еще нет определения. В данном случае таким неполным типом является `Impl`. Это работает, потому что все указатели имеют одинаковые размеры, так что компилятор знает, как зарезервировать память для указателя. После реализации PIMPL `BigClass` в видимом извне определении не зависит от `foo.h`, `bar.h` или `baz.h`. Полное определение `Impl` находится внутри файла `bigclass.cpp` (пример 7.15).

Пример 7.15. `bigclass.cpp` с определением `Impl`

```
# include "foo.h"
# include "bar.h"
# include "baz.h"
# include "bigclass.h"
```

```
class Impl {
    void g1(int a);
    void g2(float f);
    Foo foo_;
    Bar bar_;
    Baz baz_;
};
void Impl::g1(int a) {
    ...
}
char Impl::g2(float f) {
    ...
}
void BigClass::BigClass() {
    impl_ = new Impl;
}
void BigClass::f1(int a) {
    impl_->g1(a);
}
char BigClass::f2(float f) {
    return impl_->g2(f)
}
```

Во время компиляции после реализации PIMPL изменения в `foo.h`, `bar.h` или `baz.h`, или в реализации `Impl` приводят к перекомпиляции `bigclass.cpp`, но `bigclass.h` остается неизменным, ограничивая область перекомпиляции.

Во время выполнения ситуация иная. PIMPL ничего не добавляет к программе, за исключением задержек. Функции-члены `BigClass`, которые могли бы быть встроенными, в настоящее время требуют вызова функции-члена. Кроме того, каждая функция-член теперь вызывает функцию-член `Impl`. Проекты, использующие PIMPL, часто используют его во многих местах, создавая множество уровней вложенных вызовов функций. Кроме того, из-за дополнительных слоев вызовов функций отладка такого кода становится очень утомительной.

В 2016 году вряд ли есть необходимость в применении PIMPL. Время компиляции теперь составляет, наверное, порядка 1% от времени компиляции в 1990-е годы. Кроме того, даже в 1990-х годах для необходимости PIMPL `BigClass` должен был быть *очень большим* классом, зависящим от многих заголовочных файлов. Такие классы нарушают многие правила объектно-ориентированного программирования. Разбиение `BigClass` для предоставления более узконаправленных интерфейсов может быть столь же эффективным, как и реализация PIMPL.

Устранение вызовов кода в DLL

Вызовы кода из динамически компонуемых библиотек (DLL) в Windows осуществляются через указатель на функцию, который либо явно задан в программе, если библиотека DLL загружается по требованию, либо задан неявно при автоматической загрузке DLL при запуске. Linux также имеет динамические библиотеки с такой же реализацией.

Некоторые вызовы DLL необходимы. Например, приложение может использовать разработанные сторонними производителями подключаемые модули. Другие

причины для использования DLL не столь очевидны. Например, одна из причин, по которым используются библиотеки DLL, — это предоставление возможности обновлений и исправлений путем замены только библиотеки DLL. Опыт показывает, что исправления ошибок, как правило, выполняются пакетно, охватывая многие области программы одновременно. Это существенно снижает вероятность того, что все исправления ограничатся одним лишь файлом DLL.

Превращение DLL в объектный код и связывание библиотек в один выполняемый файл является еще одним способом повышения производительности вызовов функций.

Используйте статические функции-члены вместо функций-членов экземпляров

Каждый вызов функции-члена имеет дополнительный, неявный аргумент — указатель `this` на экземпляр класса, член которого вызывается. Доступ к членам-данным класса осуществляется через смещение относительно указателя `this`. Виртуальные функции-члены должны разыменовывать указатель `this`, чтобы получить указатель на таблицу виртуальных функций.

Иногда функция-член зависит только от своих аргументов, не обращаясь ни к каким членам-данным и не вызывая каких-либо иных функций-членов экземпляра. В таком случае указатель `this` в функции попросту не используется.

Такие функции-члены следует объявлять как `static`. Статические функции-члены не должны вычислять неявный указатель `this`. К ним можно обращаться с помощью обычных указателей на функции вместо более дорогих указателей на функции-члены (см. раздел "Стоимость указателей на функции" данной главы).

Перенесение виртуального деструктора в базовый класс

Деструктор любого класса, который будет иметь производные классы, должен быть объявлен виртуальным. Это необходимо, чтобы в выражении удаления производного класса с указателем на базовый класс были корректно вызваны как деструктор производного класса, так и деструктор базового класса.

Есть еще одна причина для предоставления некоторых функций базового класса любой иерархии как виртуальных: для гарантии включения указателя на таблицу виртуальных функций в базовом классе.

Базовый класс имеет особое положение в иерархии классов. Если базовый класс содержит объявление виртуальной функции-члена, указатель на таблицу виртуальных функций будет находиться с нулевым смещением в любом экземпляре класса, производном от этого базового класса. Если базовый класс объявляет переменные-члены и не объявляет ни одной виртуальной функции-члена, а некоторый производный класс объявляет таковую, то каждый вызов виртуальной функции-члена должен будет добавлять смещение к указателю `this` для получения адреса указателя таблицы виртуальных функций. Если хотя бы одна функция в базовом классе является виртуальной, смещение указателя на таблицу виртуальных функций будет нулевым, что приводит к более эффективному коду.

Отличным кандидатом для этой функции является деструктор. Он должен быть виртуальным в любом случае, если базовый класс будет иметь производные классы. Деструктор вызывается только один раз во время жизни экземпляра класса, поэтому стоимость его превращения в виртуальный является минимальной, за исключением очень уж небольших классов, которые создаются и уничтожаются в программе очень часто (такие маленькие классы очень редко используются в качестве базовых для наследования).

Это может показаться очень редким, не заслуживающим внимания случаем, но я работал над несколькими проектами, в которых важные иерархии классов имели базовый класс, содержащий счетчик ссылок, идентификатор транзакций или некоторые подобные переменные. Этот базовый класс имел мало информации о том, какие классы могут быть производными от него. Обычно первый производный класс в такой иерархии оказывался абстрактным базовым классом, объявляющим массу виртуальных функций-членов. Но базовому классу *было* точно известно, что все созданные экземпляры в конечном счете обязательно будут уничтожены.

Оптимизация выражений

Ниже уровня инструкций, на уровне арифметических выражений, включающих фундаментальные типы данных (целые числа, числа с плавающей точкой, указатели), имеется последняя возможность оптимизации. Если очень горячая функция состоит из единственного выражения, оно может оказаться единственной возможностью оптимизации.

Остановись и подумай

Современные компиляторы *очень* хороши в оптимизации выражений, включающих фундаментальные типы данных. Они настолько хороши, что их оптимизация варьируется от просто удивительной до доказуемо оптимальной. Единственный их недостаток — они не очень смелы. Они будут оптимизировать выражения только тогда, когда смогут доказать, что оптимизация не изменяет поведение программы.

Разработчики умнее компиляторов, хотя гораздо менее скрупулезны. Разработчики могут оптимизировать код в ситуациях, когда компилятор не в состоянии определить безопасность оптимизации, — потому что разработчики могут рассматривать *дизайн* и *назначение* функций, определенных в других модулях кода, не видимых компилятору.

Именно в этих относительно редких случаях разработчики могут превзойти компиляторы.

Оптимизация выражений может иметь значительное влияние на малых процессорах, которые выполняют по одной команде за раз. На процессорах

класса настольных компьютеров с многоэтапной конвейеризацией улучшение производительности хотя и обнаруживаемо, но невелико. Выражения — не место большой охоты в джунглях оптимизации. Занимайтесь ими только в тех редких случаях, когда необходимо выжать последние капли производительности в некотором раскаленном добела внутреннем цикле или функции.

Упрощение выражений

C++ вычисляет выражения в строгом порядке с учетом приоритета и ассоциативности их операторов. Выражение `a*b+a*c` вычисляется так, как если бы оно было записано как `((a*b)+(a*c))`, потому что правила приоритетов в C++ говорят, что умножение имеет более высокий приоритет, чем сложение. Компилятор C++ никогда не применяет дистрибутивность для того, чтобы вычислить это выражение более эффективно как `a*(b+c)`. Выражение `a+b+c` вычисляется так, как если бы оно было записано как `((a+b)+c)`, поскольку оператор `+` является левоассоциативным. Компилятор никогда не перепишет это выражение как `(a+(b+c))`, несмотря на то что математически для целых и действительных чисел эти записи эквивалентны.

Причина, по которой C++ не прикасается к выражениям, в том, что целочисленная модульная арифметика C++ не такая же, как математика целых чисел, а приблизительная арифметика типа `float` в C++ не такая же, как математика действительных чисел. C++ должен дать программисту возможность точно определить свои намерения, и если компилятор по ходу дела будет переупорядочивать выражения, то это может привести к переполнениям различных видов. Это означает, что за написание выражений в форме, которая использует наименьшее количество возможных операторов, отвечает разработчик.

Правило Горнера для вычисления многочленов служит примером оптимизации, состоящей в записи выражения в гораздо более эффективном виде. Хотя большинство разработчиков C++ не занимаются ежедневным вычислением полиномов, мы все знакомы с ними.

Полином $y = ax^3 + bx^2 + cx + d$ можно записать на C++ следующим образом:

```
y = a*x*x*x + b*x*x + c*x + d;
```

Эта инструкция требует шести умножений и трех сложений. Правило Горнера использует многократное применение дистрибутивности для перезаписи этого выражения в виде

```
y = (((a*x + b)*x) + c)*x + d;
```

Эта оптимизированная инструкция выполняет три умножения и три сложения. В общем случае правило Горнера упрощает выражение с $n(n-1)$ умножения до n, где n — степень полинома.

Поучительная история a/b*c

Причиной того, что C++ не переупорядочивает арифметические выражения, состоит в том, что это опасно. Численный анализ — это еще один вопрос, которому посвящены целые книги. Это один пример того, что может пойти не так.

Компилятор C++ будет вычислять выражение `a/b*c`, как если бы оно было записано как `((a/b)*c)`. Проблема с этим выражением заключается в том, что если `a`, `b` и `c` имеют целочисленные типы, то результат деления `a/b` не является точным. Таким образом, если `a = 2`, `b = 3` и `c = 10`, то `a/b*c` равно 2/3*10 = 0, в то время когда мы в действительности хотим получить 6. Проблема заключается в том, что неточность в `a/b` умножается на `c`, приводя к очень неправильному результату. Разработчик с хорошими математическими навыками может заменить это выражение на `a*c/b`, которое компилятор будет вычислять, как если бы оно было со скобками `((a*c)/b)`. Тогда результатом вычислений будет 2*10/3 = 6.

Проблема решена? На самом деле нет. Если вы первым выполняете умножение, то существует опасность переполнения. Если `a` = 86400 (количество секунд в дне), `b` = 90 000 (константа, используемая для видео выборки), а `c` = 1 000 000 (число микросекунд в секунде), то значение выражения `a*c` не поместится в 32-битном типе `unsigned int`. Исходное выражение, хотя и имеет значительные ошибки, все же оказывается менее неправильным, чем переписанное.

Разработчик единственный несет ответственность за то, что записанное выражение с аргументами ожидаемой величины будет давать правильный результат. Компилятор не может помочь ему в решении этой задачи, и именно поэтому он оставляет выражения нетронутыми.

Группирование констант

Одно из действий, которые компилятор *может* делать, — это вычисление константных выражений. Столкнувшись с выражением наподобие

```
seconds = 24 * 60 * 60 * days;
```

или

```
seconds = days * (24 * 60 * 60);
```

компилятор может вычислить константную часть выражения и получить эквивалент

```
seconds = 86400 * days;
```

Но если программист запишет

```
seconds = 24 * days * 60 * 60;
```

то компилятор должен будет выполнять умножения во время выполнения.

Всегда группируйте константные выражения с помощью круглых скобок, размещайте их в левой части выражения или, что еще лучше, вынесите их в инициализатор константной переменной или поместите их в `constexpr`-функцию (если ваш компилятор поддерживает C++11). Это позволит компилятору эффективно вычислить константное выражение во время компиляции.

Используйте менее дорогостоящие операторы

Вычисление одних арифметических операций оказывается менее дорогостоящим, чем других. Например, все используемые на сегодняшний день процессоры (за исключением самых маленьких) могут выполнять сдвиг или сложение за один внутренний такт. Некоторые специализированные цифровые процессоры в состоянии выполнять за один такт и умножение, но для персональных компьютеров умножение представляет собой итеративное вычисление, аналогичное процедуре умножения в столбик, которому учат школьников. Деление является еще более сложной итеративной процедурой. Эта иерархия стоимостей обеспечивает возможности для оптимизации.

Например, целочисленное выражение `x*8` можно вычислить более эффективно, как `x<<3`, поскольку 8 равно 2^3, а умножение на 2^k эквивалентно сдвигу влево на k бит. Любой мало-мальски стоящий компилятор уже выполняет такую оптимизацию *снижения стоимости*. Но что делать, если у нас имеется выражение `x*y` или `x*func()`? Во многих случаях компилятор не в состоянии определить, что `y` или `func()` всегда дает значение, которое является степенью двух. И здесь знания и мастерство программиста могут позволить ему превзойти компилятор.

Другой возможной оптимизацией является запись умножения как последовательности сдвигов и сложений. Например, целочисленное выражение `x*9` можно переписать как `x*8+x*1`, которое, в свою очередь, можно переписать как `(x<<3)+x`. Эта оптимизация наиболее эффективна, когда постоянный операнд не содержит много установленных битов, потому что каждый установленный бит увеличивает срок выполнения сдвига и сложения. Она эффективна на процессорах класса настольных компьютеров или телефонов с кешами команд и конвейеризацией и на очень малых процессорах, на которых "длинное" умножение представляет собой вызов подпрограммы. Данную оптимизацию, как и все прочие, обязательно следует протестировать, чтобы убедиться, что она улучшает производительность на конкретном процессоре (но, как правило, это так и есть).

Использование целочисленной арифметики вместо арифметики с плавающей точкой

Арифметика с плавающей точкой весьма дорогостоящая. Числа с плавающей точкой имеют сложное внутреннее представление с нормализованной целочисленной мантиссой, отдельным показателем степени и двумя знаками. Аппаратный блок, который реализует вычисления с плавающей точкой, на персональных компьютерах может занимать до 20% площади кристалла. Некоторые многоядерные процессоры разделяют один арифметический блок для работы с плавающей точкой, но при этом имеют несколько целочисленных арифметических блоков на ядро.

Целочисленные результаты можно вычислить по крайней мере в 10 раз быстрее, чем эквивалентные результаты с плавающей точкой, даже на процессорах с аппаратными устройствами для арифметики с плавающей точкой и даже когда результаты целочисленного деления округляются, а не усекаются. На небольших процессорах, на которых математика с плавающей точкой реализуется через библиотечные функции, целочисленная математика быстрее во много раз. И тем не менее легко найти разработчиков, которые используют вычисления с плавающей точкой для таких простых вещей, как округленное деление.

В примере 7.16 показан способ поиска округленного частного. Он преобразует целочисленные аргументы в значения с плавающей точкой, выполняет деление и округляет результат.

Пример 7.16. Целочисленное округление с плавающей точкой

```
unsigned q = (unsigned)round((double)n / (double)d));
```

Тестовый цикл, повторявший эту операцию 100 миллионов раз, занял на моем компьютере 3125 мс.

Чтобы получить округленное целочисленное частное, необходимо взглянуть на остаток от деления. Значение остатка находится в диапазоне от `0` до `d-1`, где `d` — знаменатель. Если остаток больше или равен половине знаменателя, частное должно быть округлено вверх до ближайшего целого числа. Формула для знаковых целых чисел лишь немного сложнее.

C++ предоставляет функцию `ldiv()` из библиотеки времени выполнения C, которая дает структуру, содержащую частное и остаток. В примере 7.17 представлена функция для округленного деления с использованием `ldiv()`.

Пример 7.17. Целочисленное округление с помощью `ldiv()`

```
inline unsigned div0(unsigned n, unsigned d) {
    auto r = ldiv(n, d);
    return (r.rem >= (d >> 1)) ? r.quot + 1 : r.quot;
}
```

Эта функция не идеальна. `ldiv()` ожидает аргументы типа `int` и жалуется на несоответствие значений со знаком и без. Она дает правильный результат, если оба аргумента положительны при рассмотрении их как значений типа `int`. Функция `div0()` проходит тестовый цикл за 435 мс, что в 7 раз быстрее, чем версия с плавающей точкой.

В примере 7.18 показана функция для вычисления округленного частного для двух аргументов типа `unsigned`.

Пример 7.18. Округленное целочисленное деление

```
inline unsigned div1(unsigned n, unsigned d) {
    unsigned q = n / d;
    unsigned r = n % d;
    return r >= (d >> 1) ? q + 1 : q;
}
```

Функция `div1()` вычисляет частное и остаток. Выражение `(d>>1)` представляет собой эффективную, со сниженной стоимостью запись `d/2`. Если остаток больше или равен половине знаменателя, частное округляется вверх. Ключом к успеху этой функции является оптимизация, выполняемая компилятором. У машины x86 команда процессора, которая делит два целых числа, дает и частное, и остаток. Компилятор Visual C++ достаточно умен для того, чтобы вызвать эту команду только один раз. Тот же тест, но с вызовом этой новой функции вместо использования математики с плавающей точкой занимает 135 мс, что оказывается быстрее в 23 раза.

В примере 7.19 показан еще один, даже более быстрый, способ округления `unsigned`.

Пример 7.19. Округленное целочисленное деление

```
inline unsigned div2(unsigned n, unsigned d) {
    return (n + (d >> 1)) / d;
}
```

Функция `div2()` добавляет половину знаменателя `d` к числителю `n` до деления. Слабым местом `div2()` является то, что для больших значений возможно переполнение при вычислении `n+(d>>1)`. Если разработчику известно об используемых значениях аргументов и они позволяют использовать данный метод, то `div2()` работает очень быстро, за 102 мс проходя тот же тест (почти в 31 раз быстрее, чем версия с плавающей точкой).

`double` может быть быстрее, чем `float`

На моем компьютере i7 PC с использованием Visual C++ вычисления с плавающей точкой с типом `double` быстрее, чем такие же вычисления с типом `float`. Сначала я покажу результаты, а затем объясню, с чем это может быть связано.

Следующий код итеративно вычисляет расстояние, пройденное падающим телом; типичные вычисления с плавающей точкой таковы:

```
float d, t, a = -9.8f, v0 = 0.0f, d0 = 100.0f;
for (t = 0.0; t < 3.01f; t += 0.1f) {
    d = a*t*t + v0*t + d0;
```

Десять миллионов итераций этого цикла выполняются за 1889 мс.

Тот же код, но с применением переменных и констант типа `double` имеет следующий вид:

```
double d, t, a = -9.8, v0 = 0.0, d0 = 100.0;
for (t = 0.0; t < 3.01; t += 0.1) {
    d = a*t*t + v0*t + d0;
```

Десять миллионов итераций этой версии цикла выполняются всего за 989 мс, так что версия с использованием `double` оказывается почти в два раза быстрее.

Почему это происходит? Visual C++ генерирует инструкции с плавающей точкой, которые используют старый стек регистров "сопроцессора x87". В этой схеме все вычисления с плавающей точкой выполняются в одном 80-битном формате. Тип с одинарной точностью `float` и тип с двойной точностью `double` удлиняются перед

перемещением в регистры сопроцессора. Преобразование `float` может занять больше времени, чем преобразование `double`.

Есть много способов компиляции операций с плавающей точкой. На платформе x86 использование регистров SSE позволяет выполнять вычисления с четырьмя разными размерами непосредственно. Компилятор, использующий команды SSE, может вести себя иначе, как и компилятор для процессора с архитектурой, отличной от x86.

Замена итеративных вычислений аналитическими выражениями

Что можно сказать о C++ и играх с битами? Нужна ли богатая коллекция арифметических и побитных логических операторов в C++ просто для игр с битами или же это необходимость, вызванная требованиями сегодняшнего дня, такими как обмен битной информацией с регистрами устройств или передача и получение сетевых пакетов?

Зачастую возникает необходимость в подсчете числа единичных битов, поиске старшего бита, определении четности битов в слове, определении, представляют ли биты слова степень двойки, и т.д. Большинство этих проблем имеют простые решения стоимостью O(n) с применением итераций по n битам слова. Могут быть и более сложные итеративные решения с лучшей производительностью. Но для ряда проблем имеются *аналитические решения* (или *решения в замкнутом виде*): вычисления с константным временем, которые дают решение без необходимости прибегать к итерациям.

Например, есть простой итерационный алгоритм, позволяющий определить, является ли целое число степенью двойки. Все такие значения имеют единственный установленный бит, так что одним из решений является подсчет количества установленных битов числа. Простейшая реализация этого алгоритма может выглядеть так, как показано в примере 7.20.

Пример 7.20. Итеративная проверка, является ли целое число степенью 2

```
inline bool is_power_2_iterative(unsigned n) {
    for (unsigned one_bits = 0; n != 0; n >>= 1)
        if ((n & 1) == 1)
            if (one_bits != 0)
                return false;
            else
                one_bits += 1;
    return true;
}
```

Тестовый цикл этого алгоритма выполняется за время 549 мс.

Эта задача имеет и аналитическое решение. Если x представляет собой n-ю степень двойки, то это значение имеет единственный установленный бит в n-й позиции (считая младший бит нулевым). Тогда x–1 представляет собой битную маску с установленными битами в позициях n–1,…,0, и значение x&(x–1) равно нулю. Если же x не является степенью двойки, у него имеются еще какие-то установленные биты, и x–1 обнуляет только младший установленный бит, так что значение x&(x–1) в этом случае не равно нулю.

Функция для определения, является *ли x* степенью двойки, с использованием аналитического решения показана в примере 7.21.

Пример 7.21. Проверка, является ли целое число степенью 2

```
inline bool is_power_2_closed(unsigned n) {
    return ((n != 0) && !(n & (n - 1)));
}
```

Тот же тестовый цикл, но с использованием данной функции, выполняется за 238 мс, т.е. в 2,3 раза быстрее. Есть еще более быстрые способы. Рик Риган (Rick Regan) рассмотрел 10 способов решения этой задачи и изучил их время работы (`http://bit.ly/regan-10`).

Приобретите книгу *Алгоритмические трюки для программистов*

В предыдущем разделе содержится рекомендация по эффективной работе с битами, но это — только образец, который призван заинтересовать вас этой темой. Существуют сотни маленьких хитростей для повышения эффективности арифметических операций.

Есть книга, которая должна быть в библиотеке каждого разработчика, интересующегося оптимизацией на уровне выражений: Генри С. Уоррен-мл. (Henry S. Warren, Jr.), *Hacker's Delight*[3], выдержавшая два издания. Если у вас есть хотя бы малейший интерес к эффективным выражениям, ощущение от чтения этой книги будет подобно ощущениям ребенка, открывшим свой первый конструктор. Уоррен поддерживает веб-сайт `http://hackersdelight.org/` с еще более интересными ссылками и дискуссиями.

Чтобы почувствовать вкус этой книги, вы можете ознакомится с меморандумом лаборатории искусственного интеллекта MIT, известным как HAKMEM[4] (`http://bit.ly/mithackmem`). HAKMEM является концептуальным предком упомянутой книги, переполненным всяческими хаками для работы с битами, собранными тогда, когда самые быстродействующие процессоры были в 10000 раз медленнее, чем ваш телефон.

Идиомы оптимизации потока управления

Как отмечалось в разделе "Трудное принятие решений" главы 2, "Оптимизация, влияющая на поведение компьютера", вычисления выполняются быстрее, чем поток управления, из-за замедлений конвейера, которые происходят, когда указатель

[3] Русский перевод: Генри С. Уоррен *Алгоритмические трюки для программистов. 2-е изд.* — М.: ООО "И.Д. Вильямс", 2014. — *Примеч. пер.*

[4] Beeler, Michael, Gosper, R. William, and Schroeppel, Rich, *"HAKMEM," Memo 239*, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, 1972.

команд должен быть обновлен некоторым непоследовательным адресом. Компилятор C++ выполняет огромную работу, пытаясь уменьшить число обновлений указателя команд. Имеет смысл ознакомиться с тем, что же именно компилятор делает для того, чтобы создать наиболее быстрый код, какой только возможно.

Применение `switch` вместо `if-elseif-else`

Поток управления в инструкциях `if-elseif-else` является линейным: вычисляется условие `if` и, если оно истинно, выполняется первый блок. В противном случае продолжается вычисление условий каждого `else if` и выполнение первого же блока, условие которого имеет значение `true`. Проверка переменной на равенство для каждого из n значений приводит к последовательности `if-then-elseif` с n блоками. Если все возможные значения равновероятны, такая последовательность `if-then-elseif` выполняет $O(n)$ сравнений. При очень частом выполнении (например, при диспетчеризации событий) стоимость такого кода возрастает многократно.

Инструкция `switch` также сравнивает переменную с каждым из n значений, но оператор `switch`, сравнивающий значение с рядом констант, позволяет компилятору выполнять ряд полезных оптимизаций.

В часто встречающемся случае, когда требуется проверка на равенство константам, взятым из множества последовательных или почти последовательных значений, инструкция `switch` компилируется в таблицу переходов, индексируемую тестируемым значением или выражением, производным от него. Инструкция `switch` выполняет одну операцию индексации и переходит по указанному в таблице адресу. Стоимость сравнения равна O(1), независимо от количества констант для проверки. Такие переходы в таблице не обязаны находиться в последовательном порядке; компилятор может по-своему сортировать таблицу переходов.

Когда константы образуют последовательность с большими разрывами, таблица переходов становится слишком большой. Компилятор по-прежнему может отсортировать тестируемые константы и сгенерировать код, который выполняет бинарный поиск. Для инструкции `switch`, выполняющей сравнение с n значениями, наихудшая стоимость поиска составляет $O(log_2 n)$. В любом случае компилятор никогда не сгенерирует для инструкции `switch` код, более медленный, чем для эквивалентной конструкции `if-else`.

Иногда вероятность одной ветви `if-elseif-else` оказывается гораздо выше прочих. В этом случае амортизированная производительность инструкции `if` может приближаться к константе, если первым проверяется самый вероятный случай.

Применение виртуальных функций вместо `switch` или `if`

Если до появления C++ разработчик хотел получить в программе полиморфное поведение, его приходилось кодировать с помощью типов `struct` или `union` с переменной-селектором, которая сообщает, что именно представляет конкретный тип `struct` или `union`. Программа при этом оказывалась переполненной кодом, выглядящим следующим образом:

```
if (p->animalType == TIGER) {
    tiger_pounce(p->tiger);
}
else if (p->animalType == RABBIT) {
    rabit_hop(p->rabbit);
}
else if (...)
```

Опытные разработчики знают, что это иллюстрация с плаката для начинающих объектно-ориентированных программистов "Так программировать нельзя!" Но начинающим разработчикам требуется некоторое время для укоренения объектно-ориентированного мышления. Я встречал много промышленного кода C++, содержащего такие перлы:

```
Animal::move() {
    if (this->animalType == TIGER) {
        pounce();
    }
    else if (this->animalType == RABBIT) {
        hop();
    }
    else if (...)
    ...
}
```

С точки зрения оптимизации основная проблема этого кода — в использовании блока инструкций `if` для выбора типа производного объекта. Классы C++ *уже содержат* необходимый для этого механизм: виртуальные функции-члены и указатель на таблицу виртуальных функций в качестве селектора.

Вызов виртуальной функции обращается по индексу к таблице виртуальных функций для получения адреса тела виртуальной функции. Это всегда операция с константным временем выполнения. Таким образом, виртуальная функция-член `move()` в базовом классе перекрывается в производных классах, представляющих каждое животное с соответствующим кодом для передвижения — для прыжков, плавания и т.д.

Используйте обработку исключений без стоимости

Обработка исключений является одной из тех оптимизаций, которые лучше применять во время проектирования. Дизайн метода распространения информации об ошибке волной проходит через каждую строку программы, так что модернизация программы для использования обработки исключений может оказаться слишком дорогой. Несмотря на это, использование обработки исключений ведет к программам, которые быстрее при обычном выполнении и лучше ведут себя при ошибках.

Некоторые разработчики на C++ относятся к обработке исключений с подозрением. Существует общепринятое мнение, что обработка исключений делает программы больше и медленнее и что, отключая обработку исключений с помощью параметра компилятора, мы тем самым оптимизируем программу.

Истина оказывается более сложной. Да, если программа не использует обработку исключений, то, отключая эту возможность с помощью параметра компилятора, можно сделать программу меньше и немного быстрее. Джефф Прешинг (Jeff Preshing) оценил это изменение стоимости как величину между 1,4 и 4% в своем блоге (`http://bit.ly/preshing`). Однако остается неясным, насколько хорошо будет работать полученная в результате программа. Все контейнеры стандартной библиотеки C++ используют выражения `new`, которые могут генерировать исключения. Многие другие библиотеки, включая потоки ввода-вывода и библиотеки для использования параллелизма (рассматриваемые в главе 12, "Оптимизация параллельности", данной книги), генерируют исключения. Генерирует исключение оператор `dynamic_cast`. При выключенной обработке исключений непонятно, что будет происходить, когда программа обнаружит ситуацию, в которой генерируется исключение.

Если программа не генерирует исключений, она может полностью игнорировать коды ошибок. В этом случае разработчик получает то, что заслуживает. В противном случае программа должна терпеливо и тщательно передавать коды ошибок вверх через много вызовов функций, переводя коды ошибок из одного набора кодов в другой на границах библиотек и по ходу дела освобождая все ресурсы. Она должна делать это независимо от того, как завершается операция — успешно или с ошибкой.

При использовании исключений некоторая часть стоимости обработки ошибок перемещается из удачного пути обычного выполнения программы в путь обработки ошибок. Кроме того, компилятор автоматически освобождает ресурсы путем вызова деструкторов всех автоматических переменных на пути между генерацией исключения и блоком `try/catch`, который перехватывает и обрабатывает это исключение. Это упрощает логику пути удачного выполнения программы с соответствующим влиянием на производительность.

В ранние дни C++ каждый кадр стека содержал контекст исключения: указатель на список объектов, которые были сконструированы и, соответственно, должны быть уничтожены, если сгенерированное исключение проходило через этот кадр стека. Этот контекст динамически обновлялся во время выполнения программы. Это было нежелательно, поскольку добавляло стоимость времени выполнения для удачного пути. Возможно, источник легенды о высокой стоимости обработки исключений связан именно с этим. Более новая реализация отображает объекты, которые должны быть уничтожены, на диапазон значений указателя команд. Этот механизм не имеет стоимости времени выполнения, если только не генерируется исключение, поэтому стоимость исключений очень невысока. Visual Studio использует этот более новый механизм без стоимости времени выполнения для своих 64-битных сборок, а старый механизм — 32-битных сборок. Используемый механизм может быть выбран с помощью переключателя компилятора в Clang.

Не используйте спецификации исключений

Спецификации исключений добавляются к объявлениям функций, указывая, какие именно исключения может вызвать эта функция. Функции без спецификации исключений могут генерировать исключения без проблем. Функция же со

спецификацией исключений может генерировать только исключения, перечисленные в спецификации. Если она вызывает любое иное исключение, программа немедленно и безоговорочно прекращает работу с помощью вызова `terminate()`.

Имеются две проблемы, связанные со спецификациями исключений. Одна проблема заключается в том, что разработчикам трудно выяснить, какие именно исключения могут генерироваться вызываемой функцией, особенно в незнакомой библиотеке. Это делает программы с использованием спецификаций исключений достаточно "хрупкими" и склонными к неожиданным сбоям.

Вторая проблема заключается в том, что спецификации исключений отрицательно влияют на производительность. Сгенерированные исключения должны проверяться, как если бы каждый вызов функции со спецификацией исключений входил в новый блок `try/catch`.

Традиционные спецификации исключений рассматриваются в стандарте C++11 как устаревшие.

В стандарт C++11 введена новая спецификация исключений — `noexcept`. Объявление функции как `noexcept` сообщает компилятору, что данная функция не может сгенерировать исключение. Если функция все же его генерирует, то, как и для спецификации `throw()`, вызывается функция `terminate()`. Разница заключается в том, что для реализации семантики перемещения компилятор требует, чтобы определенные перемещающие конструкторы и перемещающие операторы были объявлены как `noexcept` (семантика перемещений рассматривается в разделе "Реализация семантики перемещения" главы 6, "Оптимизация переменных в динамической памяти"). Спецификация этих функций как `noexcept` служит объявлением, что семантика перемещения более важна, чем строгая гарантия безопасности исключений для определенных объектов. Да, я понимаю, что это звучит крайне запутанно...

Резюме

- *Оптимизация на уровне инструкций не обеспечивает достаточного уровня повышения производительности, чтобы оправдать ее применение, если только не существует факторов, которые увеличивают стоимость этих инструкций.*
- *Стоимость инструкций в циклах умножается на количество итераций цикла.*
- *Стоимость инструкций в функциях умножается на количество вызовов функции.*
- *Стоимость часто используемой идиомы умножается на количество применений этой идиомы.*
- *Некоторые инструкции C++ (присваивания, инициализация, вычисление аргументов функции) содержат скрытые вызовы функций.*
- *Вызовы функций операционной системы имеют высокую стоимость.*
- *Эффективным способом снижения накладных расходов на вызов функции является встраивание функции.*

- *В настоящее время в идиоме PIMPL практически нет никакой необходимости. Время компиляции сегодня на пару порядков меньше, чем во времена изобретения идиомы PIMPL.*
- *Арифметика с использованием* `double` *может быть быстрее, чем с использованием* `float`.

ГЛАВА 8

Использование лучших библиотек

Великой библиотеке никто не придает значения, потому что она всегда рядом, и в ней всегда есть то, что необходимо людям.

— Вики Майрон (Vicki Myron), автор книги *Дьюи. Кот из библиотеки, который потряс весь мир* и библиотекарь в г. Спенсер, штат Айова

Библиотеки — одна из центральных областей в процессе оптимизации. Библиотеки предоставляют примитивы, из которых создаются программы. Библиотечные функции и классы часто используются в глубоко вложенных циклах и, таким образом, могут быть очень "горячими". Библиотеки, предоставляемые компилятором или операционной системой, требуют особого внимания команды программистов, чтобы гарантировать их эффективное использование. Библиотеки, специфичные для конкретных проектов, заслуживают тщательного проектирования, гарантирующего возможность их эффективного применения.

В этой главе сначала рассматриваются вопросы использования стандартной библиотеки C++, а затем — вопросы разработки пользовательских библиотек, связанные с оптимизацией.

Большая часть данной книги посвящена настройке функций для повышения их производительности. Данная же глава предлагает вместо этого советы проектировщикам, заинтересованным в получении высокой производительности с самого начала, основанные на моем личном опыте. Здесь представлена информация о том, как хорошие методы проектирования C++ способствуют хорошей производительности.

Оптимизация использования стандартной библиотеки

C++ предоставляет компактную стандартную библиотеку для следующих общих предназначений.

- Определение поведения, зависящего от реализации, наподобие наибольшего и наименьшего значений каждого числового типа.
- Функции, которые лучше всего писать не на C++, такие как `strcpy()` и `memmove()`.

- Простые в использовании, но сложные в написании и проверке переносимые трансцендентные математические функции, такие как синус и косинус, логарифм и возведение в степень, случайные числа и т.д.
- Переносимые обобщенные структуры данных, такие как строки, списки и таблицы, которые не зависят от операционной системы (за исключением выделения памяти).
- Переносимые обобщенные алгоритмы поиска, сортировки и преобразования данных.
- Функции, которые подключаются к базовым службам операционной системы способом, не зависящим от операционной системы, для выполнения таких задач, как выделение памяти, работа с потоками, хронометраж и потоковые операции ввода-вывода. Сюда для совместимости включается унаследованная от языка программирования C библиотека функций.

Большая часть стандартной библиотеки C++ состоит из шаблонов классов и функций, которые производят чрезвычайно эффективный код.

Философия стандартной библиотеки C++

В соответствии со своей миссией языка системного программирования язык C++ предоставляет несколько спартанскую стандартную библиотеку. Стандартная библиотека предназначена быть простой, обобщенной и быстрой. Философски говоря, функции и классы входят в стандартную библиотеку C++ либо потому, что не могут быть предоставлены другим способом, либо потому, что очень широко используются повторно в нескольких операционных системах.

Преимущества подхода C++ включают то, что программы на C++ могут работать на аппаратном оборудовании без операционной системы вовсе и что программисты могут при необходимости выбрать специализированную библиотеку с учетом особенностей конкретной операционной системы или использовать кроссплатформенную библиотеку, когда целью является независимость от платформы.

В противоположность этому некоторые языки программирования, включая C# и Java, предоставляют обширные стандартные библиотеки, которые включают каркасы оконного пользовательского интерфейса, веб-серверы, сетевые сокеты и другие крупные подсистемы. Преимуществом такого всеобъемлющего подхода является то, что разработчикам нужно выучить лишь один набор библиотек, чтобы эффективно работать на любой поддерживаемой платформе. Но такие библиотеки предъявляют много требований к операционным системам, на которых они могут работать. Библиотеки, предоставляемые с языком программирования, представляют собой наименьший общий знаменатель, менее мощный, чем сетевые возможности или возможности оконной системы любой конкретной операционной системы. Таким образом, они могут несколько ограничивать программистов, привыкших к возможностям конкретной операционной системы.

Вопросы применения стандартной библиотеки C++

Хотя эта глава, в первую очередь, адаптирована для стандартной библиотеки C++, она в равной степени относится к стандартным библиотекам Linux, библиотеке POSIX и любой другой широко используемой кроссплатформенной библиотеке. Вопросы использования библиотеки включают следующее.

Реализации стандартной библиотеки имеют ошибки

Хотя ошибки являются неизбежной частью разработки программного обеспечения, даже опытные разработчики могут не обнаружить ошибки в коде стандартной библиотеки. Таким образом, они могут поверить, что различные реализации стандартной библиотеки очень надежны. Я приношу извинения за то, что эта вера лопнет, как мыльный пузырь. При написании этой книги я наткнулся на несколько ошибок стандартной библиотеки.

Вполне вероятно, что классическая программа "Hello, World!" будет работать так, как ожидалось. Однако оптимизация заводит разработчика в такие темные углы и закоулки стандартной библиотеки, что, скорее всего, в них таятся ошибки. Разработчик должен быть готов время от времени сталкиваться с разочарованиями, когда выясняется, что некоторая перспективная оптимизация не может быть реализована или что оптимизация, которая отлично работает с одним компилятором, выдает сообщение об ошибке на другом.

Читатель вправе спросить "Как же могло случиться, что тридцатилетний код по-прежнему содержит ошибки?" Прежде всего, нужно учитывать, что стандартная библиотека все это время развивается. Определение библиотеки и реализующий ее код всегда в работе. Это далеко не код тридцатилетней давности. Стандартная библиотека поддерживается отдельно от компилятора, который также содержит ошибки. В случае GCC разработчиками библиотеки являются добровольцы. Для стандартной библиотеки Microsoft Visual C++ приобретаются компоненты сторонних производителей, которые имеют собственный график выпуска, отличающийся и от цикла выпуска Visual C++, и от цикла выпуска стандартов C++. Изменение требований стандарта, отсутствие единой ответственности, вопросы графиков выпуска и сложность стандартной библиотеки — все это оказывает неизбежное влияние на качество. На самом деле гораздо более интересным является то, что в реализации стандартной библиотеки так мало ошибок.

Реализации стандартной библиотеки могут не соответствовать стандартам C++

Вероятно, не существует такой вещи, как "соответствующая стандарту реализация". В реальном мире поставщики компиляторов рассматривают свои инструменты как таковые, если они обеспечивают достаточно большое подмножество соответствующего стандарта C++, включая наиболее важные его возможности.

Что касается библиотек, то они выпускаются по другому плану, отличному от плана выпуска компилятора, а компилятор — по плану, отличному от плана выпуска стандарта C++. Реализация стандартной библиотеки может обгонять (или отставать) компилятор в соответствии стандарту. Даже различные части библиотеки могут следовать стандарту в различной степени. Для разработчиков, заинтересованных в оптимизации, изменения в стандарте C++, такие, например, как изменение оптимальных точек вставки для новых записей отображения (см. раздел "Вставка и удаление в std::map" главы 10, "Оптимизация структур данных"), означает, что поведение некоторых функций может оказаться сюрпризом для пользователя, так как нет способа документировать или определить версию стандарта, которой соответствует определенная библиотека.

Некоторые библиотеки ограничены компилятором. Например, библиотека не может реализовывать семантику перемещения, пока компилятор не будет ее поддерживать. Несовершенная поддержка компилятором той или иной возможности может привести к ограничениям стандартной библиотеки классов, поставляемой с компилятором. Иногда попытки использовать некоторую возможность языка приводят к ошибкам компилятора; при этом разработчику невозможно определить, находится ли проблема в компиляторе или в реализации стандартной библиотеки.

Разработчик, занимающийся оптимизацией, и очень хорошо знакомый со стандартом, может быть очень разочарован, когда нечасто используемая возможность оказывается не реализованной используемым им компилятором.

Для разработчиков стандартных библиотек производительность — не самое главное

Хотя производительность для разработчиков C++ очень важна, она не обязательно является наиболее важным фактором для разработчиков стандартной библиотеки. Важное значение имеет охват стандартной библиотеки, особенно если он позволяет соответствовать списку функций последнего стандарта C++. Важны простота и удобство обслуживания, поскольку библиотека представляет собой долгоживущий проект. Значение имеет и переносимость, если реализация библиотеки должна поддерживать несколько компиляторов. Так что вопросы производительности могут занять подчиненное место по отношению к любому из этих важных факторов.

Путь от вызова функции стандартной библиотеки к соответствующей функции операционной системы может быть долгим и извилистым. Однажды я отследил вызов `fopen()` через десяток функций, занимавшихся преобразованием аргументов, до того, как управление получила функция Windows `OpenFile()`, которая в конечном итоге запросила открытие файла у операционной системы. Похоже, основной целью было сведение количества строк кода к минимуму, но эти многие слои вызовов функций никак не способствовали повышению производительности.

Реализации библиотек могут препятствовать попыткам оптимизации

Библиотека AIO в Linux (не стандартная библиотека C++, но очень полезная для разработчика в области оптимизации) призвана обеспечить очень эффективный, асинхронный интерфейс без копирования для чтения файлов. Проблема заключается в том, что AIO требует ядро определенной версии. До тех пор, пока достаточно большая часть дистрибутивов Linux не выполнит обновление ядра, AIO кодируется с использованием старых, более медленных вызовов ввода-вывода. Разработчик может использовать вызовы AIO, но не получить ее производительность.

Не все части стандарта C++ одинаково полезны

Некоторые возможности C++, такие как сложные иерархии исключений, `vector<bool>` и аллокаторы стандартной библиотеки, были добавлены к стандарту до тщательной проверки практическим использованием. Эти возможности делают кодирование не проще, а сложнее. К счастью, комитет по стандартизации, как представляется, получил прививку от ранее проявлявшегося энтузиазма при введении в стандарт новых непроверенных возможностей. Теперь предлагаемые возможности стандартной библиотеки в течение нескольких лет вызревают в библиотеке Boost (`http://www.boost.org`), прежде чем выносятся на рассмотрение комитетом по стандартизации C++.

Стандартная библиотека не столь эффективна, как лучшие функции операционной системы

Стандартная библиотека не предоставляет возможности, доступные в некоторых операционных системах, такие, например, как асинхронный файловый ввод-вывод. Обычно разработчик в поисках оптимизации может зайти не дальше, чем ему позволяет стандартная библиотека. Для получения последних капель производительности он должен опуститься до вызовов операционной системы, принося переносимость в жертву на алтаре богов скорости.

Оптимизация существующих библиотек

Оптимизация существующей библиотеки — это как разминирование минного поля. Она возможна. Она может оказаться необходимой. Но это сложная работа, которая требует терпения и внимания к деталям, иначе — *БАБАХ*!

Легче всего оптимизировать библиотеку, которая хорошо разработана, с хорошим разделением вопросов и хорошими тестами. К сожалению, такая библиотека, скорее всего, была оптимизирована до нас. Реальность такова, что если вас попросили оптимизировать библиотеку, то, вероятно, ее функциональность в слишком большом беспорядке, а ее функции или классы делают слишком много или слишком мало.

Внесение изменений в библиотеку добавляет риск, связанный с тем, что некоторая другая программа может полагаться на непреднамеренное или недокументированное поведение существующей реализации. Хотя более быстрая работа функции сама по себе не должна вызывать проблемы, некоторые сопутствующие изменения в ее поведении могут стать источником больших проблем.

Внесение изменений в библиотеку с открытым исходным кодом вносит потенциальную несовместимость версии вашего проекта с основным репозиторием. Это становится проблемой, когда впоследствии открытая библиотека пересматривается для исправления ошибок или добавления функциональности. Либо ваши изменения должны быть включены в измененные библиотеки, либо они будут потеряны в процессе обновления, либо ваша измененная библиотека становится для вашей программы окончательным вариантом и не получает преимуществ от изменений, вносимых другими авторами. Таким образом, при выборе открытой библиотеки следует убедиться, что ее сообщество приветствует ваши изменения или что она является очень зрелой и стабильной.

Тем не менее это занятие не полностью безнадежно. В последующих разделах будут рассмотрены некоторые правила работы с существующими библиотеками.

Изменения должны быть небольшими

Наилучший совет для оптимизатора библиотеки — *изменяйте как можно меньше*. Не добавляйте и не удаляйте функциональность из классов и функций и не изменяйте сигнатуры функций. Эти изменения почти гарантировано нарушат совместимость между измененной библиотекой и программами, использующими ее.

Еще одна причина ограничиться минимальным количеством изменений — это ограничение части кода библиотеки, в котором необходимо досконально разбираться.

Из истории оптимизационных войн

Однажды я работал на компанию, которая продавала и поддерживала промышленную версию OpenSSH (которая основана на программе, разработанной Тату Илоненом (Tatu Ylonen), исследователем из Хельсинкского технологического университета для личного пользования в 1995 году). Будучи новичком в мире открытого исходного кода, я заметил, что код был написан не очень опытным разработчиком и, таким образом, не был столь аккуратным, каким ему следовало бы быть. Я внес некоторые существенные изменения, чтобы сделать код более легко поддерживаемым. По крайней мере, я так считал.

И хотя мой код отличался хорошим стилем, я узнал, что он так и не был принят.

В ретроспективе причина должна была очевидной. Мои изменения сделали наш код слишком отличающимся от исходного открытого варианта. В процессе работы мы постоянно использовали информацию от сообщества о важных, связанных с безопасностью ошибках, исправления которых автоматически применялись к неизмененному коду, но для моего сильно измененного кода их реализация оказывалась неэкономной. Я мог бы предложить свои изменения для включения в версию с открытым исходным кодом, но сообщество безопасности крайне консервативно относится к любым возможным изменениям кода. И это правильно.

Мне надо было изменить только тот код, который влиял на повышение безопасности, о котором просили наши пользователи, и только в минимально возможной степени. Рефакторинг никогда не рассматривался даже как вариант, хотя код был так плох, что, казалось, он просто просит о переделке.

Добавление функций, а не изменение функциональности

Ярким пятном в мрачном мире оптимизации библиотек является то, что новые функции и классы могут быть добавлены в библиотеку относительно безопасно. Конечно, существует риск, что будущая версия библиотеки определит класс или функцию с тем же именем, что и у одного из ваших дополнений, но этот риск можно снизить с помощью управляемого выбора названий, а иногда ситуация может быть исправлена с помощью применения макросов.

Вот некоторые достаточно безопасные обновления, которые могут улучшить производительность существующих библиотек.

- Добавление функций, которые, отражая соответствующую идиому, используемую в вашем коде, переносят циклы в библиотеку.
- Реализация семантики перемещения в старых библиотеках путем добавления перегрузок функций, принимающих rvalue-ссылки. (См. более подробную информацию о семантике перемещения в разделе "Реализация семантики перемещения" главы 6, "Оптимизация переменных в динамической памяти".)

Проектирование оптимизированных библиотек

Столкнувшись с плохо спроектированной библиотекой, разработчик мало что может сделать в плане ее оптимизации. Начиная "с нуля", разработчик имеет куда более широкие возможности использовать передовой опыт и избежать ловушек.

Этот раздел не предлагает конкретные рекомендации по способам достижения каждой из рассматриваемых целей в конкретной библиотеке, а только отмечает, что наилучшие и наиболее полезные библиотеки, как правило, воплощают описанные добродетели. Если конкретная библиотека оказывается далекой от идеала, возможно, не помешает пересмотр принципов ее проектирования.

Кодировать на скорую руку — обречь себя на долгую муку

> *Жениться на скорую руку — обречь себя на долгую муку.*
> *(Marry in haste, repent at leisure.*
>
> — Сэмюэл Джонсон (Samuel Johnson) (1709–1784), английский критик, лексикограф, поэт)

Стабильность интерфейса является основным требованием к библиотеке. Библиотека, разработанная в спешке, или собранная в библиотеку куча самостоятельно написанных функций не будет подчиняться единому соглашению о вызовах и возвращаемых значениях, обладать единым поведением в отношении распределения памяти

и вряд ли будет отвечать требованиям эффективности. Необходимость сделать случайно собранную библиотеку последовательной и согласованной будет возникать практически сразу же; однако для исправления всех функций в библиотеке требуется достаточно продолжительное время, которого может просто не быть у разработчиков.

Проектирование оптимизированной библиотеки подобно проектированию другого кода C++, но с более высокими ставками. Библиотеки по определению предназначены для широкого использования. Любые недостатки в проектировании, реализации или производительности библиотеки будут общими для всех пользователей. Применение случайных практик кодирования, которые могут быть терпимы в некритичном коде, становятся куда более проблематичными при разработке библиотек. В разработке критического кода наподобие библиотек находят свое применение проверенные методы разработки старой школы, включающие первоначальные спецификации, документацию и модульное тестирование.

Совет от профессионала: важность контрольных примеров

Контрольные примеры имеют важное значение для любого программного обеспечения. Они помогают проверить правильность исходного дизайна и уменьшить вероятность того, что изменения, внесенные во время оптимизации, повлияют на правильность программы. Неудивительно, что они приобретают еще большее значение для кодирования библиотек, где ставки куда выше.

Контрольные примеры помогают обнаружить зависимости и взаимосвязи, допущенные при проектировании библиотеки. Функции хорошо продуманных библиотек можно тестировать отдельно от других. Если до тестирования целевой функции необходимо создание множества объектов, то это сигнал разработчику о том, что между компонентами библиотеки существует слишком сильная связь.

Контрольные примеры помогают разработчику практиковаться в использовании библиотеки. Без такой практики даже опытные проектировщики могут легко оставить без внимания важные функции. Контрольные примеры облегчают обнаружение проблем в проекте на ранней стадии, когда существенные изменения интерфейса библиотеки не являются столь болезненными. С помощью библиотеки проектировщику легче выявить используемые идиомы и закодировать их таким образом, чтобы это привело к более эффективным интерфейсам функций.

Контрольные примеры неплохо подходят и для хронометража. Хронометраж позволяет убедиться, что все предлагаемые оптимизации повышают производительность на самом деле. Хронометраж может быть добавлен к другим контрольным примерам, чтобы можно было убедиться, что вносимые изменения не снижают производительности.

При разработке библиотек скупость является добродетелью

Для тех немногих читателей, которые не используют это слово в своей повседневной жизни, словарь определяет *скупость* как *качество быть осторожным с деньгами или ресурсами; экономным в использовании средств для достижения цели*. Читатели могли слышать о скупости в программировании как о принципе KISS (keep it simple, stupid — делай проще, дурак). Скупость означает, что библиотека ориентирована на конкретную задачу и содержит минимальное количество кода, необходимого для эффективного решения этой задачи.

Например, более скупым решением для библиотечной функции будет передача ей в качестве аргумента ссылки на корректный поток `std::istream` для чтения данных, а не открытие ею файла с именем, переданным в качестве аргумента; обработка системозависимой семантики имен файлов и ошибок ввода-вывода не является ключевой для библиотеки обработки данных (см. пример в разделе "Создание экономной сигнатуры функции" главы 11, "Оптимизация ввода-вывода"). Получение указателя на уже выделенный буфер памяти является более скупым, чем выделение памяти и возврат указателя на нее; это также означает, что библиотека не должна обрабатывать исключения, связанные с нехваткой памяти (см. раздел "Библиотеки без копирования" главы 6, "Оптимизация переменных в динамической памяти"). Скупость является конечным результатом последовательного применения хороших принципов разработки программ на C++, таких как принцип единственной ответственности или принцип сегрегации интерфейса.

Скупые библиотеки являются простыми. Они содержат свободные функции или простые классы, являющиеся полными в имеющемся виде. Они могут быть изучены и поняты сами по себе, за один присест. Именно так большинство программистов изучают стандартную библиотеку C++, большую, но, тем не менее, скупую.

Принятие решений о выделении памяти вне библиотеки

Это частный случай правила скупости. Поскольку выделение памяти очень дорогое, решение о выделении памяти по возможности следует вынести из библиотеки. Например, заполняйте буфер, переданный в качестве аргумента в библиотечную функцию, вместо того чтобы возвращать буфер, память для которого выделяется в библиотечной функции.

Вынесение данного решения из библиотечных функций позволяет вызывающим функциям выполнять оптимизацию, описанную в главе 6, "Оптимизация переменных в динамической памяти", где это возможно, повторно используя память вместо выделения ее заново для каждого вызова.

Вынесение выделения памяти из библиотеки уменьшает также количество возможных копирований буфера данных, так как он передается от одной функции к другой.

При необходимости принимайте решения о выделении памяти в производных классах так, чтобы базовый класс содержал только указатель на выделенную память. Таким образом, новые классы могут быть производными классами, по-разному выделяющими память.

Требование выделения памяти вне библиотеки влияет на сигнатуры функций (например, передача в функцию указателя на буфер вместо возвращения выделенного буфера). Важно принять это решение на уровне проектирования библиотеки. Попытки изменять библиотеку после того, как ее функции уже используются, безусловно, вызывают по меньшей мере нарекания.

Если сомневаетесь, выбирайте скорость

В главе 1, "Обзор оптимизации", я цитировал предупреждение Дональда Кнута (Donald Knuth) — "преждевременная оптимизация является корнем всех зол". При разработке библиотек этот совет особенно опасен.

Хорошая производительность особенно важна для библиотечного класса или функции. Создатель библиотеки не может предсказать, когда библиотека может быть использована в контексте, в котором вопросы производительности будут иметь особое значение. Повышение производительности постфактум может быть трудным или невозможным, в особенности если оно включает в себя изменения сигнатур функций или их поведения. Даже библиотека, используемая в рамках одного предприятия, может использоваться во множестве программ. Если библиотека широко распространена, как в случае проектов с открытым исходным кодом, нет никакого способа обновить ее у всех пользователей (или даже просто их пересчитать). Любое изменение становится критическим.

Оптимизация функций проще оптимизации каркасов

Существует два вида библиотек: библиотеки функций и каркасы. Каркас (framework) — это концептуально большой класс, который реализует полный скелет программы, например оконное приложение или веб-сервер. Каркас декорирован небольшими функциями, которые настраивают его так, чтобы он стал конкретным оконным приложением или веб-сервером.

Вторая разновидность библиотек представляет собой набор функций и классов, являющихся компонентами, которые могут совместно использоваться для реализации программы, например анализ URI для веб-сервера или вывод текста в окне.

Обе разновидности библиотек могут воплощать в себе мощную функциональность и высокую производительность. Данный набор возможностей может быть упакован как в функции (как в Windows SDK), так и в каркас (как, например, в Windows MFC). Однако, с точки зрения оптимизатора, работать с библиотеками функций легче, чем с каркасами.

Функции имеют то преимущество, что они могут быть протестированы (а их производительность настроена) в изоляции от других. Использование каркаса требует включения всей "машинерии" одновременно, что делает более трудной задачу изоляции и тестирования изменений. Каркасы нарушают правила разделения задач и единственной персональной ответственности. Это делает их трудно оптимизируемыми.

Функции могут целенаправленно использоваться в большем приложении: здесь — подпрограмма рисования, там — синтаксический анализатор URI. При этом из библиотеки в программу компонуется только необходимая функциональность. Каркасы

же содержат "функции Бога" (см. далее в этой главе раздел "Остерегайтесь «функций Бога»"), которые сами связаны со многими частями каркаса. В результате выполнимый файл может расти в размерах и содержать код, который на самом деле никогда не используется.

Хорошо продуманные функции делают мало предположений о среде, в которой они работают. Каркасы же, напротив, основаны на большой, общей модели того, что планирует делать разработчик. Всякий раз, когда имеется несоответствие между моделью и фактическими потребностями разработчика, получаются неэффективные результаты.

Плоские иерархии наследования

Большинство абстракций требуют не более трех слоев наследования классов: базовый класс с общими функциями, один или несколько производных классов для реализации полиморфизма и, возможно, слой множественного наследования для действительно сложных случаев. Конечно же, для каждого конкретного случая самым главным является мнение разработчика. Однако существенно более глубокие иерархии наследования являются признаком того, что абстракция, представленная иерархией классов, недостаточно четко продумана и добавляет сложности, ведущие к снижению производительности.

С точки зрения оптимизации чем глубже иерархия наследования, тем больше шанс излишних вычислений при вызове функции-члена (см. раздел "Стоимость вызовов функций" главы 7, "Оптимизация инструкций"). Конструкторы и деструкторы в классах при наличии многих слоев родителей должны проходить всю длинную цепочку для выполнения своей работы. Хотя эти функции обычно часто не вызываются, они по-прежнему представляют потенциальный риск введения дорогостоящих вызовов в критически важные для производительности операции.

Упрощение цепочки вызовов

Как и в случае производных классов, *реализация большинства абстракций требует не более трех вызовов вложенных функций*: свободная функция или функция-член, реализующая стратегию, вызов функции-члена некоторого класса и вызов некоторой открытой или закрытой функции-члена, реализующего абстракцию или доступ к данным.

Если доступ к данным осуществляется во вложенной абстракции, реализованной с помощью экземпляра класса, содержащегося в данном, результат может превысить три слоя. Этот анализ выполняется рекурсивно вниз по цепочке вложенных абстракций. Библиотека с четким разделением задач не содержит длинных цепочек вложенных абстракций. Они добавляют накладные расходы в виде вызовов функций и возвратов из них.

Упрощение проектирования слоев

Иногда одна абстракция должна быть реализована в терминах другой абстракции, создавая слоистый дизайн. Как уже отмечалось, это может превратиться в крайность, которая повлияет на производительность.

Но бывают случаи, когда некоторая абстракция реализуется в "слоистой" манере. Это может быть сделано по ряду причин.

- Для реализации слоя с использованием шаблона фасада, который изменяет соглашение о вызовах, возможно, для перехода от аргументов, специфичных для конкретного проекта, к аргументам для вызовов операционной системы, перехода в аргументах от текстовых строк к числовым кодам или добавления обработки ошибок, специфичной для конкретного проекта.
- Для реализации абстракции в терминах тесно связанной с ней иной абстракции, для которой уже имеется библиотечный код.
- Для реализации перехода от функций, возвращающих коды ошибок, к функциям, генерирующим исключения.
- Для реализации идиомы PIMPL (см. раздел "Исключение применения идиомы PIMPL" главы 7, "Оптимизация инструкций").
- Для вызова DLL или подключаемого модуля.

В каждой ситуации окончательное решение — за проектировщиком, поскольку для большинства из перечисленного имеются веские причины. Однако каждый переход между слоями представляет собой дополнительный вызов функции и возврат из нее, что снижает производительность каждого вызова. Дизайнер должен просмотреть переходы между слоями, чтобы увидеть, так ли они необходимы, или два или больше слоев могут быть собраны в один. Вот несколько мыслей о такой проверке кода.

- Большое количество экземпляров шаблона фасада в одном проекте может свидетельствовать об избыточности проекта.
- Если некоторый слой встречается несколько раз (например, когда слой с возвратом кодов ошибок вызывает слой с генерацией исключений, который, в свою очередь, вызывает слой с возвратом кодов ошибок), это один из явных признаков слишком слоистого дизайна.
- Вложенные экземпляры PIMPL трудно оправдать с точки зрения изначальной цели PIMPL — обеспечения брандмауэра для перекомпиляции. Большинство подсистем просто недостаточно велики, чтобы требовать вложенные PIMPL (см. раздел "Исключение применения идиомы PIMPL" главы 7, "Оптимизация инструкций").
- Библиотеки DLL для конкретных проектов часто предлагаются для инкапсуляции исправления ошибок. Эта возможность реализовывалась в очень малом количестве проектов, так как исправление ошибок имеет тенденцию к выпуску в пакетах, которые пересекают границы DLL.

Ликвидация слоев является задачей, которая может быть решена только во время проектирования. Во время проектирования в библиотеке есть коммерческая потребность. После ее завершения независимо от ее недостатков следует сравнивать стоимость любых изменений с получаемыми от них выгодами. Опыт учит, что ни один менеджер не согласится с вашим предложением выделить время и деньги на то, чтобы подправить библиотеки, — если только не приставить к его голове пистолет.

Избегайте динамического поиска

Большие программы содержат большие профили конфигураций или длинные списки записей реестра. Сложные файлы данных, такие как аудио- или видеопотоки, содержат необязательные метаданные, описывающие основные данные. Если есть лишь несколько элементов метаданных, легко определить структуру или класс, который будет их содержать. Когда их количество переваливает за десятки или сотни, многие проектировщики склонны использовать таблицы подстановки, в которых каждые метаданные элемента можно найти с помощью ключевой строки. При записи профилей в формате JSON или XML искушение растет, потому что имеются готовые библиотеки для динамического поиска элементов в JSON- или XML-файлах. Некоторые языки программирования, такие как Objective-C, поставляются с системными библиотеками, работающими таким образом. Однако динамический поиск в символьной таблице — убийца производительности по целому ряду причин.

- Динамический поиск неэффективен по своей природе. Некоторые библиотеки, выполняющие поиск элементов JSON или XML, имеют производительность $O(n)$ по отношению к размеру файла *для каждого поиска*. Поиск в таблице может иметь производительность $O(log_2 n)$. Выборка же элемента данных из структуры выполняется за время $O(1)$ с очень малой константой пропорциональности.
- Разработчик библиотеки может не знать обо всех способах доступа к метаданным. Если профиль инициализации считывается только один раз при запуске программы, вероятно, стоимость этой операции незначительна. Но многие виды метаданных должны читаться во время обработки данных многократно и могут изменяться между последовательными выполнениями такой обработки. Хотя преждевременная оптимизация может быть корнем всех зол, библиотечный *интерфейс*, который выполняет поиск ключевой строки, никогда не сможет стать быстрее, чем поиск в таблице "ключ/значение", без внесения изменений, нарушающих существующие реализации. По-видимому, зло имеет больше одного корня!
- При использовании проекта на основе таблицы поиска тут же возникает вопрос согласованности. Содержит ли таблица все метаданные, необходимые для данного преобразования? Все ли аргументы командной строки, которые должны быть в наличии одновременно, действительно установлены? Хотя согласованность табличного репозитория поддается проверке, эта операция времени выполнения является дорогостоящей, включающей значительное количество кода и множество дорогих поисков. Доступ к данным в простой структуре происходит значительно быстрее, чем несколько поисков в таблице.
- Репозиторий на основе структуры является самодокументируемым в том смысле, что сразу видны все возможные элементы метаданных. Таблица же представляет собой просто большой непрозрачный мешок именованных значений. Команда, работающая с таким хранилищем, должна тщательно документировать метаданные, присутствующие на каждом этапе выполнения программы.

По моему опыту такая дисциплинированность встречается очень редко. Альтернативой является написание бесконечного кода, который пытается восстановить недостающие метаданные, не зная, будет ли этот код когда-либо будет вызван, и, таким образом, не зная, корректен ли он.

Остерегайтесь "функций Бога"

"Функция Бога" — это условное название функции, которая реализует стратегию высокого уровня и которая при использовании в программе заставляет компоновщик добавлять в выполнимый файл многие другие библиотечные функции. Увеличение размера выполнимого файла исчерпывает физическую память во встроенных системах и увеличивает подкачку виртуальной памяти на компьютерах настольного класса.

Многие существующие библиотеки содержат такие дорогостоящие функции. В хороших библиотеках эти функции устраняются во время проектирования. К сожалению, эти функции неизбежны в библиотеках, спроектированных как каркасы.

Из истории оптимизационных войн

Это притча о том, почему функция `printf()` не является вашим другом.

Пожалуй, одной из самых простых возможных программ на C++ (или C) является программа

```
# include <stdio.h>
int main(int, char**) {
    printf("Hello, World!\n");
return 0;
}
```

Сколько выполнимых байтов должна содержать эта программа? Если вы предположили "около 50 или 100 байтов", вы ошиблись на два порядка. Эта программа занимала более 8 Кбайтов на встроенном контроллере, для которого мне как-то пришлось программировать. И это просто код, не таблицы символов, не информация загрузчика или что-нибудь еще.

Вот еще одна программа, выполняющая те же действия:

```
# include <stdio.h>
int main(int, char**) {
    puts("Hello, World!");
return 0;
}
```

Эта программа является практически той же самой, но отличается только использованием для вывода строки `puts()` вместо `printf()`. Но вторая программа занимает на том же контроллере около 100 байтов. Что же вызывает такую разницу в размерах?

Виновницей является функция `printf()`. Она может выводить каждый конкретный тип в трех или четырех форматах. Она может интерпретировать строку формата и читать переменное количество аргументов. `printf()` — большая функция сама по себе, но что действительно делает ее большой, так это то, что она подтягивает функции стандартной библиотеки для форматирования каждого базового типа. На моем встроенном контроллере все обстояло еще хуже, потому что процессор не реализовывал арифметику с плавающей точкой аппаратно, и вместо этого использовалась обширная библиотека функций. `printf()` на самом деле представляет собой образцовый плакат "функции Бога" — функции, которая делает так много, что втягивает в программу огромные фрагменты стандартной библиотеки времени выполнения C.

Функция же `puts()` просто отправляет единственную строку на стандартный вывод. Внутренне она довольно проста, а главное — не заставляет компоноваться с программой половину стандартной библиотеки.

Резюме

- *Функции и классы входят в стандартную библиотеку C++ либо потому, что не могут быть предоставлены иным способом, либо потому, что способствуют очень широкому повторному использованию во многих операционных системах.*
- *Реализации стандартной библиотеки содержат ошибки.*
- *Не существует такой вещи, как "соответствующая стандарту реализация".*
- *Стандартная библиотека не столь эффективна, как лучшие функции операционной системы.*
- *При обновлении библиотеки вносите в нее как можно меньше изменений.*
- *Стабильность интерфейса является основным требованием к библиотеке.*
- *Контрольные примеры критичны для оптимизации библиотеки.*
- *Проектирование библиотеки сходно с проектированием другого кода на C++, но с гораздо более высокими ставками.*
- *Большинство абстракций требуют не более трех слоев наследования классов.*
- *Реализации большинства абстракций требуют не более трех вложенных вызовов функций.*

ГЛАВА 9

Оптимизация сортировки и поиска

Если есть способ сделать дело лучше — найди его.

— Томас Алва Эдисон (Thomas A. Edison) (1847–1931), американский изобретатель и оптимизатор

Программы на языке C++ выполняют множество поисков. От программирования компиляторов языка до веб-браузеров, от списков до баз данных — многие повторяющиеся действия включают поиск в самых глубоко вложенных внутренних циклах. По моему опыту поиск достаточно часто оказывается в списке самых горячих функций. Поэтому эффективному поиску стоит уделить особое внимание.

В этой главе рассматривается поиск в таблицах с точки зрения оптимизатора. Я использую поиск в качестве примера обобщенного процесса, при попытках оптимизации которого разработчик может разделить существующее решение на алгоритмы и структуры данных, а затем рассмотреть каждую часть в поисках возможностей повышения производительности. При демонстрации процесса оптимизации я также рассматриваю некоторые конкретные методы поиска.

Большинство разработчиков на C++ знают, что контейнер `std::map` стандартной библиотеки можно использовать для поиска значений, которые связаны с числовым индексом или буквенно-цифровой строкой. Такие ассоциации называются *таблицами "ключ/значение"*. Они создают *отображение ключей* на *значения*. Разработчики, знакомые с `std::map`, помнят, что этот контейнер имеет хорошую в смысле "большого О" производительность. В этой главе рассматриваются способы оптимизации поиска на основе отображений.

О том, что заголовочный файл `<algorithm>` стандартной библиотеки C++ содержит несколько алгоритмов на основе итераторов, которые выполняют поиск в последовательных контейнерах, знает меньшее количество разработчиков. Даже при оптимальных условиях не все эти алгоритмы имеют такую же эффективность в терминах большого О. Наилучший алгоритм для каждой ситуации не очевиден, а советы в Интернете не всегда указывают оптимальный метод поиска. Поиск лучшего алгоритма поиска представлен как еще один пример процесса оптимизации.

Но даже разработчики, которые хорошо знают используемые ими алгоритмы стандартной библиотеки, могли не слышать, что в C++11 в стандартную библиотеку

вошли контейнеры на основе хеш-таблиц (а задолго до этого они были доступны в библиотеке Boost (`http://www.boost.org/`)). Эти неупорядоченные ассоциативные контейнеры демонстрируют превосходную эффективность с константным средним временем поиска, но и они не являются панацеей.

Таблицы "ключ/значение" с использованием `std::map` и `std::string`

В качестве примера в этом разделе рассматривается производительность поиска и сортировки очень распространенной разновидности таблицы "ключ/значение". Типом ключа таблицы является строка ASCII-символов, которая может быть инициализирована строковым литералом C++ или храниться в `std::string`[1]. Такого рода таблицы обычно используются в ходе анализа профилей инициализации, командных строк, XML-файлов, таблиц баз данных и других приложений, требующих ограниченного набора ключей. Тип значения таблицы может как представлять собой простое целое число, так и быть произвольно сложным. Тип значения не влияет на производительность поиска, за исключением того, что действительно большое значение может снизить производительность кеша. По моему опыту в любом случае доминируют простые типы значений, так что мы будем считать, что в нашей таблице тип значения — простой `unsigned`.

Легко создать таблицу, которая отображает имена `std::string` на значения с помощью `std::map`. Такая таблица может быть определена очень просто:

```
# include <string>
# include <map>
std::map<std::string, unsigned> table;
```

Разработчики, использующие компилятор C++11, могут использовать синтаксис со списком инициализации, который позволяет легко заполнить таблицу записями:

```
std::map<std::string, unsigned> const table {
    { "alpha",   1 }, { "bravo",    2 },
    { "charlie", 3 }, { "delta",    4 },
    { "echo",    5 }, { "foxtrot",  6 },
    { "golf",    7 }, { "hotel",    8 },
    { "india",   9 }, { "juliet",  10 },
    { "kilo",   11 }, { "lima",    12 },
    { "mike",   13 }, { "november",14 },
    { "oscar",  15 }, { "papa",    16 },
    { "quebec", 17 }, { "romeo",   18 },
    { "sierra", 19 }, { "tango",   20 },
    { "uniform",21 }, { "victor",  22 },
    { "whiskey",23 }, { "x-ray",   24 },
    { "yankee", 25 }, { "zulu",    26 }
};
```

[1] Такая таблица может не удовлетворять потребностям разработчиков на арабском или китайском языке (вопросу удовлетворения подобных требований посвящены целые книги). Я надеюсь, что разработчики, использующие наборы широких символов, уже решили эти проблемы, и не рассматриваю их в своих примерах.

Если компилятор не поддерживает такой синтаксис, разработчик должен использовать код для вставки каждого элемента наподобие следующего:

```
table["alpha"] =  1;
table["bravo"] =  2;
...
table["zulu"]  = 26;
```

Получить или протестировать значения не составляет никакого труда:

```
unsigned val = table["echo"];
...
std::string key = "diamond";
if (table.find(key) != table.end())
    std::cout << "Таблица содержит " << key << std::endl;
```

Создание таблиц с помощью `std::map` является примером того, как стандартная библиотека C++ предоставляет мощные абстракции, которые обеспечивают разумную производительность с минимальными усилиями — как для мозга, так и для пальцев. Это пример общего свойства C++, упомянутого в главе 1, "Обзор оптимизации":

> *Сочетание возможностей C++ предоставляет бесконечное количество вариантов реализации, начиная с полной автоматизации и выразительности, с одной стороны, и закачивая полным тонким контролем производительности — с другой. Именно эта возможность выбора позволяет настроить программы на C++ для удовлетворения требованиям производительности.*

Инструментарий для повышения производительности поиска

Но что если профайлер укажет на функцию, содержащую поиск в таблице, как на одну из горячих функций программы? Например:

```
void HotFunction(std::string const& key) {
    ...
    auto it = table.find(key);
    if (it == table.end()) {
        // Действия, если элемент в таблице не найден
        ...
    }
    else {
        // Действия, если элемент найден в таблице
        ...
    }
    ...
}
```

Может ли разработчик поступить лучше, чем в этой простой реализации? Как нам узнать?

Глаз опытного разработчика, конечно же, сразу может увидеть недостатки, которые могут быть устранены. Он может знать, что алгоритм является субоптимальным или что существует лучшая структура данных. Я иногда работаю таким образом, хотя этот путь сопряжен с риском. Все же для разработчиков, заинтересованных в оптимизации, лучше работать методично.

- Измерить производительность существующей реализации для получения исходных показателей для сравнения.
- Идентифицировать оптимизируемую абстрактную деятельность.
- Разделить оптимизируемую деятельность на алгоритмы и структуры данных.
- Изменить или полностью заменить алгоритмы и структуры данных, которые могут быть неоптимальными, и провести эксперименты по выяснению эффективности этих действий.

Если оптимизируемая деятельность рассматривается как абстракция, задача оптимизации состоит в выборе частей базовой реализации абстракции по отдельности и создании более специализированной абстракции с лучшей производительностью.

Я предпочитаю делать это на доске, если это возможно, или хотя бы в текстовом редакторе или в ноутбуке. Процесс этот итеративный. Чем дольше вы разделяете проблему на части, тем больше частей вы обнаруживаете. *В большинстве случаев количество частей, заслуживающих оптимизации, достаточно велико для того, чтобы человек мог надежно удержать их все в голове. Бумага имеет лучшую память*, а доска или текстовый редактор — еще лучше, потому что они позволяют легко добавлять и стирать мысли.

Выполнение базовых измерений

Как уже упоминалось в разделе "Измерение базовой производительности и постановка целей" главы 3, "Измерение производительности", важно измерить производительность неоптимизированного кода, чтобы получить базовый уровень, относительно которого будут выполняться проверки возможных оптимизаций.

Я написал тест, который ищет все 26 значений в таблице, представленной в разделе "Таблицы "ключ/значение" с использованием std::map и std::string" выше в данной главе, а также 27 значений, которых нет в таблице. Я повторяю эти 53 поиска миллион раз, чтобы получить поддающиеся измерению продолжительности. Для отображений строк этот тест выполняется около 2310 мс.

Идентификация оптимизируемой деятельности

Следующим шагом является определение оптимизируемой деятельности, чтобы ее можно было разделить на части, в которых было бы легче искать кандидата для оптимизации.

Идея "оптимизируемой деятельности" остается решением разработчика. Однако есть несколько подсказок. В данном случае разработчик может видеть, что базовая реализация использует `std::map` с ключами `std::string`. Поиск горячего кода показывает вызов `std::map::find()`, функции, которая возвращает итератор найденной

записи, или `end()`. Хотя `std::map` поддерживает поиск, вставку, удаление и итерирование, горячей является только функция поиска. Может быть, необходимо взглянуть и на другие места кода, чтобы увидеть другие операции, выполняемые над таблицей. Особенно интересны места, где таблица создается и уничтожается, потому что эта деятельность может отнять много времени.

В этом случае оптимизируемая деятельность достаточно очевидна: поиск значений в таблице "ключ/значение", реализованной в виде отображения с ключами-строками. Однако важно абстрагироваться от базового решения. Разработчик не должен ограничиваться исследованием только тех таблиц, которые построены из `std::map` и `std::string`.

Для абстрагирования от базовой реализации к формулировке задачи существует метод под названием "размышления вперед и назад". Для размышлений "назад" нужно задаться вопросом "Почему?" Почему в качестве базовой реализации используется `std::map`, а не какие-то иные структуры данных? Ответ достаточно очевиден: `std::map` облегчает поиск по значению ключа. Почему базовая реализация использует `std::string`, а не `int` или указатель на `Foo`? Ответ состоит в том, что ключи являются строками ASCII-текста. Этот анализ приводит разработчика к написанию более абстрактной постановки задачи — *поиск значений в таблице "ключ/значение" с текстовыми ключами*. Обратите внимание: в формулировке нет слов *отображение* и *строка*. Это преднамеренная попытка освободить описание задачи от ее базовой реализации.

Разделение оптимизируемой деятельности

Следующим шагом является разделение оптимизируемой деятельности на алгоритмы и структуры данных. Базовое решение (отображение с ключами-строками) может использоваться в качестве примера алгоритмов и структур данных, составляющих деятельность. Однако базовое решение представляет собой одну конкретную реализацию деятельности, так что разработчик в области оптимизации ищет другие возможные реализации. Вот почему важно описывать алгоритмы и структуры данных в общем виде, чтобы не ограничивать мышление существующим решением.

Оптимизируемым действием является *поиск значений в таблице "ключ/значение" с текстовыми ключами*. Разделение этой формулировки на компоненты алгоритмов и структур данных и сравнение результатов с базовой реализацией приводит к следующему списку.

1. Таблица — структура данных, содержащая ключи и связанные с ними значения.
2. Ключ — структура данных, содержащая текст.
3. Алгоритм сравнения ключей.
4. Алгоритм поиска наличия в структуре данных таблицы определенного ключа, и, если он имеется в наличии, получение соответствующего значения.
5. Алгоритм построения таблицы, т.е. вставки в таблицу ключей.

Откуда мне известно, что нужны именно эти части? Я получил этот список, главным образом глядя на определение структуры данных базового решения, `std::map`.

1. В базовом решении таблица представляет собой `std::map`.
2. В базовом решении ключи представляют собой экземпляры `std::string`.
3. Отчасти это простая логика, но, кроме того, определение шаблона `std::map` предоставляет параметр для функции сравнения строк (который может иметь значение по умолчанию).
4. Горячая функция содержит вызов функции `std::map::find()`, а не оператора `operator[]`.
5. Этот пункт получен из понимания того, что отображение должно быть создано и уничтожено. Общие знания об `std::map` говорят мне, что это отображение реализовано как сбалансированное бинарное дерево. Отображение, таким образом, представляет собой связанную структуру данных, которая должна строиться узел за узлом, так что должен существовать алгоритм вставки.

Последний элемент, алгоритм (а значит, и стоимость) создания и уничтожения таблицы, часто упускается из виду. Эта стоимость может оказаться значительной. Даже если стоимость мала по сравнению со временем, потребляемым поиском, если таблица имеет статическую длительность хранения (см. раздел "Длительность хранения переменной" главы 6, "Оптимизация переменных в динамической памяти"), стоимость инициализации может быть добавлена к стоимости всех прочих инициализаций, выполняемых при запуске (см. раздел "Удаление кода запуска и завершения" главы 12, "Оптимизация параллельности"). Если имеется слишком много инициализаций, программа просто не будет отвечать на действия пользователя. Если таблица имеет автоматическую длительность хранения, она может быть инициализирована несколько раз во время работы программы, делая начальные затраты более значительными. К счастью, имеются реализации таблиц "ключ/значение" с низкой стоимостью создания и уничтожения (или вовсе без таковой).

Изменение или замена алгоритмов и структур данных

На этом шаге разработчик может размышлять "вперед", задаваясь вопросом "Как?" Как и какими различными способами программа может выполнить поиск значений в таблице "ключ/значение" с текстовыми ключами? Разработчик ищет неоптимальные алгоритмы в базовом решении и дорогостоящее поведение, предоставляемое структурами данных, не являющиеся необходимыми для конкретной оптимизируемой деятельности, которые, таким образом, могут быть удалены или упрощены. Затем разработчик выполняет эксперименты, чтобы увидеть, улучшают ли эти изменения производительность.

Оптимизируемые действия предлагают следующие возможности.

- Структура данных таблицы может быть изменена или сделана более эффективной. Некоторые варианты структуры данных таблицы ограничивают выбор алгоритмов поиска и вставки. Выбор структуры данных таблицы может

также влиять на производительность, если структура данных содержит динамические переменные, которые требуют вызовов диспетчера памяти.

- Структура данных ключа может быть изменена или сделана более эффективной.
- Алгоритм сравнения может быть изменен или сделан более эффективным.
- Алгоритм поиска может быть изменен или сделан более эффективным, хотя и может быть ограничен выбором структуры данных таблицы.
- Алгоритм вставки, а также то, когда и как создается и разрушается структура данных, могут быть изменены или сделаны более эффективными.

Вот некоторые наблюдения, касающиеся этих вопросов.

- Отображение `std::map` реализовано как сбалансированное бинарное дерево. Алгоритм поиска в сбалансированном бинарном дереве имеет время работы $O(log_2 n)$. Скорее всего, можно получить большой прирост производительности, если заменить `std::map` структурой данных, которая облегчает поиск при меньших затратах времени.
- Бегло взглянув на определение `std::map`, можно увидеть, что для отображений определены операции вставки элементов в отображение, поиска элементов в отображении, удаления из него элементов и обход элементов в отображении. Для облегчения этих операций `std::map` является структурой данных на основе узлов. В результате `std::map` часто вызывает диспетчер памяти во время построения и имеет плохую локальность кеша. В оптимизируемой деятельности элементы только вставляются в таблицу во время ее построения и никогда не удаляются, пока существует таблица. Менее динамичная структура данных может повысить производительность путем более редкого вызова аллокатора и иметь лучшую локальность кеша.
- Функциональность, которая действительно необходима для структуры данных ключа, — это хранение символов и сравнение двух ключей. `std::string` делает кучу дополнительных действий, помимо тех, которые реально необходимы для ключа таблицы. Строки поддерживают динамически выделенный буфер, поэтому они могут быть изменены и стать длиннее или короче, но таблица "ключ/значение" не изменяет ключи. Кроме того, поиск символьной строки вызывает дорогое преобразование к `std::string`, которое могло бы не потребоваться, если бы ключи хранились как-то иначе.
- Экземпляры `std::string` предназначены для использования в качестве *значений*. `std::string` определяет все шесть операторов сравнения таким образом, чтобы строки можно было сравнивать на равенство или упорядочивать. Контейнер `std::map` предназначен по умолчанию для работы с ключами-значениями, которые реализуют операторы сравнения. Структура данных, которая не определяет собственный оператор сравнения, может использоваться с `std::map` путем предоставления функции сравнения в качестве аргумента шаблона.

- Список инициализации в стиле C++11, показанный выше для отображения со строковыми ключами, напоминает статический составной инициализатор в стиле C, но таковым не является. Инициализатор вызывает выполнимый код, который, в свою очередь, вызывает распределитель памяти для создания `std::string` из каждого строкового литерала, а затем вызывает `insert()`, чтобы добавить каждый элемент в отображение, и эта функция вызывает диспетчер памяти еще раз, чтобы выделить новый узел в структуре данных сбалансированного дерева отображения. Преимуществом этого инициализатора является константность получающейся таблицы. Однако стоимость времени выполнения при этом примерно такая же, как и при создании отображения до C++11 путем добавления значений по одному. Структура данных, которая может быть построена с использованием реального составного инициализатора в стиле C, будет иметь нулевую стоимость построения и уничтожения во время выполнения.

Использование процесса оптимизации для пользовательских абстракций

В случае отображения со строковыми ключами разработчику просто повезло. В `std::map` могут быть запрограммированы многие действия. Можно задать тип ключа, можно изменить алгоритм сравнения типа ключа, можно указать, как должна выделяться память для узлов. Кроме того, в стандартной библиотеке C++ имеются варианты, которые позволяют совершенствовать структуру данных и алгоритм поиска, помимо `std::map`. Вот вкратце почему в C++ можно так эффективно настраивать производительность. В нескольких следующих разделах мы познакомимся с этим процессом.

Описанный здесь процесс можно использовать и для оптимизации абстракций не из стандартной библиотеки, но это может потребовать больше работы. Классы `std::map` и `std::string` представляют собой хорошо определенные и хорошо документированные структуры данных. В вебе имеются прекрасные ресурсы, показывающие, какие операции они поддерживают и как они реализованы. В случае пользовательских абстракций при их вдумчивой разработке заголовочный файл сообщает разработчику, какие операции предоставляются. Комментарии или хорошо спроектированный код рассказывает, какой алгоритм используется и как часто вызывается диспетчер памяти.

Если код беспорядочен или если интерфейс плохо разработан, а код разбросан по множеству файлов, то у меня есть и плохие, и хорошие новости. Плохая новость заключается в том, что описанный здесь процесс оптимизации не предлагает никакой конкретной помощи разработчику. Все, что у меня остается, — это банальности наподобие "Вы знали, что эта работа опасна, когда брались за нее" и "Вот потому вам столько и платят". Хорошая новость заключается в том, что чем ужаснее код, тем, скорее всего, в нем больше возможностей для оптимизации, которая будет наградой для взявшегося за него разработчика.

Оптимизация поиска с использованием std::map

Разработчик в области оптимизации может повысить производительность путем изменения структуры данных, представляющей ключи, а значит, и алгоритма сравнения ключей, оставляя нетронутой табличную структуру данных.

Применение символьных массивов фиксированного размера в качестве ключей std::map

Как отмечалось в главе 4, "Оптимизация использования строк", разработчик может захотеть избежать расходов на использование ключей `std::string` в действительно горячей таблице "ключ/значение". В стоимости создания таблицы доминирует распределение памяти. Разработчик может снизить эту стоимость вдвое, используя тип ключа, который не требует выделения динамической памяти. Кроме того, если таблица использует ключи `std::string`, а разработчик хочет найти записи, представленные строковыми литералами в стиле C, как в

```
unsigned val = table["zulu"];
```

то каждый поиск преобразует строковый литерал `char*` в `std::string`, что требует выделения памяти, которая тут же будет уничтожена.

Если максимальный размер ключа оказывается не слишком длинным, одно из решений — сделать типом ключа некоторый тип класса, содержащий массив `char`, достаточный для хранения наибольшего ключа. Использовать массив непосредственно, как

```
std::map<char[10],unsigned> table
```

невозможно, поскольку массивы C++ не имеют встроенных операторов сравнения.

Вот определение шаблонного класса `charbuf` простого массива символов фиксированного размера:

```
template <unsigned N=10, typename T=char> struct charbuf {
    charbuf();
    charbuf(charbuf const& cb);
    charbuf(T const* p);
    charbuf& operator=(charbuf const& rhs);
    charbuf& operator=(T const* rhs);

    bool operator==(charbuf const& that) const;
    bool operator<(charbuf const& that) const;

private:
    T data_[N];
};
```

Класс `charbuf` чрезвычайно прост. Он может быть инициализирован строкой в стиле C с завершающим нулевым символом или получить соответствующее значение с помощью присваивания. Его можно сравнивать с другим `charbuf`. Поскольку конструктор `charbuf(const T *)` не определен как `explicit`, этот класс можно сравнивать со строкой с завершающим нулевым символом с помощью преобразования типов. Размер массива устанавливается во время компиляции; динамическая память этим классом не используется.

C++ не знает, как сравнивать или упорядочивать два экземпляра класса. Разработчик должен определить все операторы отношений, которые намеревается использовать. Стандартная библиотека C++ обычно использует только операторы == и <. Прочие четыре оператора могут быть построены из этих двух. Операторы могут быть определены как свободные функции:

```
template <unsigned N=10, typename T=char>
bool operator<(charbuf<N,T> const& cb1, charbuf<N,T> const& cb2);
```

Но проще и лучше в смысле стиля C++ определить оператор < внутри `charbuf`, где он имеет доступ к реализации класса.

`charbuf` заставляет программиста думать. Он может работать только со строками, которые вписываются в его внутренний буфер фиксированного размера, с учетом завершающего нулевого символа. Поэтому этот класс не может дать такие же гарантии безопасности, как `std::string`. Проверка того, что все потенциальные ключи вписываются в `charbuf`, является примером переноса вычислений из времени выполнения во время разработки. Это также пример компромисса безопасности, который может потребоваться для повышения производительности. Только конкретная команда проектировщиков может сказать, перевешивают ли выгоды создаваемые риски. Ученые мужи, делающие слишком оптимистичные утверждения, не заслуживают доверия.

Я выполнил тот же тест с 53 именами миллион раз, используя `std::map` с ключами `charbuf`. Тест был выполнен за 1 331 мс. Это примерно вдвое быстрее, чем для версии с использованием `std::string`.

Использование строк в стиле C в качестве ключей `std::map`

Иногда программа имеет доступ к долгоживущим строкам в стиле C с завершающими нулевыми символами, указатели `char*` которых можно использовать в качестве ключей `std::map`. Например, таблица может быть построена с использованием строковых литералов C++. Программа может избежать расходов на строительство и уничтожение экземпляров `std::string`, используя непосредственно `char*`.

Однако с использованием `char*` в качестве типа ключа имеется проблема. `std::map` помещает пары "ключ/значение" в свою внутреннюю структуру данных с помощью отношения упорядочения для типа ключа. По умолчанию это отношение вычисляет выражение `key1<key2`. Класс `std::string` определяет оператор < для сравнения строк символов. Для `char*` также определен оператор <, но этот оператор сравнивает указатели, а не строки, на которые он указывает.

Отображение `std::map` позволяет разработчику решить эту проблему путем предоставления нестандартного алгоритма сравнения. Это пример тонкого управления поведением стандартных контейнеров C++. Алгоритм сравнения предоставляется третьим параметром шаблона `std::map`. Значением по умолчанию для алгоритма сравнения является функциональный объект `std::less<Key>`. Класс `std::less` определяет функцию-член

```
bool operator()(Key const& k1, Key const& k2);
```

которая сравнивает два ключа, возвращая результат выражения `key1<key2`.

В принципе, программа может специализировать `std::less` для `char*`. Однако такая специализация, будучи видимой во всем файле, может вызвать неожиданное поведение в другой части программы.

Вместо функционального объекта программа может предоставить свободную функцию в стиле C для выполнения сравнения, как показано в примере 9.1. Сигнатура функции становится третьим аргументом в объявлении отображения, а экземпляр отображения инициализируется указателем на функцию.

Пример 9.1. Отображение с ключами `char*` и свободной функцией в качестве функции сравнения

```
bool less_free(char const* p1, char const* p2) {
    return strcmp(p1,p2)<0;
}

...

std::map<char const*,
         unsigned,
         bool(*)(char const*,char const*)> table(less_free);
```

Тест, использующий эту версию, выполняется за время 1 450 мс, что является существенным улучшением по сравнению с версией, использующей ключи `std::string`.

Программа может также создать новый функциональный объект для инкапсуляции сравнения. В примере 9.2 `less_for_c_strings` представляет собой имя типа класса, так что оно выступает в роли аргумента типа, и передача указателя не нужна.

Пример 9.2. Отображение с ключами `char*` и функциональным объектом в качестве функции сравнения

```
struct less_for_c_strings {
    bool operator()(char const* p1, char const* p2) {
        return strcmp(p1,p2)<0;
    }
};



std::map<char const*,
         unsigned,
         less_for_c_strings> table;
```

Этот вариант проходит тест за 820 мс. Это почти в три раза быстрее, чем исходная версия, и почти в два раза быстрее, чем версия с `char*` и свободной функцией.

В C++11 есть еще один способ предоставления функции сравнения `char*` для `std::map` — определение лямбда-выражения и его передача в конструктор. Лямбда-выражения удобны, потому что могут быть определены локально и имеют компактный синтаксис. Этот подход проиллюстрирован в примере 9.3.

Пример 9.3. Отображение с ключами char* и лямбда-выражением в качестве функции сравнения

```
auto comp = [](char const* p1, char const* p2) {
    return strcmp(p1,p2)<0;
};

std::map<char const*,
         unsigned,
         decltype(comp)> table(comp);
```

Обратите внимание в этом примере на использование ключевого слова C++11 `decltype`. Третий аргумент шаблона отображения является типом. Имя `comp` является именем переменной, а `decltype(comp)` — ее типом. Тип лямбда-выражения не имеет имени, и тип каждого лямбда-выражения уникален, поэтому `decltype` — единственное средство получения типа лямбда-выражения.

В предыдущем примере лямбда-выражение ведет себя, как функциональный объект, содержащий `operator()`, так что этот механизм имеет ту же измеренную производительность, что и предыдущий, хотя лямбда-выражение должно быть передано в качестве аргумента конструктору отображения.

Лучшее время для таблицы со строками с завершающими нулевыми символами примерно в три раза меньше, чем для исходной версии, и на 55% быстрее, чем для версии с массивом фиксированного размера.

C++ и компилятор Visual Studio

Visual Studio 2010 и Visual Studio 2015 компилируют пример 9.3 корректно, но Visual Studio 2012 и 2013 выдают сообщение об ошибке из-за наличия ошибки в реализации стандартной библиотеки Visual Studio.

Интересный факт C++ — лямбда-выражение без захвата может быть преобразовано в указатель на функцию. Пользователи, которым нравится синтаксис лямбда-выражений, могут использовать лямбда-выражение даже в компиляторах Visual Studio 2012 и 2013. Для этого в качестве третьего аргумента типа отображения используйте сигнатуру указателя на функцию, к которой сводится лямбда-выражение:

```
auto comp = [](char const* p1, char const* p2) {
    return strcmp(p1, p2)<0;
};

std::map<char const*,
         unsigned,
         bool(*)(char const*, char const*)> kvmap(comp);
```

Производительность в этом случае становится такой же, как у `std::map` с указателем на свободную функцию, т.е. ниже, чем производительность с функциональным объектом.

По мере того как развивающийся стандарт C++ постепенно учит компиляторы выполнять все больше и больше выводов типов, лямбда-выражения становятся все более и более интересными. Однако по состоянию на начало 2016 года имеется не так уж много причин, чтобы рекомендовать предпочитать их функциональным объектам.

Использование `std::set`, когда ключ является значением

Некоторые программисты считают естественным определение структуры данных, содержащей ключ и другие данные и выступающей в качестве значения ключа. Действительно, `std::map` внутренне объявляет структуру, которая сочетает в себе и ключ, и значение, как если она была определена следующим образом:

```
template <typename KeyType, typename ValueType> struct value_type {
    KeyType const first;
    ValueType second;
    // ... Конструкторы и операторы присваивания
};
```

Если программа определяет такую структуру данных, она не может непосредственно использоваться в `std::map`. Отображение `std::map` требует, чтобы ключ и значение определялись отдельно, исходя из прагматических соображений. Ключ должен быть константным, поскольку изменение ключа может сделать недействительной структуру данных отображения. Кроме того, указание ключа указывает отображению, как к нему обращаться.

У `std::map` есть родственный класс, `std::set`, который облегчает использование структур данных, содержащих собственные ключи. Этот тип использует функцию сравнения, которая по умолчанию сравнивает два полных элемента с использованием `std::less`. Таким образом, чтобы использовать `std::set` и определяемую пользователем структуру данных, которая содержит собственный ключ, разработчик должен специализировать `std::less` для своей структуры данных, определить для нее `operator<` или предоставить нестандартный объект сравнения. Ни у одного из этих методов нет преимуществ перед прочими, так что выбор используемого метода диктуется предпочитаемым стилем программирования.

Я говорю об этом сейчас, потому что, когда я буду описывать применение последовательных контейнеров в качестве таблицы "ключ/значение", там точно так же нужно будет определить оператор сравнения элементов структур данных или задать функцию сравнения при вызове алгоритма поиска.

Оптимизация поиска с использованием заголовочного файла `<algorithm>`

В предыдущем разделе рассматривалось повышение производительности путем изменения структуры данных, представляющей ключи и, соответственно, алгоритма сравнения ключей. В этом разделе рассматриваются изменения алгоритма поиска и структуры данных таблицы.

В дополнение к таким структурам данных, как `std::string` и `std::map`, стандартная библиотека C++ предоставляет коллекцию алгоритмов, включающую алгоритмы поиска и сортировки. Алгоритмы стандартной библиотеки принимают в качестве аргументов *итераторы*. Итераторы абстрагируют поведение указателей, отделяя обход значений от структуры данных, которая содержит эти значения. Алгоритмы стандартной библиотеки определяются в терминах абстрактного поведения их аргументов-итераторов, а не в терминах некоторой конкретной структуры данных. Алгоритмы на основе итераторов могут использоваться с разными структурами данных, лишь бы итераторы этих структур данных имели требуемые свойства.

Алгоритмы поиска стандартной библиотеки принимают два итератора: указывающий на начало последовательности элементов, в которой выполняется поиск, и на ее конец (на элемент, следующий за последним). Все они принимают также ключ для поиска и при необходимости — функцию сравнения. Алгоритмы различаются тем, что они возвращают, и тем, что должна делать функция сравнения: определять отношение упорядочения между ключами или просто проверять их равенство.

Часть структуры данных для поиска можно описать как *диапазон* `[first,last)`, в котором квадратная скобка слева от `first` означает, что `first` включен в интервал, а круглая закрывающая скобка после `last` означает, что `last` в него не входит. Такие диапазоны часто используются в алгоритмах стандартной библиотеки.

Некоторые методы поиска на основе итераторов реализуют алгоритмы "разделяй и властвуй". Эти алгоритмы полагаются на определенные возможности некоторых итераторов, например на возможность вычислить *дистанцию* или количество элементов между двумя итераторами, и, таким образом, добиться поведения лучшего, чем линейное. Дистанция между двумя итераторами всегда может быть вычислена путем увеличения одного итератора, пока он не станет равным второму, но это приводит к стоимости вычисления дистанции, равной $O(n)$. Специальным свойством *итераторов произвольного доступа* является возможность вычисления дистанции за константное время.

Последовательными контейнерами, предоставляющими итераторы произвольного доступа, являются массивы в стиле C, `std::string`, `std::vector` и `std::deque`. Алгоритмы "разделяй и властвуй" могут быть применены и к `std::list`, но их стоимость будет $O(n)$, а не $O(log_2 n)$, из-за более высокой стоимости вычисления дистанции для двунаправленных итераторов.

Имена *строка* и *отображение* отражают их функции и могут привести начинающего пользователя к решению с использованием этих типов данных. К сожалению, не все алгоритмы поиска на основе итераторов имеют описательные имена. Они также имеют очень обобщенный характер, так что правильный выбор может привести к существенной разнице в производительности, хотя они могут иметь ту же стоимость в смысле большого O.

Таблица "ключ/значение" для поиска в последовательных контейнерах

Есть несколько причин для того, чтобы предпочесть реализацию последовательного контейнера для таблицы "ключ/значение" использованию `std::map` или `std::set`. Последовательные контейнеры потребляют меньше памяти, чем отображения. Они также могут быть менее дорогими в настройке. Одним очень полезным свойством алгоритмов стандартной библиотеки является то, что они могут работать с обычными массивами любого типа. Таким образом, они могут выполнять эффективный поиск в статически инициализированных массивах структур. Это устраняет все затраты на создание и уничтожение таблиц. Кроме того, стандарты кодирования, такие как MISRA C++ (`http://www.misra-cpp.com/`), запрещают или ограничивают использование динамически выделяемых структур данных. С помощью указанных реализаций мы можем находить эффективные решения в средах такого рода.

В примерах этого раздела использовано следующее определение структуры:

```
struct kv {             // Пары (key,value)
    char const* key;
    unsigned value;     // Может быть любым
};
```

Статический массив таких пар "ключ/значение" определен следующим образом.

```
kv names[] = {          // В алфавитном порядке
        { "alpha",   1 },   { "bravo",    2 },
        { "charlie", 3 },   { "delta",    4 },
        { "echo",    5 },   { "foxtrot",  6 },
        { "golf",    7 },   { "hotel",    8 },
        { "india",   9 },   { "juliet",  10 },
        { "kilo",   11 },   { "lima",    12 },
        { "mike",   13 },   { "november",14 },
        { "oscar",  15 },   { "papa",    16 },
        { "quebec", 17 },   { "romeo",   18 },
        { "sierra", 19 },   { "tango",   20 },
        { "uniform",21 },   { "victor",  22 },
        { "whiskey",23 },   { "x-ray",   24 },
        { "yankee", 25 },   { "zulu",    26 }
};
```

Инициализатором массива `names` является статический составной инициализатор. Компилятор C++ создает инициализированные данные для структур в стиле C во время компиляции. Стоимость создания таких массивов во время выполнения — нулевая.

Я исследовал различные алгоритмы путем поиска в этой небольшой таблице каждого из 26 ее ключей и 27 отсутствующих строк, которые выбраны таким образом, чтобы находиться между значениями таблицы. Эти 53 поиска повторяются миллион раз (это тот же тест, что и для `std::map` в предыдущем разделе).

Классы контейнеров стандартной библиотеки предоставляют функции-члены `begin()` и `end()`, так что программа может получить диапазон итераторов для выполнения поиска. Массивы в стиле C более примитивны и таких функций не имеют.

Однако немного шаблонной магии предоставляют нам возможность написать шаблонные функции для этой цели. Поскольку они принимают тип массива в качестве аргумента, массив не приводится к указателю, как это обычно происходит:

```
//   Получение размера и итераторов begin/end для массивов в стиле C
template <typename T, int N> size_t size(T (&a)[N]) {
    return N;
}
template <typename T, int N> T* begin(T (&a)[N]) {
    return &a[0];
}
template <typename T, int N> T* end(T (&a)[N]) {
    return &a[N];
}
```

В C++11 в заголовочном файле `<iterator>` в пространстве имен `std` имеются более сложные определения `begin()` и `end()`, созданные с помощью той же шаблонной магии. Этот заголовочный файл включается всякий раз при включении заголовочного файла любого контейнера стандартной библиотеки. Visual Studio 2010 предоставлял эти определения в ожидании выхода нового стандарта. К сожалению, функция `size()` не вошла в стандарт до C++14 и не реализована в Visual Studio 2010, хотя написать ее упрощенный эквивалент достаточно легко.

`std::find()`: очевидное имя, стоимость — O(*n*)

Заголовочный файл стандартной библиотеки `<algorithm>` определяет шаблонную функцию `find()` следующим образом:

```
template <class It, class T> It find(It first, It last, const T& key);
```

Алгоритм `find()` представляет собой простой линейный поиск — наиболее общий вид поиска. Он не требует никакого упорядочения данных, в которых проводится поиск, и требует только реализацию проверки равенства ключей.

Алгоритм `find()` возвращает итератор, указывающий на первую запись в последовательном контейнере, имеющую ключ, эквивалентный искомому. Аргументы-итераторы `first` и `last` определяют диапазон поиска, причем итератор `last` указывает на первый элемент после окончания данных, в которых выполняется поиск. Типы `first` и `last`, задаваемые параметром шаблона `It`, зависят от вида структуры данных, которую обходит `find()`. Соответствующий пример приведен в примере 9.4.

Пример 9.4. Линейный поиск с использованием `std::find()`

```
kv* result = std::find(std::begin(names), std::end(names), key);
```

В этом вызове `names` — имя массива, в котором выполняется поиск. `key` представляет собой значение, сравниваемое с каждой записью `kv`. Чтобы выполнить сравнение, в области видимости инстанцирования функции `find()` должна быть определена функции сравнения ключей. Эта функция говорит `std::find()` все, что ей нужно знать, чтобы выполнить сравнение. C++ позволяет перегрузить оператор равенства `bool operator==(v1,v2)` для значений произвольных пар типов. Если `key` представляет собой указатель на `char`, то такая функция имеет следующий вид:

```
bool operator==(kv const& n1, char const* key) {
    return strcmp(n1.key, key) == 0;
}
```

Хронометраж с использованием `std::find()` и набором ключей, как имеющихся в 26-элементной таблице, так и отсутствующих в ней, занимает 1 425 мс.

Вариация функции `find()` с названием `find_if()` принимает в качестве третьего аргумента функцию сравнения. Вместо определения оператора `operator==()` в области видимости `find()` разработчик может написать его как лямбда-выражение. Лямбда-выражение принимает один аргумент — элемент таблицы, с которым выполняется сравнение. Таким образом, лямбда-выражение должно захватывать значение ключа из среды выполнения.

`std::binary_search()`: не возвращает значения

Бинарный поиск является стратегией "разделяй и властвуй", которая обычно настолько полезна, что стандартная библиотека C++ предоставляет несколько различных алгоритмов на ее основе. Но по какой-то причине такое "говорящее" имя, как `binary_search`, оказалось использованным для алгоритма, который не слишком полезен для поиска значений.

Алгоритм стандартной библиотеки `binary_search()` возвращает значение типа `bool`, указывающее, находится ли ключ в отсортированной таблице. Как ни странно, связанной функции для возврата соответствующего элемента таблицы нет. Таким образом, ни одно из двух очевидных имен, `find` и `binary_search`, не дает нам искомого решения.

Если программа просто хочет знать, есть ли искомый элемент в таблице, и ее не интересует значение этого элемента, то тест с применением `std::binary_search()` выполняется за 972 мс.

Бинарный поиск с использованием `std::equal_range()`

Если последовательный контейнер отсортирован, разработчик может собрать воедино эффективную функцию поиска из фрагментов, предоставляемых стандартной библиотекой C++. К сожалению, эти элементы имеют имена, которые никак не ассоциируются с понятием бинарного поиска.

Заголовочный файл стандартной библиотеки C++ `<algorithm>` содержит шаблонную функцию `std::equal_range()`, определенную следующим образом:

```
template <class ForwardIt, class T>
    std::pair<ForwardIt,ForwardIt>
        equal_range(ForwardIt first, ForwardIt last, const T& value);
```

`equal_range()` возвращает пару итераторов, отделяющую подпоследовательность отсортированной последовательности `[first,last)`, в которой содержатся записи со значением, равным `value`. Если записей со значением `value` нет, `equal_range()` возвращает пару итераторов, указывающих на одну и ту же точку, идентифицируя таким образом пустой диапазон. Если возвращенные итераторы не равны, в контейнере имеется хотя бы одна запись, которая имеет значение `value`. По способу

построения, использованному в нашем примере, в контейнере может иметься не более одного совпадения, и первый итератор указывает на него. В примере 9.5 переменная `result` указывает на соответствующую запись в таблице (если ключ найден) или на конец таблицы (в противном случае).

Пример 9.5. Бинарный поиск с использованием `std::equal_range()`

```
auto res = std::equal_range(std::begin(names),
                            std::end(names), key);
kv* result = (res.first == res.second)
           ? std::end(names)
           : res.first;
```

Эксперимент по определению производительности `equal_range()` для приведенной ранее таблицы дает время выполнения 1 810 мс. Фактически это хуже, чем линейный поиск для таблицы такого же размера, что оказывается довольно шокирующим. Однако далее мы увидим, что `equal_range()` не является лучшим выбором в качестве функции бинарного поиска.

Бинарный поиск с использованием `std::lower_bound()`

Хотя `equal_range()` обещает стоимость поиска $O(log_2 n)$, в действительности в ней выполняется больше действий, чем требуется для простого поиска в таблице. Возможная реализация `equal_range()` может выглядеть следующим образом:

```
template <class It, class T>
    std::pair<It,It>
        equal_range(It first, It last, const T& value) {
            return std::make_pair(std::lower_bound(first, last,
                                                   value),
                                  std::upper_bound(first, last,
                                                   value));
}
```

Функция `upper_bound()` для поиска конца возвращаемого диапазона делает второй проход через таблицу в стиле "разделяй и властвуй", поскольку алгоритм `equal_range()` достаточно обобщенный для работы с отсортированными последовательностями, содержащими более одного значения с одинаковым ключом. Но в соответствии с принципом построения в таблице из нашего примера диапазон всегда будет содержать либо одну запись, либо ни одной. Таким образом, поиск может выполняться с помощью `lower_bound()` и одного дополнительного сравнения, как показано в примере 9.6.

Пример 9.6. Бинарный поиск с использованием `std::lower_bound()`

```
kv* result = std::lower_bound(std::begin(names),
                              std::end(names),
                              key);
if (result != std::end(names) && key < *result.key)
    result = std::end(names);
```

В этом примере `std::lower_bound()` возвращает итератор, указывающий на первую запись в таблице, ключ которой не меньше искомого ключа. Если все записи

меньше искомого ключа, итератор указывает за конец таблицы. Поскольку данный итератор может указывать на запись, которая больше искомого ключа, последняя инструкция `if` устанавливает `result` за конец таблицы, если любое из этих условий истинно. В противном случае `result` указывает на запись, ключ которой равен `key`.

Хронометраж с применением такого варианта поиска дает время выполнения 973 мс, что на 46% быстрее, чем `std::equal_range()`. Этого следовало ожидать, так как новый вариант делает половину работы.

Поиск с использованием `std::lower_bound` имеет производительность, конкурентоспособную с лучшей реализацией с применением `std::map`, и обладает дополнительным преимуществом нулевой стоимости построения и уничтожения статической таблицы. Тест с функцией `std::binary_search()` также выполняется за 973 мс, хотя и дает только логический результат. Похоже, это все, чего мы можем достичь с использованием алгоритмов стандартной библиотеки C++.

Самостоятельное кодирование бинарного поиска

Можно самостоятельно написать функцию бинарного поиска, принимающую те же аргументы, что и функции стандартной библиотеки. Все алгоритмы стандартной библиотеки используют одну функцию упорядочения, `operator<()`, так что необходимо предоставить только минимальный интерфейс. Поскольку эти функции в конечном итоге определяют, соответствует ли ключ записи, в конце они делают дополнительные сравнения исходя из того, что `a==b` можно определить как `!(a<b)&&!(b<a)`.

Исходная таблица занимает диапазон последовательных значений, обозначаемый как `[start,end)`. На каждом этапе поиска функция (показанная в примере 9.7) вычисляет середину диапазона и сравнивает `key` с записью в середине диапазона. Это эффективно разделяет диапазон на две половины, `[start,mid+1)` и `[mid+1,stop)`.

Пример 9.7. Функция бинарного поиска с использованием для сравнения оператора "<"

```
kv* find_binary_lessthan(kv* start, kv* end, char const* key) {
    kv* stop = end;
    while (start < stop) {
        auto mid = start + (stop-start)/2;
        if (*mid < key) {   // Поиск в правой половине [mid+1,stop)
            start = mid + 1;
        }
        else {              // Поиск в левой половине [start,mid)
            stop = mid;
        }
    }
    return (start == end || key < *start) ? end : start;
}
```

Хронометраж показывает, что время выполнения теста с данной функцией — 968 мс. Это примерно то же самое, что и для версии с использованием `std::lower_bound()`.

Самостоятельное кодирование бинарного поиска с использованием `strcmp()`

Можно продолжить наши развлечения, заметив, что `operator<()` определен в терминах `strcmp()`. Подобно оператору `operator<()`, функция `strcmp()` сравнивает два ключа. Но `strcmp()` дает большее количество информации: результат типа `int`, который меньше, равен или больше нуля, если первый ключ меньше, равен или больше второго ключа соответственно. В примере 9.8 показан соответствующий код, который мог быть написан на языке C.

На каждой итерации цикла `while` последовательность для поиска задается интервалом `[start,stop)`. На каждом шаге переменная `mid` устанавливается равной средине последовательности, в которой выполняется поиск. Значение, возвращаемое функцией `strcmp()`, делит последовательность на *три* части, а не на две: `[start,mid)`, `[mid,mid+1)` и `[mid+1,stop)`. Если `mid->key` больше `key`, то мы знаем, что `key` должен находиться в левой части, до `mid`. Если `mid->key` меньше `key`, то `key` должен находиться в правой части, начинающейся с `mid+1`. В противном случае `mid->key` равно `key`, и цикл завершает работу. Логика `if/else` делает более часто выполняющееся сравнение первым для повышения производительности.

Пример 9.8. Функция бинарного поиска с использованием `strcmp()`

```
kv* find_binary_3(kv* start, kv* end, char const* key) {
    auto stop = end;
    while (start < stop) {
        auto mid = start + (stop-start)/2;
        auto rc = strcmp(mid->key, key);
        if (rc > 0) {
            stop = mid;
        }
        else if (rc < 0) {
            start = mid + 1;
        }
        else {
            return mid;
        }
    }
    return end;
}
```

Хронометраж показал время работы теста с данной функцией, равное 771 мс. Это примерно на 21% быстрее лучшего времени с использованием функций стандартной библиотеки.

Оптимизация поиска в хешированных таблицах "ключ/значение"

В предыдущем разделе рассмотрены изменения алгоритма для конкретной табличной структуры данных, которую оказалось возможным очень эффективно настроить. В этом разделе мы рассмотрим другую структуру данных и алгоритм, а именно — хеш-таблицу.

Идеей хеш-таблицы является то, что ключ, какой бы тип он ни имел, может быть передан *хеш-функции*, которая превращает его в целочисленное *хеш-значение*. Это значение становится индексом массива, ведущим непосредственно к записи таблицы. Если эта запись таблицы соответствует ключу, поиск является успешным. Если хеш-значение всегда приводит непосредственно к искомой записи таблицы, такая хеш-таблица обеспечивает доступ за константное время. Единственная стоимость поиска оказывается равной стоимости вычисления хеш-значения. Как и линейный поиск, хеширование не предполагает наличия какого-то отношения упорядочения между ключами и требует только возможности сравнения ключей на равенство.

Самая сложная часть реализации хеш-таблицы — поиск эффективной хеш-функции. Десятисимвольная строка содержит намного больше битов, чем 32-битное целое число. Таким образом, хешироваться в одно и то же значение индекса могут несколько разных строк, и для корректной работы необходим механизм разрешения таких *коллизий*. Каждая запись в хеш-таблице может быть началом списка записей, хешированных в этот индекс. Кроме того, можно выполнять поиск пустого индекса в смежных записях.

Еще одна проблема заключается в том, что хеш-функция может не давать некоторый индекс ни для одного из корректных ключей таблицы, оставляя в таблице неиспользуемые места. Это делает хеш-таблицы потенциально бо́льшими, чем отсортированный массив тех же записей.

Плохая хеш-функция или неудачный набор ключей может привести к тому, что для множества ключей будет вычислен одинаковый индекс. Тогда производительность хеш-таблицы деградирует к O(n), что делает ее не лучше линейного поиска.

Хорошая хеш-функция создает индексы массива, которые мало коррелируют с битами ключа. Для этой цели хороши генераторы случайных чисел и криптографические кодировщики. Но если вычисление хеш-функции будет дорогим, то у хеширования может не оказаться никаких преимуществ над бинарным поиском, если только таблица не окажется очень большой.

Поиском лучших хеш-функций ученые-кибернетики занимались на протяжении многих лет. Раздел Q&A на сайте Stack Exchange (`http://bit.ly/se-hash`) предоставляет информацию о производительности и ссылки для нескольких популярных хеш-функций. Разработчику, занимающемуся настройкой хеш-таблицы, следует знать, что здесь до него частым гребнем проходилось немало народу, так что большого улучшения производительности здесь не получить.

C++ определяет стандартный объект хеш-функции под названием `std::hash`. Это шаблон со специализацией для целых чисел, чисел с плавающей точкой, указателей и `std::string`. Неспециализированное определение `std::hash`, который используется и для указателей, преобразует хешированный тип в `size_t`, а затем рандомизирует биты этого значения.

Хеширование с использованием `std::unordered_map`

В C++11 стандартный заголовочный файл `<unordered_map>` предоставляет программистам хеш-таблицу. Уже Visual Studio 2010 в ожидании нового стандарта предоставил этот заголовочный файл. Однако `std::unordered_map` нельзя использовать

с закодированной вручную статической таблицей, используемой в предыдущих примерах. Записи должны вставляться в хеш-таблицу, увеличивая тем самым стоимость запуска. Код создания хеш-кода таблицы с помощью `std::unordered_map` и вставка записей показаны в примере 9.9.

Пример 9.9. Инициализация хеш-таблицы

```
std::unordered_map<std::string, unsigned> table;
for (auto it = names; it != names+namesize; ++it)
    table[it->key] = it->value;
```

Хеш-функция, используемая `std::unordered_map` по умолчанию, — объект шаблона функции `std::hash`. Этот шаблон имеет специализацию для `std::string`, поэтому предоставлять хеш-функцию явно не требуется.

После внесения записей в таблицу можно выполнить поиск:

```
auto it = table.find(key);
```

`it` — это итератор таблицы, который либо указывает на корректную запись, либо равен `table.end()`.

Для простоты и обеспечения производительности `std::unordered_map` с ключами `std::string` использует все значения по умолчанию шаблона `std::map`. Эксперимент по измерению производительности дает значение времени выполнения 1725 мс (без учета времени на создание таблицы). Это на 25% быстрее, чем `std::map` со строковыми ключами, но едва ли его можно считать мировым рекордом. С учетом уверений в литературе о превосходной производительности хеширования этот результат кажется удивительным и разочаровывающим.

Хеширование с фиксированными символьными массивами в качестве ключей

Версия шаблона класса простого фиксированного символьного массива `charbuf` из раздела "Применение символьных массивов фиксированного размера в качестве ключей std::map" данной главы может использоваться и с хеш-таблицами. Приведенный далее шаблон расширяет `charbuf` средствами для хеширования символьной строки и оператором `operator==()` для сравнения ключей в случае коллизии:

```
template <unsigned N=10, typename T=char>
struct charbuf {
    charbuf();
    charbuf(charbuf const& cb);
    charbuf(T const* p);
    charbuf& operator=(charbuf const& rhs);
    charbuf& operator=(T const* rhs);

    operator size_t() const;

    bool operator==(charbuf const& that) const;
    bool operator<(charbuf const& that) const;
private:
    T data_[N];
};
```

Хеш-функция представляет собой `operator size_t()`. Это не слишком интуитивно понятный результат. Дело в том, что специализация по умолчанию для `std::hash()` получает аргумент типа `size_t`. В случае указателей выполняется обычное приведение битов указателя, но в случае `charbuf&` вызывается оператор приведения `charbuf::operator size_t()`, возвращающий значение типа `size_t`. Объявление хеш-таблицы с использованием `charbuf` выглядит следующим образом:

```
std::unordered_map<charbuf<>, unsigned> table;
```

Производительность этой хеш-таблицы разочаровывает. Тест выполнился за 2 277 мс, что даже хуже, чем в случае хеш-таблицы или отображения с `std::string`.

Хеширование с ключами в виде строк с завершающими нулевыми символами

> *Такие вещи требуют деликатности, иначе чары развеются...*
>
> — Злая Колдунья Запада (Маргарет Хэмилтон (Margaret Hamilton), *к/ф Волшебник страны Оз*, (1939)), размышляя о том, как снять башмачки с Дороти

Если хеш-таблица может быть инициализирована долгоживущими строками с завершающими нулевыми символами, такими как строковые литералы C++, то хеш-таблица "ключ/значение" может быть построена с использованием указателей на эти строки. Золото производительности можно добыть из шахты `std::unordered_map`, работая с ключами `char*`. Но это задача далеко не тривиальная.

Вот как выглядит полное определение `std::unordered_map`:

```
template<
    typename Key,
    typename Value,
    typename Hash = std::hash<Key>,
    typename KeyEqual = std::equal_to<Key>,
    typename Allocator = std::allocator<std::pair<const Key, Value>>
> class unordered_map;
```

`Hash` представляет собой объявление типа функционального объекта или указателя на функцию для функции, которая хеширует `Key`. `KeyEqual` — объявление типа функционального объекта или указателя на функцию для функции, которая сравнивает два экземпляра `Key` на равенство для разрешения коллизий хеша.

Если `Key` является указателем, `Hash` точно определен. Программа будет компилироваться без ошибок. Может даже показаться, что она работает. (Мои первые тесты дали отличные результаты времени и ложное чувство выполненного долга.) Но написанная таким образом программа не является правильной! Дело в том, что `std::hash` создает хеш значения указателя, а не строки, на которую он указывает. Если, например, тестовая программа загружает таблицу из массива строк, а затем проверяет, действительно ли каждую строку можно найти, то указатели, использованные в тесте для поиска, совпадают с указателями на ключи, которые использованы при инициализации таблицы, и создается впечатление, что она отлично работает!

Протестируйте таблицу с дубликатами строк, полученными из пользовательского ввода, и чары развеются — тест будет утверждать, что такой строки нет в таблице, так как указатель на искомую строку не совпадает с указателем на ключ, которым инициализировалась таблица.

Эта проблема может быть решена путем предоставления нестандартной хеш-функции вместо значения по умолчанию для третьего аргумента шаблона. Как и для отображения, эта хеш-функция может быть функциональным объектом, лямбда-выражением или указателем на свободную функцию:

```
struct hash_c_string {
    void hash_combine(size_t& seed, T const& v) {
        seed ^= v + 0x9e3779b9 + (seed << 6) + (seed >> 2);
    }

    std::size_t operator() (char const* p) const {
        size_t hash = 0;
        for (; *p; ++p)
            hash_combine(hash, *p);
        return hash;
    }
};

// Это решение не полное - см. ниже
std::unordered_map<char const*, unsigned, hash_c_string> table;
```

Я взял хеш-функцию из Boost. Она имеется в реализации стандартной библиотеки, соответствующей C++14 или более поздней версии. Увы, Visual Studio 2010 эту функцию не предоставляет.

Тщательное тестирование показывает, что это объявление все еще не корректно, хотя и компилируется без ошибок и дает программу, которая может работать для некоторых небольших таблиц. Проблема связана с `KeyEqual`, четвертым аргументом шаблона `std::unordered_map`. Значением этого аргумента по умолчанию является `std::equal_to`, функциональный объект, который применяет оператор `operator==` к двум своим операндам. Этот оператор определен для указателей, и сравнивает указатели по их порядку в памяти компьютера, а не по строкам, на которые они указывают.

Решением, конечно, является другой нестандартный функциональный объект параметра шаблона `KeyEqual`. Полное решение показано в примере 9.10.

Пример 9.10. `std::unordered_map` с ключами, являющимися строками с завершающими нулевыми символами

```
struct hash_c_string {
    void hash_combine(size_t& seed, T const& v) {
        seed ^= v + 0x9e3779b9 + (seed << 6) + (seed >> 2);
    }

    std::size_t operator() (char const* p) const {
        size_t hash = 0;
        for (; *p; ++p)
            hash_combine(hash, *p);
        return hash;
    }
};
```

```
struct comp_c_string {
    bool operator()(char const* p1, char const* p2) const {
        return strcmp(p1,p2) == 0;
    }
};

std::unordered_map<
    char const*,
    unsigned,
    hash_c_string,
    comp_c_string
> table;
```

Эта версия таблицы "ключ/значение", использующая `std::unordered_map` и `char`, выполняет тест за 993 мс. Это на 56% быстрее, чем хеш-таблица на основе `std::string`. Но это медленнее, чем лучшая реализация на основе `std::map` и ключей `char*`. И медленнее, чем бинарный поиск в простом статическом массиве записей "ключ/значение" с помощью `std::lower_bound`. Это совсем не то, чего я мог ожидать после нескольких лет шумной рекламы. (В разделе "`std::unordered_map` и `std::unordered_multimap`" главы 10, "Оптимизация структур данных", мы увидим, что бо́льшие хеш-таблицы имеют большее преимущество перед алгоритмами бинарного поиска.)

Хеширование с пользовательской хеш-таблицей

Хеш-функция для использования с неизвестными ключами должна быть очень обобщенной. Если же ключи известны заранее, как в таблице из нашего примера, может быть достаточно очень простой хеш-функции.

Хеш, создающий таблицу, в которой нет никаких коллизий для данного набора ключей, называется *идеальным*. Хеш, который создает таблицу без неиспользуемых пробелов, называется *минимальным*. Святой Грааль хеширования — минимальный совершенный хеш, создающий таблицу без коллизий и пробелов. Легко создавать идеальные хеши для разумно коротких наборов ключевых слов; идеальный минимальный хеш лишь немногим сложнее. Первая буква (или первые несколько букв), сумма букв, а также длина ключа — все это примеры хеш-функций, которые могут быть опробованы.

В примере таблицы для этого раздела 26 допустимых записей начинаются с разных букв и отсортированы, поэтому хеш на основе первой буквы является идеальным минимальным хешем. Недопустимые ключи не имеют никакого значения. Они сравниваются с корректным ключом, имеющим тот же хеш, и сравнение завершается ошибкой. В примере 9.11 показана очень простая пользовательская хеш-таблица, реализованная в стиле `std::unordered_map`.

Пример 9.11. Пример минимальной идеальной хеш-таблицы

```
unsigned hash(char const* key) {
    return key[0]%26;
}
```

```
kv* find_hash(kv* first, kv* last, char const* key) {
    unsigned i = hash(key);
    return strcmp(first[i].key, key) ? last : first + i;
}
```

Функция `hash()` отображает первую букву `key` на одну из 26 записей таблицы.

Хронометраж функции `find_hash()` показал время выполнения, равное 253 мс. Этот результат просто ошеломляет!

Хотя нам просто очень повезло с простой хеш-функцией, которая работает для нашей таблицы-примера, сама таблица не является специально созданной для достижения такого впечатляющего результата. Очень часто существует простая функция, которая представляет собой минимальный идеальный хеш. В Интернете можно найти статьи, в которых обсуждаются различные методы автоматического создания совершенной минимальной хеш-функции для небольших наборов ключевых слов. GNU Project (среди прочих) создал утилиту командной строки `gperf` (`http://www.gnu.org/software/gperf/`) для создания идеальной хеш-функции, которая часто оказывается одновременно и минимальной.

Цена абстракций Степанова

Эксперимент, который я использовал в этой главе, выполняет поиск всех 26 допустимых и 27 недопустимых записей таблицы. Это позволяет оценить своего рода среднюю производительность. Линейный поиск только для ключей, содержащихся в таблице, выглядит относительно лучше, так как завершается немедленно по нахождении искомой записи. Бинарный поиск делает примерно одинаковое количество сравнений для поиска как имеющихся в таблице ключей, так и отсутствующих.

В табл. 9.1 подытожены результаты моих экспериментов.

Таблица 9.1. Время работы различных методов поиска

	VS2010, i7, 1M итераций, мс	Улучшение по сравнению с предыдущим экспериментом, %	Улучшение в категории, %
`map<string>`	2307		
`map<char*> free function`	1453	37	37
`map<char*> function object`	820	44	64
`map<char*> lambda`	820	0	64
`std::find()`	1425		
`std::equal_range()`	1806		
`std::lower_bound`	973	46	46
`find_binary_3way()`	771	21	57
`std::unordered_map()`	509		
`find_hash()`	195	62	62

Как и ожидалось, бинарный поиск оказался быстрее, чем линейный, а хеширование быстрее, чем бинарный поиск.

Стандартная библиотека C++ предоставляет множество готовых к использованию, отлаженных алгоритмов и структур данных, которые полезны во многих ситуациях. Стандарт определяет их стоимость в наихудшем случае в O-записи, чтобы продемонстрировать их широкую применимость.

Но чрезвычайно мощный и универсальный механизм стандартной библиотеки имеет свою стоимость. Даже если существует стандартный алгоритм библиотеки с хорошей производительностью, зачастую он не может конкурировать с лучшим алгоритмом, закодированным вручную. Это может быть связано с недостатками в коде шаблона или недостатками в дизайне компилятора, или просто потому, что код стандартной библиотеки должен работать для очень общих ситуаций (например, используя только `operator<()`, а не `strcmp()`). Эта стоимость может убедить разработчика кодировать действительно важные поиски вручную.

Этот разрыв в производительности между стандартными и хорошо закодированными вручную алгоритмами называется стоимостью абстракций Степанова в честь Александра Степанова, который разработал оригинальные исходные алгоритмы стандартной библиотеки и классы контейнеров еще в то время, когда не существовало компилятора, способного их скомпилировать. Стоимость абстракции Степанова является неизбежной ценой универсального решения по сравнению с пользовательским. *Это плата за использование высокопродуктивных инструментов, таких как алгоритмы стандартной библиотеки* C++. Это не плохо, но это то, что разработчикам необходимо иметь в виду, когда им нужна очень высокая производительность.

Оптимизация сортировки с использованием стандартной библиотеки C++

Последовательные контейнеры должны быть отсортированы, прежде чем над ними можно будет выполнять эффективный поиск с помощью алгоритмов "разделяй и властвуй". Стандартная библиотека C++ предоставляет два стандартных алгоритма, `std::sort()` и `std::stable_sort()`, которые могут эффективно сортировать последовательные контейнеры.

Хотя стандарт не указывает, какой именно алгоритм сортировки используется, он написан так, что `std::sort` может быть реализован с помощью некоторых вариаций алгоритма быстрой сортировки, а `std::stable_sort` может быть реализован с помощью сортировки слиянием. Стандарт C++03 требовал, чтобы алгоритм `std::sort` имел среднюю производительность $O(n \cdot log_2 n)$. Реализации, соответствующие стандарту C++03, обычно реализуют `std::sort` с использованием быстрой сортировки, обычно с некоторыми хитрыми вариантами выбора медианы, чтобы уменьшить вероятность получения времени работы быстрой сортировки $O(n^2)$ в наихудшем случае. C++11 требует, чтобы наихудшая производительность сортировки была $O(n \cdot log_2 n)$. Реализации, соответствующие стандарту C++11, как правило, являются гибридными, такими как тимсорт или интроспективная сортировка.

Алгоритм `std::stable_sort` обычно представляет собой некоторый вариант сортировки слиянием. Своеобразная формулировка стандарта гласит, что `std::stable_sort` имеет время работы $O(n \cdot log_2 n)$, если может быть выделен

достаточный объем дополнительной памяти, в противном же случае его время работы — $O(n \cdot (log_2 n)^2)$. Типичная реализация заключается в использовании сортировки слиянием, если глубина рекурсии не слишком велика, и пирамидальной сортировки — в противном случае.

Важность устойчивой сортировки заключается в том, что программа может отсортировать диапазон записей по каждому из нескольких критериев (как, например, имя, а затем фамилия) и получить записи, отсортированные по второму критерию, а затем по первому критерию в рамках второго (например, по фамилии, а затем по имени в пределах одинаковых фамилий). Этим свойством обладает только устойчивая сортировка. Это дополнительное свойство оправдывает наличие двух видов сортировки.

В табл. 9.2 приведены результаты сортировки 100 тысяч случайным образом сгенерированных записей "ключ/значение" в контейнере `std::vector`. Интересно, что `std::stable_sort()` имеет более высокую производительность, чем `std::sort()`. Я также тестировал сортировку для уже отсортированной таблицы. Сортировку в других структурах данных я рассматриваю в главе 10, "Оптимизация структур данных".

Таблица 9.2. Результаты изучения производительности сортировок

`std::vector`, 100k элементов, VS2010, i7	Время, мс
`std::sort()` для вектора	18,61
`std::sort()` для отсортированного вектора	3,77
`std::stable_sort()` для вектора	16,08
`std::stable_sort()` для отсортированного вектора	5,01

Последовательный контейнер `std::list` обеспечивает только двунаправленные итераторы, поэтому с ним алгоритм `std::sort()` будет работать за время $O(n^2)$. Однако `std::list` предоставляет функцию-член `sort()` со временем работы $O(n \cdot log_2 n)$.

Упорядоченные ассоциативные контейнеры хранят свои данные в отсортированном порядке, поэтому их сортировать не нужно. Неупорядоченные ассоциативные контейнеры поддерживают свои данные в определенном порядке, который не представляет интереса для пользователей. Такие контейнеры не могут быть отсортированы.

Заголовочный файл `<algorithm>` стандартной библиотеки C++ содержит фрагменты различных алгоритмов сортировки, из которых могут быть построены более сложные виды сортировки для входных данных, обладающих дополнительными специальными свойствами.

- `std::heap_sort` преобразует диапазон, обладающий свойством пирамиды, в отсортированный диапазон. Сортировка `heap_sort` не является устойчивой.
- `std::partition` выполняет базовое действие по разбиению диапазона для быстрой сортировки.
- `std::merge` выполняет базовое действие сортировки слиянием.
- Член `insert` различных последовательных контейнеров выполняет базовое действие для сортировки вставками.

Резюме

- *Сочетание различных возможностей в C++ обеспечивает огромное множество вариантов реализации, начиная от полной автоматизации и выразительности, с одной стороны, и заканчивая полным тонким контролем над производительностью — с другой. Именно эта возможность выбора позволяет настраивать программы C++ для удовлетворения требованиям производительности.*
- *В большинстве случаев имеется такое огромное количество возможностей оптимизации, что человеческая память не в состоянии достоверно помнить их все. Бумага имеет лучшую память.*
- *В тесте поиска в таблице* `std::unordered_map` *с 26 строковыми ключами только на 25% быстрее, чем поиск в аналогичном* `std::map`*. С учетом рекламы выигрыша производительности при хешировании этот результат выглядит более чем скромно.*
- *Цена абстракций Степанова является платой за использование таких высокопродуктивных инструментов, как алгоритмы стандартной библиотеки C++.*

ГЛАВА 10

Оптимизация структур данных

В прекрасной вещи жизнь и радость вечны.[1]
— Джон Китс (John Keats) (1818)

Если вы не прекращаете восторгаться классами контейнеров стандартной библиотеки C++ (ранее — стандартной библиотеки шаблонов, или STL), возможно, сейчас вам все же придется это сделать. Во времена, когда она была введена в черновик стандарта в 1994 году, *стандартная библиотека шаблонов Степанова была первой возможностью повторного использования библиотеки эффективных контейнеров и алгоритмов.* До появления STL каждый проект разрабатывал собственный связанный список и реализации двоичного дерева, возможно, адаптируя исходный код других пользователей. Язык программирования C не имеет аналогов этой библиотеке. Контейнеры стандартной библиотеки последние 20 лет позволяют многим программистам забыть собственные алгоритмы и структуры данных и выбирать готовые из меню готовых контейнеров стандартной библиотеки.

Знакомство с контейнерами стандартной библиотеки

Есть много вещей, которые делают контейнеры стандартной библиотеки C++ привлекательными, например единообразное именование и согласованное и последовательное понятие итераторов для обхода контейнеров. Но для оптимизации особенно важное значение имеют некоторые иные свойства, к которым относятся следующие:

- гарантии производительности (в O-терминах) вставки и удаления;
- амортизированная константная стоимость добавления к последовательным контейнерам;
- возможность тонкого контроля распределения динамической памяти контейнером.

Контейнеры стандартной библиотеки C++ кажутся похожими настолько, что могут заменить друг друга несмотря на их явно различные реализации. Но это иллюзия. Контейнеры стандартной библиотеки слишком стары, чтобы покупать в магазине

[1] Перевод А. Гастева.

горячительное. Как и другие части C++, контейнеры стандартной библиотеки развивались независимо один от другого. Их интерфейсы перекрываются только частично. Производительность одной и той же операции меняется от контейнера к контейнеру. Что еще более важно, семантика некоторых функций-членов варьируется от одного контейнера к другому, даже если они имеют одинаковые имена. Разработчик должен досконально знать каждый класс контейнера, чтобы понимать, как оптимально их использовать.

Последовательные контейнеры

Последовательные контейнеры `std::string`, `std::vector`, `std::deque`, `std::list` и `std::forward_list` хранят элементы в том порядке, в котором они были вставлены в контейнер. Соответственно, каждый контейнер имеет начало и конец. Все последовательные контейнеры имеют средство для вставки элементов и все они, за исключением `std::forward_list`, имеют функции-члены с константным временем работы для вставки элементов в конец контейнера. Однако только `std::deque`, `std::list` и `std::forward_list` могут эффективно вставлять элементы в начало контейнера.

Элементы в `std::string`, `std::vector` и `std::deque` пронумерованы от 0 до *size*–1 и могут быть эффективно получены с помощью индекса. Контейнеры `std::list` и `std::forward_list` различны, но оба не имеют оператора индексации.

Контейнеры `std::string`, `std::vector` и `std::deque` построены вокруг внутреннего "массивообразного" скелета. При вставке элемента в контейнер все элементы, следующие за ним, должны быть перенесены в соседние местоположения в массиве, так что стоимость вставки в произвольное место контейнера (но не в его конец!) составляет $O(n)$, где n — количество элементов в контейнере. При вставке элементов внутренние массивы могут быть перераспределены, что приводит к недействительности всех имеющихся итераторов и указателей. Напротив, у `std::list` и `std::forward_list` недействительными становятся только итераторы и указатели на удаленные из списка элементы. Два экземпляра списков `std::list` и `std::forward_list` могут даже соединяться вместе без потери корректности итераторов. Вставка в средину `std::list` или `std::forward_list` выполняется за константное время в предположении, что итератор уже указывает на точку вставки.

Ассоциативные контейнеры

Ассоциативные контейнеры хранят вставленные в них элементы на основе некоторого свойства упорядочения элементов, а не в порядке их вставки. Все ассоциативные контейнеры обеспечивают эффективный доступ к содержащимся в них элементам за время, меньшее линейного.

Отображения и множества предоставляют различные интерфейсы. Отображения хранят отдельно определенные ключ и значение и, таким образом, обеспечивают эффективное отображение ключа на значение. Множества хранят уникальные упорядоченные значения и обеспечивают эффективную проверку наличия значения в множестве. Мультиотображения отличаются от отображений (как и мультимножества от множеств) только тем, что допускают вставку нескольких элементов, которые при сравнении считаются равными.

В плане реализации имеется четыре упорядоченных ассоциативных контейнера: `std::map`, `std::multimap`, `std::set` и `std::multiset`. Упорядоченные ассоциативные контейнеры требуют определения упорядочивающего отношения `operator<()` для ключей (`std::map`) или самих элементов (`std::set`). Упорядоченные ассоциативные контейнеры реализованы в виде сбалансированных бинарных деревьев. В сортировке упорядоченных ассоциативных контейнеров необходимости нет. Обход элементов такого контейнера выполняется в порядке их упорядочения. Вставка или удаление элементов имеет амортизированное время выполнения $O(log_2 n)$, где n — количество элементов в контейнере.

Хотя для отображений и множеств возможны различные реализации, на практике все четыре ассоциативных контейнера реализуются как отдельные фасады поверх одной и той же структуры данных сбалансированного бинарного дерева. Это верно как минимум для всех компиляторов, которые я использовал. Поэтому я не предоставляю отдельные результаты хронометража для мультиотображений, множеств и мультимножеств.

Со стандартом C++11 в стандартную библиотеку вошли также четыре *неупорядоченных ассоциативных контейнера*: `std::unordered_map`, `std::unordered_multimap`, `std::unordered_set` и `std::unordered_multiset`. Эти контейнеры появились в Visual C++ еще в версии 2010 года. Неупорядоченные ассоциативные контейнеры требуют определения только равенства отношения для ключей (`std::unordered_map`) или элементов (`std::unordered_set`). Неупорядоченные ассоциативные контейнеры реализованы в виде хеш-таблиц. Порядок обхода неупорядоченного ассоциативного контейнера не определен. Вставка или удаление элементов в среднем выполняется за константное время, хотя время вставки в наихудшем случае — $O(n)$.

Ассоциативные контейнеры являются очевидным выбором для таблиц поиска. Разработчик может также хранить элементы, имеющие отношение упорядочения, в последовательном контейнере, после сортировки которого поиск становится относительно эффективным — со временем работы $O(log_2 n)$.

Эксперименты с контейнерами стандартной библиотеки

Я создал несколько разнотипных контейнеров со 100 тысячами элементов в каждом и измерил производительность вставки, удаления и обхода элементов. В последовательных контейнерах я также измерил стоимость сортировки.

Это количество элементов является достаточным, чтобы амортизированная стоимость вставки приблизилась к своему асимптотическому поведению, указанному для каждого контейнера — 100 тысяч элементов достаточно, чтобы основательно загрузить кеш-память. С любой точки зрения это никак не маленький контейнер, но и не такой непомерно большой, чтобы быть крайне редким.

Производительность в терминах "большого О" — это не вся история. Я обнаружил, что одни контейнеры оказывались во много раз быстрее других, даже когда та или иная операция имела асимптотическую стоимость O(1) для обоих сравниваемых контейнеров.

Я также обнаружил, что неупорядоченные отображения с их стоимостью поиска O(1) быстрее, чем отображения, но не с таким большим отрывом, какого я ожидал. Кроме того, значительной оказывается стоимость памяти для получения такой производительности.

Большинство типов контейнеров предоставляют несколько способов вставки элементов. Я обнаружил, что одни из них на 10–15% быстрее других, причем причина этого не ясна.

Стоимость вставки 100 тысяч элементов в контейнер состоит из двух частей: стоимость выделения памяти и стоимость копирующего создания элементов в памяти. Стоимость выделения памяти фиксирована для элементов определенного размера, в то время как стоимость копирующего построения не ограничена и зависит от прихоти программиста. В случае очень дорогого копирующего конструктора стоимость копирования элементов будет доминировать над стоимостью построения контейнера. В этом случае все контейнеры будут давать примерно одинаковую производительность при тестировании вставок.

Большинство типов контейнеров также предоставляют несколько способов перебора элементов. Здесь я также нашел, что одни способы демонстрируют бо́льшую скорость по сравнению с другими при отсутствии какой-то очевидной причины для такого поведения. Интересно, что разница во времени обхода разных контейнеров оказалась меньшей, чем я ожидал.

Я проверил стоимость сортировки последовательных контейнеров, чтобы понять, могут ли они заменить ассоциативные контейнеры в приложениях, выполняющих поиск в таблицах. Одни контейнеры выполняют сортировку элементов в процессе вставки; другие контейнеры нельзя сортировать вовсе.

Полученные мною результаты достаточно значительны, чтобы быть интересными, но, вероятно, достаточно недолговечны. По мере развития реализаций с течением времени самым быстрым методом может стать другой. Например, алгоритм `stable_sort()` постоянно превосходит алгоритм `sort()`. Я подозреваю, что, когда `stable_sort()` был добавлен в библиотеку алгоритмов, дела обстояли иначе.

Тип данных элемента

Для элементов в последовательных контейнерах я использовал структуру "ключ/значение". Ассоциативные контейнеры создают очень похожие структуры, построенные поверх `std::pair`:

```
struct kvstruct {
    char key[9];
    unsigned value;   // Может быть чем угодно
    kvstruct(unsigned k) : value(k)
    {
        if (strcpy_s(key, stringify(k)))
            DebugBreak();
    }
    bool operator<(kvstruct const& that) const {
        return strcmp(this->key, that.key) < 0;
    }
    bool operator==(kvstruct const& that) const {
        return strcmp(this->key, that.key) == 0;
    }
};
```

Копирующий конструктор для этого класса создается компилятором, так как в данном случае его работа нас устраивает; но в общем случае это нетривиальная

операция, которой недостаточно побитового копирования содержимого одного объекта `kvstruct` в другой. Как и ранее, моей целью было сделать копирование и сравнение хотя бы немного дорогими для имитации реальных структур данных.

Сами ключи представляют собой строки в стиле C с завершающими нулевыми символами, состоящие из семи цифр. Ключи были получены с помощью равномерного случайного распределения с использованием заголовочного файла C++ `<random>`. То же значение сохранялось и как целое беззнаковое число в качестве поля `value` элемента. Повторяющиеся ключи устранялись, чтобы получить исходный вектор из 100 тысяч различных значений в произвольном порядке.

Примечания о проектировании эксперимента

Некоторые контейнеры очень недороги при вставке или обходе, даже со 100 тысячами элементов. Чтобы получить тест, работающий измеримое количество времени, я должен был повторить вставку или обход 1000 раз. Но здесь есть проблема. Каждый раз при вставке элементов в контейнер необходимо также удалить из контейнера имеющиеся в нем элементы, что влияет на общее время работы. Например, следующий код измеряет стоимость присваивания одного вектора другому. Это неизбежно добавляет стоимость создания новой копии `random_vector` и последующего ее удаления:

```
{   Stopwatch sw("Присваивание вектора вектору + удаление×1000");
    std::vector<kvstruct> test_container;
    for (unsigned j = 0; j < 1000; ++j) {
        test_container = random_vector;
        std::vector<kvstruct>().swap(test_container);
    }
}
```

Для получения отдельных стоимостей присваивания и удаления я создал более сложную версию кода, по отдельности накапливающую время, затраченное на создание новой копии и ее удаление:

```
{   Stopwatch sw("Присваивание вектора вектору", false);
    Stopwatch::tick_t ticks;
    Stopwatch::tick_t assign_x_1000 = 0;
    Stopwatch::tick_t delete_x_1000 = 0;
    std::vector<kvstruct> test_container;
    for (unsigned j = 0; j < 1000; ++j) {
        sw.Start("");
        test_container = random_vector;
        ticks = sw.Show("");
        assign_x_1000 += ticks;
        std::vector<kvstruct>().swap(test_container);
        delete_x_1000 += sw.Stop("") - ticks;
    }
    std::cout << " присваивание вектора вектору x 1000: "
              << Stopwatch::GetMs(assign_x_1000)
              << "мс" << std::endl;
    std::cout << " удаление вектора x 1000: "
              << Stopwatch::GetMs(delete_x_1000)
              << "мс" << std::endl;
}
```

Первая инструкция в цикле, `sw.Start("");`, запускает секундомер без вывода на экран. Следующая инструкция, `test_container = random_vector;`, тратит время на копирование вектора. Третья инструкция, `ticks = sw.Show("");`, устанавливает переменную `ticks` равной прошедшему до этого момента времени.

Что же за значение находится в переменной `ticks`? Источник тактов в экземпляре `sw` класса `Stopwatch` имеет длительность такта, равную 1 мс. Время, затраченное на присваивание, гораздо меньше 1 мс, поэтому в основном значение в `ticks` равно 0. Но не всегда. Часы идут независимо от кода, так что иногда возникает ситуация, например, когда секундомер начал отсчет в начале 987-й микросекунды миллисекундного такта, так что к концу завершения копирования засчитывается прошедший такт и значение `ticks` равно 1. Если присваивание занимает 500 мкс, это происходит примерно в половине случаев. Если же присваивание занимает 10 мкс, то `ticks` равно 1 примерно в 1% случаев. Учитывая достаточное количество повторений цикла, мы получаем точное совокупное время присваивания.

Значения `ticks` накапливаются в переменной `assign_x_1000`, которая подсчитывает время, требуемое для присваиваний. Затем инструкция `std::vector().swap(test_container);` удаляет содержимое вектора `test_container`. Наконец инструкция `delete_x_1000 += sw.Stop("") - ticks;` получает значения еще от одного счетчика тактов, который также равен либо нулю, либо единице, вычитает счетчик тактов от конца присваивания и накапливает разность в переменной `delete_x_1000`. Измеренная стоимость 1000-кратного удаления вектора составляет 111 мс, или 0,111 мс на одно удаление.

Вооружившись стоимостью удаления контейнера со 100 тысячами записей, время выполнения оставшегося кода можно получить с помощью арифметической операции. Приведенный далее код представляет собой еще один цикл, который 1000 раз заполняет контейнер и который также должен включать стоимость удаления контейнера:

```
{   Stopwatch sw("vector iterator insert() + delete x 1000");
    std::vector<kvstruct> test_container;
    for (unsigned j = 0; j < 1000; ++j) {
        test_container.insert(test_container.begin(),
                              random_vector.begin(),
                              random_vector.end());
        std::vector<kvstruct>().swap(test_container);
    }
}
```

Тестовый запуск приведенного выше кода за 696 мс 1000 раз заполнял и удалял контейнер. Если принять стоимость 1000 удалений вектора равной измеренной выше величине 111 мс, то стоимость одного вызова `insert()` составит (696–111)/1000 = 0,585 мс.

Примечания о современном кодировании на C++

В состав стандартной библиотеки C++ входит малозаметная библиотека `<random>` для генерации случайных чисел. После того как я обнаружил эту библиотеку, она стала одним из моих любимых инструментов для генерации ключей для их случайного поиска. Например, в примере 10.1 приведен код, который я использовал для генерации случайных строк для моих тестов с контейнерами.

Пример 10.1. Создание уникальных экземпляров **`kvstruct`**

```
# include <random>
// Построение вектора с count элементами,
// содержащего уникальные случайные строки
void build_rnd_vector(std::vector<kvstruct>& v, unsigned count){
    std::default_random_engine e;
    std::uniform_int_distribution<unsigned> d(count, 10*count-1);
    auto randomizer = std::bind(d,e);
    std::set<unsigned> unique;
    v.clear();
    while (v.size() < count) {
        unsigned rv = randomizer();
        if (unique.insert(rv).second == true) { // Элемент вставлен
            kvstruct keyvalue(rv);
            v.push_back(keyvalue);
        }
    }
}
```

Первая строка `build_rnd_vector()` создает *генератор* случайных чисел, основной источник случайности. Вторая строка создает *распределение* случайных чисел, объект, который преобразует последовательности случайных чисел, получаемых от генератора, в последовательности чисел, которые отвечают некоторому распределению вероятностей. В данном случае распределение является равномерным, а это означает, что с равной вероятностью может встретиться любое значение — от минимального значения `count` до максимального значения `10*count-1`. Таким образом, если `count` равно 100 000, значения, предоставляемые распределением, будут варьироваться от 100 000 до 999 999 (т.е. все они будут шестизначными). Третья строка создает объект, который применяет генератор в качестве аргумента распределения, таким образом, чтобы вызов оператора `operator()` объекта генерировал случайное число.

Все генераторы документированы и имеют известные свойства. Есть даже генератор, называемый `std::random_device`, который создает значения из источника истинной случайности, если таковой доступен.

Распределения обеспечивают мощность этой библиотеки. Например, ниже приведено несколько полезных распределений.

```
std::uniform_int_distribution<unsigned> dice(1, 6);
```

Распределение выпадений честной шестигранной кости, которое с одинаковой вероятностью дает числа от 1 до 6. Кости с 4, 20 или 100 сторонами можно моделировать путем изменения второго аргумента.

```
std::binomial_distribution<unsigned> coin(1, 0.5);
```

Распределение выпадений честной монеты, которое дает значение 0 или 1 с одинаковой вероятностью. Не совсем честную монету можно моделировать путем корректировки второго аргумента, делая его отличным от 0,5.

```
std::normal_distribution<double> iq(100.0, 15.0);
```

Распределение оценочных данных о количестве населения, возвращающее значения типа `double`, такие, что между 85,0 и 115,0 находится около двух третей результатов.

Для более изысканных ценителей есть такие распределения, как распределение Пуассона или экспоненциальные распределения для построения моделей событий, а также ряд иных распределений.

`std::vector` и `std::string`

"Список характеристик" этих структур данных имеет следующий вид.

- Последовательный контейнер.
- Время вставки: амортизированная вставка в конец — O(1), в произвольное место — O(n).
- Время индексации: по позиции, за время O(1).
- Сортировка за время O($n \cdot log_2 n$).
- Поиск за время O($log_2 n$) в отсортированном контейнере, в противном случае за время O(n).
- Итераторы и ссылки становятся недействительными при перераспределении памяти для внутреннего массива.
- Итераторы обходят элементы от начала до конца и от конца до начала.
- Разумный контроль над выделенной памятью независимо от размера.

Исторически `std::string` было разрешено иметь иные реализации, но в C++11 определение строки зафиксировано. Реализация Visual Studio может быть производным от `std::vector` классом со специализированными функциями-членами для работы со строками. Комментарии об `std::vector` в равной степени относятся к `std::string` в Visual Studio.

`std::vector` представляет собой массив с динамически изменяемым размером (рис. 10.1). Элементы массива являются экземплярами параметра типа шаблона `T`, которые создаются копированием в вектор. Хотя копирующие конструкторы

элементов могут выделять память для членов, запросы к диспетчеру памяти для перераспределения внутреннего буфера при добавлении элементов может делать только `std::vector`. Такая плоская структура делает `std::vector` необычайно эффективным. Создатель C++ Бьярне Страуструп (Bjarne Stroustrup) рекомендует использовать в качестве контейнера `std::vector`, если только у вас нет серьезной причины выбрать другой контейнер. В этом разделе будет показано, почему это так.

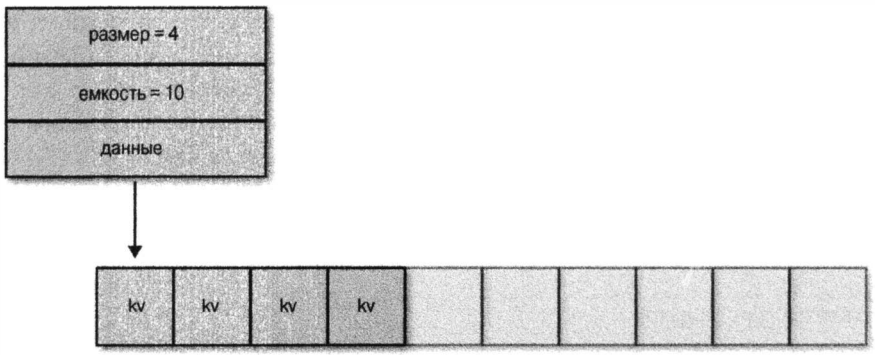


Рис. 10.1. Возможная реализация `std::vector`

Многие операции над `std::vector` являются эффективными в терминах "большого О", выполняясь за константное время. Среди этих операций — добавление нового элемента в конец вектора и получение ссылки на *i*-й элемент. Из-за простой внутренней структуры вектора эти операции выполняются довольно быстро и в абсолютном выражении. Итераторы `std::vector` представляют собой итераторы с произвольным доступом, а это означает, что вычисление расстояния между двумя итераторами в одном векторе может быть выполнено за константное время. Это свойство делает эффективными поиск и сортировку "разделяй и властвуй" в `std::vector`.

Следствия перераспределения для производительности

`std::vector` имеет *размер*, который описывает, как много элементов в настоящее время имеется в векторе, и *емкость*, который говорит о том, насколько велик внутренний буфер, содержащий элементы. Когда размер равен емкости, любые дальнейшие вставки приводят к выполнению дорогостоящего расширения: перераспределение внутреннего буфера и копирование элементов вектора в новое место в памяти, которые делают недействительными все итераторы и ссылки на старый буфер. Когда перераспределение становится необходимым, емкость нового буфера получается путем умножения прежней на некоторый коэффициент. В результате агрегированная стоимость вставки элемента оказывается константной, хотя одни вставки являются дорогостоящими, а другие — нет.

Один секрет эффективного использования `std::vector` заключается в том, что емкость может быть зарезервирована заранее путем вызова `void reserve(size_t n)`, таким образом предотвращая выполнение ненужных циклов перераспределения и копирования.

Еще одним секретом эффективности `std::vector` является то, что при удалении элементов он не возвращает автоматически память диспетчеру памяти. Если программа поместит миллион элементов в вектор, а затем удалит их, вектор останется с памятью, в которой безо всяких распределений можно разместить миллион элементов. Разработчики должны иметь в виду этот факт, используя `std::vector` в ограниченных средах.

На емкость `std::vector` влияют некоторые его функции-члены, но стандарт очень уклончив в отношении каких-либо гарантий. `void clear()` сбрасывает *размер* контейнера до нуля, но не гарантирует перераспределение его внутреннего буфера для уменьшения *емкости*. В C++11 и в Visual Studio 2010 `void shrink_to_fit()` является подсказкой вектору о сокращении емкости в соответствии с текущим размером, но выполнение перераспределения не является обязательным.

Чтобы гарантировать освобождение памяти вектора во всех версиях C++, воспользуйтесь следующим трюком:

```
std::vector<Foo> x;
...
vector<Foo>().swap(x);
```

Эта инструкция создает временный пустой вектор, обменивает его содержимое с содержимым вектора `x`, а затем удаляет временный вектор, так что диспетчер памяти освобождает всю память, принадлежавшую `x`, и последний становится пустым вектором.

Вставка и удаление в `std::vector`

Существует несколько способов заполнения вектора данными. Я исследовал стоимость построения вектора со 100 тысячами экземпляров `kvstruct`, чтобы найти, определенно, более быстрые и более медленные методы.

Самый быстрый способ заполнения вектора — это его присваивание:

```
std::vector<kvstruct> test_container, random_vector;
...
test_container = random_vector;
```

Присваивание является очень эффективным, поскольку знает размер копируемого вектора и ему требуется вызвать диспетчер памяти только один раз для создания внутреннего буфера памяти в целевом векторе. Хронометраж показал, что вектор со 100 тысячами записей копируется за 0,445 мс.

Если данные находятся в другом контейнере, скопировать их в вектор можно с помощью `std::vector::insert()`:

```
std::vector<kvstruct> test_container, random_vector;
...
test_container.insert(test_container.end(),
                      random_vector.begin(),
                      random_vector.end());
```

Хронометраж показал, что данная инструкция копирует вектор со 100 тысячами записей за 0,696 мс.

Функция-член `std::vector::push_back()` может эффективно (т.е. за константное время) добавить новый элемент в конец вектора. Так как элементы находятся в другом векторе, их нужно как-то получать. Для этого есть три способа.

- Использовать итератор вектора.

```
std::vector<kvstruct> test_container, random_vector;
...
for (auto it=random_vector.begin(); it!=random_vector.end(); ++it)
    test_container.push_back(*it);
```

- Использовать функцию-член `std::vector::at()`.

```
std::vector<kvstruct> test_container, random_vector;
...
for (unsigned i = 0; i < nelts; ++i)
    test_container.push_back(random_vector.at(i));
```

- Код может использовать индекс вектора.

```
std::vector<kvstruct> test_container, random_vector;
...
for (unsigned i = 0; i < nelts; ++i)
    test_container.push_back(random_vector[i]);
```

В моих тестах эти три метода дали схожие времена — 2,26, 2,05 и 1,99 мс соответственно. Однако это в шесть раз превосходит количество времени, требуемого оператором простого присваивания.

Причина, по которой этот код оказывается более медленным, — он вставляет элементы в вектор по одному. Вектор не знает, сколько элементов будет в него вставлено, и поэтому наращивает свой внутренний буфер постепенно. Несколько раз в течение цикла вектор перераспределяет внутренний буфер и копирует все элементы в новый буфер. `std::vector` гарантирует, что амортизированное время работы `push_back()` константное, но это не означает, что стоимости у этой функции-члена нет.

Разработчик может использовать дополнительные знания для повышения эффективности этого цикла, предварительно запросив буфер достаточного размера, чтобы поместить в нем все элементы. Вариант этого кода с итераторами выглядит следующим образом:

```
std::vector<kvstruct> test_container, random_vector;
...
test_container.reserve(nelts);
for (auto it=random_vector.begin(); it != random_vector.end(); ++it)
    test_container.push_back(*it);
```

Такой цикл выполняется за респектабельные 0,674 мс.

Есть и другие способы вставки элементов в вектор. Можно воспользоваться другим вариантом функции-члена `insert()`:

```
std::vector<kvstruct> test_container, random_vector;
...
for (auto it=random_vector.begin(); it != random_vector.end(); ++it)
    test_container.insert(test_container.end(), *it);
```

Этот код должен бы иметь ту же стоимость, что и `push_back()`, но это не так (по крайней мере в Visual Studio 2010). Все три варианта (с итераторами, `at()` и индексами) выполняются примерно за 2,7 мс. Резервирование внутреннего буфера уменьшает это время до 1,45 мс, но эта величина не может конкурировать с любым из предыдущих результатов.

Наконец, мы добрались до самого больного места `std::vector`: до вставки элементов в начало вектора. `std::vector` не предоставляет член `push_front()`, потому что его стоимость равна O(*n*). Вставка в начало вектора неэффективна, потому что, чтобы освободить место для новой записи, должен быть скопирован каждый элемент вектора. И это действительно неэффективная операция. Цикл

```
std::vector<kvstruct> test_container, random_vector;
...
for (auto it=random_vector.begin(); it != random_vector.end(); ++it)
    test_container.insert(test_container.begin(), *it);
```

выполняется за 8065 мс. Да, здесь нет опечатки — этот цикл выполняется *почти в три тысячи раз дольше* вставки в конец вектора.

Таким образом, чтобы эффективно заполнить вектор, используйте присваивание, `insert()` с итераторами из другого контейнера, `push_back()` и `insert()` в конец вектора — в указанном порядке.

Итерирование `std::vector`

Обойти вектор с посещением каждого элемента — недорого. Но, как и в случае со вставкой, стоимость доступных методов существенно различается.

Есть три способа перебора элементов вектора: с помощью итератора, функции-члена `at()` и индекса. Если тело цикла стоит дорого, разница в стоимости различных методов обхода становится несущественной. Однако разработчики часто выполняют только простые быстрые действия с данными из каждого элемента. В приведенном примере цикл суммирует элементы вектора, что занимает очень незначительное количество времени (а также заставляет компилятор воздержаться от оптимизации, которая выбросила бы прочь весь цикл как ничего не делающий):

```
std::vector<kvstruct> test_container;
...
unsigned sum = 0;
for (auto it=test_container.begin(); it!=test_container.end(); ++it)
    sum += it->value;

std::vector<kvstruct> test_container;
...
unsigned sum = 0;
for (unsigned i = 0; i < nelts; ++i)
    sum += test_container.at(i).value;

std::vector<kvstruct> test_container;
...
unsigned sum = 0;
for (unsigned i = 0; i < nelts; ++i)
    sum += test_container[i].value;
```

Разумно ожидать, что эти циклы будут практически эквивалентны по стоимости, но на самом деле они таковыми не являются. Версия с итератором выполняется за 0,236 мс. Версия с `at()` немногим лучше, выполняясь за 0,230 мс. Но, как и в случае со вставкой, версия с индексом в Visual Studio 2010 оказывается на 46% эффективнее, чем с итератором.

Сортировка `std::vector`

Чтобы использовать бинарный поиск в векторе, вектор следует отсортировать. Стандартная библиотека C++ имеет два алгоритма сортировки, `std::sort()` и `std::stable_sort()`. Оба они имеют время работы $O(n \cdot log_2 n)$, если итераторы контейнера являются итераторами с произвольным доступом, как в `std::vector`. Оба алгоритма работают несколько быстрее с данными, которые уже отсортированы. Сортировка выполняется с помощью одного краткого вызова:

```
std::vector<kvstruct> sorted_container, random_vector;
...
sorted_container = random_vector;
std::sort(sorted_container.begin(), sorted_container.end());
```

Результаты подытожены в табл. 10.1.

Таблица 10.1. Стоимость сортировки вектора со 100 тысячами элементов

std::vector	VS2010, i7, 100k элементов, мс
`std::sort()` вектор	18,61
`std::sort()` отсортированный вектор	3,77
`std::stable_sort()` вектор	16,08
`std::stable_sort()` отсортированный вектор	5,01

Поиск в `std::vector`

Приведенный далее фрагмент программы ищет каждый ключ из `random_vector` в отсортированном контейнере:

```
std::vector<kvstruct> sorted_container, random_vector;
...
for (auto it=random_vector.begin(); it!=random_vector.end(); ++it) {
    kp = std::lower_bound(sorted_container.begin(),
                          sorted_container.end(),
                          *it);
    if (kp != sorted_container.end() && *it < *kp)
        kp = sorted_container.end();
}
```

Эта программа выполняет поиск 100 тысяч ключей в отсортированном векторе за 28,92 мс.

`std::deque`

"Список характеристик" этой структуры данных имеет следующий вид.

- Последовательный контейнер.
- Время вставки: в конец или в начало — O(1), в произвольное место — O(n).
- Время индексации: по позиции, за время O(1).
- Сортировка за время O($n \cdot log_2 n$).
- Поиск за время O($log_2 n$), если контейнер отсортирован, иначе за O(n).
- Итераторы и ссылки становятся недействительными после перераспределения внутреннего массива.
- Итераторы обходят элементы от начала до конца и от конца до начала.

`std::deque` — специализированный контейнер для создания очереди (первым вошел — первым вышел, FIFO). Вставка и удаление в любом конце выполняются за константное время. Индексация также является операцией с константным временем выполнения. Итераторы `std::deque`, как и итераторы `std::vector`, являются итераторами с произвольным доступом, так что контейнер `std::deque` может быть отсортирован за время O($n \cdot log_2 n$).

Поскольку `std::deque` дает те же гарантии производительности (в терминах "большого O"), что и `std::vector`, а также обеспечивает константное время вставки с обоих концов, так и хочется спросить — а зачем нам тогда `std::vector`? Однако коэффициент пропорциональности во всех этих операциях у дека больше, чем у вектора. Измеренная производительность основных операций с участием деков в 3–10 раз меньше производительности у соответствующих операций с векторами. Светлым лучом у деков являются только обход, сортировка и поиск, которые примерно на 30% медленнее, чем у векторов.

`std::deque` обычно реализуется как массив массивов (рис. 10.2). Для получения элемента из дека необходимы два косвенных обращения, что уменьшает локальность кеша; стоимость более частых вызовов диспетчера памяти у дека больше, чем у `std::vector`.

Для вставки элемента в любой конец дека может потребоваться более двух вызовов аллокатора: для добавления еще одного блока элементов и (гораздо реже) для расширения базового массива дека. Такое поведение аллокатора более сложное, чем у векторов, и, таким образом, о нем труднее судить, чем о поведении аллокаторов в векторах. `std::deque` не предлагает каких-либо эквивалентов функции-члена `std::vector::reserve()` для выделения достаточного места для размещения элементов во внутренних структурах данных. Дек может показаться очевидным кандидатом на реализацию очереди FIFO. Существует даже шаблон адаптера контейнера, именуемый `std::queue`, для которого дек является реализацией по умолчанию. Однако нет никакой гарантии, что производительность распределения памяти будет при таком применении дека достаточно хорошей.

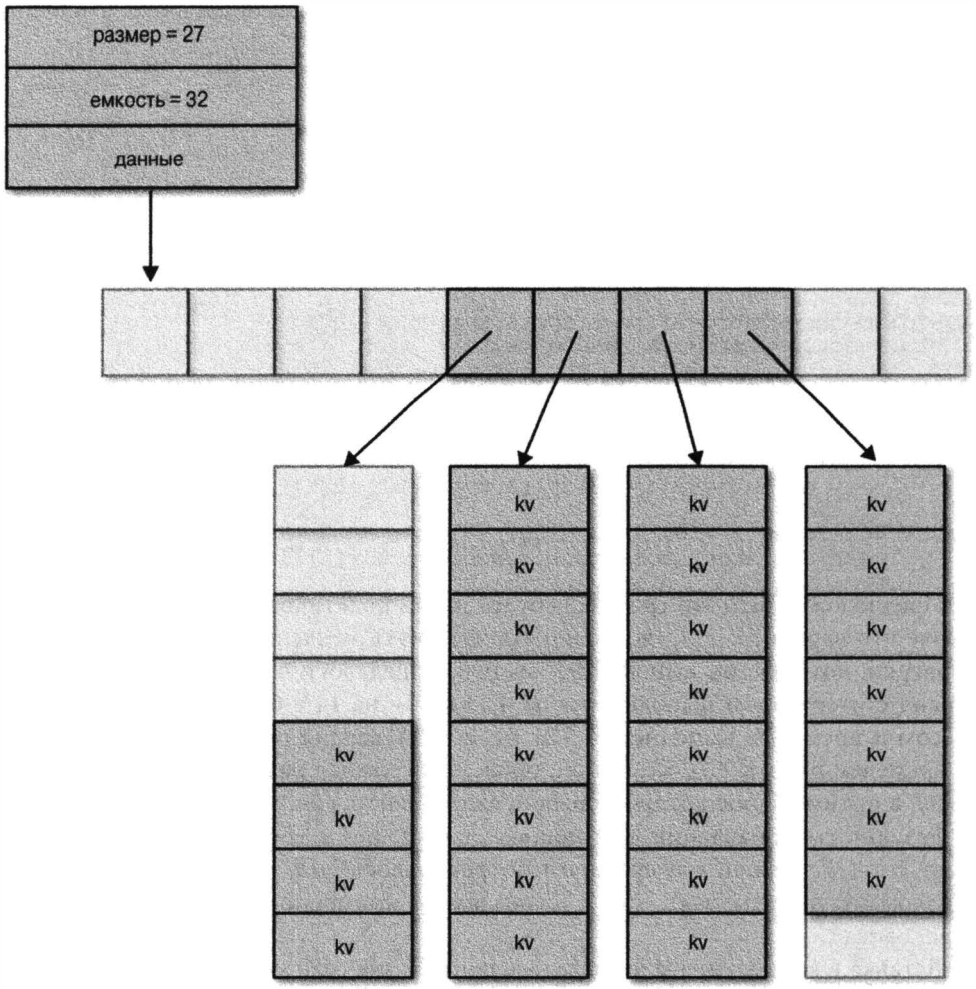


Рис. 10.2. Возможная реализация `std::deque` *(состояние после нескольких вставок и удалений)*

Вставка и удаление в `std::deque`

`std::deque` обеспечивает тот же интерфейс для вставки, что и `std::vector`, плюс функцию-член `push_front()`.

Показанная ниже операция присваивания одного дека другому выполняется за 5,70 мс:

```
std::deque<kvstruct> test_container;
std::vector<kvstruct> random_vector;
...
test_container = random_vector;
```

Вставка в дек с использованием пары итераторов, показанная ниже, выполняется за 5,28 мс:

```
std::deque<kvstruct> test_container;
std::vector<kvstruct> random_vector;
...
test_container.insert(test_container.end(),
                      random_vector.begin(),
                      random_vector.end());
```

Вот три способа копирования элементов из вектора в дек с использованием `push_back()`:

```
std::deque<kvstruct> test_container;
std::vector<kvstruct> random_vector;
...
for (auto it=random_vector.begin(); it!=random_vector.end(); ++it)
    test_container.push_back(*it);

for (unsigned i = 0; i < nelts; ++i)
    test_container.push_back(random_vector.at(i));

for (unsigned i = 0; i < nelts; ++i)
    test_container.push_back(random_vector[i]);
```

Предположив, что эти три цикла будут иметь практически одинаковую стоимость (разве что вариант с `at()` должен быть чуть-чуть медленнее из-за дополнительной проверки, которую он выполняет), вы будете недалеки от истины. Действительно, версия с итератором выполняется за 4,33 мс — на 14% быстрее, чем версия с индексом и временем выполнения 5,01 мс, а `at()`-версия располагается между ними со временем работы 4,76 мс. Это не такая уж огромная разница, по крайней мере не такая, в оптимизацию которой обычно вкладываются большие усилия.

Результаты добавления с помощью `push_front()` похожи на результаты для `push_back()`. Версии с итератором потребовалось 5,19 мс, с индексом — 5,55 мс. В целом результаты `push_front()` оказались примерно на 15% медленнее, чем у `push_back()`.

Вставка в конце и вначале оказалась почти в два раза медленнее, чем использование `push_back()` и `push_front()` соответственно.

Теперь взглянем на производительность `std::vector` по сравнению с `std::deque`. Вектор выполняет присваивание в 13 раз быстрее при том же количестве записей. Вектор в 22 раза быстрее при удалении, в 9 раз быстрее при вставке с помощью итератора, вдвое быстрее при работе с `push_back()` и втрое — при вставке в конец.

Из истории оптимизационных войн

В самом начале тестирования производительности дека я столкнулся с неприятным сюрпризом: операции с `std::deque` были в тысячу раз медленнее, чем эквивалентные операции с `std::vector`. Сначала я сказал себе: "Что ж, имеем то, что имеем. Дек — просто ужасный выбор в качестве структуры данных". И только когда я стал выполнять "окончательный" набор тестов для таблиц в этой книге, я понял, какого дурака свалял.

Обычно при разработке я запускаю тестовые программы в отладчике, потому что для этого в интегрированной среде разработки имеется большая жирная кнопка. Мне было известно, что при отладке с программой связывается библиотека времени выполнения C++ с дополнительными возможностями отладки. Но я никогда не видел, чтобы это приводило к большему, чем несущественное различие в производительности. Для таблиц же в книге я выполнял окончательный запуск вне отладчика, чтобы получать более последовательные и точные данные хронометража. Так я обнаружил, что `std::deque` под отладчиком имеет чудовищную стоимость из-за диагностического кода, добавленного к процедуре выделения памяти. Этот результат выпадал из всего моего опыта измерений относительной производительности в отладочной и окончательной, производственной версиях сборки. Имеется возможность управлять тем, как при отладке работает диспетчер памяти — в отладочном или производственном режиме (см. врезку "Совет от профессионала" в разделе "Профилирование выполнения программы" главы 3, "Измерение производительности").

Итерирование `std::deque`

Итерирование элементов дека выполняется за 0,828 мс для версии с индексами и за 0,450 мс для кода на основе итератора. Интересно, что в случае дека гораздо быстрее работает код на основе итератора, в то время как для вектора быстрее работает код с использованием индексов. Но самый быстрый метод обхода дека в два раза медленнее самого быстрого метода обхода вектора (что продолжает выявленную ранее тенденцию к отставанию дека от вектора).

Сортировка `std::deque`

`std::sort()` обрабатывает 100 тысяч записей в деке за 24,82 мс, что примерно на треть дольше, чем сортировка вектора. Что касается `std::stable_sort()`, то здесь отставание сортировки для дека находится в пределах 10% от сортировки вектора. В обоих случаях отсортированный контейнер сортировался быстрее неотсортированного.

Поиск в `std::deque`

Поиск всех 100 тысяч ключей в отсортированном деке выполняется за 35 мс. Поиск в деке примерно на 20% медленнее поиска в векторе.

`std::list`

"Список характеристик" этой структуры данных имеет следующий вид.

- Последовательный контейнер.
- Время вставки: O(1) в любой позиции.
- Сортировка за время $O(n \cdot log_2 n)$.

- Поиск за время O(n).
- Итераторы и указатели никогда не становятся недействительными, за исключением только удаленных элементов.
- Итераторы обходят элементы от начала до конца и от конца до начала.

`std::list` имеет много общих свойств с `std::vector` и `std::deque`. Как и в случае вектора и дека, добавление элементов в конец списка выполняется за константное время. Как и у дека (но не у вектора) вставка элементов в начало списка также выполняется за константное время. Кроме того, в отличие от вектора или дека, вставка элементов в середину списка выполняется за константное время (при наличии итератора, указывающего на точку вставки). Как и дек и вектор, список может быть эффективно отсортирован. Но, в отличие от векторов и деков, не существует эффективного способа поиска в списке. Лучшее, что можно сделать, — применить алгоритм `std::find()`, который имеет временную сложность O(n).

Общеизвестная мудрость — что `std::list` слишком неэффективен, чтобы быть полезным, но после того как я исследовал его производительность, я так не считаю. В то время как копирование или создание `std::list` может быть в 10 раз дороже соответствующей операции для `std::vector`, список вполне может конкурировать с деком. Последовательное добавление элементов в хвост списка менее чем вдвое дороже добавления к вектору. Обход и сортировка списка только на 30% дороже, чем те же операции над вектором. Для большинства операций, которые я тестировал, `std::list` был дешевле, чем `std::deque`.

В соответствии с тем же общеизвестным мнением `std::list`, с его прямыми и обратными связями и методом `size()` с константным временем работы, дороже, чем это необходимо для функций, которые он предоставляет. Это мнение в конечном итоге привело к включению в стандарт C++11 списка `std::forward_list`. Однако стоит немного протестировать производительность, как становится очевидным, что стоимость операций `std::list` практически идентична стоимости операций `std::forward_list`, по крайней мере на персональных компьютерах.

Поскольку `std::list` не имеет в своей основе массива, который требуется перераспределять, итераторы и ссылки на элементы списка никогда не становятся недействительными из-за вставки. Они становятся таковыми только тогда, когда удаляются элементы, на которые они указывают.

Светлым местом `std::list` является то, что списки могут быть разрезаны (за время O(1)) и объединены без копирования элементов списка. Даже такие операции, как сращивание и сортировка списков, не делают итераторы `std::list` недействительными. Вставка в средину списка выполняется за константное время *при условии, что программе уже известно, куда выполнять вставку*. Таким образом, приложение, которое создает списки элементов, а затем перетасовывает их, *может* быть более эффективным при использовании `std::list`, чем с `std::vector`.

`std::list` взаимодействует с диспетчером памяти простым и предсказуемым способом. Каждый элемент списка при необходимости выделяется отдельно. В списке нет никакой скрытой неиспользованной емкости (рис. 10.3).

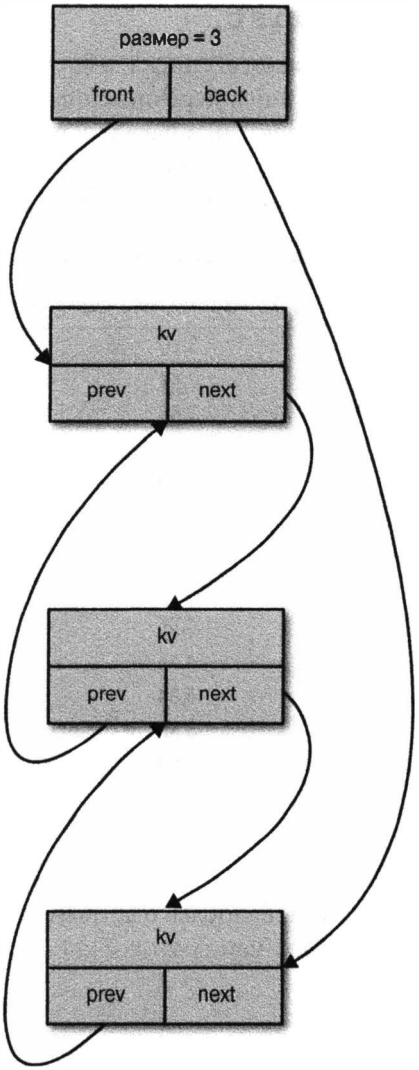


Рис. 10.3. Возможная реализация `std::list` *(состояние после нескольких вставок и удалений)*

Память, выделенная для каждого элемента списка, имеет один и тот же размер. Это помогает сложным диспетчерам памяти работать эффективно и с меньшим риском фрагментации памяти. Можно также определить простой пользовательский аллокатор для `std::list`, который использует это свойство для повышения эффективности (см. раздел "Аллокатор блоков фиксированного размера" главы 13, "Оптимизация управления памятью").

Вставка и удаление в std::list

Алгоритмы для копирования одного списка в другой с помощью insert(), push_back() и push_front() идентичны перечисленным для вектора и дека, за исключением объявления структуры данных в начале кода. Очень простая структура std::list не дает компилятору особых шансов улучшить код во время компиляции. Хронометраж всех этих механизмов подытожен в табл. 10.2.

Таблица 10.2. Итоги экспериментов по производительности std::list

std::list, 100k элементов, VS2010 release, i7	Время, мс	Список по сравнению с вектором, %
Присваивание	5,10	1046
Удаление	2,49	2141
insert(end())	3,69	533
push_back() с итераторами	4,26	88
at() push_back()	4,50	120
push_back() с индексами	4,63	132
push_front() с итераторами	4,77	
at() push_front()	4,82	
push_front() с индексами	4,99	
Итераторная вставка в конец	4,75	75
at() вставка в конец	4,84	77
Индексная вставка в конец	4,88	75
Итераторная вставка в начало	4,84	
at() вставка в начало	5,02	
Индексная вставка в начало	5,04	

Вставка в конец списка оказалась наиболее быстрым способом построения списка; по ряду причин она даже быстрее, чем operator=().

Итерирование std::list

Для списков не существует операторов индексации. Единственным вариантом обхода списка является использование итераторов.

Обход всех 100 тысяч элементов списка выполняется за 0,326 мс. Это всего на 38% дольше, чем обход вектора.

Сортировка std::list

Итераторы std::list являются двунаправленными итераторами, менее мощными, чем итераторы с произвольным доступом у std::vector. В частности, одно из свойств этих итераторов — найти расстояние, или количество элементов между двумя двунаправленными итераторами, можно за время O(n). Таким образом, std::sort() при сортировке std::list имеет производительность O(n^2). Компилятор по-прежнему без замечаний будет компилировать std::sort() для списка, но его производительность будет гораздо хуже, чем может ожидать разработчик.

К счастью, `std::list` имеет более эффективную встроенную сортировку, выполняющуюся за время $O(n \cdot log_2 n)$. Сортировка списка с помощью `std::list::sort()` выполнилась за время 23,2 мс, только на 25% дольше, чем сортировка эквивалентного вектора.

Поиск в `std::list`

Поскольку `std::list` предоставляет только двунаправленные итераторы, алгоритмы бинарного поиска для списков дают время работы $O(n)$. Поиск с использованием `std::find()` также дает $O(n)$, где n — количество записей в списке. Это делает `std::list` плохим кандидатом на замену ассоциативных контейнеров.

`std::forward_list`

"Список характеристик" этой структуры данных имеет следующий вид.

- Последовательный контейнер.
- Время вставки: O(1) в любой позиции.
- Сортировка за время $O(n \cdot log_2 n)$.
- Поиск за время $O(n)$.
- Итераторы и указатели никогда не становятся недействительными, за исключением только удаленных элементов.
- Итераторы обходят элементы от начала до конца.

`std::forward_list` представляет собой последовательный контейнер, упрощенный до предела. Он содержит один указатель на головной узел списка. Он был разработан специально для того, чтобы сделать его эквивалентом однонаправленного списка, закодированного вручную. У него нет функций-членов `back()` или `rbegin()`.

`std::forward_list` взаимодействует с диспетчером памяти очень простым и предсказуемым способом. Каждый элемент односвязного списка выделяется при необходимости отдельно. Нет никакой неиспользованной емкости, скрытой в таком списке (рис. 10.4). Выделяемая для каждого элемента списка память имеет один и тот же размер. Это помогает сложным диспетчерам памяти работать эффективно и с меньшим риском фрагментации памяти. Можно также определить простой пользовательский аллокатор для `std::forward_list`, который использует это свойство для повышения эффективности (см. раздел "Аллокатор блоков фиксированного размера" главы 13, "Оптимизация управления памятью").

Односвязный список отличается от списка тем, что предлагает только однонаправленные итераторы, как и предполагается из его названия. Такой список можно обходить обычным циклом:

```
std::forward_list<kvstruct> flist;
// ...
unsigned sum = 0;
for (auto it = flist.begin(); it != flist.end(); ++it)
    sum += it->value;
```

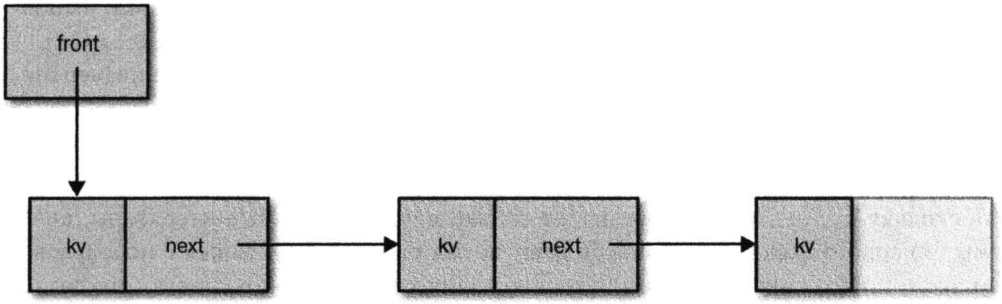


Рис. 10.4. Возможная реализация `std::forward_list`

Вставка, однако, требует иного подхода. Вместо метода `insert()` класс `std::forward_list` имеет метод `insert_after()`. Существует также функция `before_begin()` для получения итератора, указывающего на (несуществующий) элемент перед первым элементом списка (другого способа вставки перед первым элементом нет, поскольку все элементы содержат указатели только на следующий элемент):

```
std::forward_list<kvstruct> flist;
std::vector<kvstruct> vect;
// ...
auto place = flist.before_begin();
for (auto it = vvect.begin(); it != vect.end(); ++it)
    place = flist.insert_after(place, *it);
```

На моем компьютере `std::forward_list` не опережал `std::list` сколь-нибудь значительно. Все то, что делает `std::list` медленным (поэлементное выделение памяти, плохая локальность кеша), остается такой же большой проблемой и для `std::forward_list`. Он может быть полезен на меньших процессорах с более жесткими ограничениями памяти, но для процессоров класса настольных компьютеров и телефонов мало что можно сказать в его пользу.

Вставка и удаление в `std::forward_list`

`std::forward_list` выполняет вставку за константное время в любую позицию при наличии итератора, указывающего на предыдущую позицию. Вставка 100 тысяч записей в односвязный список заняла 4,24 мс, примерно столько же, сколько и для `std::list`.

`std::forward_list` имеет функцию-член `push_front()`. Вставка 100 тысяч записей с ее помощью выполнилась за 4,16 мс — вновь примерно за то же время, что и для `std::list`.

Итерирование `std::forward_list`

Для `std::forward_list` не существует оператора индексации. Единственным вариантом обхода списка является использование итераторов.

Обход всех 100 тысяч элементов списка выполняется за 0,343 мс. Это всего на 45% дольше, чем обход вектора.

Сортировка std::forward_list

Как и std::list, std::forward_list имеет встроенную функцию-член, которая выполняет сортировку за время $O(n \cdot log_2 n)$. Производительность сортировки аналогична таковой для std::list, — сортировка 100 тысяч элементов занимает 23,3 мс.

Поиск в std::forward_list

Поскольку std::forward_list предоставляет только однонаправленные итераторы, алгоритмы бинарного поиска для списков дают время работы $O(n)$. Поиск с использованием std::find() также дает $O(n)$, где n — количество записей в списке. Это делает однонаправленный список плохим кандидатом на замену ассоциативных контейнеров.

std::map and std::multimap

"Список характеристик" этих структур данных имеет следующий вид.

- Упорядоченный ассоциативный контейнер.
- Время вставки: $O(log_2 n)$.
- Время индексации: по ключу $O(log_2 n)$.
- Итераторы и ссылки никогда не становятся недействительными, за исключением только удаленных элементов.
- Итераторы обходят элементы в порядке сортировки или в обратном ему порядке.

std::map отображает экземпляры типа ключа на соответствующие экземпляры некоторого типа значения. std::map представляет собой структуру данных на основе узлов, как std::list. Однако отображение упорядочивает свои узлы в соответствии со значением ключа. Внутренне отображение реализуется как сбалансированное бинарное дерево с дополнительными ссылками для облегчения обхода на основе итератора (рис. 10.5).

Хотя отображение std::map реализовано с использованием дерева, это не дерево. Нет никакого способа обратиться к ссылкам, выполнить обход в ширину или другие операции над деревом, работая с отображением.

Взаимодействие std::map с диспетчером памяти простое и предсказуемое. Память для каждого элемента отображения при необходимости выделяется отдельно. std::map не содержит базового массива, который может быть перераспределен, поэтому итераторы и ссылки на элементы отображения никогда не становятся недействительными из-за вставки. Они станут недействительными, только если будут удалены элементы, на которые они указывают.

Выделяемая для каждого элемента отображения память имеет один и тот же размер. Это помогает сложным диспетчерам памяти работать эффективно и с меньшим риском фрагментации памяти. Можно также определить простой пользовательский аллокатор для std::map, который использует это свойство для повышения

эффективности (см. раздел "Аллокатор блоков фиксированного размера" главы 13, "Оптимизация управления памятью").

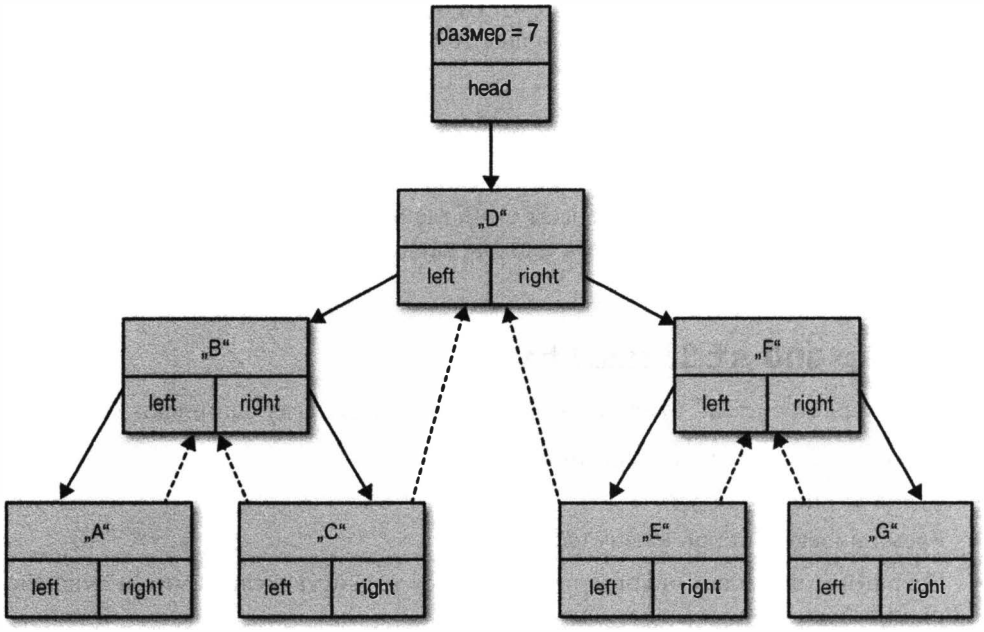


Рис. 10.5. Упрощенная возможная реализация `std::map` *и* `std::set`

Вставка и удаление в `std::map`

Вставка 100 тысяч случайных записей из вектора в `std::map` выполнилась за 33,8 мс.

Вставка в отображение обычно имеет стоимость O($log_2 n$) из-за необходимости поиска во внутреннем дереве отображения точки вставки. Эта стоимость достаточно высока для того, чтобы класс `std::map` предоставлял версию `insert()`, которая принимает дополнительный итератор отображения, выступающий в качестве подсказки места вставки, и которая может оказаться куда более эффективной. Если подсказка оптимальна, амортизированное время вставки — O(1).

Что касается подсказки, то здесь есть и хорошие, и плохие новости. Хорошей новостью является то, что вставка с подсказкой никогда не может быть дороже, чем обычная вставка. Плохая новость — оптимальное значение, рекомендуемое в качестве подсказки для вставки, изменено в стандарте C++11. До C++11 оптимальное значение подсказки для вставки было позицией *перед* новым элементом, т.е., если элементы вставляются в отсортированном порядке, оптимальная подсказка представляет собой положение предыдущей вставки. Начиная с C++11 оптимальной подсказкой для вставки является позиция *после* нового элемента, т.е., если элементы вставляются в порядке сортировки, позиция подсказки должна быть `end()`. Чтобы сделать ранее вставленный элемент оптимальной подсказкой, программа должна

выполнять проход по отсортированным вводимым данным в обратном порядке, как показано в примере 10.2.

Пример 10.2. Вставка из отсортированного вектора с использованием подсказки в стиле C++11

```
ContainerT test_container;
std::vector<kvstruct> sorted_vector;
...
std::stable_sort(sorted_vector.begin(), sorted_vector.end());
auto hint = test_container.end();
for (auto it = sorted_vector.rbegin();
     it != sorted_vector.rend(); ++it)
    hint = test_container.insert(hint, value_type(it->key,
                                                   it->value));
```

Как показывает мой опыт с GCC и Visual Studio 2010, реализация стандартной библиотеки может как опережать последний стандарт, так и отставать от него. В результате программа, оптимизированная с помощью подсказки в старом стиле, может замедлить работу при использовании более нового компилятора, даже если этот компилятор не полностью соответствует стандарту C++11.

Я выполнял хронометраж вставки, используя три варианта подсказки: `end()`, итератор на предыдущий узел (использовавшийся в стандартных библиотеках до C++11) и итератор, указывающий на узел-преемник, как требует C++11. Результаты показаны в табл. 10.3. Для выполнения этого теста входные данные также должны быть отсортированы.

Таблица 10.3. Производительность вставки с подсказкой в `std::map`

Эксперимент	Время, мс
Отсортированный вектор: `insert()`	18,00
Отсортированный вектор: `insert()` с подсказкой `end()`	9,11
Отсортированный вектор: `insert()` с подсказкой до C++11	14,40
Отсортированный вектор: `insert()` с подсказкой C++11	8,56

Похоже, что Visual Studio 2010 реализует подсказку в стиле C++11. Но и та, и другая подсказки оказываются лучше, чем отсутствие подсказки вообще, и лучше, чем версия `insert()` без подсказки и неотсортированный ввод.

Оптимизация идиомы проверки и обновления

В часто встречающейся идиоме программа проверяет, есть ли некоторые ключи в отображении, а затем выполняет действия в зависимости от полученного результата. Оптимизация производительности возможна, когда действие включает в себя вставку или обновление значения, соответствующего ключу поиска.

Ключом к пониманию оптимизации является то, что и `map::find()`, и `map::insert()` имеют стоимость $O(log_2 n)$ из-за необходимости проверки наличия ключа и поиска точки вставки. Обе эти операции выполняют обход одинакового набора узлов в бинарном дереве отображения:

```
iterator it = table.find(key); // O(log n)
if (it != table.end()) {
    // Ключ найден
    it->second = value;
}
else {
    // Ключ не найден
    it = table.insert(key, value); // O(log n)
}
```

Результат первого поиска программа может использовать как подсказку для `insert()`, что делает стоимость вставки равной O(1). Есть два способа улучшить эту идиому, в зависимости от потребностей программы. Если все, что вам нужно, — это знать, был ли ключ найден, можно использовать версию `insert()`, которая возвращает пару, содержащую итератор, указывающий на найденную или вставленную запись, и логическое значение типа `bool`, равное `false`, если запись с таким ключом была найдена, а потому не вставлена; и `true`, если она была успешно вставлена. Это решение работает, если программа знает, как инициализировать запись, прежде чем выяснить, присутствует ли она в отображении, или если ее значение недорого обновить:

```
std::pair<value_t, bool> result = table.insert(key, value);
if (result.second) {
    // key успешно вставлен в отображение
}
else {
    // key уже имеется в отображении
}
```

Второй метод предусматривает поиск ключа или точки вставки с помощью вызова `upper_bound()` для подсказки в стиле C++98 или `lower_bound()` для подсказки в стиле C++11. `lower_bound()` возвращает итератор, указывающий на наименьшую запись в отображении, ключ которой не меньше, чем искомый, или `end()`, если таковой не найден. Этот итератор является подсказкой для вставки, если ключ должен быть вставлен, и указывает на существующий ключ, если необходимо обновить существующую запись. Этот метод не делает никаких предположений о записи, которую требуется вставить:

```
iterator it = table.lower_bound(key);
if (it == table.end() || key < it->first) {
    // Ключ не найден
    table.insert(it, key, value);
}
else {
    // Ключ найден
    it->second = value;
}
```

Итерирование `std::map`

Обход всех 100 тысяч элементов отображения выполняется за 1,34 мс, примерно в 10 раз медленнее, чем в случае вектора.

Сортировка std::map

Отображения отсортированы по самой своей природе. Итерирование отображения воспроизводит записи в упорядоченном по ключам виде в соответствии с используемым предикатом поиска. Обратите внимание, что невозможно повторно отсортировать отображение с использованием другого предиката сортировки без копирования всех элементов в другое отображение.

Поиск в std::map

Поиск всех 100 тысяч записей в отображении занял 42,3 мс. Тот же поиск всех записей в отсортированном векторе занял 28,9 мс и 35,1 мс — в отсортированном деке, с использованием `std::lower_bound()`. В табл. 10.4 подытожены производительности вектора и отображения при использовании в качестве таблицы поиска.

Таблица 10.4. Время вставки и поиска в векторе и отображении

	Вставка + сортировка, мс	Поиск, мс
Вектор	19,1	28,9
Отображение	33,8	42,3

Реализация на основе вектора 100 000-элементной таблицы, которая строится вся сразу, а затем в ней многократно осуществляется поиск, будет быстрее, чем на основе отображения. Если таблица будет часто меняться путем вставок или удалений, то повторная сортировка таблицы на основе вектора "съест" любое преимущество, полученное благодаря более быстрому поиску.

std::set и std::multiset

"Список характеристик" этих структур данных имеет следующий вид.

- Упорядоченный ассоциативный контейнер.
- Время вставки: $O(log_2 n)$.
- Время доступа по ключу: $O(log_2 n)$.
- Итераторы и ссылки никогда не становятся недействительными, за исключением только удаленных элементов.
- Итераторы обходят элементы в порядке сортировки или обратном ему.

Я не выполнял тест производительности для `std::set`. В Windows `std::set` и `std::multiset` используют ту же структуру данных, что и `std::map` (см. рис. 10.5), так что характеристики производительности у них те же, что и для `std::map`. Хотя множество, в принципе, может быть реализовано с использованием иной структуры данных, нет никаких причин, по которым следует поступать именно так.

Единственное различие между `std::map` и `std::set` заключается в константности возвращаемых элементов в `std::set`. Но эта проблема меньше, чем кажется. Если вы действительно хотите использовать абстракцию множества, то поля в типе данных

элементов, которые не участвуют в определении отношения упорядочения, могут быть объявлены `mutable`. Конечно, компилятор безоговорочно доверяет разработчику, поэтому важно и в самом деле не изменять члены, которые участвуют в определении отношения, иначе структура данных множества станет недействительной.

`std::unordered_map` и `std::unordered_multimap`

"Список характеристик" этих структур данных имеет следующий вид.

- Неупорядоченный ассоциативный контейнер.
- Время вставки: O(1) в среднем, O(*n*) в наихудшем случае.
- Время индексации по ключу: O(1) в среднем, O(*n*) в наихудшем случае.
- Итераторы становятся недействительными при перехешировании, ссылки становятся недействительными только для удаленных элементов.
- Емкость может увеличиваться и уменьшаться независимо от размера.

`std::unordered_map` отображает экземпляры с типом ключа на соответствующие экземпляры с типом значения. Таким образом, он похож на `std::map`. Однако отображение реализовано иначе, чем класс `std::unordered_map`, который реализован как хеш-таблица. Ключи преобразуются в целочисленный хеш-код, который используется в качестве индекса массива для поиска значения за константное среднее амортизированное время.

Стандарт C++ ограничивает реализацию `std::unordered_map` так же, как и `std::string`. Таким образом, хотя хеш-таблица может быть реализована несколькими разными способами, стандарту, пожалуй, соответствует только дизайн с динамически выделяемым базовым массивом *сегментов*, указывающих на связанные списки динамически выделенных узлов.

Неупорядоченные отображения дороги в построении. Они содержат динамически выделенные узлы для каждой записи таблицы, а также массив сегментов с динамически изменяемым размером, который периодически перераспределяется по мере роста таблицы (рис. 10.6). Таким образом, для достижения лучшей производительности поиска используется значительный объем памяти. Итераторы становятся недействительными при каждом перераспределении массива сегментов. Однако ссылки, указывающие на узлы, становятся недействительными только при удалении соответствующих элементов.

Хеш-таблицы, такие как `std::unordered_map`, имеют несколько параметров настройки для достижения оптимальной производительности. В этом их сила — или слабость, в зависимости от точки зрения разработчика.

Количество записей в неупорядоченном отображении называется его *размером*. Вычисляемое отношение *размер/емкость* называется *коэффициентом загрузки*. Коэффициент загрузки, больший 1,0, означает, что некоторые сегменты содержат цепочки из нескольких записей, что приводит к снижению производительности поиска этих ключей (другими словами, хеш не является совершенным). В реальных хеш-таблицах коллизии ключей приводят к появлению цепочек записей даже при коэффициенте

загрузки, меньшем 1,0. Коэффициент загрузки, меньший, чем 1,0, означает, что имеются неиспользуемые сегменты, потребляющие место в базовом массиве неотсортированного отображения (другими словами, хеш не является минимальным). Когда коэффициент загрузки меньше 1,0, значение *1–коэффициент загрузки* представляет собой нижнюю границу количества пустых сегментов, но поскольку хеш-функции могут быть несовершенными, объем неиспользуемого пространства обычно больше вычисленного по этой формуле.

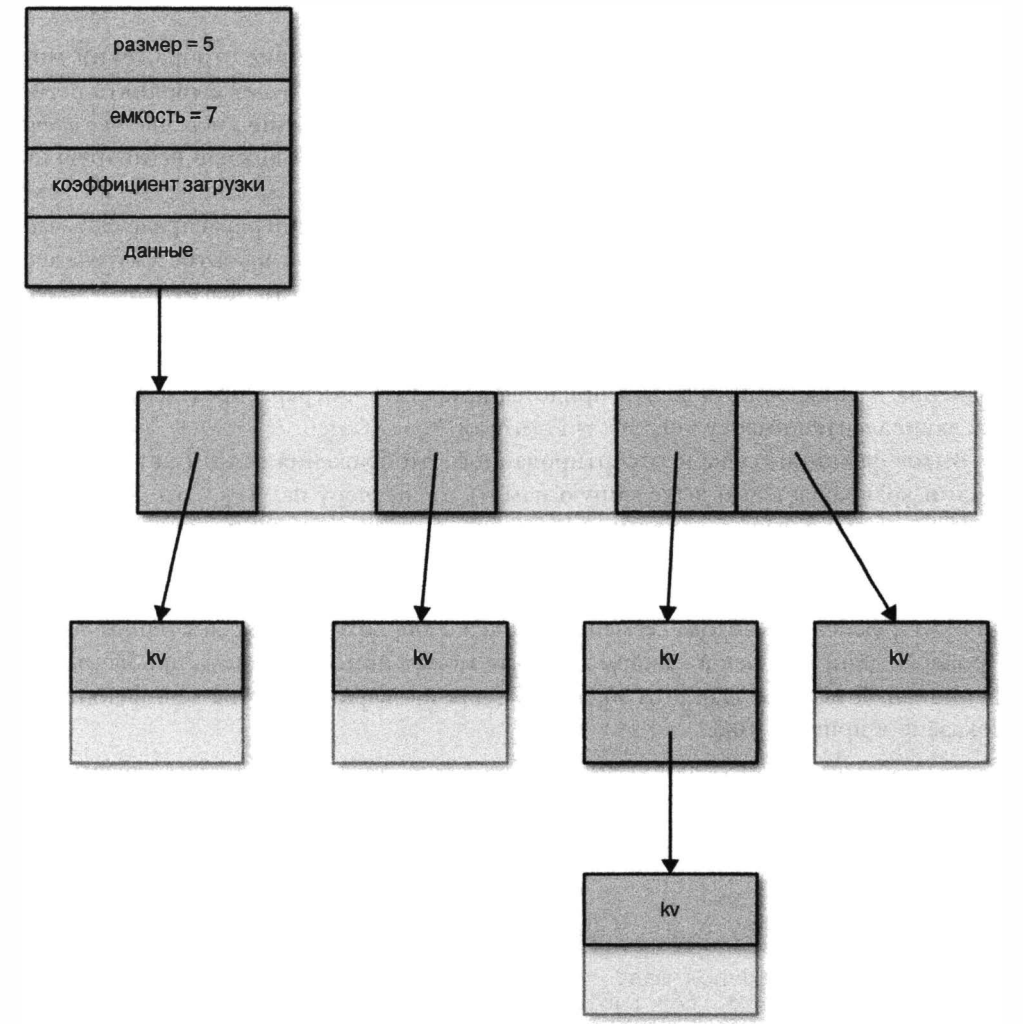


Рис. 10.6. Возможная реализация `std::unordered_map`

Коэффициент загрузки является зависимой переменной в `std::unordered_map`. Программа может наблюдать его значение, но не может установить его непосредственно или предсказать его значение после перераспределения. Когда в неупорядоченное

отображение вставляется новая запись, то, если коэффициент загрузки превышает заранее установленный программой *максимальный коэффициент загрузки*, массив сегментов перераспределяется, и все записи перехешируются в новый массив. Поскольку количество сегментов всегда увеличивается на коэффициент, больший, чем 1, амортизированная стоимость вставки оказывается равной O(1). Эффективность вставки и поиска существенно уменьшится, если максимальный фактор загрузки больше 1,0 (значение по умолчанию). Производительность можно несколько улучшить путем уменьшения максимального коэффициента загрузки до значения, меньшего 1,0.

Изначальное количество сегментов в неотсортированном отображении можно указать в качестве аргумента конструктора. Контейнер не будет выполнять перераспределение до тех пор, пока его размер не превысит значение *емкость * коэффициент загрузки*. Программа может увеличить количество сегментов в неотсортированном отображении путем вызова функции-члена `rehash()`. `rehash(size_t n)` устанавливает количество сегментов равным как минимум `n`, перераспределяет массив сегментов и перестраивает таблицу, перемещая все записи в соответствующие сегменты в новом массиве. Если `n` меньше текущего количества сегментов, `rehash()` может как уменьшить размер таблицы, так и не уменьшать его.

Вызов `reserve(size_t n)` может зарезервировать память, достаточную для сохранения `n` записей, прежде чем придется прибегать к перераспределению. Этот вызов эквивалентен вызову `rehash(ceil(n/max_load_factor()))`.

Вызов функции-члена неотсортированного отображения `clear()` стирает все записи и возвращает всю выделенную память диспетчеру памяти. Это более строгое поведение, чем в случае функции-члена `clear()` вектора или строки.

В отличие от других контейнеров стандартной библиотеки C++, `std::unordered_map` предоставляет структуру своей реализации путем предоставления интерфейса для обхода сегментов, а также для обхода записей в одном сегменте. Проверка длин цепочек в каждом сегменте может помочь выявить проблемы с хеш-функцией. Я использовал этот интерфейс для проверки качества хеш-функции, как показано в примере 10.3.

Пример 10.3. Изучение поведения `std::unordered_map`

```
template<typename T> void hash_stats(T const& table) {
    unsigned zeros = 0;
    unsigned ones = 0;
    unsigned many = 0;
    unsigned many_sigma = 0;
    for (unsigned i = 0; i < table.bucket_count(); ++i) {
        unsigned how_many_this_bucket = 0;
        for (auto it = table.begin(i); it != table.end(i); ++it) {
            how_many_this_bucket += 1;
        }
        switch(how_many_this_bucket) {
        case 0:
            zeros += 1;
            break;
```

```
            case 1:
                ones += 1;
                break;
            default:
                many += 1;
                many_sigma += how_many_this_bucket;
                break;
            }
        }
        std::cout << "unordered_map с " << table.size()
                  << " записями" << std::endl
                  << "    " << table.bucket_count() << " сегментами"
                  << " коэффициентом загрузки " << table.load_factor()
                  << ", максимальным коэффициентом загрузки "
                  << table.max_load_factor() << std::endl;
        if (ones > 0 && many > 0)
            std::cout << "    " << zeros << " пустых сегментов, "
                      << ones << " сегментов с одной записью, "
                      << many << " сегментов с несколькими записями, "
                      << std::endl;
        if (many > 0)
            std::cout << "    средняя длина цепочки с "
                      << "несколькими записями "
                      << ((float) many_sigma) / many << std::endl;
    }
```

Я обнаружил при использовании хеш-функции из проекта Boost, что коллизии охватывают 15% моих записей и что автоматическое выделение создает таблицу с коэффициентом загрузки 0,38, а это означает, что 62% базового массива не используются. Хеш-функции, которые я пытался создавать, оказались гораздо хуже, чем я ожидал.

Вставка и удаление в `std::unordered_map`

Подобно `std::map`, `std::unordered_map` предоставляет два метода вставки — с подсказкой и без. Но в отличие от отображения неупорядоченное отображение не использует эту подсказку. Этот интерфейс предусмотрен только для совместимости и дает даже небольшое уменьшение производительности, несмотря на то что он ничего не делает.

Тест вставки выполнился за 15,5 мс. Производительность вставки можно несколько улучшить с помощью предварительного выделения с помощью вызова `reserve()` достаточного количества сегментов для предотвращения перехешированый. Улучшение производительности в моем тесте оказалось всего лишь 4%, что уменьшило время вставки до 14,9 мс.

Перераспределение может быть отложено путем установки очень высокого значения максимального коэффициента загрузки. Я решил посмотреть, будет ли повышена производительность, если сделать это, а затем перехешировать таблицу после вставки всех элементов. Для версии `unordered_map` из стандартной библиотеки Visual Studio 2010 это оказалось катастрофической деоптимизацией. Видимо, в реализации Visual Studio элементы вставляются в конец цепочки коллизии, поэтому стоимость каждой вставки имела значение от константы до O(n). Вполне возможно, что более сложные реализации окажутся без этого слабого места.

Итерирование std::unordered_map

В примере 10.4 приведен код обхода элементов std::unordered_map.

Пример 10.4. Обход всех записей в неупорядоченном отображении

```
for (auto it = test_container.begin();
          it != test_container.end();
        ++it) {
    sum += it->second;
}
```

Неупорядоченное отображение нельзя отсортировать, а обход контейнера выдает элементы в неотсортированном порядке, как и указывает название контейнера. Обход неупорядоченного отображения относительно эффективен — за 0,34 мс. Это всего лишь в 2,8 раза медленнее, чем обход std::vector.

Поиск в std::unordered_map

Поиск — смысл существования std::unordered_map. Я сравнил производительность поиска в таблице "ключ/значение" со 100 тысячами записей на основе неупорядоченного отображения с поиском в отсортированной таблице на базе вектора и с помощью std::lower_bound().

std::unordered_map выполнил мой тест со 100 тысячами запросов за 10,4 мс. Как видно из табл. 10.5, это в 3 раза быстрее, чем для std::map, и в 1,7 раза быстрее, чем поиск в отсортированном std::vector с помощью std::lower_bound(). Да, это быстрее — гораздо быстрее. Но о хеш-таблицах отзываются с таким благоговением, что я ожидал удивительного (скажем, на порядок) увеличения производительности. Но я его не увидел.

Неупорядоченное отображение быстрее по сравнению с std::map в построении и гораздо быстрее при поиске. Недостатком неупорядоченных отображений является используемый ими объем памяти. В ограниченной среде может быть необходимо использовать более компактную таблицу, основанную на std::vector. В противном случае неупорядоченное отображение является однозначным победителем в плане производительности.

Таблица 10.5. Время вставки и поиска в vector, map и unordered_map

	Вставка + сортировка, мс	Поиск, мс
map	33,8	42,3
vector	19,1	28,9
unordered_map	15,5	10,4

Другие структуры данных

Контейнеры стандартной библиотеки отнюдь не являются последним словом в структурах данных. Библиотека Boost (http://www.boost.org/) содержит ряд струк-

тур данных, имитирующих контейнеры стандартной библиотеки. Boost предоставляет следующие библиотеки, содержащие альтернативные контейнеры.

`boost::circular_buffer` (`http://bit.ly/circ-buffer`)

Во многих аспектах схож с `std::deque`, но более эффективен.

`Boost.Container` (`http://bit.ly/boost-cont`)

Вариации на тему контейнеров стандартной библиотеки, таких как устойчивый вектор (вектор, в котором перераспределение не делает итераторы недействительными), `map`/`multimap`/`set`/`multiset`, реализованные как адаптер контейнера для `std::vector`, статический вектор с переменной длиной не свыше фиксированного максимума и вектор с оптимизированным поведением, когда в нем содержится лишь несколько элементов.

`dynamic_bitset` (`http://bit.ly/b-bitset`)

Выглядит как вектор битов.

Fusion (`http://bit.ly/b-fusion`)

Контейнеры и итераторы над кортежами.

Boost Graph Library (*BGL*) (`http://bit.ly/b-graphlib`)

Алгоритмы и структуры данных для обхода графов.

`boost.heap` (`http://bit.ly/b-heap/`)

Очередь с приоритетами с повышенной производительностью и более тонким поведением по сравнению с простым адаптером `std::priority_queue`.

Boost.Intrusive (`http://bit.ly/b-intrusive`)

Предоставляет интрузивные контейнеры (контейнеры, основанные на типах узлов, которые явно содержат ссылки). Цель интрузивных контейнеров — повысить производительность в очень горячем коде. Эта библиотека содержит одно- и двусвязные списки, ассоциативные контейнеры, неупорядоченные ассоциативные контейнеры и различные явные реализации сбалансированных деревьев. Наличие `make_shared`, наряду с семантикой перемещения и функциями-членами `emplace()`, добавленными в большинство контейнеров, уменьшает потребность в интрузивных контейнерах.

`boost.lockfree` (`http://bit.ly/b-lockfree`)

Очередь и стек без блокировок.

Boost.MultiIndex (`http://bit.ly/b-multi`)

Контейнеры, которые имеют несколько индексов с различным поведением.

В этом списке представлены не все типы контейнеров, имеющиеся в Boost.

Еще один значительный вклад внесен играми компании Electronic Arts, версия классов контейнеров стандартной библиотеки которых EASTL доступна с открытым кодом (`http://bit.ly/ea-stl`). Классы контейнеров Electronic Arts включают следующие.

- Более простое и более разумное определение `Allocator`.
- Более строгие гарантии у некоторых контейнеров, включая гарантии, что контейнеры не вызывают диспетчеры памяти, пока программа не размещает в них элементы.
- Большие возможности программирования `std::deque`.
- Ряд контейнеров, похожих на предоставляемые Boost.

Резюме

- *Стандартная библиотека шаблонов Степанова была первой повторно используемой библиотекой эффективных контейнеров и алгоритмов.*
- *Производительность в терминах большого О не дает полной информации о классах контейнеров. Одни контейнеры во много раз более быстрые, чем другие.*
- *Контейнер с самой быстрой вставкой, удалением, итерированием и сортировкой — `std::vector`.*
- *Поиск с помощью `std::lower_bound` в отсортированном `std::vector` может конкурировать с `std::map`.*
- *`std::deque` немного быстрее, чем `std::list`.*
- *`std::forward_list` не быстрее, чем `std::list`.*
- *Хеш-таблица `std::unordered_map` быстрее, чем `std::map`, но отнюдь не на порядок по величине, как можно было бы предположить из ее репутации.*
- *В Интернете имеется масса ресурсов с контейнерами, имитирующими контейнеры стандартной библиотеки.*

ГЛАВА 11

Оптимизация ввода-вывода

Программа — это заклинания над компьютером, превращающие входные данные в сообщения об ошибках.

— Аноним

В этой главе рассмотрено эффективное использование потоковых функций ввода-вывода C++ для распространенных примеров чтения и записи текстовых данных. Чтение и запись данных являются настолько распространенными действиями, что разработчики их просто не замечают, однако эта деятельность требует достаточно много времени.

Вращение пластины диска для сегодняшних ультрабыстрых компьютерных чипов такое же медленное, как вращение Земли для нас. Считывающие головки имеют инерцию, которую необходимо преодолеть, чтобы переместить их от одной дорожки к другой. Физические законы яростно сопротивляются попыткам повышения производительности аппаратного обеспечения. В мире ограниченных пропускных каналов и занятых серверов Интернета задержка ответа может измеряться в секундах, а не в миллисекундах. Даже скорость света становится существенным фактором при потоковой передаче данных с удаленных компьютеров.

Еще одной проблемой ввода-вывода является большое количество кода между программой пользователя и вращающимися пластинами жесткого диска или картой сетевого интерфейса. Для эффективного ввода-вывода необходимо управление стоимостью всего этого кода.

Рецепты для чтения файлов

В вебе имеется множество вариантов чтения и записи файлов (`http://bit.ly/read-cpp`), и некоторые из них даже, как утверждают, весьма быстрые. Их производительность в моих тестах варьируется на целый порядок. Читатели, которые уже достаточно умны, чтобы не использовать непроверенные советы из Интернета (например, рецепты для изготовления взрывчатых веществ, приготовления мета[1] или получения разъема на 3,5 мм в iPhone 7 с помощью дрели), могут добавить в этот список рецепты чтения файлов в C++.

[1] Мет — сленговое название метамфетамина. — *Примеч. пер.*

Из истории оптимизационных войн

Моя родственница Марсия любит печь пироги, но ей никак не удавалось найти рецепт, в котором получалась бы устраивающая ее корочка. Она была то слишком светлой, то рыхлой, то еще какой-то не такой. Марсия — не разработчик программного обеспечения, но она оказалась довольно хорошим оптимизатором, судя по тому как она поступила.

Она собрала несколько рецептов пирогов; во всех них, как утверждалось, получался "пирог с наилучшей корочкой, которую вы когда-либо видели". Она заметила, что все ингредиенты повторяются: мука, соль, вода, масло. Тогда она обратила внимание на различия. В одних использовалось сливочное масло или смалец вместо растительного масла. В других ингредиенты или получившееся тесто охлаждались. Кто-то добавлял немного сахара, столовую ложку уксуса или яйцо. Марсия взяла все отличительные черты из рецептов и в течение нескольких месяцев терпеливо ставила вкусные эксперименты. В результате она сумела скомбинировать советы из нескольких рецептов и получить пирог с удивительной, полностью устроившей ее корочкой.

Как и моя родственница, я обнаружил, что есть несколько методов улучшения производительности функций чтения файлов. Многие из этих методов могут быть скомбинированы. Я также обнаружил некоторые методы, от которых слишком мало толку.

В примере 11.1 представлена простая функция для чтения текстового файла в строку. Я часто встречал этот код, в основном в качестве прелюдии для анализа строки как блока XML или JSON.

Пример 11.1. Исходная функция `file_reader()`

```
std::string file_reader(char const* fname) {
    std::ifstream f;
    f.open(fname);
    if (!f) {
        std::cout << "Невозможно открыть " << fname
                  << " для чтения" << std::endl;
        return "";
    }

    std::stringstream s;
    std::copy(std::istreambuf_iterator<char>(f.rdbuf()),
              std::istreambuf_iterator<char>(),
              std::ostreambuf_iterator<char>(s));
    return s.str();
}
```

Аргумент `fname` содержит имя файла. Если файл не может быть открыт, `file_reader()` выводит сообщение об ошибке на стандартный вывод и возвращает пустую строку. В противном случае `std::copy()` копирует буфер `f` в буфер переменной `s` типа `std::stringstream`.

Создание экономной сигнатуры функции

С точки зрения дизайна библиотеки функция `file_reader()` может быть усовершенствована (см. раздел "При разработке библиотек скупость является добродетелью" главы 8, "Использование лучших библиотек"). Она делает несколько различных вещей: открывает файл; выполняет обработку ошибок (в форме сообщения о невозможности открыть файл); читает поток открытый, корректный поток в строку. Такое сочетание обязанностей делает `file_reader()` трудно используемой в качестве библиотечной функции. Например, если клиентская программа реализует обработку исключений или хочет использовать ресурс Windows для строки сообщений об ошибках, или даже если просто хочет выводить сообщения об ошибках в `std::cerr`, `file_reader()` использоваться не может. `file_reader()` также создает новую память и возвращает ее — шаблон, который потенциально выполняет несколько копирований при возвращении значения вызывающей функции (см. раздел "Библиотеки без копирования" главы 6, "Оптимизация переменных в динамической памяти"). Если файл не может быть открыт, `file_reader()` возвращает пустую строку. Он также возвращает пустую строку, если файл читаем, но не содержит символов. Было бы неплохо получить более детальную информацию, позволяющую различать эти случаи.

В примере 11.2 представлен обновленный вариант `file_reader()`, который разделяет проблемы открытия файлов и чтения потоков и имеет сигнатуру, которую пользователи не захотят немедленно изменить.

Пример 11.2. `stream_read_streambuf_stringstream()`, теперь экономная!

```
void stream_read_streambuf_stringstream(std::istream& f,
                                        std::string& result) {
    std::stringstream s;
    std::copy(std::istreambuf_iterator<char>(f.rdbuf()),
              std::istreambuf_iterator<char>(),
              std::ostreambuf_iterator<char>(s));
    std::swap(result,s.str());
}
```

Последняя строка `stream_read_streambuf_stringstream()` обменивает динамическую память `result` с таковой в `s.str()`. Вместо этого можно выполнить присваивание `s.str()` переменной `result`, но это вызовет выделение памяти и копирование, если только компилятор и реализация строк не обеспечивают семантику перемещений. Функция `std::swap()` специализирована для многих классов стандартной библиотеки путем вызова функции-члена `swap()`. Эта функция-член, в свою очередь, обменивает указатели, что гораздо дешевле, чем операции выделения памяти и копирования.

В качестве бонуса за экономию `f` становится `std::istream`, а не `std::ifstream`. Такая функция может работать с любыми видами входных потоков, как, например, с `std::stringstream`. Эта функция короче и проще для понимания и содержит только один концептуальный блок вычислений.

`stream_read_streambuf_stringstream()` может быть вызвана таким клиентским кодом:

```
std::string s;
std::ifstream f;
f.open(fname);
if (!f) {
    std::cerr << "Ошибка открытия " << fname
              << " для чтения" << std::endl;
}
else {
    stream_read_streambuf_stringstream(f, s);
}
```

Обратите внимание, что клиент теперь отвечает за открытие файла и отчет об ошибках в том виде, в каком это нужно клиенту, в то время как вся "магия" чтения из потока находится в `stream_read_streambuf_stringstream()` — только эта версия функции не такая уж и магическая.

Я провел эксперимент, который состоял в чтении файла с 10 тыс. строк 100 раз. Этот эксперимент в основном испытывает именно стандартные операции ввода-вывода C++. Из-за повторения чтения операционная система почти наверняка кеширует содержимое файла. В реальных условиях, которые трудно имитировать в цикле теста, эта функция, вероятно, потребует больше времени, чем измеренные 1,548 мс в как Visual Studio 2010, так и в 2015.

Поскольку эта программа считывает диск, а на моем компьютере он только один, данный тест более чувствителен к другой дисковой активности на компьютере, чем прочие мои тесты. Мне пришлось закрыть ненужные программы и остановить ненужные службы. Даже тогда я должен был сделать несколько тестовых запусков и принять в качестве результата наименьшее полученное время.

`stream_read_streambuf_stringstream()` использует идиомы стандартной библиотеки, но не особенно эффективно. Он использует символьные итераторы для посимвольного копирования. Разумно предположить, что каждый выбранный символ задействует значительное количество инструкций в `std::istream`, а возможно, и в API файловой подсистемы операционной системы. Также разумно представить, что `std::string` в `std::stringstream` продлевается по одному символу за раз, что обусловливает значительное количество вызовов диспетчера памяти.

Существует несколько вариантов идеи копирования потокового итератора. В примере 11.3 `std::string::assign()` используется для копирования итератора из входного потока в `std::string`.

Пример 11.3. Еще одна функция чтения файла

```
void stream_read_streambuf_string(std::istream& f,
                                  std::string& result) {
    result.assign(std::istreambuf_iterator<char>(f.rdbuf()),
                  std::istreambuf_iterator<char>());
}
```

Этот код выполняется в моем тесте за 1 510 мс при компиляции с помощью Visual Studio 2010 и за 1 787 мс при использовании Visual Studio 2015.

Сокращение цепочек вызовов

Пример 11.4 — еще одна версия посимвольного чтения. `std::istream` имеет `operator<<()`, который принимает в качестве аргумента `streambuf`. Вероятно, `operator<<()` обходит API `istream` для непосредственного вызова `streambuf`.

Пример 11.4. Добавление потока к `stringstream` по одному символу

```
void stream_read_streambuf(std::istream& f, std::string& result) {
    std::stringstream s;
    s << f.rdbuf();
    std::swap(result, s.str());
}
```

Этот код выполняется в моем тесте за 1 294 мс при компиляции с помощью Visual Studio 2010 и за 1 181 мс при использовании Visual Studio 2015 — соответственно на 17 и 51% быстрее, чем в случае функции `stream_read_streambuf_string()`. Моя гипотеза заключается в том, что используемый код просто выполняет меньше действий.

Снижение количества перераспределений

`stream_read_streambuf_string()` содержит семена надежды на оптимизацию. Несмотря на отсутствие очевидного способа оптимизации, `std::string`, в который считывается файл, может заранее выделить необходимую память путем вызова `reserve()` и тем самым предотвратить дорогостоящее перераспределение памяти по мере роста строки символ за символом. В примере 11.5 проверяется, как эта идея влияет на производительность.

Пример 11.5. `stream_read_string_reserve()`: предварительное выделение памяти

```
void stream_read_string_reserve(std::istream& f,
                                std::string& result)
{
    f.seekg(0,std::istream::end);
    std::streamoff len = f.tellg();
    f.seekg(0);
    if (len > 0)
        result.reserve(static_cast<std::string::size_type>(len));

    result.assign(std::istreambuf_iterator<char>(f.rdbuf()),
                  std::istreambuf_iterator<char>());
}
```

`stream_read_string_reserve()` определяет длину потока, позиционируя его указатель в конец и считывая смещение, а затем сбрасывая указатель потока в начало. Фактически `istream::tellg()` возвращает небольшую структуру, которая дает позицию, включая смещение частично считанного многобайтного символа UTF-8. К счастью, эта структура реализует преобразование в знаковый целочисленный тип. Этот тип должен быть знаковым, поскольку `tellg()` может завершиться ошибкой, возвращая при этом значение `-1`, если, например, поток не был открыт, произошла ошибка, или если достигнут конец файла. Если `tellg()` возвращает значение

смещения, а не -1, то это значение можно использовать в качестве указания для std::string::reserve() для выделения достаточного количества памяти для всего файла заранее, подобно remove_ctrl_reserve() в примере 4.3.

stream_read_string_reserve() проверяет гипотезу о том, что стоимость двойного перемещения указателя файла меньше, чем стоимость перераспределений памяти, устраняемых с помощью вызова reserve(). Заранее это неизвестно. Если переход к концу файла приводит к чтению всех секторов диска, принадлежащих файлу, то это очень существенные затраты. С другой стороны, прочтя эти секторы диска, операционная система может хранить их в кеше, сокращая другие расходы. Это поведение может зависеть от размера файла. C++ может быть способен переместить файловый указатель в конец файла без чтения чего бы то ни было, кроме записи каталога для этого файла, а может быть, для этого имеется функция, зависящая от операционной системы (может, даже и не одна).

Когда имеется множество разных догадок, опытный оптимизатор признает, что стоимость поиска ответов на все эти вопросы оказывается высокой, а выгода — неизвестной. Эксперимент быстро укажет, полезна ли идиома перехода в конец файла для получения его размера. В Windows stream_read_string_reserve() имеет не бо́льшую производительность при тестировании, чем stream_read_streambuf_string(). Однако это может означать всего лишь, что степень улучшения незаметна по сравнению с другими "неэффективностями" этого метода чтения файла.

Внимание, спойлер: методика определения длины потока и предварительное выделение памяти оказывается полезными инструментами. Хороший дизайн библиотеки разделяет повторно используемые инструменты на отдельные функции. Функция stream_size(), реализующая этот метод, приведена в примере 11.6.

Пример 11.6. stream_size(): вычисление длины потока

```
std::streamoff stream_size(std::istream& f) {
    std::istream::pos_type current_pos = f.tellg();
    if (-1 == current_pos)
        return -1;
    f.seekg(0,std::istream::end);
    std::istream::pos_type end_pos = f.tellg();
    f.seekg(current_pos);
    return end_pos - current_pos;
}
```

Функция чтения потока может быть вызвана с уже частично считанным потоком. stream_size() учитывает эту возможность путем сохранения текущей позиции в файле, перехода в конец потока и вычисления разности между концом потока и текущей позицией. Указатель потока восстанавливается в сохраненной позиции до возврата из функции. Такой подход на самом деле более правильный, чем упрощенные вычисления из stream_read_string_reserve(). Это еще один пример того, как хороший дизайн библиотеки делает функции более гибкими и обобщенными.

В примере 11.7 представлена версия stream_read_string_reserve() с вычислением размера файла, выполняемым за пределами функции. Это позволяет разработчику предоставить оценку при вызове функции для потоков, которые не могут

определить свой размер. Поведение функции при передаче значения по умолчанию такое же, как и поведение функции `stream_read_string_reserve()`.

Пример 11.7. `stream_read_string_2()`: обобщенная версия `stream_read_string_reserve()`

```
void stream_read_string_2(std::istream& f,
                          std::string& result,
                          std::streamoff len = 0)
{
    if (len > 0)
        result.reserve(static_cast<std::string::size_type>(len));

    result.assign(std::istreambuf_iterator<char>(f.rdbuf()),
                  std::istreambuf_iterator<char>());
}
```

Этот код выполняется в моем тесте за 1 566 мс при компиляции с помощью Visual Studio 2010 и за 1 766 мс при использовании Visual Studio 2015. Стоимость дополнительного вызова функции `stream_size()`, предположительно не равная нулю, в моем тесте не наблюдалась. С другой стороны, функция `stream_read_string_2()` не была заметно быстрее, чем функция `stream_read_string_reserve()`. Предложенный метод не работает? Мы узнаем об этом позже.

Использование большего входного буфера

Потоки C++ содержат класс, производный от `std::streambuf`, что повышает производительность в случае чтения данных большими блоками. Данные поступают в буфер внутри `streambuf`, из которого они затем извлекаются побайтно с помощью итераторов, как было показано ранее. В некоторых статьях в Интернете утверждается, что увеличение размера входного буфера позволяет повысить производительность. В примере 11.8 показан простой способ сделать это.

Пример 11.8. Увеличение размера внутреннего буфера `std::streambuf`

```
std::ifstream in8k;
in8k.open(filename);
char buf[8192];
in8k.rdbuf()->pubsetbuf(buf, sizeof(buf));
```

Многие в Интернете сообщают о наличии трудностей с использованием `pubsetbuf()`. Эта функция должна быть вызвана после открытия потока и до считывания из потока любых символов. Вызов завершается неудачей, если установлен любой из битов состояния потока (`failbit`, `eofbit`). Буфер должен оставаться корректным до тех пор, пока поток не будет закрыт. Увеличив размер буфера, я получил небольшое, но измеримое (20–50 мс) улучшение производительности при тестовом запуске для большинства функций ввода в этом разделе. Значения свыше 8 Кбайт давали слишком малое дополнительное влияние на время выполнения. Это был довольно разочаровывающий факт, потому что в свое время при увеличении размера аналогичного буфера при работе с `FILE` в C наблюдалось резкое улучшение написанного мною кода. Это еще раз показывает, как предыдущий опыт может увести

разработчика в сторону от правильного пути. Данное улучшение составляет около 5% при времени работы теста 1500 мс, но оно может стать относительно бо́льшим, если удастся сократить общее время выполнения.

Использование построчного чтения

Во введении к этому разделу я отметил, что зачастую файлы для чтения являются текстовыми. Для текстового файла, разделенного на строки, разумно предположить, что функция, которая считывает файл построчно, может уменьшить количество вызовов функций. Кроме того, если результирующая строка обновляется реже, будет меньше копирований и перераспределений. На самом деле в стандартной библиотеке имеется функция под названием `getline()`, и пример 11.9 может использоваться для проверки этой гипотезы.

Пример 11.9. `stream_read_getline()`: построчное чтение файла

```
void stream_read_getline(std::istream& f, std::string& result) {
    std::string line;
    result.clear();
    while (getline(f, line))
        (result += line) += "\n";
}
```

Функция `stream_read_getline()` выполняет добавление к переменной `result`. В начале функции `result` следует очистить, потому что ничто не требует, чтобы эта переменная была пустой при передаче в функцию. `clear()` не возвращает динамический буфер строки диспетчеру памяти. Эта функция просто устанавливает длину строки равной нулю. В зависимости от того, как строковый аргумент использовался до вызова, он может иметь динамический буфер значительного размера, что приведет к уменьшению расходов на распределение памяти.

Тестирование `stream_read_getline()` подтвердило эту гипотезу: код, выполняющий 100 итераций чтения 10 000-строчного файла, выполняется в моем тесте за 1284 мс при компиляции с помощью Visual Studio 2010 и за 1440 мс при использовании Visual Studio 2015.

Хотя `result` может иметь достаточно длинный буфер, который предотвратит выполнение перераспределения, резервирование места на всякий случай — неплохая идея. К `stream_read_getline()` можно добавить такой же механизм, как и используемый в `stream_read_string_2()`, получив в результате функцию, приведенную в примере 11.10.

Пример 11.10. `stream_read_getline_2()`: построчное чтение, предварительное выделение памяти

```
void stream_read_getline_2(std::ifstream& f,
                           std::string& result,
                           std::streamoff len = 0)
{
    std::string line;
    result.clear();
```

```
    if (len > 0)
        result.reserve(static_cast<std::string::size_type>(len));

    while (getline(f, line))
        (result += line) += "\n";
}
```

Эта оптимизация дает едва измеримое улучшение производительности на 3% по сравнению с `stream_read_getline()`. Это улучшение может сочетаться со `streambuf` большего размера, что дает время работы тестов 1193 мс (VS2010) и 1404 мс (VS2015).

Еще один способ увеличения считываемых блоков состоит в использовании функции-члена `sgetn()` класса `std::streambuf`, которая считывает произвольное количество данных в буфер, передаваемый в качестве аргумента. Для файлов разумного размера можно извлечь файл целиком за одно большое чтение. Функция `stream_read_sgetn()` в примере 11.11 иллюстрирует этот подход.

Пример 11.11. `stream_read_sgetn()`

```
bool stream_read_sgetn(std::istream& f, std::string& result) {
    std::streamoff len = stream_size(f);

    if (len == -1)
        return false;

    result.resize(static_cast<std::string::size_type>(len));

    f.rdbuf()->sgetn(&result[0], len);
    return true;
}
```

В `stream_read_sgetn()` функция `sgetn()` копирует данные непосредственно в строку `result`, которая может быть сделана достаточно большой, чтобы хранить данные. Размер потока, таким образом, должен быть определен до вызова `sgetn()`. В данном случае, в отличие от `stream_read_string_2()` из примера 11.7, это значение не является необязательным. Размер определяется вызовом `stream_size()`.

Как отмечалось ранее, `stream_size()` может завершиться неудачно. Было бы неплохо получить указание на происшедшую ошибку из `stream_read_sgetn()`. К счастью, поскольку эта библиотечная функция использует идиому отсутствия копирования (см. раздел "Библиотеки без копирования" главы 6, "Оптимизация переменных в динамической памяти"), возвращаемое значение может указывать успешность завершения функции.

Функция `stream_read_sgetn()` — очень быстрая. Мой тест с ней завершился за 307 мс (VS2010) и за 148 мс (VS2015), что более чем в 4 раза быстрее, чем `stream_read_streambuf()`. В сочетании с `rdbuf` большего размера время выполнения тестов удалось уменьшить до 244 мс (VS2010) и 134 мс (VS2015). Скромные улучшения от `rdbuf` большего размера дают больший вклад, если общее время работы меньше.

Еще одно сокращение цепочек вызовов

`std::istream` предоставляет функцию-член `read()`, которая копирует символы непосредственно в буфер. Эта функция имитирует функции низкого уровня `read()` в Linux и `ReadFile()` в Windows. Если `std::istream::read()` подключается непосредственно к этой функции низкого уровня, минуя буферизацию и прочий багаж потоков ввода-вывода в C++, этот вызов должен быть более эффективным. Кроме того, если весь файл может быть прочитан сразу, это может дать нам очень эффективный вызов. В примере 11.12 реализуется эта функциональность.

Пример 11.12. `stream_read_string()` использует `read()` для чтения в строку

```
bool stream_read_string(std::istream& f, std::string& result) {
    std::streamoff len = stream_size(f);
    if (len == -1)
        return false;

    result.resize (static_cast<std::string::size_type>(len));
    f.read(&result[0], result.length());
    return true;
}
```

Этот код выполняется в моем тесте за 267 мс при компиляции с помощью Visual Studio 2010 и за 144 мс при использовании Visual Studio 2015 — примерно на 25% быстрее, чем в случае функции `stream_read_sgetn()`, и в 5 раз быстрее `file_reader()`.

Проблемой в случае `stream_read_sgetn()` и `stream_read_string()` является предположение о том, что указатель `&s[0]` указывает на непрерывный блок памяти. До C++11 стандарт C++ не требовал от символов строки последовательного хранения, хотя все реализации стандартной библиотеки, которые я знаю, были закодированы именно таким образом. Стандарт C++11 в разделе 21.4.1 ясно указывает, что строка хранится в памяти непрерывно.

Я тестировал функцию, которая динамически выделяет массив символов, в который затем считываются данные, копируемые после в строку с помощью вызова `assign()`. Эта функция может использоваться, даже если реализация строки нарушает правило непрерывного размещения в памяти:

```
bool stream_read_array(std::istream& f, std::string& result) {
    std::streamoff len = stream_size(f);
    if (len == -1)
        return false;

    std::unique_ptr<char> data(new char[static_cast<size_t>(len)]);
    f.read(data.get(), static_cast<std::streamsize>(len));
    result.assign(data.get(), static_cast<std::string::size_type>(len));
    return true;
}
```

Этот код выполняется в моем тесте за 307 мс при компиляции с помощью Visual Studio 2010 и за 186 мс при использовании Visual Studio 2015, что только немного медленнее, чем `stream_read_string()`.

Бесполезные вещи

Я встречал даваемый вполне серьезно совет создать собственный `streambuf` для повышения производительности. В примере 11.13 приведена одна такая функция, пойманная в диких дебрях Интернета.

Пример 11.13. Дети, не вздумайте делать это дома!

```
// Источник: http://stackoverflow.com/questions/8736862
class custombuf : public std::streambuf
{
public:
    custombuf(std::string& target): target_(target) {
        this->setp(this->buffer_, this->buffer_ + bufsize - 1);
    }
private:
    std::string& target_;
    enum { bufsize = 8192 };
    char buffer_[bufsize];
    int overflow(int c) {
        if (!traits_type::eq_int_type(c, traits_type::eof())) {
            *this->pptr() = traits_type::to_char_type(c);
            this->pbump(1);
        }
        this->target_.append(this->pbase(),
                             this->pptr() - this->pbase());
        this->setp(this->buffer_, this->buffer_ + bufsize - 1);
        return traits_type::not_eof(c);
    }
    int sync() { this->overflow(traits_type::eof()); return 0; }
};

std::string stream_read_custombuf(std::istream& f) {
    std::string data;
    custombuf sbuf(data);
    std::ostream(&sbuf) << f.rdbuf() << std::flush;
    return data;
}
```

Проблема этого кода в том, что он является попыткой оптимизировать неэффективный алгоритм. Как отмечалось ранее (в `stream_read_streambuf()`), вставка `streambuf` в `ostream` не дает особого эффекта. Хронометраж дает значения времени 1312 мс (VS2010) и 1182 мс (VS2015), что не лучше, чем у `stream_read_streambuf()`. Небольшое улучшение, вероятно, вызвано использованием буфера объемом 8 Кбайт в пользовательском `streambuf`, чего можно добиться с помощью всего лишь пары строк кода.

Запись файлов

Чтобы проверить свои функции чтения файлов, мне нужно было создавать эти файлы. Это позволило мне заодно протестировать функции записи в файлы. Моя первая попытка записи файла выглядела так, как показано в примере 11.14.

Пример 11.14. `stream_write_line()`

```
void stream_write_line(std::ostream& f, std::string const& line) {
    f << line << std::endl;
}
```

Эта функция вызывается 10 тысяч раз для создания файла, и эти вызовы повторяются 100 раз для хронометража, который показывает время работы теста 1972 мс (VS2010) и 2 110 мс (VS2015).

Функция `stream_write_line()` завершает каждую строку `std::endl`. Я не знал об `std::endl` того, что его вывод выполняет сброс выходных данных. Без `std::endl` запись должна выполняться быстрее, потому что `std::ofstream` передает операционной системе для записи только большие блоки данных. Эта гипотеза проверяется в примере 11.15.

Пример 11.15. `stream_write_line_noflush()`

```
void stream_write_line_noflush(std::ostream& f,
                               std::string const& line)
{
    f << line << "\n";
}
```

Конечно, после вызова `stream_write_line_noflush()` должен быть вызов `f.flush()` или закрытие потока, чтобы был сброшен последний буфер. Функция `stream_write_line_noflush()` создала файл за 367 мс (VS2010) и 302 мс (VS2015), что примерно в 5 раз быстрее, чем при использовании `stream_write_line()`.

Я также вызывал `stream_write_line_noflush()` с передачей всего файла как одной строки. Как и ожидалось, это оказалось гораздо быстрее, так что тестовый цикл был завершен за 132 мс (VS2010) и 137 мс (VS2015). Это примерно в 1,7 раза быстрее, чем при построчном выводе файла.

Чтение из `std::cin` и запись в `std::cout`

При чтении из стандартного ввода следует знать, что `std::cin` связан с `std::cout`. Запрос ввода из `std::cin` сбрасывает буфер `std::cout`, чтобы интерактивная консольная программа могла отображать свои запросы. Вызов `istream::tie()` дает указатель на связанный поток, если таковой имеется. Вызов `istream::tie(nullptr)` разрушает существующую связь. Сброс буферов, как было показано в предыдущем разделе, является довольно дорогостоящей операцией.

Еще один момент, о котором следует знать в связи с `std::cin` и `std::cout`, — потоки C++ концептуально связаны с объектами C типа `FILE*` (с `stdin` и `stdout`). Это позволяет программе использовать операторы ввода-вывода C++ и C и осмысленно чередовать ввод и вывод на разных языках. Способ подключения `std::cout` к `stdout` зависит от реализации. Большинство реализаций стандартной библиотеки по умолчанию связывают `std::cout` непосредственно с `stdout`. Поток `stdout` линейно буферизован по умолчанию, и этот режим отсутствует у потоков C++. Когда `stdout` видит символ новой строки, он сбрасывает буфер.

Производительность повышается, если эту связь разорвать. Вызов статической функции-члена `std::ios_base::sync_with_stdio(false)` разрывает эту связь, повышая производительность за счет непредсказуемого чередования вывода, если программа использует одновременно функции C и C++.

Насколько это повышает производительность, я не тестировал.

Резюме

- *Код "быстрого" файлового ввода-вывода в Интернете не обязательно быстр, независимо от того, в чем вас пытаются убедить.*
- *Увеличение размера* `rdbuf` *дает улучшение производительности чтения файлов на несколько процентов.*
- *Наилучшая производительность при чтении файлов получается при использовании функции* `std::streambuf::sgetn()` *для заполнения строкового буфера размером, равным размеру файла.*
- `std::endl` *сбрасывает выходные буфера. Если вы не выполняете вывод на консоль, это достаточно дорогое удовольствие.*
- `std::cout` *связан с* `std::cin` *и* `stdout`. *Разрыв этой связи может повысить производительность.*

ГЛАВА 12

Оптимизация параллельности

Трудно делать прогнозы, особенно о будущем.

— Йоджи Берра (Yogi Berra) (1925–2015), легенда бейсбола и непреднамеренный юморист

Все современные компьютеры, кроме разве что самых маленьких, выполняют несколько потоков одновременно. Они содержат несколько ядер процессора, графические процессоры с сотнями простых ядер, звуковые процессоры, контроллеры, сетевые карты и даже клавиатуры с отдельной вычислительной мощностью и памятью. Нравится нам это или нет, но параллельные вычисления завоевали этот мир, так что разработчики обязаны понимать, как программировать параллельно выполняемые действия.

Методы параллельного программирования эволюционировали в мире одноядерных процессоров. С появлением в прошлом десятилетии многоядерных микропроцессоров, обеспечивающих истинный параллелизм (в отличие от основанного на временны*х* интервалах, выделяемых разным процессам), изменились принципы разработки, передовые практики программирования оснастились новыми принципами и правилами. Эти правила могут быть незнакомы даже разработчикам, имеющим опыт работы с параллельностью в однопроцессорных системах.

Если будущее направление развития процессоров приведет к появлению коммерческих устройств с десятками или сотнями ядер, лучшие практики программирования будут продолжать изменяться. Однако аппаратное обеспечение общего назначения со множеством ядер пока еще не стало распространенным[1], сообщество разработчиков еще не выработало окончательно устоявшиеся методы программирования, а явно выраженных лидеров среди решений для мелкозернистого параллелизма до сих пор нет. Будущее пока что не определено. Об этом можно долго говорить, но, увы, мне надо раскрыть другие темы, в большей степени связанные с названием книги.

Параллельность может предоставляться программам с помощью различных механизмов. Некоторые из них находятся за пределами C++, в операционной системе или оборудовании. Код C++ представляется работающим в нормальном режиме как

[1] Да, да, я слышал о GPU. Когда миллионы программистов станут работать непосредственно с GPU, это будет основным направлением в программировании. Но пока что это время еще не наступило.

отдельная программа или группы программ, общающихся между собой с помощью системы ввода-вывода. Тем не менее эти подходы к параллелизму имеют определенное влияние на разработку программ C++.

Стандартная библиотека C++ непосредственно поддерживает модель параллелизма с общей памятью. Многие разработчики лучше знакомы с функциями параллелизма своей операционной системы или вызовами библиотеки POSIX Threads (pthreads) на языке C. Мой опыт показывает, что разработчики гораздо меньше знакомы с функциями параллелизма в C++ и параллелизмом в целом, чем с другими областями программирования C++. По этой причине я предоставлю более обширные заметки о том, как могут быть использованы эти функции C++ по сравнению с другими функциями, описываемыми в этой книге.

Поддержка параллелизма в стандартной библиотеке C++ является областью, над которой ведется интенсивная работа. Хотя стандарт добился огромных успехов в обеспечении основополагающих концепций и возможностей, некоторые возможности до сих пор проходят процедуру стандартизации и не появятся в стандарте языка до C++17 или более поздней версии.

В этой главе рассматривается несколько способов повышения производительности параллельных программ на основе потоков. Предполагается, что читатели на базовом уровне уже знакомы с параллелизмом на уровне потоков и примитивами синхронизации и ищут пути оптимизации многопоточных программ. Обеспечение "базового уровня знаний" параллелизма на уровне потоков должно быть темой еще одной книги.

Введение в параллельные вычисления

Параллелизм представляет собой одновременное (или кажущееся одновременным) выполнение нескольких потоков управления. Целью параллельной обработки является не сокращение количества выполненных инструкций или доступа к словам данных, как таковым. Это скорее увеличение эффективности использования вычислительных ресурсов, сокращающее общее время работы программы.

Параллелизм повышает производительность, позволяя одним программам продолжать работу в то время, когда другие ожидают наступления некоторого события или когда ресурс станет доступным. Это позволяет вычислительным ресурсам использоваться с большей эффективностью. Чем больше одновременно выполняемых действий, тем больше ресурсов используется и больше программ ожидают событий и ресурсов. Положительная обратная связь увеличивает общее использование вычислительных ресурсов и ресурсов ввода-вывода до некоторого момента насыщения, который, я надеюсь, близок к 100%-ному использованию имеющихся ресурсов. Результатом является уменьшение времени работы по сравнению с тем, которое потребовалось бы, если бы каждая задача полностью выполнялась до начала следующей задачи, с простоем компьютера в ожидании событий.

С точки зрения оптимизации задача параллелизма — найти достаточное количество независимых задач для использования имеющихся вычислительных ресурсов в полной мере, даже если некоторые задачи должны будут ожидать внешних событий или доступности ресурсов.

C++ предлагает скромную библиотеку для параллелизма на основе потоков с общей памятью. Это отнюдь не единственный способ, которым программы C++ могут реализовывать систему взаимодействующих программ. Другие виды параллелизма также оказывают влияние на программы на C++ и, таким образом, стоят хотя бы беглого взгляда на них.

В этом разделе рассматриваются модель памяти C++ и основной инструментарий для использования общей памяти в многопоточных программах. На мой взгляд, это самая сложная тема в C++. Дело в том, что наши печально маленькие обезьяньи мозги мыслят одним потоком. Они могут естественно рассуждать одновременно только об одном причинно-следственном потоке.

Экскурсия по зоопарку параллелизма

Параллелизм может предоставляться программам аппаратным обеспечением компьютеров, операционными системами, библиотеками функций и возможностями самого C++. В этом разделе описывается ряд возможностей параллелизма и их влияние на C++.

Тот факт, что одни функции параллелизма встроены в C++, в то время как другие предоставляются кодом библиотеки или операционной системой, не должен рассматриваться как превосходство одной модели параллелизма над другими. Некоторые возможности являются встроенными потому, что они должны быть таковыми — иного способа их предоставления нет. Наиболее известные формы параллелизма включают следующие.

Квантование времени

Это функция планировщика операционной системы. При квантовании времени операционная система поддерживает список одновременно выполняющихся программ и задач системы и выделяет интервалы времени для каждой программы. Всякий раз, когда программа ожидает события или ресурса, она убирается из списка работающих программ операционной системы, делая свою долю процессора доступной для других программ.

Операционная система зависит от процессора и аппаратного обеспечения. Она использует таймер и периодические прерывания для планирования процессов. C++ остается в неведении о квантовании времени.

Виртуализация

В одной типичной разновидности виртуализации легковесная операционная система под названием *гипервизор* выделяет интервалы процессорного времени *гостевой виртуальной машине*. Гостевая виртуальная машина содержит образ файловой системы и образ памяти, типичные для операционной системы, в которой работает одна или несколько программ. Когда гипервизор запускает гостевую виртуальную машину, некоторые команды процессора и доступ к определенным областям памяти перехватываются гипервизором, позволяя низкоуровневой оболочке эмулировать устройства ввода-вывода и другие аппаратные ресурсы. При другом типе виртуализации обычная операционная

система выступает в роли принимающей системы (хоста) для гостевых виртуальных машин, с использованием возможностей ввода-вывода операционной системы для более эффективной эмуляции операций ввода-вывода при одной и той же операционной системе хоста и гостевых виртуальных машин.

Преимущества виртуализации таковы.

- Гостевые виртуальные машины, когда они не запущены, представляют собой файлы на диске. Гостевые виртуальные машины могут быть проверены и сохранены, загружены и возобновлены, а также скопированы и работать на нескольких компьютерах.
- Несколько виртуальных машин могут работать одновременно, если позволяют наличные ресурсы. Каждая гостевая виртуальная машина изолирована от других машин гипервизором (в сотрудничестве с аппаратной защитой виртуальной памяти компьютера).
- Гостевые виртуальные машины можно настроить на использование только части имеющихся ресурсов (физической памяти, ядер процессора) компьютера. Вычислительные ресурсы могут быть адаптированы в соответствии с требованиями программ, выполняемых каждой гостевой виртуальной машиной, обеспечивая согласованные уровни производительности и предотвращая случайное взаимодействие между несколькими виртуальными машинами, работающими одновременно на одном оборудовании.

Как и при традиционном квантовании времени, C++ по-прежнему не знает, что он работает на гостевой виртуальной машине под управлением гипервизора. Программам на C++ может быть косвенно известно, что они имеют ограниченные ресурсы. Виртуализация влияет на разработку программ C++, потому что она может ограничивать вычислительные ресурсы, потребляемые программой, так что программе нужно знать, какие ресурсы при работе ей действительно доступны, и работать с ними.

Контейнеризация

Контейнеризация похожа на виртуализацию в том, что контейнеры содержат образы файловой системы и памяти, хранящие состояние записанной программы. Она отличается тем, что контейнером является операционная система, так что операции ввода-вывода и системные ресурсы могут быть предоставлены непосредственно, а не с помощью менее эффективной эмуляции гипервизором.

Контейнеризация имеет те же преимущества, что и виртуализация (упаковка, настройка и изоляция), а кроме того, контейнеры могут работать более эффективно.

Контейнеризация невидима для программ на C++, запущенных в контейнере. Она влияет на разработку программ на C++ так же, как и виртуализация.

Симметричная мультипроцессорная обработка

Симметричная мультипроцессорная обработка представляет собой компьютер, содержащий несколько исполнительных устройств, выполняющих один и тот же машинный код и имеющих доступ к одной и той же физической памяти.

Современные многоядерные процессоры являются симметричными мультипроцессорами. Выполняющиеся в данный момент программы и системные задачи могут быть запущены на любом доступном исполнительном устройстве, хотя выбор блока может влиять на производительность.

Симметричная мультипроцессорная обработка выполняет несколько потоков управления с истинным аппаратным параллелизмом. Если имеется *n* блоков, общее время выполнения программы может быть уменьшено (не более чем) в *n* раз. Эти аппаратные потоки отличны от рассматриваемых далее программных потоков, которые могут выполняться на различных аппаратных потоках (но могут и не выполняться таким образом), и, соответственно, могут привести к уменьшению общего времени выполнения (или оставить его неизменным).

Одновременная многопоточная обработка

Некоторые процессоры разработаны таким образом, что каждое аппаратное ядро имеет два (или более) набора регистров и выполняет два или более соответствующих потоков команд. Когда один поток приостанавливается (например, для доступа к основной памяти), могут выполняться команды другого потока. Ядро процессора с такой характеристикой работает как два или более ядер, так что "четырехъядерный процессор" в действительности может поддерживать восемь аппаратных потоков. Как мы увидим позже, в разделе "Создавайте потоков столько же, сколько имеется ядер", это важно, потому что наиболее эффективное использование программных потоков происходит тогда, когда количество программных потоков соответствует числу аппаратных потоков.

Несколько процессов

Процессы представляет собой параллельно выполняющиеся потоки, которые имеют собственные пространства защищенной виртуальной памяти. Процессы обмениваются информацией с использованием каналов, очередей, сетевого ввода-вывода или некоторых других не разделяемых механизмов[2]. Для синхронизации процессов используются примитивы синхронизации или ожидание входных данных (т.е. блокирование до тех пор, пока входные данные не становятся доступными).

Основным преимуществом процессов является то, что операционная система изолирует один процесс от другого. Если один процесс аварийно завершает работу, другие продолжают работать, хотя и могут ничего не делать.

Главная слабость процессов заключается в том, что они имеют много состояний: таблицы виртуальной памяти, несколько контекстов исполнительных устройств, а также контексты всех приостановленных потоков. Они медленнее запускаются и останавливаются; переключение между ними выполняется медленнее, чем между потоками.

[2] Некоторые операционные системы разрешают процессам разделять специально выделенный общий блок памяти. Такие механизмы совместного использования памяти зависят от операционных систем и здесь не рассматриваются.

C++ не работает с процессами непосредственно. Как правило, программы на языке C++ представлены процессами операционной системы. В C++ нет средств для управления процессами, поскольку даже не все операционные системы имеют концепцию процесса. На небольших процессорах программы могут работать параллельно с использованием квантования времени, не будучи защищенными одна от другой, так что они действуют в большей степени как потоки.

Распределенные вычисления

Распределенные вычисления представляют собой разделение вычислений среди набора процессоров (не обязательно на одной машине), взаимодействующих через связи, которые обычно гораздо медленнее по сравнению с процессором. Примером таких вычислений является кластер облачных экземпляров, взаимодействующих по TCP/IP. Примером распределенных вычислений на одном компьютере являются также интеллектуальные аппаратные решения, разгружающие драйверы путем переноса обработки в устройства дисков или сетевых карт. Еще одним примером является передача графических задач от процессора многим видам специализированных графических процессоров (GPU).

В типичной системе распределенных вычислений данные пересылаются через конвейер или сети процессов, в которых каждый процесс выполняет некоторые преобразования входных данных, передавая обработанные данные на следующий этап конвейера. Эта модель (такая же старая, как конвейеры командной строки Unix) позволяет эффективно взаимодействовать относительно тяжеловесным процессам. Процессы в конвейере — долгоживущие, что позволяет сократить стоимость их запуска. Процессы могут непрерывно выполнять обработку, поэтому при наличии входных данных они используют кванты времени полностью. Самое главное — процессы не используют общую память, не синхронизируются один с другим и поэтому работают на полной скорости.

Хотя в C++ нет понятия процесса, распределенные вычисления имеют отношение к разработке C++, потому что они влияют на дизайн и структуру программ. Совместное использование памяти не масштабируется за пределы нескольких потоков. Некоторые трактовки параллелизма вообще выступают за полный отказ от общей памяти. Системы распределенных вычислений часто естественным образом распадаются на подсистемы, приводя к модульным, понятным, перенастраиваемым архитектурам.

Потоки

Потоки (threads) представляют собой параллельные потоки выполнения в рамках процесса, которые разделяют общую память. Потоки синхронизируются с использованием примитивов синхронизации и взаимодействуют один с другим с использованием общей памяти.

Их сильной стороной, по сравнению с процессами, является то, что потоки потребляют меньше ресурсов, быстрее создаются и обеспечивают более быстрое переключение между ними.

Однако потоки не лишены недостатков. Поскольку все потоки процесса разделяют одно пространство памяти, поток, записывающий данные в некоторое местоположение в памяти, может перезаписать структуры данных в других потоках, приводя к аварийному завершению их работы или непредсказуемому поведению. Кроме того, доступ к общей памяти во много раз медленнее доступа к неразделенной памяти и должен быть синхронизирован между потоками, иначе содержимое памяти будет трудно истолковать.

Большинство операционных систем поддерживают потоки с помощью библиотек, зависящих от операционных систем. До недавнего времени разработчики на C++ с опытом в области параллелизма активно использовали библиотеки потоков, предоставляемые операционными системами, или библиотеку потоков POSIX (pthreads), которая обеспечивает кроссплатформенное решение, предоставляющее основные службы.

Задания

Задание — это единица действия, которая может быть выполнена асинхронно в контексте отдельного потока. В параллелизме на основе заданий управление заданиями и потоками выполняется отдельно и явным образом, так что задание назначается потоку для выполнения. Напротив, в параллелизме на основе потоков как единое целое управляются поток и выполнимый код, выполняющийся в этом потоке.

Параллелизм на основе заданий строится поверх потоков, поэтому задания разделяют сильные и слабые стороны потоков.

Дополнительным преимуществом параллелизма на основе заданий является то, что количество активных программных потоков может соответствовать количеству аппаратных потоков, так что потоки будут выполняться эффективно. Программа может назначать приоритеты заданиям и ставить их в очереди на выполнение. Напротив, в системе, основанной на потоках, приоритеты потокам назначает операционная система, при этом непрозрачным и зависимым от операционной системы способом.

Стоимость дополнительной гибкости заданий выражается в большей сложности приложений. Программа должна реализовывать некоторые средства для определения приоритетности или последовательности заданий. Программа также должна управлять пулом потоков, которые выполняют задания.

Чередующееся выполнение

У астрономов достаточно интересный взгляд на Вселенную. Водород составляет 73% видимой материи во Вселенной, а гелий — 25%. Все остальное составляет менее 2%. Астрономы могут описать большинство свойств наблюдаемой Вселенной, как если бы она состояла исключительно из водорода и гелия. Они называют все прочие элементы — те, из которых состоят планеты, процессоры и люди — "металлами", как будто они не являются отдельными элементами, и в значительной степени игнорируют их наличие.

Параллельные программы также могут быть абстрагированы как состоящие из загрузок, сохранений и ветвлений, причем ветвления в основном игнорируются, как если бы вся сложность программирования не имела никакого значения. В обсуждении параллелизма (в том числе в этой книге) часто для иллюстрации концепций параллелизма используются простые фрагменты кода, состоящие преимущественно из последовательностей операторов присваивания.

Одновременное выполнение двух потоков управления может быть смоделировано как чередование простых инструкций загрузки и сохранения двух потоков. Если поток 1 и поток 2 состоят из одного выражения, возможными чередованиями являются "12" и "21". Если каждый поток имеет по две инструкции, количество потенциальных чередований растет: "1122", "1212", "2112", "1221", "2121", "2211". В реальных программах количество вариантов чередования представляет собой очень большое число.

Во времена одноядерных процессоров параллелизм был реализован с помощью квантования времени в операционной системе. Состояния гонки были сравнительно редки, поскольку выполнялось много инструкций одного потока до того, как операционная система передавала управление другому потоку. Чередования на практике имели вид "1111...11112222...2222".

При использовании современных многоядерных процессоров возможны чередования отдельных инструкций, поэтому состояния гонки встречаются куда чаще. Разработчики, которые писали параллельные программы в прошлом, для однопроцессорных систем, могут испытывать основанную на прошлом опыте самоуспокоенность, но это чувство больше не является оправданным.

Последовательная согласованность

Как упоминалось в главе 2, "Оптимизация, влияющая на поведение компьютера", C++ верит в простую, интуитивно понятную модель компьютера. Она включает требования последовательной согласованности программ. Это означает, что программы ведут себя так, как если бы их инструкции выполнялись в том порядке, в котором они написаны, с учетом инструкций управления потоком выполнения C++. Понятно, что слова "как если бы" в предыдущем предложении разрешают многие оптимизации компиляторов и инновационные конструкции микропроцессоров.

Например, фрагмент программы в примере 12.1 последовательно согласован, если `y` получает значение `0` до `x` или `x` получает значение `1` после того, как `y` получает значение `1`, или даже если `x` получает значение `1` после сравнения `y == 1` в инструкции `if`, лишь бы присваивание `x = 1` было выполнено до `assert(x == 1)`, т.е. до того, как будет использовано это значение.

Пример 12.1. Последовательная согласованность означает выполнение "как если бы" по порядку

```
int x = 0, y = 0;
x = 1;
y = 1;
if (y == 1) {
    assert(x == 1);
```

Читатель может спросить: "А зачем компилятору менять порядок инструкций?" Тому имеется множество причин, и все они порождены темной магией создания оптимального кода компиляторами, о которой я просто не осмеливаюсь говорить вслух. И этим перемещением инструкций занимается не только компилятор; современные микропроцессоры также меняют порядок загрузок и сохранений (см. раздел "Множественные потоки выполнения" главы 2, "Оптимизация, влияющая на поведение компьютера").

Совокупный эффект оптимизации компилятора, выполнения команд вне порядка, кеширования и буферизации записи можно моделировать с помощью общей метафоры загрузок и сохранений, свободно разгуливающих по листингу программы и перемещающихся вверх и вниз по коду в точку до или после их предполагаемого положения. Эта метафора охватывает все эти эффекты без необходимости детального объяснения или понимания оптимизации конкретного компилятора и поведения конкретного процессора.

В конкретной комбинации "компилятор/процессор" на самом деле происходят не все возможные перемещения загрузок и сохранений. Таким образом, свободное перемещение загрузок и сохранений совместно используемых переменных иллюстрирует наихудший сценарий. Но пытаться рассуждать о конкретной аппаратной архитектуре, когда она имеет три уровня кеша и поведение, которое зависит от того, в какой строке кеша находится каждая переменная, безумно трудно. Это занятие попросту бесполезно, потому что большинство программ предназначены для работы на более чем одном устройстве за время их жизненного цикла.

Важно то, что программа до тех пор остается последовательно согласованной при перемещении использования переменной вверх или вниз по отношению к другим инструкциям, пока оно не проскакивает мимо обновления этой переменной. Аналогично программа до тех пор остается последовательно согласованной при перемещении обновления переменной вверх или вниз, пока оно не заходит за ее использование.

Гонки

Параллелизм создает проблему для языка C++, который не предоставляет возможности сказать, когда две функции могут выполняться одновременно и какие переменные являются совместно используемыми. Оптимизация с перемещением кода, которая вполне разумна при рассмотрении одновременного выполнения только одной функции, может вызвать проблемы, если две функции будут выполняться одновременно.

Если поток 1 состоит из инструкции `x = 0`, а поток 2 — из инструкции `x = 100`, то результат работы программы зависит от *гонки* между этими двумя потоками. Гонка возникает, когда результат параллельного выполнения инструкций зависит от того, в каком порядке они чередуются при данном выполнении программы. Чередование "12" дает результат `x == 100`, а чередование "21" — `x == 0`. Результат этой программы, как и результат любой программы, содержащей гонку, *недетерминированный*, т.е. непредсказуемый.

Модель стандартной памяти C++ гласит, что программы до тех пор ведут себя, как если бы они были последовательно согласованными, пока они не содержат гонок. *Если программа содержит гонки, последовательная согласованность может быть нарушена.*

Пример 12.2 представляет собой многопоточную версию примера 12.1, в которой переменным даны более значащие имена.

Пример 12.2. Последовательная согласованность нескольких потоков

```
// Поток 1, выполняется на ядре 1
shared_result_x = 1;
shared_flag_y   = 1;

...

// Поток 2, выполняется на ядре 2
while (shared_flag_y != 1)
    /* Ожидание, пока shared_flag_y получит значение 1 */ ;
assert(shared_result_x == 1);
```

`shared_result_x` представляет собой результат вычислений в потоке 1, используемый потоком 2. `shared_flag_y` представляет собой флаг, устанавливаемый потоком 1, который говорит потоку 2, что результат готов к использованию. Если процессор или компилятор меняет порядок двух инструкций в потоке 1 так, что значение `shared_flag_y` устанавливается до значения `shared_result_x`, то поток 2 может (но не обязательно должен) выйти из своего цикла `while`, увидев новое значение `shared_flag_y`, но не пройти вызов `assert`, потому что он по-прежнему видит старое значение `shared_result_x`. Каждый поток последовательно согласован, но взаимодействие двух потоков представляет собой гонку.

Последовательная согласованность переменных, совместно используемых несколькими потоками, гарантируется некоторыми языками, которые не имеют досадных "как если бы" в своих определениях. В других языках предусматривается, что такие совместно используемые переменные должны быть явно объявлены как таковые; компилятор не будет их перемещать и сгенерирует специальный код, гарантирующий, что и аппаратное обеспечение не будет их как-либо перемещать. Параллельная программа на C++ должна явным образом обеспечить конкретное чередование для сохранения последовательной согласованности.

Синхронизация

Синхронизация — это обеспечение принудительного порядка чередования инструкций в нескольких потоках. Она позволяет разработчикам говорить о порядке, в котором в многопоточной программе происходят те или иные действия. Без синхронизации такой порядок может быть непредсказуемым, что затрудняет координацию деятельности потоков.

Примитивы синхронизации представляют собой программные конструкции, целью которых является достижение синхронизации путем обеспечения конкретного чередования в параллельных программах. Все примитивы синхронизации работают, заставляя один поток *ожидать* другой поток. Примитивы синхронизации предотвращают гонки путем обеспечения конкретного порядка выполнения.

За последние 50 лет программирования были предложены и реализованы различные примитивы синхронизации. Microsoft Windows имеет богатый набор примитивов синхронизации, включая события, которых могут ожидать потоки, два вида мьютексов, весьма обобщенный семафор и сигналы в стиле Unix. Linux имеет собственный богатый, но отличающийся набор.

Важно понимать, что примитивы синхронизации существуют только как концепции. Нет никакого голоса свыше, который точно говорит о том, что такое семафор или как именно должен быть реализован монитор. В том, что в Windows называется семафором, трудно распознать оригинальное описание Дейкстры[3]. Кроме того, все множество предлагаемых примитивов синхронизации может быть синтезировано из достаточно богатого набора базовых элементов таким же образом, как любая булева функция может быть синтезирована аппаратно с помощью логических вентилей НЕ-И и НЕ-ИЛИ. Таким образом, в разных операционных системах можно ожидать различные реализации примитивов синхронизации.

Классические примитивы синхронизации взаимодействуют с операционной системой для перемещения потоков между активным состоянием и состоянием ожидания. Эта реализация подходит только для компьютеров с только одним относительно медленным исполнительным устройством. Однако задержка запуска и остановки потоков с помощью операционной системы могут быть значительными. Когда имеется несколько процессоров, выполняющих потоки команд в истинно параллельном режиме, синхронизация с помощью ожидания общих переменных может приводить к очень небольшим временам ожидания. Проектировщики библиотек синхронизации могут также принять гибридный подход.

Атомарность

Операция над общей переменной (в частности, над экземпляром класса, имеющем несколько членов) является *атомарной*, если ни один поток не может увидеть выполняющееся другим потоком обновление в незавершенном состоянии. Если операции обновления не являются атомарными, то некоторое чередование двух потоков позволяет одному потоку получить доступ к совместно используемой переменной, когда та находится в несогласованном состоянии, поскольку ее обновление все еще выполняется в другом потоке. Можно сформулировать и так: атомарность — это обещание, что такие нежелательные чередования произойти не могут.

Атомарность с помощью взаимоисключений

Традиционно атомарность обеспечивается с помощью взаимных исключений. Каждый поток, желающий получить доступ к общей переменной, перед этим должен *захватить* мьютекс, а по завершении операции *освободить* его. Часть программы, ограниченная захватом и освобождением мьютекса, называется критическим разделом. Если один поток получил мьютекс, все остальные потоки приостанавливаются, когда пытаются захватить этот мьютекс. Таким образом, одновременно выполнять

[3] Edsger W. Dijkstra, *"Cooperating Sequential Processes (EWD-123)"*, (`http://bit.ly/ewd-123`). E.W. Dijkstra Archive, Center for American History, University of Texas at Austin (September 1965).

операции над совместно используемыми данными может только один поток. Говорят, что этот поток *удерживает* мьютекс. Мьютекс *сериализует* потоки так, что критические разделы выполняются один за другим.

Загрузки и сохранения совместно используемых переменных должны происходить в критическом разделе, где работает только один поток, иначе может возникнуть гонка с непредсказуемыми результатами. Но, как описано выше, в разделе "Последовательная согласованность", и компилятор, и процессор могут перемещать загрузки и сохранения. Механизм, именуемый *барьером памяти* (memory fence), предотвращает загрузки и сохранения совместно используемых переменных от выхода за пределы критического раздела. В процессоре специальные команды требуют от него не перемещать загрузки и сохранения за барьеры памяти. В компиляторе барьер памяти носит концептуальный характер. Оптимизатор не перемещает загрузки через вызовы функций, поскольку любой вызов функции может содержать критический раздел.

Барьеры памяти в верхней части критического раздела должны препятствовать утечке загрузок совместно используемых переменных. Говорят, что такой барьер имеет *семантику захвата*, потому что он связан с получением мьютекса потоком. Аналогично барьер памяти в нижней части критического раздела предотвращает утечку сохранений общих переменных через дно критического раздела. Этот барьер памяти имеет *семантику освобождения*, потому что это происходит, когда поток освобождает мьютекс.

Во времена одноядерных процессоров барьеры памяти были не нужны. Компиляторы не переносили загрузки и сохранения через вызовы функций, и операционная система синхронизировала память почти случайно при переключении потоков. Но в многоядерном мире программисты вынуждены бороться с этой новой проблемой. Разработчикам, использующим примитивы синхронизации, предоставляемые стандартной библиотекой C++ или библиотекой синхронизации операционной системы, не нужно беспокоиться о барьерах памяти, но программисты, реализующие примитивы синхронизации или структуры данных без блокировок, должны проявить к этой теме повышенный интерес.

Атомарные аппаратные операции

Реализация атомарности путем взаимных исключений имеет расходы, которые делают ее достаточно тяжеловесным инструментом.

- Поскольку удерживать мьютекс может только один поток, операции над общими переменными не могут выполняться параллельно. Чем больше времени выполняется критический раздел, тем больше время, в течение которого программа не использует преимущества параллельности. Точно так же чем больше потоков работают с общей переменной, тем больше время, которое критический раздел отбирает у параллельного выполнения программы, лишая ее преимуществ использования параллельности.
- Когда поток освобождает мьютекс, другой поток, находящийся в состоянии ожидания этого мьютекса, может захватить его. Но мьютекс не может ничего

гарантировать в отношении того, *какой* именно другой поток его захватит, потому что такая гарантия стоит очень дорого. Если много потоков находятся в ожидании мьютекса, теоретически некоторые потоки могут никогда не захватить его, а вычисления, выполняемые этими потоками, могут оставаться невыполненными. Такая ситуация называется *голоданием*.

- Если поток удерживает один мьютекс, и ему необходимо захватить второй мьютекс, может возникнуть ситуация, когда поток может навсегда остаться в таком заблокированном состоянии, потому что другой поток захватил второй мьютекс и теперь желает захватить первый. Такая ситуация называется *клинчем*. Поток может заблокировать сам себя, если попытается захватить один и тот же мьютекс дважды. Любое количество потоков может попасть в состояние взаимной блокировки из-за циклической зависимости мьютексов между потоками. Нет никаких известных способов гарантировать отсутствие взаимоблокировок в программе, которая пытается захватить несколько мьютексов, хотя имеются различные стратегии предотвращения взаимоблокировок.

Для простых переменных, таких как целые числа и указатели, определенные операции на некоторых компьютерах могут выполняться атомарно, поскольку эти операции выполняются с помощью одной машинной команды. Эти специальные атомарные команды содержат барьеры памяти, гарантирующие, что команда не будет прервана до полного ее завершения.

Атомарные команды формируют основу для реализации мьютексов и других примитивов синхронизации. Использование только тех операций, которые можно выполнить атомарно на аппаратном уровне, позволяет реализовать на удивление сложные структуры данных, безопасные с точки зрения параллельности. Это так называемое *программирование без блокировок*, поскольку работающему таким образом коду не приходится ожидать захвата мьютекса.

Программы без блокировок масштабируются для большего числа параллельных потоков, но они не являются панацеей. Потоки по-прежнему сериализуются атомарными операциями, включая те, которые выполняются с помощью единственной команды. Однако продолжительность критического раздела оказывается на порядок меньше производительности *са*мого эффективного мьютекса.

Возможности параллельности в C++

В ныне действующем стандарте C++14 поддержка параллелизма в стандартной библиотеке C++ по сравнению с богатыми возможностями популярных операционных систем выглядит несколько спартанской. Отчасти это связано с тем, что стандарт C++ должен содержать только то поведение, которое осуществимо во всех операционных системах. Отчасти дело в том, что в настоящее время все еще идет работа над стандартизацией параллелизма в C++, и в стандарте C++17 запланирован ряд существенных улучшений. Преимущество использования возможностей параллелизма C++, по сравнению с использованием вызовов операционной системы, заключается в том, что определенные в стандарте возможности обеспечивают одинаковое поведение на разных платформах.

Возможности параллельности стандартной библиотеки C++ сочетаются одна с другой, как кирпичи, что позволяет возводить здание параллельных вычислений от потоковой библиотеки операционной системы в стиле C до полностью использующих возможности C++ потоковых решений, с которыми можно обмениваться произвольными списками аргументов, возвращаемыми значениями, исключениями, и которые можно хранить в контейнерах.

Потоки

Заголовочный файл `<thread>` предоставляет шаблонный класс `std::thread`, который позволяет программе создавать объекты потоков в качестве тонких оболочек вокруг потоковых возможностей операционной системы. Конструктор `std::thread` принимает в качестве аргумента *вызываемый объект* (указатель на функцию, функциональный объект, лямбда-выражение иди выражение `bind`) и выполняет его в контексте нового программного потока. C++ использует возможности шаблонов с произвольным количеством аргументов для вызова функций с произвольными списками аргументов, тогда как базовый вызов потока операционной системы обычно принимает указатель на функцию с одним аргументом типа `void*`, которая ничего не возвращает.

Класс `std::thread` представляет собой RAII-класс для управления потоками операционной системы. Он предоставляет функцию-член `get()`, которая возвращает дескриптор потока операционной системы. Это дает программе возможность доступа к в общем случае более богатому набору функций операционной системы для работы с потоками.

В примере 12.3 показан простой пример использования `std::thread`.

Пример 12.3. Запуск нескольких простых потоков

```
void f1(int n) {
    std::cout << "thread " << n << std::endl;
}

void thread_example() {
    std::thread t1;         // Переменная потока, но не поток
    t1 = std::thread(f1,1); // Присваивание потока потоковой переменной
    t1.join();              // Ожидание завершения потока
    std::thread t2(f1, 2);
    std::thread t3(std::move(t2));
    std::thread t4([](){ return;}); // Лямбда-выражения также допустимы
    t4.detach();
    t3.join();
}
```

Поток `t1` изначально пуст. Потоки не могут копироваться, поскольку каждый из них содержит уникальный дескриптор для используемого ресурса операционной системы, но перемещающий оператор присваивания позволяет присвоить rvalue пустому потоку. `t1` может владеть любым потоком, который выполняет функцию, принимающую целочисленный аргумент. Указатель на функцию `f1` и целочисленный аргумент передаются конструктору. Второй аргумент передается вызываемому объекту (`f1`), который запускается в конструкторе `std::thread`.

Поток `t2` запускается с той же функцией, но другим вторым аргументом. Поток `t3` является примером перемещающего конструктора в действии. После вызова перемещающего конструктора `t3` выполняет поток, начатый как `t2`, а `t2` становится пустым. Поток `t4` показывает, что потоки можно запускать с лямбда-выражениями в качестве вызываемых объектов.

Поток операционной системы, представленный `std::thread`, должен быть утилизирован до уничтожения `std::thread`. Поток может быть подключен, как в инструкции `t3.join()`, а это означает, что текущий поток ожидает завершения подключенного потока. Поток операционной системы также может быть отключен от объекта `std::thread`, как в инструкции `t4.detach()`. В этом случае поток продолжает выполняться, но он становится невидим для запустившего его объекта `std::thread`. Отключенный поток завершается, когда выполняется возврат из вызываемого объекта. Если возврат из вызываемого объекта не происходит, поток представляет собой утечку ресурсов, так как продолжает потреблять ресурсы до тех пор, пока не завершится вся программа. Если перед уничтожением `std::thread` не вызваны ни `join()`, ни `detach()`, его деструктор вызывает функцию `terminate()`, которая аварийно завершает выполнение программы.

Хотя можно работать с `std::thread` непосредственно, более продуктивно рассматривать `std::thread` как строительный блок для более сложных объектов. Любое значение, возвращаемое функциональным объектом, игнорируется. Любое исключение, сгенерированное функциональным объектом, вызывает немедленное безусловное завершение программы с помощью вызова функции `terminate()`. Эти ограничения делают вызовы `std::thread` столь хрупкими, что создается впечатление, что разработчики стандарта стремились всячески воспрепятствовать их использованию.

Обещания и фьючерсы

Классы шаблонов C++ `std::promise` и `std::future` представляют собой соответственно отправителя и получателя сообщений от одного потока к другому. Обещания и фьючерсы позволяют потокам асинхронно создавать значения и генерировать исключения. Обещание и фьючерс совместно используют динамически выделенную переменную, которая называется *общим состоянием* и может содержать либо значение определенного типа, либо исключение любого типа, инкапсулированное в стандартной оболочке. Поток выполнения может ожидать фьючерс, так что последний может выступать в качестве устройства синхронизации.

Обещания и фьючерсы могут использоваться просто для реализации асинхронного вызова функции и возврата из нее. Однако на самом деле обещание и фьючерс гораздо более обобщенные, чем это простое использование. Они могут реализовывать граф динамически изменяющихся точек связи между потоками. Но они не предоставляют механизм структурирования, так что совершенно дикие и сумасшедшие графы связей может оказаться трудно отлаживать.

Заголовочный файл C++ `<future>` содержит функциональность обещаний и фьючерсов. Экземпляр шаблона `std::promise` позволяет потоку настроить общее состояние либо как значение определенного типа, либо как исключение. Поток-

отправитель не ожидает чтения общего состояния; он может сразу же продолжить свою работу.

Общее состояние обещания не *готово*, пока не будет установлено значение или исключение. Общее состояние должно быть установлено только один раз. В противном случае происходит следующее.

- Если поток пытается задать значение или исключение более чем один раз, вместо этого общее состояние устанавливается как исключение `std::future_error` с кодом ошибки `promise_already_satisfied`, и общее состояние становится *готовым* освободить любые фьючерсы, ожидающие обещания.
- Если поток не устанавливал значение или исключение, то, когда обещание уничтожается, деструктор устанавливает общее состояние как исключение `std::future_error` с кодом ошибки `broken_promise`, и общее состояние становится *готовым* освободить любые фьючерсы, ждущие обещания. Чтобы получить этот полезный индикатор ошибки, обещания должны уничтожаться в вызываемом объекте потока.

`std::future` позволяет потоку получить значение или исключение, сохраненное в общем состоянии обещания. Фьючерс — это примитив синхронизации; поток-приемник переходит в состояние ожидания в вызове функции-члена `get()` до тех пор, пока соответствующее обещание не будет сделано *готовым* путем вызова для задания значения или исключения.

Фьючерс является недействительным до тех пор, пока не будет построен или связан с обещанием. Поток-приемник не может перейти в состояние ожидания фьючерса до тех пор, пока этот фьючерс не станет действительным. Фьючерс должен быть построен из обещания до выполнения отправляющего потока. В противном случае принимающий поток может попытаться ожидать фьючерс до того, как этот фьючерс станет *действительным*.

Обещания и фьючерсы не могут быть скопированы. Они являются сущностями, которые представляют определенные коммуникационные соединения. Они могут быть сконструированы или созданы перемещением, и обещание может быть связано с фьючерсом. В идеале обещание создается в потоке-отправителе, а фьючерс — в потоке-получателе. Существует идиома, в которой обещание создается в потоке-отправителе, затем передается как rvalue-ссылка с использованием `std::move(promise)` потоку-получателю, так что его содержимое перемещается в обещание, принадлежащее принимающему потоку. Эту странную магию выполняет вызов `std::async()`. Можно также передать обещание по ссылке в поток-отправитель. В примере 12.4 показано, как управлять взаимодействием потоков с помощью обещаний и фьючерсов.

Пример 12.4. Обещания, фьючерсы и потоки

```
void promise_future_example() {
    auto meaning = [](std::promise<int>& prom) {
        prom.set_value(42);      // Вычисление смысла жизни
    };

    std::promise<int> prom;
    std::thread(meaning, std::ref(prom)).detach();
```

```
    std::future<int> result = prom.get_future();
    std::cout << "Смысл жизни: " << result.get() << "\n";
}
```

В примере 12.4 обещание `prom` создается до запуска `std::thread`. Это не столь идеально, как написано, потому что не выполнен тест обещания. Он необходим, потому что если `prom` не построен до запуска потока, то нет никакой возможности гарантировать, что фьючерс `result` будет *действительным* до вызова `result.get()`.

Затем программа создает анонимный `std::thread`. В качестве аргумента ему передается лямбда-выражение `meaning`, которое является вызываемым объектом, который должен быть выполнен, и `prom`, обещание, которое является аргументом `meaning`. Обратите внимание, что поскольку `prom` является ссылочным аргументом, он должен быть обернут в `std::ref()`, чтобы передача аргументов работала правильно. Вызов `detach()` отключает выполняющийся поток от анонимного `std::thread`, который будет уничтожен.

Теперь происходят следующие события: операционная система готовится выполнить `meaning`, а программа создает фьючерс `result`. Программа может выполнить `prom.get_future()`, прежде чем поток начнет работу. Вот почему `prom` был создан до того, как был построен поток, — чтобы фьючерс был действительным, и программа могла приостановиться в ожидании выполнения потока.

В `result.get()` программа ожидает, пока поток установит общее состояние равным `prom`. Поток вызывает метод `prom.set_value(42)`, делая общее состояние *готовым* и позволяя программе продолжаться. Программа заканчивается путем вывода "Смысл жизни: 42".

Во фьючерсах нет ничего магического. Разработчик, желающий создать поток, который сначала возвращает значение типа `int`, а затем `std::string`, может сделать два обещания. Программа-получатель создает два соответствующих фьючерса.

Переход фьючерса в состояние *готовности* сигнализирует, что вычисление завершено. Поскольку программа может находиться в состоянии ожидания фьючерса, ей не обязательно дожидаться окончания потока. Это будет иметь важное значение в разделе "Асинхронные задания", в котором мы обсуждаем `std::async()`, и в разделе "Реализуйте очередь заданий и пул потоков", в котором рассматриваются пулы потоков, потому что этот подход более эффективен для повторного использования потоков, чем их уничтожение и создание заново.

Асинхронные задания

Шаблонный класс задания стандартной библиотеки C++ оборачивает вызываемый объект в блок `try` и сохраняет возвращаемое значение или сгенерированное исключение в обещании. Задания позволяют асинхронно выполнять потоками вызываемые объекты.

Параллельность на основе заданий в стандартной библиотеке C++ сформирована только наполовину. C++11 предоставляет шаблонную функцию `async()`, которая упаковывает вызываемый объект как задание и вызывает его в повторно используемом потоке. `async()` в определенном смысле является "функцией Бога" (см. раздел "Остерегайтесь «функций Бога»" главы 8, "Использование лучших библиотек"), скрывая многие сочные детали пулов потоков и очереди заданий.

Задания представлены в заголовочном файле <future> стандартной библиотеки C++. Шаблон класса std::packaged_task обертывает любой *вызываемый* объект (который может быть указателем на функцию, функциональным объектом, лямбда-выражением или выражением bind) так, чтобы он мог быть вызван асинхронно. Упакованное задание само по себе является вызываемым объектом, который может выступать в роли аргумента std::thread. Большим преимуществом заданий по сравнению с другими вызываемыми объектами является то, что задание может возвращать значение или генерировать исключение без внезапного завершения программы. Возвращаемое значение или сгенерированное исключение задания хранится в общем состоянии, доступ к которому можно получить через объект std::future.

Пример 12.5 представляет собой упрощенную версию примера 12.4, использующую packaged_task.

Пример 12.5. packaged_task и поток

```
void promise_future_example_2() {
    auto meaning = std::packaged_task<int(int)>(
                        [](int n) { return n; });
    auto result  = meaning.get_future();
    auto t       = std::thread(std::move(meaning), 42);

    std::cout << "Смысл жизни: " << result.get() << "\n";
    t.join();
}
```

Переменная meaning типа packaged_task содержит вызываемый объект и std::promise. Это решает проблему вызова деструктора обещания в контексте потока. Обратите внимание, что лямбда-выражение в meaning просто возвращает значение; весь механизм установки обещания удобно скрыт.

В этом примере я подключил поток вместо его отключения. Хотя это не вполне очевидно из данного примера, но основная программа и поток могут продолжать выполняться одновременно, после того как основная программа получает значение фьючерса.

Библиотека <async> предоставляет функциональность на основе заданий. Шаблонная функция std::async() выполняет передаваемый в качестве аргумента *вызываемый объект*, и этот вызываемый объект может быть выполнен в контексте нового потока. Однако std::async() возвращает std::future, способный хранить возвращаемое значение или исключение, сгенерированное вызываемым объектом, выполняемым std::async(). Кроме того, реализация может предпочесть выделение потоков std::async() из пула для повышения эффективности. Это проиллюстрировано в примере 12.6.

Пример 12.6. Задания и async()

```
void promise_future_example_3() {
    auto meaning = [](int n) { return n; };
    auto result  = std::async(std::move(meaning), 42);
    std::cout << "Смысл жизни: " << result.get() << "\n";
}
```

Лямбда-выражение `meaning` и его аргумент передаются `std::async()` в качестве аргументов. Для определения параметров шаблона `std::async()` используется выведение типов. `std::async()` возвращает фьючерс, способный получить результат типа `int` (или исключение), который перемещается в `result`. Вызов `result.get()` приостанавливает выполнение до тех пор, пока поток, вызванный `std::async()`, выполняет свои обещания, возвращая переданный ему аргумент `int`. Завершение потока управляется в функции `std::async()`, которая может оставить поток в пуле потоков.

В приведенном примере кода явное управление завершением потока не требуется. `std::async()` может использовать пул потоков, поддерживаемый системой среды выполнения C++, для повторного использования потока, если это дешевле, чем уничтожение и повторное создание потоков при необходимости. Явные пулы потоков могут быть добавлены в стандарт C++17.

Мьютексы

C++ предоставляет несколько шаблонов мьютексов, чтобы обеспечить взаимное исключение для критических разделов. Определение шаблона мьютекса достаточно простое, чтобы его можно было специализировать для конкретных классов мьютексов операционной системы.

Заголовочный файл `<mutex>` содержит четыре шаблона мьютексов.

`std::mutex`

Простой, относительно эффективный мьютекс. В Windows этот класс сначала пытается получить мьютекс без ожидания, а затем, если это не получается, осуществляет соответствующий вызов операционной системы.

`std::recursive_mutex`

Мьютекс, который позволяет потоку, уже удерживающему его, захватить его снова, как в случае вложенного вызова функции. Этот класс может быть менее эффективным из-за необходимости подсчитывать количество захватов.

`std::timed_mutex`

Мьютекс с тайм-аутом, который позволяет хронометрировать попытки захвата мьютекса. Требование ограниченных по времени попыток захвата, как правило, требует вмешательства операционной системы, что значительно увеличивает задержку этого типа мьютекса по сравнению с `std::mutex`.

`std::recursive_timed_mutex`

Мьютекс, который одновременно является рекурсивным и использующим тайм-аут (а еще приправленный горчицей, кетчупом и секретным соусом; вкусный, но очень дорогой).

Мой опыт подсказывает, что применение рекурсивных мьютексов и мьютексов с тайм-аутом может служить предупреждением о том, что такой дизайн можно упростить. Применение рекурсивного мьютекса трудно обосновать; это скорее приманка для взаимоблокировок. Со стоимостью этих мьютексов следует мириться только тогда, когда это совершенно необходимо, и их следует всячески избегать при проектировании нового кода.

В C++14 добавлен заголовочный файл <shared_mutex>, содержащий поддержку *общих (совместно используемых) мьютексов*, известных также как *мьютексы "читатель/писатель"*. Для того чтобы атомарно обновить структуру данных, один поток может заблокировать общий мьютекс в монопольном режиме. Несколько потоков могут блокировать общий мьютекс в режиме общего использования для атомарного чтения структуры данных, но для монопольного доступа этот мьютекс будет заблокирован до тех пор, пока его не освободят все читатели. Общий мьютекс может обеспечить большему количеству потоков доступ к структуре данных без ожидания при условии, что большинство обращений выполняются для чтения. Доступны следующие общие мьютексы.

`std::shared_timed_mutex`

Совместно используемый мьютекс с поддержкой захвата с тайм-аутом и без такового.

`std::shared_mutex`

Более простой совместно используемый мьютекс, ожидающий включения в C++17.

По моему опыту, мьютексы "читатель/писатель" ведут к голоданию потока писателя, если только чтение не выполняется достаточно редко; но в этом случае величина оптимизации чтения/записи оказывается незначительной. Как и в случае с рекурсивными мьютексами, разработчики должны иметь веские основания для использования этого более сложного мьютекса и в общем случае выбирать более простой и более предсказуемый мьютекс.

Блокировки

В C++ слово *блокировка* (lock) означает класс RAII, который захватывает и освобождает мьютекс структурированным образом. Использование этого слова может несколько вводить в заблуждение, потому что мьютексы также иногда называют блокировками. Захват мьютекса называется также *блокировкой* мьютекса, а освобождение — *разблокированием*. В C++ функция-член мьютекса для его захвата называется `lock()`. Я использую мьютексы более 20 лет, но мне по-прежнему приходится делать усилия, чтобы не путаться в этих понятиях.

Стандартная библиотека C++ предоставляет простую блокировку для захвата одного мьютекса и более обобщенную блокировку для захвата нескольких мьютексов. Обобщенная блокировка реализует алгоритм предотвращения взаимоблокировок.

Заголовочный файл <mutex> содержит два шаблона блокировок.

`std::lock_guard`

Простая блокировка RAII. В конструкторе класса программа ожидает захвата блокировки и освобождает ее в деструкторе `lock_guard`. Реализация этого класса до принятия стандарта часто называлась `scope_guard`.

`std::unique_lock`

Класс — владелец мьютекса общего назначения, который предлагает RAII-блокировку, отложенную блокировку, попытку блокировки с тайм-аутом, передачу владения мьютексом и использование с условными переменными.

В стандарте C++14 в заголовочный файл `<shared_mutex>` добавлена блокировка для совместно используемых мьютексов.

`std::shared_lock`

Класс — владелец мьютекса для общих (читатель/писатель) мьютексов. Он предлагает все сложные возможности `std::unique_lock` плюс управление общими мьютексами.

Один поток может заблокировать общий мьютекс в монопольном режиме, чтобы атомарно обновить структуру данных. Несколько потоков могут заблокировать общий мьютекс в режиме совместного использования для атомарного чтения структуры данных, но он будет заблокирован и для монопольного доступа до тех пор, пока все читатели не освободят этот мьютекс.

Условные переменные

Условные переменные позволяют программам на C++ реализовать концепцию *монитора*, предложенную известными учеными-кибернетиками Ч.Э.Р. Хоаром (C.A.R. Hoare) и Пером Бринч-Хансеном (Per Brinch-Hansen) и широко применяемую в Java в качестве синхронизированных классов[4].

Монитор разделяет структуру данных между несколькими потоками. Когда поток успешно входит в монитор, он получает во владение мьютекс, который позволяет обновлять общую структуру данных. Поток может покинуть монитор после обновления структуры данных, отказываясь от монопольного доступа. Он также может ожидать условную переменную, временно отказавшись от монопольного доступа до момента, когда произойдет определенное изменение.

Монитор может иметь одну или несколько *условных переменных*. Каждая условная переменная суммирует концептуальные события изменения состояния в структуре данных. Когда поток, работающий внутри монитора, обновляет структуру данных, он должен уведомить все условные переменные, на которые влияет это обновление, что произошло событие, которое они представляют.

C++ предоставляет две реализации условных переменных в заголовочном файле `<condition_variable>`. Они различаются обобщенностью аргумента блокировки, который принимают.

`std::condition_variable`

Наиболее эффективная условная переменная, требующая использования `std::unique_lock` для блокировки мьютекса.

[4] См. C.A.R Hoare, "Monitors: An Operating System Structuring Concept", *ACM Communications 17* (Oct 1974): 549–557.

`std::condition_variable_any`

Условная переменная, которая может использовать любую блокировку `BasicLockable`, т.е. любую блокировку, предоставляющую функции-члены `lock()` и `unlock()`. Эта условная переменная может быть менее эффективной, чем `std::condition_variable`.

Когда поток освобожден условной переменной, он должен убедиться, что структура данных находится в ожидаемом состоянии. Дело в том, что некоторые операционные системы могут выдавать ложные уведомления условных переменных. (Мой опыт говорит, что это может случиться и из-за ошибки программиста.) Пример 12.7 представляет собой расширенный пример использования условных переменных для реализации многопоточной модели производителя/потребителя.

Пример 12.7. Простой производитель и потребитель с использованием условных переменных

```
void cv_example() {
    std::mutex m;
    std::condition_variable cv;
    bool terminate = false;
    int shared_data = 0;
    int counter = 0;

    auto consumer = [&]() {
        std::unique_lock<std::mutex> lk(m);
        do {
            while (!(terminate || shared_data != 0))
                cv.wait(lk);
            if (terminate)
                break;
            std::cout << "потребление " << shared_data << std::endl;
            shared_data = 0;
            cv.notify_one();
        } while (true);
    };

    auto producer = [&]() {
        std::unique_lock<std::mutex> lk(m);
        for (counter = 1; true; ++counter) {
            cv.wait(lk,[&]() {return terminate || shared_data == 0;});
        if (terminate)
            break;
        shared_data = counter;
        std::cout << "производство " << shared_data << std::endl;
        cv.notify_one();
        }
    };

    auto p = std::thread(producer);
    auto c = std::thread(consumer);
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    {
        std::lock_guard<std::mutex> l(m);
        terminate = true;
    }
```

```
    std::cout << "всего произведено " << counter << std::endl;
    cv.notify_all();
    p.join();
    c.join();
    exit(0);
}
```

Поток производителя в примере 12.7 "производит" путем присваивания одной целочисленной переменной под названием `shared_data` ненулевого значения. Потребитель "потребляет" `shared_data`, возвращая ему нулевое значение. Основной поток программы запускает потоки производителя и потребителя, а затем приостанавливается на 1000 мс. Когда основной поток активизируется вновь, он блокирует мьютекс `m` для краткого входа в монитор и устанавливает флаг `terminate`, который заставляет потоки производителя и потребителя завершить работу. Основная программа уведомляет условную переменную о том, что состояние завершения изменилось, подключает два потока и завершает работу.

`consumer` входит в монитор, блокируя мьютекс `m`. Потребитель состоит из одного цикла, ожидающего условную переменную с именем `cv`. Во время ожидания `cv` потребитель не находится в мониторе; мьютекс `m` доступен. Когда появляется что-то, что можно потребить, уведомляется условная переменная `cv`. Потребитель активизируется, захватывает мьютекс `m` и возвращается из вызова `cv.wait()`, концептуально вновь входя в монитор.

В примере 12.7 используется одна условная переменная, которая имеет смысл "структура данных была обновлена". Потребитель в основном ждет выполнения условия `shared_data! = 0`, но он должен "просыпаться" и при выполнении условия `terminate == true`. Это экономичное использование примитивов синхронизации, в отличие от массива сигналов в функции Windows `WaitForMultipleObjects()`. Аналогичный пример можно закодировать с помощью одной условной переменной для потребителя, а второй — чтобы будить производителя.

Потребитель вызывает `cv.wait()` в цикле, проверяя всякий раз при активизации, выполнено ли условие. Дело в том, что некоторые реализации могут давать ложные срабатывания, активизируя ожидающие условную переменную потоки случайно. Если условие выполняется, цикл `while` завершен. Если условие, которое активизировало потребителя, было `terminate == true`, то потребитель завершает внешний цикл и работу своей функции. В противном случае выполнилось условие `shared_data! = 0`. Потребитель выводит сообщение и указывает, что он потребил данные, устанавливая переменную `shared_data` равной нулю и уведомляя `cv`, что общие данные были изменены. В этот момент потребитель все еще находится внутри монитора, удерживая блокировку мьютекса `m`, но при продолжении выполнения цикла он вновь вызывает `cv.wait()`. Мьютекс при этом разблокируется, и концептуально потребитель выходит из монитора.

Производитель похож на потребителя. Он находится в состоянии ожидания до тех пор, пока не сможет заблокировать мьютекс `m`, а затем входит во внешний цикл, который продолжается до выполнения условия `terminate == true`. Производитель ожидает условную переменную `cv`. В данном примере производитель использует

версию wait(), которая принимает аргумент предиката. Пока предикат возвращает значение false, ложные срабатывания будут приводить к продолжению ожидания условной переменной. Таким образом, предикат представляет собой условие, о котором должна сообщить условная переменная. Эта вторая форма является просто "синтаксическим сахаром", скрывающим цикл while. Первоначально shared_data равно нулю, поэтому производитель не ждет уведомления cv. Он обновляет shared_data, уведомляет cv, а затем переходит к следующей итерации цикла и вновь входит в cv.wait(), освобождая мьютекс и концептуально выходя из монитора.

Атомарные операции над общими переменными

Заголовочный файл стандартной библиотеки C++ <atomic> предоставляет низкоуровневый инструментарий для построения многопоточных примитивов синхронизации: барьеров памяти и атомарных загрузок и сохранений.

std::atomic предоставляет стандартный механизм обновления произвольных структур данных, лишь бы они были конструируемыми путем копирования или перемещения. Любая специализация std::atomic должна предоставлять следующие функции для любого типа T.

load()
: std::atomic<T> предоставляет функцию-член T load(memory_order), которая атомарно копирует объект T из std::atomic<T>.

store()
: std::atomic<T> предоставляет функцию-член void store(T,memory_order), которая атомарно копирует объект T в std::atomic<T>.

is_lock_free()
: is_lock_free() возвращает значение типа bool, которое равно true, если все операции, определенные для этого типа, реализованы без использования взаимоисключений, как, например, единой машинной командой чтения-изменения-записи.

Имеются специализации std::atomic для интегральных типов и указателей. Там, где это поддерживает процессор, эти специализации синхронизируют память без вызова примитивов синхронизации операционной системы. Специализации предоставляют целый ряд операций, которые могут быть реализованы на современном оборудовании атомарно.

Производительность std::atomic зависит от процессора, для которого компилируется код.

- Персональные компьютеры на базе архитектуры Intel имеют множество команд чтения-изменения-записи, и стоимость атомарного доступа зависит от барьеров памяти; при этом некоторые барьеры памяти не имеют стоимости вовсе.
- На одноядерных процессорах с командами чтения-изменения-записи std::atomic может не генерировать никакого дополнительного кода.

- На процессорах, которые не имеют команд чтения-изменения-записи, `std::atomic` может быть реализован с использованием дорогостоящих взаимных исключений.

Барьеры памяти

Большинство функций-членов `std::atomic` принимает необязательный аргумент `memory_order`, который выбирает барьер памяти вокруг данной операции. Если аргумент `memory_order` не указан, принимается значение по умолчанию — `memory_order_acq_rel`. Этим обеспечивается полный барьер, который всегда безопасен, но может быть весьма дорогим. Могут быть выбраны более тонкие варианты барьеров, но это могут делать только опытные пользователи, понимающие, что именно они делают.

Барьеры памяти синхронизируют основную память с кешем нескольких аппаратных потоков. В общем случае для синхронизации одного потока с другим барьер памяти выполняется в каждом из двух потоков. C++ допускает следующие барьеры памяти.

`memory_order_acquire`

`memory_order_acquire` может рассматриваться как означающий "получить всю работу, проделанную другими потоками". Он гарантирует, что последующие загрузки не перемещаются перед текущей загрузкой или любыми предыдущими. Это делается, как ни парадоксально, путем ожидания завершения операций сохранения, в настоящее время выполняемых между процессором и основной памятью. Без этого барьера, если поток выполняет загрузку из того же адреса, куда в настоящий момент выполняется сохранение, поток получит старую информацию, как если бы загрузка внутри программы была перемещена.

`memory_order_acquire` может быть дешевле полного барьера, применяемого по умолчанию. Например, вполне можно применять `memory_order_acquire` при атомарном чтении флага в цикле ожидания `while`.

`memory_order_release`

`memory_order_release` может рассматриваться как означающий "освобождение всей работы, выполненной данным потоком к этому моменту". Он гарантирует, что предыдущие загрузки и сохранения, выполненные этим потоком, не перемещаются в точку после текущего сохранения. Это делается путем ожидания завершения текущих операций сохранения в этом потоке.

`memory_order_release` может быть дешевле полного барьера, применяемого по умолчанию. Этот барьер может пригодиться, например, при установке флага в конце самодельного мьютекса.

`memory_order_acq_rel`

Этот барьер объединяет две предыдущие гарантии, создавая полный барьер.

`memory_order_consume`

`memory_order_consume` потенциально более слабая (и быстрая) версия `memory_order_acquire`, которая требует только того, чтобы текущая загрузка

выполнялась до других операций, зависящих от ее данных. Например, когда загрузка указателя помечена как `memory_order_consume`, последующие операции, разыменовывающие этот указатель, не будут перемещены в точку перед ним.

`memory_order_relaxed`

При этом значении разрешены все переупорядочения.

В настоящее время реализация барьеров памяти для большинства процессоров является весьма грубым инструментом. Барьеры памяти блокируют дальнейшие действия, пока не будут завершены *все* текущие операции записи. В действительности достаточно завершить только запись в совместно используемые местоположения в памяти, но ни C++, ни x86-совместимые процессоры не имеют средств для идентификации этого более ограниченного набора местоположений, в особенности когда этот набор варьируется от одного потока к другому.

Остановись и подумай

Многоядерные процессоры x86, выпускаемые на сегодняшний день, имеют очень предупредительную модель памяти. Все загрузки имеют семантику захвата, а все сохранения имеют семантику освобождения. То есть архитектура x86 обеспечивает сильное упорядочение доступа к памяти. Вероятно, необходимо отказаться от более агрессивной модели памяти в архитектуре x86 для поддержания совместимости с множеством старых программ, возможности параллелизма которых были выполнены фактически неправильно. Все процессоры PowerPC, ARM и Itanium имеют более слабое упорядочение (и более высокую производительность).

Это означает, что разработчики, которые пишут параллельные программы на C++ с использованием Visual C++ на x86, в настоящее время могут счастливо избежать неприятностей в многопоточной программе, но, перекомпилировав ту же программу для процессоров ARM или PowerPC, они могут быть удивлены количеством новых ошибок в их якобы отлаженном и рабочем коде.

Атомарный доступ — не панацея. Барьеры памяти *стоят* довольно дорого. Чтобы узнать, насколько дорого, я провел эксперимент, хронометрируя атомарную операцию по сравнению с простым сохранением. Пример 12.8 выполняет в цикле простое атомарное сохранение (с полным барьером по умолчанию).

Пример 12.8. Атомарное сохранение

```
typedef unsigned long long counter_t;
std::atomic<counter_t> x;
for (counter_t i = 0, iterations = 10'000'000 * multiplier;
        i < iterations; ++i)
    x = i;
```

Этот тест на моем персональном компьютере выполняется 15 318 мс. Неатомарная версия этого теста в примере 12.9 выполняется за 992 мс — примерно в 15 раз быстрее. В примере с большим количеством записей "на лету" различие может стать еще более существенным.

Пример 12.9. Неатомарное сохранение

```
typedef unsigned long long counter_t;
counter_t x;
for (counter_t i = 0, iterations = 10'000'000 * multiplier;
        i < iterations; ++i)
    x = i;
```

Если `std::atomic` реализованы с помощью мьютексов операционной системы, как это может быть на некоторых небольших процессорах, разница в производительности может быть в несколько порядков по величине. Таким образом, `std::atomic` должны использоваться с определенными знаниями о целевом оборудовании.

Если вам интересно, что делают одиночные кавычки в числовых константах, то возможность добавлять разделители групп в числовые константы является одним из нескольких небольших, но замечательных дополнений в C++14. Подобные изменения в C++14 могут показаться мелочной суетой, но лично я считаю их очень полезными.

Будущие возможности параллелизма C++

Сообщество разработчиков демонстрирует огромный интерес к параллельным вычислениям, мотивируемое, без сомнения, наличием и необходимостью использования быстро растущих ресурсов компьютеров. Многие разработчики получали опыт работы с параллелизмом на основе потоков с использованием вызовов операционной системы или функций библиотеки POSIX Threads (pthreads). Некоторые из них создавали собственные обертки в стиле классов C++ вокруг вызовов операционной системы. Лучшие из таких разработок становились общедоступными, собирая сообщества пользователей и положительно влияя друг на друга. Многие разработанные таким образом предложения по возможностям параллелизма в C++ теперь проходят процесс стандартизации. Стандарт C++17 может предоставить гораздо более широкую поддержку параллелизма, но о будущем можно сказать с уверенностью только то, что оно наступает.

Перечисленное ниже — просто примеры того, что может принести (а может и не принести) стандарт C++17.

Кооперативная многопоточность

В кооперативной многопоточности два или более программных потоков используют явно указываемые инструкции таким образом, что на самом деле одновременно работает только один поток. Примером кооперативной многопоточности являются сопрограммы.

Кооперативная многопоточность имеет ряд ключевых преимуществ.

- Каждый поток может поддерживать собственный контекст, когда он не выполняется.

- Поскольку одновременно работает только один поток, переменные не являются общими, и взаимные исключения оказываются ненужными.

Сопрограммы (coroutines), похоже, появятся в C++17, и доступны уже сейчас в Boost (`http://www.boost.org/doc/libs/1_59_0/libs/coroutine/doc/html/index.html`). Ряд строительных блоков для различных инновационных параллельных схем перечислены в недавнем предложении (`http://bit.ly/doc-n4399`) рабочей группы C++ Concurrency TR.

Инструкции SIMD

"SIMD" является аббревиатурой от *single instruction multiple data* (одна инструкция, много данных). В процессорах с поддержкой SIMD некоторые инструкции работают с вектором регистров. Процессор выполняет одно и то же действие одновременно со всеми регистрами вектора, сокращая накладные расходы по сравнению со скалярными операциями.

Обычно компилятор C++ не генерирует SIMD-команды, потому что их поведение является сложным и не слишком хорошо соответствует тому, как C++ описывает программы. Зависящие от компилятора директивы `pragma` или встраиваемые ассемблерные команды позволяют вставлять инструкции SIMD в функции, которые затем предоставляются библиотеками для специализированных задач типа цифровой обработки сигналов или компьютерной графики. Таким образом, SIMD-программирование является зависимым одновременно и от процессора, и от компилятора. Среди множества веб-страниц, посвященных SIMD-командам, имеется Stack Exchange Q&A (`http://bit.ly/simd-c-lib`), где, в свою очередь, содержится много ссылок по использованию SIMD в C++.

Оптимизация многопоточных программ C++

Бесплатных завтраков не существует.

— Фраза впервые появилась в статье "*Economics in Eight Words*" в *El Paso Herald-Post* от 27 июня 1938 года. Многие фанаты фантастики узнают цитату из романа Роберта Э. Хайнлайна (Robert A. Heinlein) 1966 года "*Луна — суровая хозяйка*"

По состоянию на начало 2016 года широко используются микропроцессоры класса настольных компьютеров с несколькими ядрами с высокой степенью конвейеризации и несколькими уровнями кеш-памяти. Их архитектуры подходят для высокопроизводительного выполнения нескольких потоков управления. Для выполнения многих потоков часто необходимо дорогостоящее переключение контекста.

Об этой особенности архитектуры не следует забывать при разработке параллельных программ. Попытки обеспечить "мелкозернистую" модель параллельности на имеющихся в настоящее процессорах могут приводить к программам, неэффективно использующим параллелизм.

В светлом будущем личных летающих автомобилей и городов в облаках языки программирования будут автоматически эффективно распараллеливать программы. Но пока мы живем на поверхности Земли, поиск задач, которые могут выполняться одновременно, — дело разработчиков. Хотя возможности параллелизма так же разнообразны, как и программы, есть некоторые "рыбные" места, в которые имеет смысл заглянуть, чтобы поискать код, который можно вынести в отдельный поток. Я перечисляю несколько из них в следующих разделах.

Потоковое поведение программы может быть глубоко погружено в ее структуру. Таким образом, изменить потоковое поведение может оказаться более трудной задачей, чем оптимизация вызовов распределения памяти или функций. Несмотря на это, есть несколько методов проектирования параллельных программ, которые оптимизируют их производительность. Некоторые из этих методов развились только в последние несколько лет, поэтому они могут не быть знакомы даже опытным программистам.

Предпочитайте `std::async`, а не `std::thread`

С точки зрения производительности значительной проблемой `std::thread` является то, что каждый вызов запускает новый программный поток. Запуск потока влечет прямые и косвенные расходы, которые делают эту операцию очень дорогой.

- Прямые расходы включают вызов операционной системы для выделения памяти для потока в таблицах операционной системы, выделение памяти для стека потока, инициализацию набора регистров потока и планирование потока для выполнения. Если поток получает новый квант времени от планировщика, имеется задержка перед началом его работы. Если же он получает остаток кванта времени работавшего потока, имеется задержка, вызванная сохранением регистров этого потока.
- Косвенными затратами на создание потоков является увеличение объема используемой памяти. Каждый поток должен резервировать память для собственного стека вызываемых функций. Если часто запускается и останавливается большое количество потоков, это может привести к пробуксовке кеша при конкуренции выполняющихся потоков за доступ к ограниченному кешу.
- Еще одна косвенная затрата возникает, когда количество программных потоков превышает число аппаратных потоков. Все потоки при этом начинают замедляться, так как они должны быть распланированы операционной системой.

Я измерял стоимость запуска и остановки потока, хронометрируя цикл с фрагментом кода из примера 12.10. Функция внутри потока выполняет немедленный возврат. Это кратчайшая возможная функция. Вызов `join()` приводит к ожиданию завершения потока основной программой, так что потоки вызываются один за другим, не дублируя один другой. Это плохой дизайн параллелизма, если только он не имеет конкретной цели измерения стоимости запуска потоков.

Пример 12.10. Запуск и остановка std::thread

```
std::thread t;
t = std::thread([]() { return; });
t.join();
```

На самом деле этот тест, вероятно, занижает стоимость вызова потока, поскольку такой поток ничего не пишет в кеш. Но результаты все равно оказываются значительными: 10 тысяч вызовов потока занимают около 1 350 мс, или около 135 мкс на один запуск и остановку потока в Windows. Это во много тысяч раз превосходит стоимость выполнения лямбда-выражения. `std::thread` является баснословно дорогим способом выполнения кратких вычислений, даже если они могут выполняться одновременно.

Избежать этой стоимости невозможно, поскольку она представляет собой задержку. Это время, которое потребляется до того, как выполняется возврат из вызова конструктора `std::thread`, а также время до окончания вызова `join()`. Даже если программа отключает потоки вместо подключения, тест по-прежнему выполняется более 700 мс.

Полезной оптимизацией в параллельном программировании является повторное использование потоков вместо создания для каждого использования новых. Поток может остановиться в ожидании некоторой условной переменной, пока он не потребуется вновь, затем быть освобожденным и выполнить вызываемый объект. Хотя одни издержки переключения потоков (сохранение и восстановление регистров и сброс и заполнение кеша) остаются теми же, другие, такие как выделение памяти для потока и планирование операционной системы, устраняются или сокращаются.

Шаблонная функция `std::async()` выполняет вызываемый объект в контексте потока, однако реализации разрешено использовать потоки повторно. Стандарт подсказывает, что `std::async()` может быть реализована с помощью пула потоков. В Windows `std::async()` оказывается значительно быстрее. Я хронометрировал цикл с фрагментом кода из примера 12.11 для того, чтобы измерить улучшение кода.

Пример 12.11. Запуск и остановка с использованием async()

```
std::async(std::launch::async, []() { return; });
```

`std::async()` возвращает `std::future` (в данном случае возвращаемое значение представляет собой неименованную временную переменную). Программа вызывает деструктор этого неименованного фьючерса `std::future`, как только завершается работа `std::async()`. Деструктор ожидает, пока фьючерс не станет *готов*, так что он может генерировать любое исключение, которое может произойти. Явный вызов `join()` или `detach()` не требуется. В примере 12.11, как и в примере 12.10, выполнения потоков следуют один за другим.

Повышение производительности оказалось значительным: 10 тысяч вызовов простого лямбда-выражения выполнились за 86 мс, примерно в 14 раз быстрее, чем если бы новый поток запускался каждый раз заново.

Создавайте потоков столько же, сколько имеется ядер

В старых учебниках по параллелизму советуется создавать столько потоков, сколько вам удобно, т.е. дается тот же совет, что и для создания динамических переменных. Это мышление отражает античный мир, в котором потоки соревновались за внимание одного процессора. В современном мире многоядерных процессоров этот совет оказывается слишком упрощенным.

Разработчик, заботящийся об оптимизации, может различать два вида потоков с различным поведением.

Работающие непрерывно потоки

Такой поток потребляет 100% вычислительных ресурсов ядра, на котором работает. Если имеется *n* ядер, то выполнение потоков на каждом ядре может уменьшить время выполнения программы в пределе до *1/n*. Однако, когда на каждом доступном ядре выполняется свой поток, планирование дополнительных потоков не дает никаких дальнейших улучшений времени выполнения. Вместо этого потоки просто начинают крошить пирог имеющегося оборудования на все меньшие и меньшие куски. Фактически имеется предел этого нарезания пирога, когда за отпущенный квант времени успевают выполниться только запуск и остановка, а на вычисления времени уже не остается. С ростом количества работающих потоков общая производительность падает и в конечном итоге может стремиться к нулю.

Ожидающие потоки с кратковременной работой и долгим ожиданием событий

Такой поток потребляет всего несколько процентов вычислительных ресурсов, имеющихся у ядра. Для планирования работы нескольких потоков ожидания на одном ядре используется большая доля доступных ресурсов, если выполнение этих потоков чередуется таким образом, что вычисления в одном потоке происходят во время ожидания второго потока. Это сокращает время выполнения программы до точки насыщения, в которой полностью используются все вычислительные ресурсы.

В одноядерном прошлом от планирования вычислений как работающих потоков не было никакой пользы. Весь прирост производительности получался только из чередования ожидающих потоков с другими. Но все меняется, когда имеется несколько ядер.

C++ предоставляет функцию под названием `std::thread::hardware_concurrency()`, которая возвращает количество доступных ядер. Эта функция учитывает ядра, выделяемые гипервизором для других виртуальных машин, и ядра, которые ведут себя, как два или более логических ядер из-за одновременной многопоточности. Это позволяет программе подстраиваться под оборудование, содержащее большее (или меньшее) количество ядер.

Чтобы проверить влияние нескольких потоков на производительность, я написал функцию `timewaster()` (пример 12.12), которая многократно выполняет занимающие большое время вычисления. Если все итерации выполняются в одном цикле в основной программе, вычисление занимает 3 087 мс.

Пример 12.12. Итерации потребителя компьютерного времени

```
void timewaster(unsigned iterations) {
    for (counter_t i = 0; i < iterations; ++i)
        fibonacci(n);
}
```

Затем я написал функцию для создания потоков, каждый из которых вычисляет `timewaster()`, разделив общее количество итераций на число потоков (пример 12.13). Я использовал эту функцию для тестирования производительности при разном количестве потоков.

Пример 12.13. Многопоточный потребитель времени

```
void multithreaded_timewaster(unsigned iterations,
                              unsigned threads)
{
    std::vector<std::thread> t;
    t.reserve(threads);
    for (unsigned i = 0; i < threads; ++i)
        t.push_back(std::thread(timewaster, iterations/threads));
    for (unsigned i = 0; i < threads; ++i)
        t[i].join();
}
```

Вот мои наблюдения.

- `std::thread::hardware_concurrency()` на моем компьютере возвращает `4`. Как и ожидалось, максимальная скорость, 1 870 мс, достигается для четырех потоков.
- Немного удивительно, но лучшее время было ближе к половине (а не к четверти) времени работы для одного потока.
- Хотя общая тенденция для экспериментов более чем с четырьмя потоками состояла в увеличении времени работы, наблюдалось значительное отклонение от запуска к запуску; таким образом, результат не был надежным.

Разумеется, существует практическое препятствие, ограничивающее число потоков, запущенных программой. В большой программе, над которой работают несколько разработчиков, или при применении сторонних библиотек может быть трудно узнать, сколько потоков начато в другом месте программы. Потоки создаются при необходимости и в любом месте в программе. Хотя операционная система служит контейнером, который знает все работающие потоки, это знание недоступно для C++ с помощью внутренних средств.

Реализуйте очередь заданий и пул потоков

Решением проблемы отсутствия знания о количестве работающих потоков является явное создание потоков: предоставить *пул потоков*, структуру данных, содержащую фиксированное количество долгоживущих потоков, и *очереди заданий*, структуру данных, содержащую список вычислений, которые следует выполнить и которые обслуживаются потоками из пула потоков.

В *программировании, ориентированном на задания*, программа записывается в виде коллекции объектов выполнимых заданий, которые выполняются потоками из пула потоков. Когда поток становится доступным, он выбирает задачу из очереди задач. Когда поток завершает задачу, сам он не завершается; он либо выполняет следующую задачу, либо блокируется до получения новой задачи для выполнения.

Программирование, ориентированное на задания, имеет ряд преимуществ.

- Программы, ориентированные на задания, могут эффективно обрабатывать события завершения ввода-вывода от неблокирующих вызовов ввода-вывода, добиваясь эффективного использования процессора.
- Наличие пула потоков и очереди заданий устраняет накладные расходы запуска потоков для краткосрочных задач.
- Программирование, ориентированное на задания, централизует асинхронную обработку в одном наборе структур данных, что позволяет легко ограничить количество используемых потоков.

Недостатком программ, ориентированных на задания, является проблема под названием *инверсия управления*. Вместо явного управления потоком выполнения программы он становится неявным и управляется порядком полученных сообщений о событиях. Это может сделать программы, ориентированные на задания, трудными для понимания или отладки, хотя мой личный опыт говорит, что такая путаница на практике встречается редко.

Стандарты пулов потоков и очередей заданий пока что отсутствуют; возможно их появление в C++17. В настоящее время они доступны в библиотеках Boost и Threading Building Blocks от Intel (`http://www.threadingbuildingblocks.org/`).

Выполняйте ввод-вывод в отдельном потоке

Физическая реальность в виде вращающихся дисков и удаленных сетевых подключений создает задержку между временем, когда программа запрашивает данные, и временем, когда эти данные становятся доступными. Таким образом, операции ввода-вывода являются идеальным местом для поиска возможностей параллелизма. Для ввода-вывода типично также, что программа должна выполнять преобразование данных, прежде чем они смогут быть записаны или после их прочтения. Например, из Интернета может быть считан файл с XML-данными. Затем эти данные анализируются для извлечения информации, интересующей программу. Поскольку данные не могут использоваться, пока не будут преобразованы, весь процесс, включая чтение и анализ, становится очевидным кандидатом для перемещения в отдельный поток.

Программа без синхронизации

Синхронизация и взаимные исключения замедляют многопоточные программы. Избавление от этой синхронизации может повысить производительность. Существуют три простых и один трудный способ программировать без явной синхронизации.

Программирование, ориентированное на события

В программировании, ориентированном на события, программа записывается в виде коллекции функций обработки событий, которые вызываются из программного каркаса. Каркас в основе программы диспетчеризует каждое событие из очереди событий, передавая его функции-обработчику этого события. Программирование, ориентированное на события, во многом похоже на программирование, ориентированное на задания. Каркас в программе, ориентированной на события, действует в качестве планировщика, а обработчики событий аналогичны заданиям. Важное различие заключается в том, что в программе, ориентированной на события, каркас однопоточный, и функции обработки событий выполняются не одновременно.

Программирование, ориентированного на события, имеет ряд преимуществ.

- Поскольку лежащий в основе каркас однопоточный, синхронизация не требуется.
- Программы, ориентированные на события, могут эффективно обрабатывать события завершения ввода-вывода от неблокирующих вызовов ввода-вывода. Такая программа достигает такой же высокой эффективности использования процессора, как и многопоточная.

Как и для программ, ориентированных на задания, главным недостатком является проблема под названием *инверсия управления*, т.е. поток управления становится неявным и определяется порядком полученных сообщений о событиях. Это может сделать программы, ориентированные на события, трудными для понимания или отладки.

Сопрограммы

Сопрограммы являются исполняемыми объектами, которые явно передают выполнение от одного объекта другому, но запоминают свои указатели выполнения, так что могут возобновить выполнение при повторном вызове. Как и программы, ориентированные на события, сопрограммы не являются истинно многопоточными, поэтому синхронизация для них не нужна, если только они не управляются несколькими потоками.

Можно выделить два вида сопрограмм. Первый вид имеет собственный стек и может передать контроль другой сопрограмме в любой момент выполнения. Второй вид заимствует стек у другого потока и может передать управление только на своем верхнем уровне.

Сопрограммы могут появиться в стандарте C++17.

Передача сообщений

В программах с передачей сообщений потоки управления принимают входные данные из одного или нескольких источников, преобразуют их и передают одному или нескольким выходным приемникам. Соединения выходов и входов образуют граф с точно определенными узлами входа и выхода. Элементы, считываемые и записываемые потоком на каждом этапе, могут быть реализованы

как сетевые датаграммы, символьные потоки ввода-вывода или структуры данных в явных очередях.

Конвейеры командной строки UNIX и веб-службы являются примерами программирования с передачей сообщений. Компонентами распределенной системы обработки являются программы, передающие сообщения.

Среди преимуществ программ с передачей сообщений можно выделить следующие.

- Синхронизация выходных данных каждого этапа со входом следующего этапа является неявной. Она либо обрабатывается операционной системой, когда этапы взаимодействуют с помощью датаграмм, либо обеспечивается очередью, подключенной к этапам. Параллелизм системы происходит за пределами этих этапов, поэтому обычно этапы могут рассматриваться как однопоточный код, который может блокировать действия ввода и вывода.
- Поскольку выводы одного этапа подключены к одному входу следующего этапа, вопросы голодания и равнодоступности встают гораздо реже.
- Синхронизация выполняется реже, над более крупными блоками данных. Это увеличивает долю времени, когда несколько потоков могут выполняться параллельно.
- Поскольку этапы конвейера не используют общие переменные, они не замедляются мьютексами и барьерами памяти.
- Между этапами конвейера могут передаваться более крупные блоки работы, так что каждый этап использует все кванты времени вместо остановки и запуска взаимного исключения. Это повышает эффективность использования процессора.

Недостатки программ с передачей сообщений таковы.

- Сообщения не являются объектно-ориентированными по своей природе. Разработчик на C++ должен писать код для маршалинга входящих сообщений в вызовы функций-членов.
- Восстановление после ошибок может оказаться большой проблемой при аварийном завершении этапа конвейера.
- Не каждая задача имеет очевидное решение в виде конвейера независимых программ с передачей сообщений.

Остановись и подумай

В этой книге описаны возможности C++ по совместному использованию переменных и синхронизации потоков, так как эти возможности должны быть встроены в язык C++ для эффективного функционирования. Программы с передачей сообщений используют библиотеки, не входящие в C++, поэтому здесь об этих программах говорится очень кратко. Но это не означает, что передача сообщений не имеет никакого значения.

Большое и шумное сообщество проектировщиков считает, что явная синхронизация и общие переменные — плохая идея, ведущая к сложным, склонным к гонкам и немасштабируемым решениям. Эти разработчики полагают, что единственный масштабируемый способ параллельного программирования — это применение конвейеров.

Высокопараллельные архитектуры GPU даже не предоставляют общую память, поэтому для этих процессоров необходимо программирование с использованием передачи сообщений.

Программы, свободные от блокировок

Hic Sunt Dracones (Здесь живут драконы)

— Надпись на глобусе Ханта-Ленокса (ок. 1503–1507), указывающая на опасности неизвестных берегов

Программирование без блокировок означает практики программирования, которые допускают многопоточные изменения структур данных без необходимости прибегать ко взаимным исключениям. В такой программе дорогостоящие мьютексы заменяются атомарными операциями, синхронизируемыми с помощью оборудования. Структуры данных без блокировок могут работать значительно лучше, чем обычные контейнеры под охраной мьютексов, в особенности когда к одному и тому же контейнеру обращается множество потоков.

В свет были выпущены классы контейнеров C++ без блокировок, представляющие массивы, очереди и хеш-таблицы. В библиотеке Boost имеются стеки и очереди без блокировок (`http://bit.ly/b-lock-free`), но они были протестированы только для компиляторов GCC и Clang. В Intel Threading Building Blocks (`http://www.threadingbuildingblocks.org/`) имеются классы контейнеров без блокировок, представляющие массив, очередь и хеш-отображение. Эти контейнеры не являются идентичными эквивалентным контейнерам стандартной библиотеки C++ из-за требования программирования без блокировки.

Безумно трудно аргументированно рассуждать о структурах данных без блокировки. Даже известные эксперты спорят о правильности опубликованных алгоритмов. По этой причине я рекомендую придерживаться широко опубликованного и хорошо поддерживаемого кода, а не пытаться использовать собственные структуры данных без блокировки.

Удаление кода запуска и завершения

Программа может запустить столько потоков, сколько необходимо для параллельного выполнения задач или для использования нескольких ядер процессора. Однако есть часть программы, которую довольно трудно выполнять параллельно: это код, выполняемый до того, как управление будет передано функции `main()`, и после выхода из нее.

Перед запуском `main()` инициализируются все переменные со статической длительностью хранения (см. раздел "Длительность хранения переменной" главы 6, "Оптимизация переменных в динамической памяти"). Для простых типов данных инициализация имеет нулевую стоимость. Компоновщик указывает на переменные с данными инициализации. Но для переменных типа класса со статической длительностью хранения процесс инициализации последовательно вызывает конструкторы каждой переменной в одном потоке, в порядке, определяемом стандартом.

Легко забыть, что переменные, такие как строки, которые инициализируются статическими строковыми константами, в действительности выполняют код во время инициализации. Аналогично конструкторы, которые содержат вызовы функций или неконстантные выражения в их списках инициализаторов, также выполняются во время выполнения, даже если тело конструктора является пустым.

Эти стоимости, каждая из которых по отдельности может быть небольшой, в сумме могут составить большую величину, приводя к тому, что большие программы могут не отвечать на запросы в течение нескольких секунд после запуска.

Из истории оптимизационных войн

Браузер Google Chrome представляет собой массивную программу, годами разрабатываемую сотнями людей. Число таблиц, которые должны быть инициализированы в Chrome, ошеломляет. Чтобы производительность при запуске не продолжала деградировать, администраторы проекта Chromium добавили к процедуре проектирования правило, требующее утверждения каждой статической переменной, инициализируемой с выполнением кода.

Более эффективная синхронизация

Синхронизация является накладными расходами параллелизма с общей памятью. Уменьшение этих накладных расходов представляет собой важный путь к оптимальной производительности.

Общеизвестная мудрость гласит, что примитивы синхронизации, такие как мьютексы, дороги. Истина, конечно, несколько сложнее. Например, в Windows и Linux мьютексы, на которых основан `std::mutex`, представляют собой гибридный дизайн, который короткое время ожидает атомарную переменную, а затем, если не может быстро получить мьютекс, переходит в состояние ожидания сигнала операционной системы. Стоимость "везения", если никакой другой поток не удерживает мьютекс, низка. Стоимость перехода потока в режим ожидания сигнала измеряется в миллисекундах. Важной стоимостью мьютекса является стоимость ожидания, когда другой поток освободит мьютекс, а не стоимость самого вызова `lock()`.

Многопоточные программы наиболее часто сталкиваются с проблемами, когда имеется много потоков, желающих захватить один и тот же мьютекс. При малой конкуренции удержание мьютекса обычно не замедляет другие потоки. При высокой конкуренции такой удерживаемый мьютекс приводит к тому, что потоки работают по

очереди, в каждый момент времени только один поток, перечеркивая все попытки разработчика выполнить некоторые части программы одновременно.

Параллельные программы на C++ значительно сложнее, чем однопоточные. Куда труднее создавать и примеры или тестовые случаи, которые давали бы значимые результаты, поэтому в этом разделе я вынужден опуститься до эвристики.

Уменьшайте критические разделы

Критический раздел представляет собой область кода, ограниченного захватом мьютекса и его освобождением. Во время выполнения критического раздела никакой другой поток не сможет получить доступ к общим переменным, контролируемым этим мьютексом. И вот проблема в двух словах. Если критический раздел делает что-то, *помимо* доступа к общим переменным, остальные потоки должны находиться в состоянии ожидания без уважительной причины.

Чтобы проиллюстрировать эту проблему, взгляните еще раз на пример 12.7. Два лямбда-выражения, `producer` и `consumer`, выполняются одновременно в двух потоках. Оба потока немедленно пытаются войти в монитор, контролируемый мьютексом `m`. Оба потока остаются в мониторе, за исключением случаев, когда они находятся в состоянии вызова `cv.wait()`. `producer` устанавливает значение общей переменной `shared_data` равным следующему значению последовательности. `consumer` сбрасывает значение `shared_data`. Но и производитель, и потребитель делают еще одну вещь: выводят строку текста в `cout`.

Написанный код `cv_example` может производить приблизительно 35–45 обновлений `shared_data` за 1000 мс на моей системе i7 при использовании компилятора Visual Studio 2015 в производственном режиме. Не так уж очевидно, много это или мало. Но если я закомментирую две инструкции вывода, `cv_example` сможет выполнить 1,25 млн обновлений. Вывод символов на консоль — очень медленная операция, выполняющая массу действий, так что в ретроспективе полученный результат не так уж удивителен.

Мы получили два урока. Урок первый в том, что концепция монитора, в котором код всегда находится в мониторе, за исключением времени ожидания условной переменной, может оказаться трудно используемой эффективно. Урок второй в том, что *выполнение операций ввода-вывода в критическом разделе не приводит к оптимальной производительности.*

Ограничивайте количество параллельных потоков

Как уже упоминалось в разделе "Создавайте потоков столько же, сколько имеется ядер", *количество работающих потоков не должно превышать количество ядер процессора* для устранения издержек переключения контекста. Чтобы указать вторую причину для этой рекомендации, сначала нужно объяснить, как реализуются мьютексы.

Класс мьютекса, предоставляемый Windows, Linux и большинством других современных операционных систем, представляет собой гибридный дизайн, оптимизированный для многоядерных процессоров. Поток t_1 пытается захватить мьютекс,

и если он не заблокирован, то захватывается немедленно. Если же t_1 пытается захватить мьютекс, удерживаемый другим потоком, t_2, то поток t_1 сначала в режиме активного ожидания (постоянного опроса) ожидает захвата в течение небольшого интервала времени. Если t_2 освобождает мьютекс до завершения активного ожидания, t_1 захватывает мьютекс и продолжает работать, эффективно используя весь свой квант времени. Если же t_2 не освобождает мьютекс за время активного ожидания, t_1 переходит в режим ожидания события операционной системы, отдавая свой квант времени. Поток t_1 отправляется в "список ожидания" операционной системы.

Если потоков больше, чем ядер, то лишь некоторые потоки назначаются ядрам и фактически работают в конкретный момент времени. Остальные потоки ожидают в соответствующем списке операционной системы и в конечном итоге получают свой временной квант. Операционная система периодически активизируется для принятия решения о том, какой поток следует запустить на выполнение. Эта активизация достаточно медленная по сравнению со скоростью выполнения отдельных инструкций. Таким образом, поток в очереди работающих потоков может ожидать много миллисекунд, прежде чем операционная система выделит для него ядро, на котором он будет выполняться.

Если поток t_2 удерживает мьютекс, но ожидает в очереди работающих потоков операционной системы своего кванта времени вместо фактического выполнения, он не может освободить мьютекс. Когда t_1 пытается получить мьютекс, его время активного ожидания истекает, и t_1 переходит в режим ожидания события операционной системы; это означает, что t_1 отдает свой квант времени и ядро и переходит в список ожидания операционной системы. В конце концов t_2 получает ядро для работы и освобождает мьютекс, т.е. происходит событие, в ожидании которого находится поток t_1. Операционная система отмечает, что произошло ожидаемое событие, и перемещает t_1 в очередь работающих потоков. Но операционная система не обязательно немедленно выделяет этому потоку ядро для работы[5]. Потоки, которые работают в настоящее время, до конца используют свои кванты времени, если только не будут раньше заблокированы и не перейдут в режим ожидания. Операционная система не выделяет ядро для вновь активизированного потока t_1, пока какой-нибудь другой поток не исчерпает свой квант времени, что может занять много миллисекунд.

Именно этот медленный процесс разработчик пытается устранить путем ограничения количества активных потоков. Избежать этого медленного пути означает, что вместо сотен взаимноблокируемых операций в секунду потоки t_1 и t_2 могут за секунду выполнять миллионы таких операций.

Идеальное число потоков, конкурирующих за краткий критический раздел, равно двум. Когда есть только два потока, нет вопросов равнодоступности или голодания и никаких шансов для проблемы "громового стада", описанной в следующем разделе.

[5] Я опускаю массу подробностей. Windows повышает приоритет потока, чтобы поскорее дать ему возможность выполнения. Другие операционные системы выполняют иные действия в надежде уменьшить продолжительности критических разделов. Но в общем случае наличие критического раздела, как правило, означает длительное ожидание.

Избегайте громового стада

Явление, получившее название *громовое стадо*, происходит, когда много потоков находятся в ожидании события, например доступности работы, которую может выполнять одновременно только один поток. При наступлении события все потоки переводятся в режим работающих, но запустить немедленно из-за ограниченного количества ядер можно только некоторые из них. Один из этих потоков и берется за работу. Все остальные потоки в конечном итоге будут рано или поздно запущены. Каждый поток обнаруживает, что событие уже обработано, так что пора снова погружаться в спячку, т.е. в результате целое "стадо" потоков проснулось только для того, чтобы без толку занять процессорное время и опять уйти в спячку в ожидании следующего события.

Избежать громового стада очень просто — надо ограничить число потоков, созданных для обслуживания этого события. Два потока может быть лучше, чем один, но 100 потоков, вероятно, *не* лучше (см. выше раздел "Ограничивайте количество параллельных потоков"). Ограничение числа потоков может быть проще для тех проектов программного обеспечения, которые реализуют очередь заданий, чем для конструкций, которые связывают поток с каждым рабочим элементом.

Избегайте очереди на блокировку

Блокировка колонны (lock convoy), или очередь на блокировку, происходит при синхронизации большого количества потоков, ожидающих ресурс или вход в критический раздел. Это вызывает дополнительные заторы, поскольку все потоки пытаются действовать одновременно, но в каждый момент времени продвигаться вперед может только один из них, как будто все они находятся в колонне.

В простом случае мы снова и снова сталкиваемся с громовым стадом. Имеется количество конкурирующих за мьютекс потоков, достаточное для того, чтобы множество потоков находилось в состоянии ожидания сигнала операционной системы. Когда поток, удерживавший мьютекс, освобождает его, выдается сигнал о наступлении события, и все ожидающие потоки становятся активными. Поток, первым получивший доступ к процессору, снова блокирует мьютекс. Все остальные потоки, рано или поздно получая доступ к процессору, обнаруживают заблокированный мьютекс и возвращаются в состояние ожидания. В результате операционная система тратит много времени на перезапуск потоков, но большинство потоков так и не выполняются. Кроме того, все эти потоки по-прежнему остаются синхронизированными, так что, когда следующий поток освободит мьютекс, все опять повторится сначала.

В более сложном случае стадо потоков, которые одновременно пытаются захватить второй мьютекс или выполнить некоторые действия, такие как чтение файла, окажется узким местом для физических свойств устройства. Поскольку потоки синхронизированы, все они попытаются получить доступ ко второму ресурсу в одно и то же время. Производительность резко снижается, поскольку потоки запрашивают ресурс одновременно, вызывая сериализацию. Будучи несинхронизированными, они могли бы добиться куда большего прогресса.

Блокировка колонны проявляется в виде системы, которая обычно хорошо работает, но иногда перестает отвечать на запросы в течение нескольких секунд.

Сокращение числа потоков или планирование запуска потоков в разное время может облегчить блокировку колонны, хотя всегда есть риск, что она просто проявится в другом месте. Иногда лучше просто признать, что некоторые группы задач не могут выполняться одновременно, потому что они разделяют некоторые аппаратные устройства или другие узкие места.

Уменьшайте конкуренцию

Потоки в многопоточной программе могут конкурировать за ресурсы. В любой момент, когда двум или более потокам требуется один и тот же ресурс, взаимное исключение приводит к режиму ожидания и потере параллелизма. Имеется несколько методов решения проблемы конкуренции.

Помните, что память и ввод-вывод являются ресурсами

Не каждый разработчик понимает, что диспетчер памяти является ресурсом. Диспетчер памяти должен сериализовать доступ в многопоточных системах, иначе будут испорчены его структуры данных. Большое число потоков, пытающихся одновременно выделить память для динамических переменных (в особенности этим отличаются `std::string`), могут приводить к резкому падению производительности с ростом количества потоков.

Ввод-вывод в файлы также является ресурсом. Дисковые головки могут быть одновременно только в одном месте. Попытка выполнить операции ввода-вывода с несколькими файлами одновременно может привести к внезапному падению производительности.

Сетевой ввод-вывод также является ресурсом. Разъем Ethernet представляет собой узкий канал, через который тонкой струйкой текут биты. Современные процессоры могут ухитриться переполнить даже гигабитные Ethernet-кабели. Еще проще сделать то же самое с WiFi-соединениями.

Когда встает проблема производительности, приходит время сделать шаг назад и спросить: "Что вся программа делает прямо сейчас?" Ведение журнала большую часть времени не является проблемой, но может замедлять работу всей системы, если в этот момент кто-то еще работает с диском или сетевым интерфейсом. Имеющаяся динамическая структура данных может не масштабироваться для большого количества потоков.

Дублируйте ресурсы

Иногда вместо того, чтобы несколько потоков боролись за ресурсы, например за общее отображение или хеш-таблицу, можно просто дублировать таблицы таким образом, чтобы каждый поток имел собственную копию, что устранит конфликты. Хотя хранение двух экземпляров структуры данных означает большую работу, оно может привести к снижению реального времени работы по сравнению с общей структурой данных.

Для повышения пропускной способности могут дублироваться даже аппаратные ресурсы, такие как диски и сетевые карты.

Разделяйте ресурсы

Иногда вместо того, чтобы несколько потоков конкурировали за одну структуру данных, можно разбить ее так, чтобы каждый поток обращался только к той части данных, с которой он работает.

Используйте более мелкозернистую блокировку

Вместо блокировки всей структуры данных с помощью одного мьютекса можно использовать несколько мьютексов. Например, в хеш-таблице один мьютекс может блокировать хеш-таблицу от модификации (например, вставки и удаления записей), а другой мьютекс может блокировать записи от изменения. В этом случае хорошим выбором могут оказаться блокировки читателя/писателя. Для доступа к элементу хеш-таблицы поток захватывает блокировку чтения хеш-таблицы и блокировку чтения или записи для элемента таблицы. Чтобы вставить или удалить запись, поток должен захватить блокировку записи самой хеш-таблицы.

Используйте структуры данных без блокировок

Использование структур данных без блокировки, таких как хеш-таблица, снимает необходимость во взаимных исключениях. Это последний экстент, к которому может быть применена мелкозернистая блокировка.

Используйте планировку ресурсов

Некоторые ресурсы, такие как жесткий диск, невозможно дублировать или разбить. Но все еще можно запланировать работу с диском таким образом, чтобы не вся она выполнялась одновременно или чтобы одновременно выполнялся доступ к близлежащим частям диска. Операционная система может планировать чтения и записи на мелкозернистом уровне, но программа может так установить последовательность операций, таких как чтение файлов конфигурации, чтобы они не выполнялись одновременно.

Не пользуйтесь активным ожиданием в одноядерных системах

Возможности параллелизма C++ позволяют разработчикам реализовывать высокопроизводительные примитивы синхронизации, которые используют активное ожидание. Но это не всегда является хорошей идеей.

В одноядерном процессоре единственным способом синхронизации потоков является вызов примитивов синхронизации операционной системы. Активное ожидание в этом случае до смешного неэффективное. Такое ожидание заставляет поток использовать его квант времени впустую, потому что поток, удерживающий мьютекс, не в состоянии выполнять действия до завершения критического раздела до тех пор, пока ожидающий поток не отдаст ему процессор.

Не ждите вечно

Что происходит с потоком, который безоговорочно ожидает события? Если программа работает правильно, возможно, ничего страшного. Но что делать, если пользователь пытается остановить программу? Пользовательский интерфейс выключается, но программа не останавливается, потому что есть по-прежнему работающий поток. Если функция `main()` попытается подключиться к ожидающему потоку, она "зависнет". Если ожидающий поток отключить, функция `main()` может завершиться. Что именно при этом происходит, зависит от того, чего и как именно ожидает поток. Если он ожидает установки флага, он ожидает вечно. Если он ожидает события операционной системы, он ожидает вечно. Если же он ожидает объект C++, то очень многое зависит от того, не удаляет ли некоторый незаблокированный поток этот объект. Это может привести к завершению ожидающего потока (или не привести).

Вечное ожидание является врагом восстановления после ошибок. Есть существенная разница между программой, которая в основном работает, но изредка сбоит, и надежной программой, на поведение которой могут положиться пользователи.

Собственные мьютексы могут быть неэффективными

Не так трудно разработать простой класс, который действует как мьютекс, который находится в активном ожидании до тех пор, пока другой поток обновит атомарную переменную. Такой класс может даже быть быстрее, чем предоставляемые системой мьютексы, если конкуренция низкая, а критические разделы краткие. Однако мьютексы, которые поставляются операционной системой, часто знают секреты операционной системы и то, как она планирует задачи, что повышает их производительность или позволяет избежать проблемы инверсии приоритетов в данной конкретной операционной системе.

Проектирование надежных мьютексов должно учитывать дизайн операционной системы, в которой они должны работать. Применение собственных мьютексов не ведет к оптимизации.

Ограничивайте длину очереди вывода производителя

В программе производителя/потребителя всякий раз, когда производитель быстрее потребителя, данные создают очередь между производителем и потребителем. Среди множества проблем в этой ситуации можно указать следующие.

- Производитель конкурирует за процессор, выделение памяти и другие ресурсы, что приводит к дальнейшему замедлению потребителя и усугубляет проблему.
- Производитель в конечном итоге потребит все системные ресурсы памяти, вызвав аварийное завершение программы.
- Если программа разработана с восстановлением после исключений, то может потребоваться обработать все данные в очереди перед перезагрузкой, что увеличит время восстановления.

Эта ситуация особенно вероятна в случаях, когда программа частично принимает входные данные из потокового источника с максимальной скоростью, ограничивающей скорость работы производителя, а частично из фиксированного источника, такого как файл, в котором производитель может работать непрерывно.

Решение заключается в том, чтобы ограничить размер очереди и блокировать производителя, когда очередь заполнена. Но очередь должна быть достаточно большой, чтобы сгладить любые изменения в производительности потребителя. Зачастую достаточно лишь нескольких записей. Любые дополнительные записи очереди приводят только к тому, что производитель продолжает дольше работать, увеличивая потребление ресурсов без вклада в параллелизм.

Библиотеки для параллельных вычислений

Есть целый ряд библиотек для эффективного использования возможностей параллелизма. Разработчику, который ищет реализацию параллелизма с передачей сообщений, можно посоветовать рассмотреть и принять один из предлагаемых инструментов. В частности, Threading Building Blocks предлагает некоторые возможности для параллелизма, которые еще не вошли в стандарт C++. Вот возможные варианты инструментов.

Boost.Thread (`http://bit.ly/b-thread`)
и Boost.Coroutine (`http://bit.ly/b-coroutine`)

Потоковая библиотека от Boost представляет собой ожидания того, как должна будет выглядеть стандартная потоковая библиотека в стандарте C++17. Некоторые ее части все еще находятся в стадии активного эксперимента. Boost.Coroutine также является экспериментальной библиотекой.

POSIX Threads

POSIX Threads (pthreads) является кроссплатформенной библиотекой потоков и примитивов синхронизации — возможно, старейшей и наиболее широко используемой библиотекой для параллельной обработки. Это библиотека функций в стиле C, которая предоставляет традиционные возможности параллельности. Она обильно документирована, широко доступна в дистрибутивах Linux, а также для Windows (`http://sourceware.org/pthreads-win32/`).

Threading Building Blocks (TBB) (`http://www.threadingbuildingblocks.org/`)

TBB — амбициозный, хорошо документированный C++ API для работы с потоками, с уклоном в сторону шаблонов. Он предоставляет параллельные циклы `for`, задания и пулы потоков, параллельные контейнеры, классы передачи сообщений и примитивы синхронизации. TBB разработан компанией Intel для содействия эффективному использованию многоядерных микропроцессоров. Он имеет открытый исходный код, обширную документацию, в том числе как минимум одну хорошую книгу (Джеймс Рейндерс (James Reinders) *Intel Threading Building Blocks* издательства O'Reilly), и работает в Windows и Linux.

0mq (`http://zeromq.org/`) *(другое название — ZeroMQ)*

0mq представляет собой коммуникационную библиотеку для соединения программ на основе передачи сообщений. Она поддерживает различные парадигмы связи и стремится к эффективности и экономности. Мой личный опыт работы с этой библиотекой был весьма положительным. 0mq — библиотека с открытым исходным кодом, хорошо документированная и активно поддерживаемая. Существует также пересмотренная версия 0mq под названием "nanomsg" (`http://www.nanomsg.org`), которая решает некоторые проблемы, имеющиеся в 0mq.

Message Passing Interface (MPI) (`http://computing.llnl.gov/tutorials/mpi/`)

MPI представляет собой API для передачи сообщений в распределенной сети компьютеров. Реализации являются библиотеками функций в стиле C. MPI ведет свое происхождение из Ливерморской национальной лаборатории им. Э. Лоуренса в Калифорнии, давно ассоциирующейся с кластерами суперкомпьютеров и физикой высоких энергий. MPI хорошо документирована в старинном стиле DoD 1980-х годов. Имеются реализации для Linux и Windows, включая одну в Boost (`http://bit.ly/b-mpi`), но они не всегда охватывают полную спецификацию.

OpenMP (`http://openmp.org`)

OpenMP представляет собой API для "многоплатформенного параллельного программирования с общей памятью на C/C++ и Fortran". При использовании разработчик добавляет в программы на языке C++ прагмы, определяющие его параллельное поведение. OpenMP предоставляет мелкозернистую модель параллелизма с акцентом на численные задачи и эволюционирует в сторону программирования GPU. OpenMP доступен на Linux для GCC и Clang и на Windows для Visual C++.

C++ AMP (`http://bit.ly/cpp-accel`)

C++ AMP — открытая спецификация библиотеки C++, предназначенная для выполнения параллельных вычислений на устройствах GPU. Версия от Microsoft разрешается в вызовы DirectX 11.

Резюме

- *Многопоточная программа на языке C++ является последовательно согласованной, если в ней нет гонок.*
- *Большое и шумное сообщество проектировщиков считает, что явная синхронизация и общие переменные — плохая идея.*
- *Выполнение ввода-вывода в критическом разделе не ведет к оптимальной производительности.*
- *Количество работающих потоков не должно превышать количество ядер процессора.*
- *Идеальное число потоков, конкурирующих за краткий критический раздел, равно двум.*

ГЛАВА 13

Оптимизация управления памятью

Эффективность делает лучше то, что уже сделано.

— Питер Ф. Друкер (Peter F. Drucker) (1909–2005), американский консультант в области управления

Диспетчер памяти — это набор функций и структур данных из системы времени выполнения C++, которые контролируют выделение памяти для динамических переменных. Диспетчер памяти должен отвечать множеству требований. Эффективное удовлетворение этим требованиям и сегодня является открытой исследовательской задачей. Во многих программах C++ функции диспетчера памяти весьма горячие, очень часто вызываемые в процессе работы программы. Если удастся увеличить его производительность, это будет иметь глобальное влияние на производительность программы. По этим причинам диспетчер памяти является естественной целью для усилий оптимизации.

Но, на мой взгляд, есть множество других мест, более достойных просмотра, в первую очередь, в поисках улучшения производительности, которые, скорее всего, будут более плодотворными, чем возня с диспетчером памяти. Диспетчеры памяти, будучи очень горячим кодом, как правило, очень хорошо отработаны. Управление памятью в лучшем случае является частью от части общего времени выполнения программы. Закон Амдала ограничивает общее улучшение, которое можно получить, даже сведя стоимость управления памятью до нуля. В больших программах в одном исследовании улучшение производительности от оптимизации управления памятью варьировались от незначительной величины до значения около 30%.

Диспетчер памяти C++ — высоконастраиваемый, с существенным API. Хотя многим программистам никогда не приходится использовать этот API, он предоставляет множество способов оптимизации производительности. Имеется ряд высокопроизводительных диспетчеров памяти, которые могут быть подключены к C++ путем замены функций C `malloc()` и `free()`. Кроме того, для горячих классов и контейнеров стандартной библиотеки разработчик может заменить диспетчер памяти специализированным.

API управления памятью C++

Инструментарий C++ для управления динамическими переменными был описан в разделе "API динамических переменных C++" главы 6, "Оптимизация переменных в динамической памяти". Этот инструментарий содержит интерфейс диспетчера памяти, состоящий из выражений `new` и `delete`, функций управления памятью и аллокаторов шаблонов классов стандартной библиотеки.

Жизненный цикл динамических переменных

Динамическая переменная проходит через пять этапов. Наиболее распространенные вариации выражения `new` выполняют этапы *выделения* памяти и *размещения* в ней. После этапа *использования* выражение `delete` выполняет этапы *уничтожения* и *освобождения*. C++ предоставляет средства для управления каждой из этих стадий отдельно.

Выделение

Программа запрашивает у диспетчера памяти указатель на непрерывную область памяти, содержащую как минимум указанное в запросе количество байтов нетипизированной памяти. Выделение может завершиться ошибкой, если память недоступна. Этот этап управляется функцией `malloc()` библиотеки C и различными перегрузками `operator new()` в C++.

Размещение

Программа устанавливает начальное значение динамической переменной, помещая значение в выделенную память. Если переменная является экземпляром класса, вызывается один из конструкторов. Если переменная имеет простой тип, ее инициализация не является обязательной. Размещение может быть неудачным, если конструктор генерирует исключение, требующее возврата выделенной памяти диспетчеру памяти. Выражения `new` участвуют в этом этапе.

Использование

Программа считывает значения из динамической переменной, вызывает функции-члены динамической переменной и записывает в нее значения.

Уничтожение

Если переменная является экземпляром класса, программа вызывает ее деструктор для выполнения последней операции над динамической переменной. Уничтожение является возможностью для динамической переменной вернуть все системные ресурсы, которые она удерживает, выполнить очистку, сказать свое последнее слово и приготовиться к вечному сну. Уничтожение может дать сбой, если деструктор генерирует исключение, которое не обрабатывается в теле деструктора. Если это происходит, программа безоговорочно завершает работу. Этот этап управляется выражением `delete`. Переменную можно уничтожить без освобождения ее памяти путем явного вызова ее деструктора.

Освобождение

Программа возвращает память, ранее принадлежавшую уничтоженной динамической переменной, диспетчеру памяти. Этап освобождения выполняют функция библиотеки C `free()` и различные перегрузки `operator delete()` C++.

Функции для выделения и освобождения памяти

C++ предоставляет набор функций управления памятью вместо простых `malloc()` и `free()` в C. Эти функции обеспечивают богатое поведение выражений `new`, описанных ниже, в разделе "Построение динамических переменных с помощью выражений new". Перегрузки `operator new()` выделяют память для одного экземпляров любого типа. Перегрузки `operator new[]()` выделяют память для массивов любого типа. Когда операторы и для массива, и для одного экземпляра работают одинаково, я буду описывать их вместе, как `operator new()`, подразумевая наличие эквивалентного `operator new[]()`.

`operator new()` реализует выделение памяти

Выражение `new` вызывает одну из нескольких версий `operator new()` для получения памяти для динамической переменной или `operator new[]()` — для получения памяти для динамического массива. C++ предоставляет реализации по умолчанию этих операторов. Он также неявно объявляет их, поэтому они могут быть вызваны программой без необходимости включать заголовочный файл `<new>`. При желании программа может перекрыть операторы, предоставляемые по умолчанию, собственной реализацией.

Для целей оптимизации следует хорошо изучить `operator new()`, поскольку используемый по умолчанию диспетчер памяти имеет высокую стоимость. В некоторых случаях программа может выделять память очень эффективно, предоставляя специализированную реализацию.

C++ определяет ряд перегрузок `operator new()`.

`void* ::operator new(size_t)`

По умолчанию память для всех динамически выделенных переменных выделяется путем вызова этой перегрузки `operator new()` с указанием минимального количества выделяемых байтов в качестве аргумента. Если не хватает памяти для выполнения запроса, реализация этой перегрузки в стандартной библиотеке вызывает исключение `std::bad_alloc`.

Реализации всех других перегрузок `operator new()` вызывают данную. Программа может глобально изменить способ выделения памяти, предоставляя определение `::operator new(size_t)` в любой единице компиляции.

Хотя стандарт C++ этого не требует, версия этой перегрузки в стандартной библиотеке обычно реализуется с помощью вызова `malloc()`.

`void* ::operator new[](size_t)`

Память для массивов выделяется путем вызова этой перегрузки. Реализация стандартной библиотеки вызывает `::operator new(size_t)`.

`void* ::operator new(size_t, const std::nothrow_tag&)`

Выражение `new` наподобие `Foo* p = new(std::nothrow) Foo(123);` вызывает перегрузку `operator new()`, не генерирующую исключения. Если память недоступна, эта версия вместо генерации исключения `std::bad_alloc` возвращает `nullptr`. Реализация стандартной библиотеки вызывает `operator new(size_t)` и перехватывает все исключения, которые этот оператор может сгенерировать.

`void* ::operator new[](size_t, const std::nothrow_tag&)`

Это версия не генерирующего исключения `operator new()` для выделения памяти для массивов.

Выражение `new` может вызвать `operator new()` с произвольной сигнатурой при условии, что первый аргумент имеет тип `size_t`. Все эти перегрузки `operator new()` называются размещающим `operator new()`. Выражение `new` сопоставляет типы переданных аргументов с сигнатурами функций доступных операторов `operator new()`, чтобы определить, какая именно функция используется.

Стандартная библиотека предоставляет и неявно объявляет две перегрузки размещающего `operator new()`. Они не выделяют память (первый этап жизненного цикла динамической переменной), а содержат дополнительный аргумент, принимающий указатель на память, уже выделенную программой. Вот эти перегрузки.

`void* ::operator new(size_t, void*)`

Это размещающий `operator new()` для переменных. Он принимает указатель на память в качестве второго аргумента и просто возвращает этот указатель.

`void* ::operator new[](size_t, void*)`

Это версия размещающего `operator new()` для массивов. Она принимает указатель на память в качестве второго аргумента и просто возвращает этот указатель.

Эти две перегрузки размещающего `operator new()` вызываются размещающим выражением `new`, которое имеет вид `new(p) type`, где `p` — указатель на корректную область памяти. Стандарт гласит, что эти перегрузки не могут быть заменены кодом разработчика. Если они заменены, и новый код не делает ничего, кроме возвращения указателя, переданного в качестве аргумента, бо́льшая часть стандартной библиотеки перестает работать. Это важно знать, потому что некоторые компиляторы C++ не налагают запрет в отношении замены этих перегрузок; это означает, они могут быть заменены, например кодом, который выводит диагностику.

Помимо этих двух размещающих перегрузок `operator new()`, другие перегруженные версии размещающего `operator new()` размещения не имеют определенного значения в C++ и могут использоваться разработчиком, как ему угодно.

operator delete() освобождает выделенную память

Выражение delete вызывает operator delete() для возврата памяти, выделенной для динамической переменной, среде выполнения, и operator delete[]() для возврата памяти, выделенной для динамического массива.

Операторы new и delete работают вместе над выделением и освобождением памяти. Если программа определяет operator new(), который выделяет память из специального пула памяти или специальным образом, то она должна определить и соответствующий operator delete() в той же области видимости для возврата памяти в пул, из которого она была выделена; в противном случае поведение является неопределенным.

Функции управления памятью из библиотеки C

C++ предоставляет функции библиотеки C malloc(), calloc() и realloc() для выделения памяти и функцию free() для освобождения памяти, которая больше не нужна. Эти функции предоставляются для совместимости с программами на языке программирования C.

void* malloc(size_t size) реализует этап выделения памяти жизненного цикла динамической переменной и возвращает указатель на блок памяти, достаточный для хранения size байтов, или nullptr, если доступной памяти нет.

void free(void*p) реализует этап освобождения и возвращает память, на которую указывает указатель p, диспетчеру памяти.

void* calloc(size_t count, size_t size) реализует этап выделения памяти для динамического массива. Функция выполняет простое вычисление, преобразуя число count элементов массива, каждый размером size байтов, в количество байтов, а затем возвращает значение вызова malloc().

void* realloc(void*p, size_t size) изменяет размер блока памяти, перемещая его в новое место в памяти в случае необходимости. Содержимое старого блока будет скопировано в новый блок (в количестве байтов, равном минимальному значению из старого и нового размеров). Функция realloc() должна использоваться очень осторожно. Иногда она перемещает указанный блок в новое место и удаляет старый блок. При этом все указатели в старый блок становятся недействительными. Иногда функция повторно использует существующий блок, который оказывается больше, чем запрошенный размер.

В соответствии со стандартом C++ функции malloc() и free() работают с областью памяти, именуемой "куча", в то время как перегрузки operator new() и operator delete() работают с областью памяти, именуемой "свободная память". Тщательно выверенный язык стандарта сохраняет для разработчиков библиотек возможность реализации двух наборов функций по-разному. С учетом сказанного, требования к управлению памятью в C и C++ схожи. Просто неразумно иметь две параллельные, но разные реализации, так что во всех реализациях стандартной библиотеки, которые я знаю, operator new() вызывает malloc() для фактического выделения памяти. Программа может глобально изменить способ управления памятью путем замены malloc() и free().

Построение динамических переменных с помощью выражений new

Программа на C++ запрашивает создание динамической переменной или массива с использованием выражения `new`. Это выражение содержит ключевое слово `new`, за которым следует тип, указатель на который и возвращается данным выражением. Выражение `new` может также содержать инициализатор, задающий начальное значение переменной или каждого элемента массива. Выражение `new` возвращает типизированный указатель на полностью инициализированную переменную C++ или массив, а не просто `void`-указатель на неинициализированную память, возвращаемую C-функциями управления памятью или оператором `operator new()`.

Синтаксис выражения `new` имеет следующий вид:

$$::_{opt} \quad \texttt{new} \quad (\textit{параметры\_размещения})_{opt} \quad (\textit{тип}) \quad \textit{инициализатор}_{opt}$$

или

$$::_{opt} \quad \texttt{new} \quad (\textit{параметры\_размещения})_{opt} \quad \textit{тип} \quad \textit{инициализатор}_{opt}$$

(Индекс `opt` означает необязательность данной части выражения.)

Разница между этими двумя строками — в дополнительных скобках вокруг *типа*, которые иногда необходимы, чтобы помочь компилятору отличить конец *параметров_размещения* от начала сложного *типа* или окончание *типа* от начала *инициализатора*. Более полную информацию по этому вопросу можно найти, среди прочего, в Интернете, например на сайте *cppreference* (`http://en.cppreference.com/w/cpp/language/new`).

Если *тип* объявляет массив, то его наивысшая (т.е. крайняя слева)[1] размерность может быть описана неконстантным выражением, позволяя указать размер массива во время выполнения. Это единственное место в C++, где объявление может иметь переменный размер.

Выражение `new` возвращает rvalue-указатель на динамическую переменную или первый элемент динамического массива. (То, что этот указатель представляет собой rvalue, важно; подробности см. в разделе "Реализация семантики перемещения" главы 6, "Оптимизация переменных в динамической памяти".)

Все версии выражения `new` выполняют куда бо́льшую работу, чем просто вызывают функцию `operator new()` для выделения памяти. Если вызов `operator new()` является успешным, версия для создания отдельного объекта (не массива) создает объект указанного *типа*. Если конструктор генерирует исключение, его члены и базовые классы уничтожаются, а выделенная память возвращается диспетчеру с помощью вызова `operator delete()`, сигнатура которого соответствует функции `operator new()`, использовавшейся для выделения памяти. Если соответствующий `operator delete()` отсутствует, память диспетчеру памяти не возвращается, что потенциально вызывает утечку памяти. Выражение `new` возвращает указатель, повторно генерирует исключение или возвращает `nullptr`, если использовалось выражение `new`, не генерирующее исключений.

[1] В C++ *n*-мерный массив представляет собой массив n–1-мерных массивов. Таким образом, крайняя слева размерность является наивысшей.

Выражение new для массива работает таким же образом, но с дополнительной сложностью: любой из нескольких конструкторов может сгенерировать исключение, что требует от выражения new для массива уничтожить все успешно построенные к этому моменту экземпляры и освободить память до возврата из выражения или повторной генерации исключения.

Почему имеются два вида выражения new: одно — для массивов, а второе — для экземпляров? Выражение для массива может выделить память для хранения числа элементов массива в дополнение к памяти, выделяемой для самого массива. Этим достигается отсутствие необходимости передавать данное значение в выражение delete для массива. Такие дополнительные накладные расходы не являются необходимыми для одного экземпляра. Такое поведение C++ было разработано во времена, когда память была гораздо дороже, чем сегодня.

Не генерирующий исключений оператор new

Если *параметры_размещения* состоят из дескриптора std::nothrow, выражение new не генерирует исключения std::bad_alloc. Вместо этого при недоступности памяти оно возвращает значение nullptr, не пытаясь строить объект.

Исторически были времена, когда многие компиляторы C++ не реализовывали должным образом обработку исключений. Коду, написанному для этих старых компиляторов или перенесенному из C, была необходима функция выделения памяти, возвращающая при нехватке памяти нулевое значение.

Некоторые отрасли — в особенности аэрокосмическая и автомобильная промышленности — ввели стандарты кодирования, которые запрещают генерацию исключений. Это вызывает проблему, поскольку выражения new определяются как генерирующие исключения. Таким образом, возникла необходимость выражения new, которое не будет генерировать исключения.

"Всем известно", что обработка исключений является медленным процессом, а потому выражения new, не генерирующие исключений, должны выполняться быстрее. Однако современные компиляторы C++ реализуют механизм обработки исключений с очень низкой стоимостью времени выполнения, если исключение не генерируется, так что эта общеизвестная истина может зависеть от конкретного компилятора. В разделе "Используйте обработку исключений без стоимости" главы 7, "Оптимизация инструкций", стоимость обработки исключений рассматривается более подробно.

Размещающий оператор new не выделяет память

Если *параметры_размещения* представляют собой указатель на имеющуюся корректно выделенную память, выражение new не вызывает диспетчер памяти, а просто размещает объект указанного *типа* в местоположении, на которое указывает указатель (в этом местоположении должно быть достаточно места для хранения *типа*). Размещающий оператор new используется следующим образом:

```
char mem[1000];
class Foo {...};
Foo* foo_p = new (mem) Foo(123);
```

В этом примере экземпляр класса Foo размещается в начале массива mem. Размещающий оператор new вызывает конструктор класса для выполнения инициализации экземпляра класса. Для базового типа размещающий оператор new выполняет инициализацию вместо конструирования.

Поскольку размещающий оператор new не выделяет память, соответствующего размещающего оператора delete нет. Экземпляр Foo, размещенный в начале массива mem, при выходе mem из области видимости автоматически не уничтожается. От разработчика требуется должным образом уничтожить экземпляры, созданные с использованием размещающих операторов new, путем явного вызова деструктора класса. В самом деле, если экземпляр Foo размещен в памяти, которая объявлена как экземпляр Bar, будет вызван деструктор Bar с неопределенным (и обычно катастрофическим) результатом. Таким образом, размещающий оператор new следует использовать с памятью, возвращаемой оператором operator new(), или с памятью, занимаемой массивом элементов типа char или некоторого другого базового типа.

Размещающий operator new() используется в параметре Allocator шаблонов контейнеров стандартной библиотеки, которые должны размещать экземпляры класса в ранее выделенной, но пока что неиспользуемой памяти. Более детально этот вопрос рассматривается ниже, в разделе "Пользовательские аллокаторы стандартной библиотеки".

Пользовательский размещающий оператор new

Если *параметры_размещения* представляют собой нечто отличное от std::nothrow или простого указателя, выражение new называется *пользовательским размещающим оператором new*. C++ не приписывает никакого конкретного смысла пользовательскому *размещающему выражению new*. Оно доступно для использования разработчиками для выделения памяти не указанным образом. Пользовательское размещающее выражение new выполняет поиск перегрузки operator new() или operator new[](), первый аргумент которого соответствует size_t, а последующие аргументы соответствуют типам из списка выражения new. Если конструктор динамического объекта генерирует исключение, размещающее выражение new ищет перегрузку operator delete() или operator delete[](), первым параметром которой является void*, а последующие аргументы соответствуют типам списка выражения.

Размещающий оператор new полезен, если программе необходимо несколько механизмов создания динамических переменных или для передачи аргументов для диагностики диспетчера памяти.

Проблемой пользовательского размещающего оператора new является то, что нет никакого способа указать соответствующее "пользовательское размещающее выражение delete". Таким образом, в то время как в случае генерации исключения в конструкторе во время выполнения выражения new вызываются различные размещающие перегрузки оператора operator delete(), выражения delete эти перегрузки вызывать не могут. Это создает проблему для разработчиков, так как стандарт гласит, что если оператор operator delete() не соответствует оператору operator new(), который выделяет память для динамической переменной, то поведение является

неопределенным. Необходимо объявить соответствующий размещающий оператор operator delete(), так как именно он вызывается в выражении new, если конструктор объекта генерирует исключение. Однако нет способа вызывать его из выражения delete. Комитет по стандартизации работает над решением этой проблемы в будущих версиях стандарта C++.

Простейшее решение заключается в том, чтобы заметить, что если размещающий оператор operator new() совместим с обычным оператором operator delete(), то поведение, будучи неопределенным, на само деле является вполне предсказуемым. Другое решение заключается в том, чтобы заметить, что выражения delete вовсе не столь сложные и в них нет никакой магии, так что при необходимости их можно закодировать как свободные функции.

operator new() класса обеспечивает тонкий контроль выделения памяти

Выражение new ищет оператор operator new() в области видимости создаваемого типа. Таким образом, класс может предоставить реализации этих операторов, чтобы получить тонкий контроль над выделением памяти только для этого класса. Если класс не определяет специфичный для данного класса оператор operator new(), то используется глобальный оператор. Чтобы использовать глобальный operator new() вместо версии данного оператора для конкретного класса, программист может использовать оператор глобальной области видимости :: в выражении new:

```
Foo* foo_p = ::new Foo(123);
```

Определенный для данного класса operator new() вызывается только для выделения памяти для экземпляров класса, который определяет эту функцию. Функции-члены класса, которые содержат выражения new с другими классами, используют operator new(), определенный для этого другого класса (если таковой имеется), или глобальный operator new() по умолчанию.

Определенный для конкретного класса operator new() может быть более эффективным, поскольку выделяет объекты единственного размера. Таким образом, первый же свободный блок всегда пригоден к использованию. Если класс не используется в нескольких потоках, operator new() этого класса может обойтись без накладных расходов на обеспечение безопасности с точки зрения параллельности его внутренних структур данных.

operator new() для конкретного класса определен как статическая функция-член. Это имеет смысл, поскольку operator new() выделяет память для каждого экземпляра.

Если класс реализует пользовательский operator new(), он должен реализовать соответствующий operator delete(), иначе будет вызван глобальный operator delete() с неопределенными и обычно нежелательными результатами.

Уничтожение динамических переменных с помощью выражения delete

Программа возвращает память, используемую динамической переменной, диспетчеру памяти с помощью выражения delete. Выражение delete обрабатывает два последних этапа жизненного цикла динамической переменной: уничтожение

переменной и освобождение памяти, которую она ранее занимала. Выражение `delete` содержит ключевое слово `delete`, за которым следует выражение, производящее указатель переменной для удаления. Синтаксис выражения `delete` выглядит следующим образом:

`::`$_{opt}$ `delete` *выражение*

или:

`::`$_{opt}$ `delete []` *выражение*

(Индекс `opt` означает необязательность данной части выражения.)

Первая форма выражения `delete` удаляет динамическую переменную, созданную с помощью выражения `new`. Вторая форма удаляет динамический массив, созданный с помощью выражения `new[]`. Для обычных переменных и массивов применяются отдельные выражения `delete`, поскольку массивы могут быть созданы иначе, чем обычные переменные. Большинство реализаций выделяют дополнительное место для хранения числа элементов в выделенном массиве, с тем, чтобы вызывалось правильное количество деструкторов. Использование неверной версии выражения `delete` для динамической переменной может привести к разрушительному хаосу, который стандарт C++ называет "неопределенным поведением".

Явный вызов деструктора уничтожает динамическую переменную

Можно выполнить только уничтожение динамической переменной, без освобождения ее памяти, путем явного вызова деструктора вместо использования выражения `delete`. Имя деструктора представляет собой просто имя класса с предшествующей ему тильдой (~):

```
foo_p->~Foo();
```

Явные вызовы деструктора используются там же, где и размещающие `new`; в шаблонах `Allocator` стандартной библиотеки, где деструкция и освобождение происходят раздельно.

Не существует явного вызова конструктора... или существует?

Раздел 13.1 стандарта C++ гласит: "Конструкторы не имеют имени". Поэтому программа не может вызывать конструктор непосредственно. Конструктор вызывается с помощью выражения `new`. Стандарт требователен по отношению к конструкторам, поскольку память, занятая экземпляром класса, до конструктора представляет собой неинициализированное хранилище, а после него — экземпляр. Трудно пояснить это волшебное превращение.

На самом деле это не так уж сложно. Если программа хочет явно вызвать конструктор для уже сконструированного экземпляра класса, синтаксис размещающего `new` достаточно прост:

```
class Blah {
public:
    Blah() {...}
    ...
};
```

```
Blah* b = new char[sizeof(Blah)];
Blah myBlah;
...
new (b) Blah;
new (&myBlah) Blah;
```

Конечно же, компоновщик отлично знает, что имя конструктора `Blah` — `Blah::Blah()`. В Visual C++ инструкция

```
b->Blah::Blah();
```

успешно компилируется и вызывает конструктор `Blah`. Это не кодирование, а ужас, летящий на крыльях ночи, делающий данную книгу одной из первых книг по C++ для "готов". Компилятор GCC в Linux более строго подчиняется стандартам и выдает сообщение об ошибке. Конструктор предназначен для вызова через размещающее `new`.

Высокопроизводительные диспетчеры памяти

По умолчанию все запросы на выделение памяти проходят через `::operator new()`, а на освобождение — через `::operator delete()`. Эти функции образуют диспетчер памяти C++ по умолчанию. Диспетчер памяти C++ по умолчанию должен отвечать многим требованиям.

- Он должен быть эффективным, поскольку, скорее всего, будет очень "горячим".
- Он должен корректно работать в многопоточной программе. Доступ к структурам данных в диспетчере памяти по умолчанию должен быть сериализован.
- Он должен эффективно выделять много объектов одного и того же размера (например, узлы списков).
- Он должен эффективно выделять много объектов разного размера (например, строки).
- Он должен выделять как очень большие структуры данных (буфера ввода-вывода, массивы из миллиона `int`), так и малые структуры данных (наподобие отдельного указателя).
- Для максимальной эффективности он должен быть осведомлен о выравнивании границ для указателей, строк кеша и страниц виртуальной памяти, по крайней мере для больших выделяемых блоков памяти.
- Его производительность со временем не должна уменьшаться.
- Он должен эффективно повторно использовать возвращаемую ему память.

Множество требований к диспетчеру памяти C++ делают его разработку открытой и развивающейся задачей, являющейся темой текущих научных исследований, в настоящее время ведущихся в том числе поставщиками компиляторов. В ряде случаев диспетчер памяти может не удовлетворять всем перечисленным требованиям. Оба эти факта открывают разработчикам возможности для оптимизации.

Версия `::operator new()`, поставляемая с большинством компиляторов C++, является тонкой оболочкой вокруг функции C `malloc()`. В ранние дни C++ эти реализации `malloc()` были разработаны для удовлетворения относительно простых потребностей программ на C в небольшом количестве динамических буферов, а не с учетом приведенного выше длинного списка требований к диспетчеру памяти со стороны программ на C++. Замена простого, предлагаемого поставщиками `malloc()` сложным диспетчером памяти была в те времена настолько успешной оптимизацией, что разработчики могли построить свою репутацию мастера на одной этой простой оптимизации.

Было разработано множество более или менее автономных версий библиотек диспетчеров памяти, которые, как утверждалось, выигрывают в производительности у диспетчеров памяти по умолчанию. Если программа активно использует динамические переменные, включая строки и стандартные контейнеры, заманчиво рассматривать эти замены `malloc()` как некую серебряную пулю, повышающую производительность везде и всегда ценой одного изменения при компоновке и без всего этого скучного профилирования. Но в настоящее время, когда современные диспетчеры памяти имеют выдающуюся производительность, есть причины не хвастать преждевременно преимуществами таких изменений.

- Хотя последние диспетчеры памяти продемонстрировали значительное улучшение производительности над наивными реализациями `malloc()`, конкретный уровень отсчета часто явно не указывается. До меня доходили слухи, что и Windows, и Linux недавно принялись за обновление своих диспетчеров памяти более современными, так что в Linux, начиная с версии 3.7, и в Windows, начиная с Windows 7, может наблюдаться небольшой прирост производительности, связанный с изменениями диспетчеров памяти.
- Более быстрый диспетчер памяти может повысить производительность только там, где выделение и освобождение динамических переменных доминирует во времени выполнения программы. Пусть даже программа выделяет память для структуры данных из несметного количества узлов, но если эта структура долгоживущая, то эффект от улучшения диспетчера памяти ограничивается законом Амдала (см. раздел "Закон Амдала" главы 3, "Измерение производительности"). Хотя сам код диспетчера памяти может оказаться в 3–10 раз быстрее диспетчера по умолчанию, *улучшение общей производительности программы варьируется от незначительной величины до 30% в нескольких крупных программах с открытым исходным кодом.*
- Уменьшение числа вызовов диспетчера памяти средствами, описанными в главе 6, "Оптимизация переменных в динамической памяти", обеспечивает повышение производительности независимо от того, насколько быстрым является диспетчер памяти, а усилия по оптимизации могут быть нацелены на горячий код, обнаруживаемый с помощью профайлера.
- Современные диспетчеры памяти могут платить за повышенную производительность потреблением значительных объемов памяти для различных кешей и пулов свободных блоков. Ограниченные среды выполнения могут попросту не иметь необходимой дополнительной памяти.

Для более старых операционных систем и разработки для встраиваемых устройств имеется несколько замен `malloc()`, считающихся высокопроизводительными.

Hoard (`http://www.hoard.org/`)
: Hoard является коммерциализированной версией многопроцессорного диспетчера памяти из Университета Техаса. Утверждается, что он имеет 3–7-кратное превосходство над `malloc()`. Для коммерческого использования Hoard нужна лицензия.

mtmalloc (`http://bit.ly/mtmalloc`)
: mtmalloc представляет собой замену `malloc()` для многопоточной работы с высокой нагрузкой в Solaris. Он использует принцип выделения памяти "fast fit".

ptmalloc (*glibc malloc*) (`http://bit.ly/1VFcqux`)
: ptmalloc представляет собой `malloc()`, предоставляемый Linux 3.7 и более поздними версиями. Для снижения конкуренции в многопоточных программах использует динамические области памяти, выделяемые для потоков.

TCMalloc (`http://bit.ly/tcmalloc`) (*Thread-Caching `malloc()`*)
: TCMalloc (из пакета *gperftools*), представляет собой замену `malloc()` от Google. Он имеет специализированный аллокатор для малых объектов и тщательно разработанные циклические блокировки для управления большими блоками. По мнению дизайнеров, этот диспетчер памяти ведет себя лучше, чем `malloc()` в glibc. *TCMalloc* протестирован только на Linux.

Для небольших встроенных проектов реализация собственного диспетчера памяти не является неподъемной задачей. Поиск в Интернете "fast-fit memory allocation" предоставляет массу ссылок для начала работы. Я сам реализовывал такой диспетчер для встроенного проекта и получал хорошие результаты. Тем не менее дизайн многопоточных диспетчеров памяти общего назначения сам по себе является темой, которая заполнила бы целую книгу. Люди, которые пишут диспетчеры памяти, являются высококлассными специалистами. Чем более сложны программы и их операционные среды, тем меньше вероятность того, что ваше собственное решение будет производительным и без ошибок.

Диспетчеры памяти для конкретных классов

Даже современный `malloc()` представляет собой компромисс, который оставляет возможности для оптимизации. `operator new()` также может быть перекрыт на уровне класса (см. выше раздел "operator new() класса обеспечивает тонкий контроль выделения памяти"). Когда код, динамически создающий экземпляры класса, является горячим, диспетчер памяти для конкретного класса может помочь повысить производительность.

Если класс реализует `operator new()`, эта функция вызывается вместо глобального `operator new()` при запросе памяти для экземпляров класса. `operator new()` для конкретного класса может использовать дополнительные знания, недоступные

версии по умолчанию. *Все запросы на выделение памяти для экземпляров определенного класса запрашивают одинаковое число байтов. Диспетчеры памяти для запросов одинакового размера особенно легки в написании и эффективно работают* по ряду различных причин.

- Диспетчеры памяти для блоков фиксированного размера эффективно используют возвращаемую память. Они не подвержены фрагментации, поскольку все запросы имеют одинаковый размер.
- Диспетчеры памяти для блоков фиксированного размера могут быть реализованы с малыми накладными расходами памяти или вовсе без них.
- Диспетчеры памяти для блоков фиксированного размера могут обеспечивать гарантии верхней границы количества потребленной памяти.
- Функции диспетчеров памяти для блоков фиксированного размера, которые выделяют и освобождают память, внутренне достаточно просты, чтобы быть эффективно встраиваемыми функциями. Функции диспетчера памяти C++ по умолчанию не могут быть встраиваемыми. Они должны быть вызовами функций, чтобы их можно было перекрывать функциями, определяемыми разработчиком. Функции управления памятью в C `malloc()` и `free()` должны быть обычными функциями по той же причине.
- Диспетчеры памяти для блоков фиксированного размера имеют хорошее поведение кеша. Последний освобожденный узел может быть очередным выделяемым узлом.

Многие разработчики никогда не встречались с диспетчером памяти для конкретного класса. Я подозреваю, что это связано с тем, что они состоят из ряда частей, которые должны быть написаны и собраны вместе, так что это достаточно сложная для изучения тема. Кроме того, даже в большой программе лишь несколько классов могут быть достаточно горячими, чтобы использовать эту оптимизацию. Это не совсем то, что требует многократного применения в обычной программе.

Диспетчер памяти для блоков фиксированного размера

В примере 13.1 определен простой диспетчер памяти для блоков фиксированного размера, который выделяет блоки из одного статически объявленного блока памяти под названием *арена*. Диспетчеры памяти такого рода наиболее часто встречаются во встроенных проектах в качестве альтернативы выделения из свободной памяти. `fixed_block_memory_manager` внутренне очень прост: он представляет собой однонаправленный список блоков свободной памяти. Этот простой дизайн будет использоваться в данной главе для нескольких целей. Давайте рассмотрим его детальнее.

Пример 13.1. Диспетчер памяти для блоков фиксированного размера

```
template <class Arena> struct fixed_block_memory_manager {
    template <int N>
        fixed_block_memory_manager(char(&a)[N]);
    fixed_block_memory_manager(fixed_block_memory_manager&)
        = delete;
```

```
    ~fixed_block_memory_manager() = default;
    void operator=(fixed_block_memory_manager&) = delete;

    void*  allocate(size_t);
    size_t block_size() const;
    size_t capacity() const;
    void   clear();
    void   deallocate(void*);
    bool   empty() const;

private:
    struct free_block {
        free_block* next;
    };
    free_block* free_ptr_;
    size_t      block_size_;
    Arena       arena_;
};

#include "block_mgr.inl"
```

Конструктор, определенный в примере 13.2, принимает в качестве аргумента массив `char` в стиле C. Этот массив формирует арену, из которой будут выделяться блоки памяти. Конструктор представляет собой шаблонную функцию, которая позволяет передавать размер массива в качестве параметра шаблона.

Пример 13.2. Определение конструктора `fixed_block_memory_manager`

```
template <class Arena>
    template <int N>
        inline fixed_block_memory_manager<Arena>
        ::fixed_block_memory_manager(char(&a)[N]) :
            arena_(a), free_ptr_(nullptr), block_size_(0) {
            /* Пусто */
        }
```

Примечания о современном кодировании на C++

Чтобы сохранить аккуратность определения шаблонного класса, можно определить функции-члены шаблонного класса вне определения самого шаблонного класса. Я обычно храню определения функций-членов в файле с суффиксом `.inl` (от слова *inl*ine). Однако, когда определение функции находится вне шаблона класса, необходим более подробный синтаксис, который должен помочь компилятору связать определение с объявлением в теле шаблона класса. Первая строка в примере 13.2, `template<class Arena>`, объявляет параметр шаблона класса. Вторая строка, `template<int N>`, относится к самому конструктору, который является шаблонной функцией. Когда определение функции-члена находится вне тела шаблонного класса, необходимо явным образом указать ключевое слово `inline` (так как иначе функция-член считается встраиваемой, только когда ее определение находится внутри класса).

Функция-член `allocate()` в примере 13.3 убирает блок из списка свободных блоков (если имеется хотя бы один свободный блок) и возвращает его адрес. Если свободный список пуст, `allocate()` пытается получить новый список свободных блоков от диспетчера арены, который я опишу ниже. Если диспетчер арены не имеет больше памяти для выделения, он возвращает `nullptr`, а `allocate()` генерирует исключение `std::bad_alloc`.

Пример 13.3. Функция `allocate()` класса `fixed_block_memory_manager`

```
template <class Arena>
    inline void* fixed_block_memory_manager<Arena>
            ::allocate(size_t size) {
    if (empty()) {
        free_ptr_ = reinterpret_cast<free_block*>
                    (arena_.allocate(size));
        block_size_ = size;
        if (empty())
            throw std::bad_alloc();
    }
    if (size != block_size_)
        throw std::bad_alloc();
    auto p = free_ptr_;
    free_ptr_ = free_ptr_->next;
    return p;
}
```

Функция-член `deallocate()` очень проста — она возвращает блок в список свободных блоков:

```
template <class Arena>
    inline void fixed_block_memory_manager<Arena>
            ::deallocate(void* p) {
    if (p == nullptr)
        return;
    auto fp = reinterpret_cast<free_block*>(p);
    fp->next = free_ptr_;
    free_ptr_ = fp;
}
```

Далее приведены определения остальных функций-членов. Копирование и присваивание диспетчера памяти отключены в определении класса с помощью синтаксиса C++11.

```
template <class Arena>
    inline size_t fixed_block_memory_manager<Arena>
            ::capacity() const {
    return arena_.capacity();
}

template <class Arena>
    inline void fixed_block_memory_manager<Arena>::clear() {
    free_ptr_ = nullptr;
    arena_.clear();
}
```

Арена блоков

Единственная сложность в `fixed_block_memory_manager` связана с созданием первоначального списка свободных блоков. Эта сложность выделена в отдельный шаблон класса. Представленная здесь реализация называется `fixed_arena_controller` и определена в примере 13.4. Здесь *арена* означает замкнутое пространство в памяти, в котором выполняются некоторые действия. `fixed_arena_controller` предоставляет единый статический блок памяти, из которого выделяется память для узлов фиксированного размера.

Пример 13.4. Арена для диспетчера памяти для блоков фиксированного размера

```
struct fixed_arena_controller {
    template <int N>
        fixed_arena_controller(char(&a)[N]);
    fixed_arena_controller(fixed_arena_controller&) = delete;
    ~fixed_arena_controller() = default;
    void operator=(fixed_arena_controller&) = delete;

    void*  allocate(size_t);
    size_t block_size() const;
    size_t capacity() const;
    void   clear();
    bool   empty() const;
private:
    void*  arena_;
    size_t arena_size_;
    size_t block_size_;
};
```

Класс `fixed_arena_controller` предназначен для создания списка блоков памяти. Все блоки памяти имеют один и тот же размер. Размер устанавливается при первом вызове `allocate()`. Каждый блок памяти должен быть достаточно большим, чтобы вместить запрошенное число байтов, но, кроме того, он должен таким, чтобы вместить указатель, который используется, когда блок находится в списке свободных блоков.

Шаблонный конструктор принимает массив от `fixed_block_memory_manager`, сохраняя его размер и указатель на его начало:

```
template <int N>
    inline fixed_arena_controller
            ::fixed_arena_controller(char (&a)[N]) :
    arena_(a), arena_size_(N), block_size_(0) { /* Пусто */
}
```

Все действие происходит в функции-члене `allocate()`. Она вызывается функцией-членом `allocate()` класса `fixed_block_memory_manager`, когда список свободных блоков пуст (что происходит при первом запросе на выделение памяти).

`fixed_arena_controller` имеет один блок памяти для выделения памяти. Если этот блок использован, `allocate()` вызывается снова и должна вернуть признак ошибки, которым в данном случае является значение `nullptr`. Другие виды контроллеров арены могут делить большие блоки памяти, полученные, например, с помощью

вызова ::operator new(). Для других контроллеров арены может быть вполне допустим многократный вызов allocate().

При первом вызове allocate() устанавливает размер блока и емкость (в виде количества блоков). Фактически создание свободного списка — это упражнение по превращению нетипизированных байтов памяти в типизированные указатели. Массив char рассматривается как набор блоков, расположенных один за другим. Первые байты каждого блока представляют собой указатель на следующий блок. Указатель в последнем блоке имеет значение nullptr.

fixed_arena_controller не управляет размером массива арены. В ней может быть несколько неиспользуемых байтов в конце, которые никогда не выделяются. Код для установки указателей свободных блоков не слишком красив; он должен постоянно интерпретировать один вид указателя как иной, выходя за рамки системы типов C++ в царство поведения, зависящего от реализации. Увы, это справедливо в отношении диспетчеров памяти в общем случае и, к сожалению, неизбежно.

Код выделения и освобождения у fixed_arena_controller прост. Он просто "накладывает" список свободных узлов поверх предоставленной конструктору памяти и возвращает указатель на первый элемент списка. Этот код выглядит следующим образом:

```
inline void* fixed_arena_controller
        ::allocate(size_t size) {
    if (!empty())
        return nullptr; // Арена уже выделена

    block_size_ = std::max(size, sizeof(void*));
    size_t count = capacity();

    if (count == 0)      // Арены недостаточно
        return nullptr; // даже для одного элемента

    char* p;
    for (p = (char*)arena_; count > 1; --count, p += size) {
        *reinterpret_cast<char**>(p) = p + size;
    }
    *reinterpret_cast<char**>(p) = nullptr;
    return arena_;
}
```

Далее приведена остальная часть fixed_arena_controller:

```
inline size_t fixed_arena_controller::block_size() const {
    return block_size_;
}

inline size_t fixed_arena_controller::capacity() const {
    return block_size_ ? (arena_size_ / block_size_) : 0;
}

inline void fixed_arena_controller::clear() {
    block_size_ = 0;
}
```

```
inline bool fixed_arena_controller::empty() const {
    return block_size_ == 0;
}
```

Добавление operator new() для конкретного класса

В примере 13.5 приведен очень простой класс с операторами `new` и `delete` для данного класса. Класс содержит также статический член `mgr_`, который является диспетчером памяти с блоками фиксированного размера, описанным выше, в разделе "Диспетчер памяти для блоков фиксированного размера". Операторы `new` и `delete` являются встраиваемыми функциями, которые передают запросы функциям-членам `allocate()` и `deallocate()` объекта `mgr_`.

Пример 13.5. Класс с собственным оператором new

```
class MemMgrTester {
    int contents_;
public:
    MemMgrTester(int c) : contents_(c) {}

    static void* operator new(size_t s) {
        return mgr_.allocate(s);
    }
    static void operator delete(void* p) {
        mgr_.deallocate(p);
    }
    static fixed_block_memory_manager<fixed_arena_controller> mgr_;
};
```

Статический член `mgr_` объявлен как `public`, так что, чтобы облегчить написание тестов производительности, я могу повторно инициализировать список свободных блоков с помощью вызова `mrg_.clear()`. Если объект `mgr_` инициализируется только один раз при запуске программы и не должен быть инициализирован повторно, он может быть сделан закрытым.

Диспетчер памяти, который можно сбросить, как описанный выше, называется диспетчером *пула* памяти, а арена, которой он управляет, называется *пулом памяти*. Диспетчер пула памяти может быть полезен в тех случаях, когда структура данных строится, используется, а затем уничтожается. Если можно быстро повторно инициализировать весь пул памяти, программа может отказаться от освобождения структур данных узел за узлом.

`mgr_` объявлен как статический член класса `MemMgrTester`. Где-то в программе статические члены должны быть определены. Определение статического члена имеет такой вид, как в примере 13.6. Этот код сначала определяет арену памяти, а затем — объект `mgr_`, конструктор которого принимает арену в качестве аргумента.

Пример 13.6. Инициализация диспетчера памяти

```
char arena[4004];
fixed_block_memory_manager<fixed_arena_controller>
    MemMgrTester::mgr_(arena);
```

Приведенный пример не определяет `operator new[]()` конкретного класса для выделения памяти для массивов. Диспетчер памяти для блоков фиксированного размера не будет работать для массивов, которые по определению могут иметь различное количество элементов. Если программа пытается выделить массив `MemMgrTester`, выражение `new` использует глобальный `operator new[]()`, поскольку соответствующий оператор в классе не определен. Память для отдельных экземпляров выделяется с помощью диспетчера памяти для блоков фиксированного размера, а массивы используют `malloc()`.

Производительность диспетчера памяти для блоков фиксированного размера

Диспетчер памяти для блоков фиксированного размера очень эффективен. Методы выделения и освобождения имеют фиксированную стоимость, а кроме того, их код может быть встраиваемым. Но насколько этот диспетчер памяти быстрее, чем `malloc()`?

Я провел два эксперимента для проверки производительности оператора `new` конкретного класса. В первом тесте я выделил миллион экземпляров `MemMgrTester`. Выделение с помощью оператора `new` класса и моего диспетчера памяти для блоков фиксированного размера заняло 4 мс. Выделение же с помощью глобального оператора `new`, который использует `malloc()`, заняло 64 мс. Получается, что в моем тесте диспетчер памяти для блоков памяти фиксированного размера в 16 раз быстрее, чем `malloc()`. Впрочем, этот результат, вероятно, существенно выше улучшения производительности, которое может быть достигнуто в фактической программе. Из закона Амдала вытекает, что чем больше вычислений происходит между выделениями памяти, тем меньше будет выигрыш в производительности от ускорения выделения.

Во втором эксперименте я создал массив из 100 указателей на `MemMgrTester`. Я создал миллион экземпляров `MemMgrTester`, присваивая каждый из них случайной позиции массива и удаляя экземпляр, уже имевшийся там. При использовании диспетчера памяти для блоков фиксированного размера тест занял 25 мс; при использовании диспетчера глобальной памяти по умолчанию потребовалось 107 мс, используя диспетчер глобальной памяти по умолчанию. Итак, применение диспетчера для блоков фиксированного размера оказалось в 4,3 раза быстрее.

Вариации диспетчера памяти для блоков фиксированного размера

Базовая структура диспетчера памяти для блоков фиксированного размера чрезвычайно проста. Различные вариации (любую из которых можно найти, если потратить немного времени на поиск диспетчеров памяти в Интернете) можно проверить на практике и посмотреть, не подойдет ли какая-то из них в большей степени для оптимизируемой программы.

- Вместо того чтобы начинать работу с фиксированной ареной, память (при пустом списке свободных блоков) можно выделять с помощью `malloc()`, а освобождаемые блоки кешировать в собственном списке для быстрого повторного использования.

- Арена может быть не фиксированного размера, а создаваться путем вызова `malloc()` или `::new`. При необходимости можно поддерживать цепочку блоков арены, так что не будет никаких ограничений на то, сколько небольших блоков может быть выделено. Диспетчер памяти для блоков фиксированного размера по-прежнему сохранит свои преимущества скорости и компактности, даже если он будет иногда вызывать `malloc()`.
- Если экземпляры класса используются некоторое время, а затем уничтожаются все вместе, диспетчер памяти может использоваться в качестве пула памяти. В пуле памяти распределение происходит как обычно, но память не освобождается вообще. Когда программа завершает работу с экземплярами класса, они все убираются одним махом путем реинициализации статической арены или возврата динамически выделяемой арены системному диспетчеру памяти. Даже если все они удаляются вместе, каждый выделенный блок по-прежнему должен быть освобожден с помощью вызова деструктора. Многие пулы памяти в Интернете забывают эту маленькую, но ответственную деталь.

Можно разработать диспетчер памяти общего назначения, который обслуживает запрос каждого размера из другой арены и возвращает блоки каждого размера в соответствующий свободный список. Если все запрашиваемые размеры округлять до следующей более высокой степени двойки, то в результате получится диспетчер памяти "fast-fit". Обычно такой диспетчер памяти выделяет объекты до определенного максимального размера, отсылая бо́льшие запросы к диспетчеру памяти по умолчанию. Код такого диспетчера слишком большой, чтобы воспроизвести его в этой книге, но он доступен в Интернете.

В Boost имеется диспетчер памяти для блоков фиксированного размера под названием "Pool" (`http://www.boost.org/doc/libs/release/libs/pool/`).

Небезопасные с точки зрения параллельности диспетчеры более эффективны

Диспетчер памяти для блоков фиксированного размера обязан некоторой долей своей эффективности тому, что он небезопасен с точки зрения потоков. Такие небезопасные с точки зрения параллельности диспетчеры являются эффективными по двум причинам. Во-первых, они не требуют механизмов синхронизации для сериализации критических разделов. Синхронизация является дорогостоящей операцией, потому что в центре каждого примитива синхронизации находится очень медленный барьер памяти (см. разделы "Атомарность с помощью взаимоисключений" и "Барьеры памяти" главы 12, "Оптимизация параллельности"). Даже если диспетчер памяти вызывается только одним потоком (что достаточно типично), эти дорогостоящие инструкции замедляют его работу.

Во-вторых, небезопасные диспетчеры являются эффективными, поскольку они никогда не ожидают примитивы синхронизации. Если программа имеет несколько потоков, они конкурируют за диспетчер памяти, как за ресурс. Чем больше потоков в системе, тем больше конкуренция и тем больше обращений к аллокатору сериализует деятельность потоков.

Когда класс реализует свой диспетчер памяти, то даже если программа в целом является многопоточной, но данный класс используется только в одном потоке, ему никогда не приходится ждать. Вызовы же диспетчера памяти по умолчанию, напротив, конкурируют в многопоточных программах даже для объектов, используемых только в одном потоке.

Небезопасные диспетчеры также гораздо проще писать, чем потокобезопасные, поскольку задача сведения к минимуму критического раздела диспетчера памяти для повышения его эффективности может быть весьма сложной.

Пользовательские аллокаторы стандартной библиотеки

Классы контейнеров стандартной библиотеки C++ интенсивно используют динамическую память. Они являются естественным местом для поиска возможностей оптимизации, включая пользовательские диспетчеры памяти, подобные описанным выше, в разделе "Диспетчер памяти для блоков фиксированного размера".

Однако имеется проблема. Динамически выделенные переменные в `std::list<T>` не имеют пользовательского типа `T`. Они имеют некоторый невидимый снаружи тип, что-то вроде `listitem<T>`, который содержит указатели на предыдущий и последующий элементы списка, а также тип данных `T`. Динамически выделенные переменные в `std::map<K,V>` имеют другой скрытый тип, что-то вроде `treenode<std::pair<const K,V>>`. Эти шаблоны классов похоронены в заголовочных файлах компилятора. Изменить эти классы, чтобы добавить к ним операторы `new` и `delete`, невозможно[2]. И, кроме того, шаблоны являются обобщенными. Разработчик обычно хочет изменить диспетчер памяти для некоторых экземпляров обобщенного шаблона в конкретной программе, а не для всех экземпляров во всех программах. К счастью, стандарт C++ предоставляет механизм для определения диспетчера памяти, используемого каждым контейнером. Контейнеры стандартной библиотеки принимают аргумент `Allocator`, который дает те же возможности настройки управления памятью, что и оператор `new` для конкретного класса.

`Allocator` представляет собой шаблонный класс, который управляет памятью. Он делает три главных действия: получает память от диспетчера памяти и возвращает ее ему, а также конструирует копированием сам себя из связанного аллокатора. Казалось бы, просто? но это не так. Аллокаторы имеют длинную и даже болезненную историю, которая рассматривается ниже, в разделе "Дополнительные определения для аллокатора C++98". По мнению некоторых влиятельных разработчиков, аллокаторы представляют собой самую никчемную часть C++. Тем не менее, если код достаточно горячий, а контейнер является одним из контейнеров на основе узлов (`std::list`, `std::forward_list`, `std::map`, `std::multimap`, `std::set` или `std::multiset`), то реализация пользовательского аллокатора может быть оправданной и помочь в улучшении производительности.

[2] Надеюсь, вы не хотите возразить "Ну почему же? Надо просто зайти в каталог, где компилятор хранит свои стандартные заголовочные файлы, и немного их переписать..." Это можно сделать. Но это выглядит так грязно! Брррр...

Реализации аллокаторов варьируются от простейших до совершенно заумных. Аллокатор по умолчанию `std::allocator<T>` представляет собой тонкую оболочку вокруг `::operator new()`. Разработчик может предоставить нестандартный аллокатор с поведением, отличным от поведения по умолчанию.

Имеются две основные разновидности аллокаторов. Простейшая разновидность является аллокатором *без состояния*, т.е. аллокатором, не имеющим нестатических членов-данных. Аллокатор по умолчанию для контейнеров стандартной библиотеки, `std::allocator<T>`, является аллокатором без состояния. Такие аллокаторы обладают рядом хороших свойств.

- Такой аллокатор может быть создан с помощью конструктора по умолчанию. Нет никакой необходимости создавать явный экземпляр аллокатора без состояния, чтобы передать его конструктору класса контейнера. `std::list<myClass,myAlloc>my_list;` строит список экземпляров `myClass`, память для которых выделяется аллокатором без состояния `myAlloc`.
- Аллокатор без состояния не занимает никакого пространства в экземпляре контейнера. Большинство классов контейнеров стандартной библиотеки наследуют свои аллокаторы, используя преимущества оптимизации пустого базового класса для получения базового класса нулевого размера.
- Два экземпляра аллокатора без состояния `my_allocator<T>` являются неразличимыми. Это означает, что память для объекта, выделенная одним аллокатором, может быть освобождена другим. Это, в свою очередь, делает возможными такие операции, как функция-член `splice()` класса `std::list`. *Иногда, но не всегда* может быть так, что два аллокатора без состояний для разных типов, наподобие `AllocX<T>` и `AllocX<U>`, оказываются эквивалентными. Это справедливо, например, для `std::allocator`.

  Эквивалентность также означает возможность эффективного перемещающего присваивания и функции `std::swap()`. Если два аллокатора не эквивалентны, содержимое одного контейнера должно быть скопировано в другой контейнер с использованием глубокого копирования и аллокатора целевого контейнера.

  Обратите внимание, что хотя эквивалентность может иметь место даже для экземпляров совершенно несвязанных типов аллокаторов наподобие `AllocX<T>` и `AllocY<U>`, это свойство не является особо важным. Тип контейнера включает тип аллокатора. Вы не можете выполнить соединение `std::list<T,AllocX>` с `std::list<T,AllocY>` точно так же, как не можете выполнить его для `std::list<int>` и `std::list<string>`.

Конечно, основной недостаток аллокатора без состояния является тем же, что и его основное преимущество. Все экземпляры аллокатора без состояния по своей природе получают память от одного и того же диспетчера памяти. Это глобальный ресурс, и мы имеем зависимость от глобальной переменной.

Аллокатор с внутренним состоянием более сложно создавать и использовать по следующим причинам.

- В большинстве случаев аллокатор с локальным состоянием не может быть построен по умолчанию. Аллокатор должен быть создан, а затем передан в конструктор контейнера:

```
char arena[10000];
MyAlloc<Foo> alloc(arena);
std::list<Foo, MyAlloc<Foo>> foolist(alloc);
```

- Состояние аллокатора должно храниться в каждой переменной, увеличивая ее размер. Это очень болезненно для таких контейнеров, как `std::list` и `std::map`, которые создают много узлов, но именно эти контейнеры, как правило, пытаются оптимизировать программисты.
- Два аллокатора одного типа могут не совпасть при сравнении из-за того, что они имеют различные внутренние состояния. В результате одни операции над контейнерами с этим типом аллокатора оказываются невозможными, а другие — крайне неэффективными.

Однако аллокаторы с состояниями имеют одно важное преимущество — гораздо проще создать несколько видов арен памяти для различных целей, если все запросы не обязаны проходить через единый глобальный диспетчер памяти.

Для разработчиков, пишущих пользовательские аллокаторы для повышения производительности, выбор между аллокаторами с состояниями и без таковых сводится к вопросу, со сколькими классами намерен работать программист. Если достаточно горячим для оптимизации является только один класс, проще воспользоваться аллокатором без состояния. Если же разработчик хочет оптимизировать больше чем пару классов, более гибким решением будет применение аллокаторов с локальным состоянием. Однако разработчик должен доказать необходимость применения пользовательского аллокатора с использованием профайлера. Написание пользовательских аллокаторов для многих контейнеров может не дать результат, который стоил бы затраченных усилий и времени.

Минимальный аллокатор в C++11

Если программисту повезло и в его руках компилятор и стандартная библиотека, которые полностью соответствуют стандарту C++11, он может создать *минимальный аллокатор*, который требует лишь нескольких определений. В примере 13.7 представлен аллокатор, выполняющий приблизительно те же действия, что и `std::allocator`.

Пример 13.7. Минимальный аллокатор C++11

```
template <typename T> struct my_allocator {
    using value_type = T;

    my_allocator() = default;

    template <class U> my_allocator(const my_allocator<U>&) {}

    T* allocate(std::size_t n, void const* = 0) {
        return reinterpret_cast<T*>(::operator new(n*sizeof(T)));
    }
```

```
    void deallocate(T* ptr, size_t) {
        ::operator delete(ptr);
    }
};

template <typename T, typename U>
    inline bool operator==(const my_allocator<T>&,
                           const my_allocator<U>&) {
    return true;
}

template <typename T, typename U>
    inline bool operator!=(const my_allocator<T>& a,
                           const my_allocator<U>& b) {
    return !(a == b);
}
```

Минимальный аллокатор содержит следующие несколько функций.

`allocator()`

Это конструктор по умолчанию. Если аллокатор имеет конструктор по умолчанию, разработчику не нужно явным образом создавать экземпляр для передачи его конструктору контейнера. Конструктор по умолчанию в аллокаторах без состояния обычно пуст и обычно отсутствует во всех аллокаторах с нестатическим состоянием.

`template <typename U> allocator(U&)`

Этот копирующий конструктор позволяет преобразовать `allocator<T>` в связанный аллокатор для закрытого класса, такого как `allocator<treenode<T>>`. Это важно, потому что в большинстве контейнеров память для узлов типа `T` не выделяется.

Копирующий конструктор обычно пуст в аллокаторах без состояния, но должен копировать или клонировать состояния в аллокаторах с нестатическими состояниями.

`T* allocate(size_type n, const void* hint = 0)`

Эта функция выделяет достаточную память для хранения `n` байтов и возвращает указатель на них или генерирует исключение `std::bad_alloc`. Подсказка `hint` предназначена для помощи аллокатору в решении вопросов, связанных с "локальностью". Я еще не встречался с реализацией, в которой использовалась бы эта подсказка.

`void deallocate(T* p, size_t n)`

Эта функция освобождает память, которая ранее была возвращена вызовом `allocate()`, на которую указывает указатель `p` и которая занимает `n` байтов, передавая ее диспетчеру памяти. `n` должно быть тем же самым, что и аргумент вызова `allocate()`, который выделил память, на которую указывает `p`.

`bool operator==(allocator const& a) const`
`bool operator!=(allocator const& a) const`

Эти функции проверяют два экземпляра аллокатора одного и того же типа на равенство. *Если экземпляры равны, то память для объектов, выделенная одним экземпляром, может быть безопасно освобождена другим.* Это значит, что эти два экземпляра выделяют память для объектов из одной области памяти.

Равенство имеет важное значение. Например, оно означает, что элементы `std::list` из одного списка можно соединять с другим только в том случае, если два списка имеют аллокаторы одинакового типа и экземпляры аллокаторов равны. Тип аллокатора является частью типа экземпляра контейнера, поэтому если аллокаторы имеют разные типы, то не имеет значения, используют ли они одну и ту же память.

Аллокаторы без состояния возвращают при проверке на равенство, безусловно, `true`. Аллокаторы с нестатическими состояниями должны сравнивать состояния для определения равенства или просто возвращать значение `false`.

Дополнительные определения для аллокатора C++98

C++11 прилагает значительные усилия, чтобы аллокаторы было проще разрабатывать. Это упрощение достигается ценой усложнения классов контейнеров. Разработчик, который должен написать аллокатор для контейнера стандартной библиотеки до C++11, поймет, о чем я говорю.

Аллокаторы изначально не предназначались для управления памятью (или, по крайней мере, предназначались не только для этого). Концепция аллокаторов развилась в 1980-е годы, когда микропроцессоры и разработчики пытались вырваться из ограниченного 16-битного адресного пространства. Персональные компьютеры того времени формировали адрес из регистра сегмента плюс смещение. Каждая программа компилировалась для *модели памяти*, которая описывала способ работы указателей по умолчанию. Существует множество моделей памяти. Одни были эффективными, но имели ограниченный объем памяти программы или ее данных. Другие модели памяти разрешали использовать больше памяти, но были медленнее. Компиляторы С тех времен использовали расширения с указанием дополнительных типов модификаторов таким образом, чтобы отдельные указатели могли быть объявлены как `far` или `near`, основываясь на том, доступ к какому количеству памяти с их помощью хотел получить разработчик.

Изначально аллокаторы задумывались как средство для наведения порядка в этом бедламе моделей памяти. Но к моменту появления аллокаторов в C++ производители оборудования уже не могли не реагировать на возмущенные вопли тысяч разработчиков на С и реализовали унифицированную модель памяти без сегментных регистров. Кроме того, решение с использованием аллокаторов было неэффективным при использовании компиляторов тех времен.

До появления C++11 каждый аллокатор содержал все функции рассмотренного выше минимального аллокатора, а также весь перечисленный ниже багаж.

`value_type`

Тип объекта, для которого выделяется память.

`size_type`

Интегральный тип, достаточно большой для хранения максимального количества байтов, которые может выделить данный аллокатор.

Для аллокаторов, используемых в качестве параметров шаблонов контейнеров стандартной библиотеки, это определение должно иметь вид `typedef size_t size_type;`.

`difference_type`

Интегральный тип, достаточно большой для хранения максимальной разности двух указателей.

Для аллокаторов, используемых в качестве параметров шаблонов контейнеров стандартной библиотеки, это определение должно иметь вид `typedef ptrdiff_t difference_type;`.

`pointer`
`const_pointer`

Тип указателя на (`const`) `T`.

Для аллокаторов, используемых в качестве параметров шаблонов контейнеров стандартной библиотеки, это определение должно иметь вид

```
typedef T* pointer;
typedef T const* const_pointer;
```

Для прочих аллокаторов `pointer` может быть классом в стиле указателя, который реализует оператор разыменования `operator*()`.

`reference`
`const_reference`

Тип ссылки на (`const`) `T`.

Для аллокаторов, используемых в качестве параметров шаблонов контейнеров стандартной библиотеки, это определение должно иметь вид

```
typedef T& reference;
typedef T const& const_reference;
```

`pointer address(reference)`
`const_pointer address(const_reference)`

Функции, которые для данной ссылки на (`const`) `T` дают указатель на (`const`) `T`.

Для аллокаторов, используемых в качестве параметров шаблонов контейнеров стандартной библиотеки, это определение должно иметь вид

```
pointer address(reference r) { return &r; }
const_pointer address(const_reference r) { return &r; }
```

Эти функции были предназначены для абстрактных моделей памяти. К сожалению, они мешают совместимости с контейнерами стандартной библиотеки, которая требует от `pointer` *быть* `T*`, чтобы обеспечить эффективность итераторов с произвольным доступом и работу бинарного поиска.

Несмотря на то что эти определения имеют фиксированные значения для аллокаторов, используемых контейнерами стандартной библиотеки, они необходимы, поскольку код контейнера в C++98 их использует, например:

```
typedef size_type allocator::size_type;
```

Некоторые разработчики выводят свои шаблоны аллокаторов из `std::allocator`, чтобы получить эти определения без их явного написания. Однако эта практика является спорной. В конце концов, когда-нибудь `std::allocator` может и измениться. Так, он часто изменялся в первые годы и снова изменился при принятии стандарта C++11, так что подобные опасения являются вполне обоснованными. Другой подход заключается в том, чтобы выделить наиболее неизменные из них, например, следующим образом:

```
template <typename T> struct std_allocator_defs {
    typedef T value_type;
    typedef T* pointer;
    typedef const T* const_pointer;
    typedef T& reference;
    typedef const T& const_reference;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

    pointer address(reference r) { return &r; }
    const_pointer address(const_reference r) { return &r; }
};
```

Доводя эту идею до логического завершения, данные определения можно сделать классом свойств, как у некоторых из более сложных шаблонов аллокаторов, которые можно найти в вебе. Это также то, что делает минимальный аллокатор C++11, только класс свойств работает в обратном направлении. Класс свойств выполняет поиск этих определений в шаблоне аллокатора и предоставляет стандартное определение, если в аллокаторе соответствующего определения нет. Затем код контейнера ссылается на класс `allocator_traits`, а не на класс аллокатора, как здесь:

```
typedef std::allocator_traits<MyAllocator<T>>::value_type value_type;
```

Теперь пришло время взглянуть на важные определения (помните, что сюда включаются и определения минимального распределителя из приведенного выше раздела "Минимальный аллокатор в C++11").

`void construct(pointer p, const T& val)`

Эта функция конструирует экземпляр `T` копированием с использованием размещающего `new`:

```
new(p) T(val);
```

В случае C++11 эта функция может быть определена так, что список аргументов передается конструктору `T`:

```
template <typename U, typename... Args>
void construct(U* p, Args&&... args) {
    new(p) T(std::forward<Args>(args...));
}
```

`void destroy(pointer p);`

Эта функция уничтожает указатель на `T`, вызывая `p->~T();`.

`rebind::value`

Объявление `struct rebind` находится в самом сердце аллокатора. Обычно оно имеет следующий вид:

```
template <typename U> struct rebind {
    typedef allocator<U> value;
};
```

`rebind` дает формулу для создания аллокатора для нового типа `U`, имея `allocator<T>`. Каждый аллокатор должен предоставлять такую формулу. Именно так контейнер наподобие `std::list<T>` выделяет память для экземпляров `std::list<T>::listnode<T>`. В большинстве контейнеров память для узлов типа `T` никогда не выделяется.

В примере 13.8 приведен полный аллокатор в стиле C++98, эквивалентный минимальному аллокатору C++11 из примера 13.7.

Пример 13.8. Аллокатор C++98

```
template <typename T> struct my_allocator_98 :
        public std_allocator_defs<T> {

    template <typename U> struct rebind {
        typedef my_allocator_98<U, n> other;
    };

    my_allocator_98() {/* Пусто */}
    my_allocator_98(my_allocator_98 const&) {/* Пусто */}

    void construct(pointer p, const T& t) {
        new(p) T(t);
    }
    void destroy(pointer p) {
        p->~T();
    }
    size_type max_size() const {
        return block_o_memory::blocksize;
    }
    pointer allocate(
            size_type n,
            typename std::allocator<void>::const_pointer = 0) {
        return reinterpret_cast<T*>(::operator new(n*sizeof(T)));
    }
    void deallocate(pointer p, size_type) {
        ::operator delete(ptr);
    }
};

template <typename T, typename U>
    inline bool operator==(const my_allocator_98<T>&,
                           const my_allocator_98<U>&) {
    return true;
}
```

```
template <typename T, typename U>
    inline bool operator!=(const my_allocator_98<T>& a,
                           const my_allocator_98<U>& b) {
    return !(a == b);
}
```

При изучении исходных текстов аллокаторов, найденных в Интернете, разработчики будут сталкиваться со множеством различных вариантов написания типов. Очень осторожный, заботящийся о соответствии стандарту разработчик запишет сигнатуру функции `allocate()` как

```
pointer allocate(size_type n,
                 typename std::allocator<void>::const_pointer = 0);
```

в то же время менее осторожный и аккуратный может записать ту же сигнатуру для аллокатора, предназначенного строго для использования с контейнерами стандартной библиотеки, как

```
T* allocate(size_t n, void const* = 0);
```

Первая сигнатура технически соответствует стандарту в наибольшей степени, но и вторая сигнатура будет успешно компилироваться и имеет преимущество краткости. Так нередко случается в мире шаблонов.

Еще одна проблема, связанная с исходным текстом аллокаторов, которые могут быть найдены в Интернете, заключается в том, что функция `allocate()` должна генерировать исключение `std::bad_alloc`, если ей не удается удовлетворить запрос. Так, например, следующий код, который вызывает `malloc()` для выделения памяти, не соответствует стандарту, потому что `malloc()` может возвращать `nullptr`:

```
pointer allocate(size_type n,
                 typename std::allocator<void>::const_pointer = 0) {
    return reinterpret_cast<T*>(malloc(n*sizeof(T)));
}
```

Аллокатор блоков фиксированного размера

Классы контейнеров стандартной библиотеки `std::list`, `std::map`, `std::multimap`, `std::set` и `std::multiset` создают структуру данных из множества одинаковых узлов. Такие классы могут воспользоваться простыми аллокаторами, реализуемыми с помощью диспетчера памяти для блоков фиксированного размера, описанного выше, в разделе "Диспетчер памяти для блоков фиксированного размера". Частичное определение в примере 13.9 демонстрирует две функции — `allocate()` и `deallocate()`. Другие определения идентичны определениям стандартного аллокатора, приведенного в примере 13.8.

Пример 13.9. Аллокатор блоков фиксированного размера

```
extern fixed_block_memory_manager<fixed_arena_controller>
    list_memory_manager;

template <typename T> class StatelessListAllocator {
public:
    ...
```

```
    pointer allocate(
            size_type count,
            typename std::allocator<void>::const_pointer = nullptr) {
        return reinterpret_cast<pointer>
                (list_memory_manager.allocate(count * sizeof(T)));
    }
    void deallocate(pointer p, size_type) {
        string_memory_manager.deallocate(p);
    }
};
```

Как упоминалось ранее, `std::list` никогда не пытается выделять память для узлов типа `T`. Вместо этого `std::list` использует параметр шаблона `Allocator` для построения `listnode<T>` с помощью вызова `list_memory_manager.allocate(sizeof (<listnode<T>>))`.

Аллокатор списка требует изменения ранее определенного диспетчера памяти. Реализация `std::list`, поставляемая с Microsoft Visual C++ 2015, выделяет специальный ограничивающий узел, размер которого отличен от размера других узлов списка. Он меньше, чем обычный узел списка, так что можно внести в диспетчер памяти для фиксированных блоков небольшие изменения, которые позволят ему работать. Измененная версия показана в примере 13.10. Изменение заключается в том, что вместо тестирования текущего запроса на равенство сохраненному размеру блока `allocate()` проверяет только, не больше ли запрашиваемый размер сохраненного размера блока.

Пример 13.10. Модифицированная функция `allocate()`

```
template <class Arena>
inline void* fixed_block_memory_manager<Arena>
            ::allocate(size_t size) {
    if (empty()) {
        free_ptr_ = reinterpret_cast<free_block*>
                    (arena_.allocate(size));
        block_size_ = size;
        if (empty())
            throw std::bad_alloc();
    }
    if (size > block_size_)
        throw std::bad_alloc();
    auto p = free_ptr_;
    free_ptr_ = free_ptr_->next;
    return p;
}
```

Производительность аллокатора блоков фиксированного размера

Я написал программу для тестирования производительности аллокатора блоков фиксированного размера. Программа представляет собой цикл, который многократно создает список из 1000 целых чисел, а затем удаляет его. При использовании аллокатора по умолчанию этот цикл выполняется за 76,2 мкс. Та же программа с использованием аллокатора блоков фиксированного размера выполняется за 11,6 мкс, т.е. приблизительно в 6,6 раза быстрее. Это впечатляющие показатели,

однако их следует рассматривать с некоторым подозрением. Выгоду от этой оптимизации получает только создание и уничтожение списка. Прирост общей производительности программы, которая, кроме того, обрабатывает список, будет гораздо более скромным.

Я также строил отображение с 1000 целочисленных ключей. Создание и уничтожение отображения с использованием аллокатора по умолчанию занимает 142 мкс, по сравнению с 67,4 мкс с аллокатором блоков фиксированного размера. Это более скромное улучшение показывает, что любая дополнительная деятельность программы (в данном случае балансировка дерева, в котором хранится отображение) существенно влияет на повышение производительности, достигаемое путем оптимизации аллокатора.

Аллокатор блоков фиксированного размера для строк

Класс `std::string` хранит свое содержимое в динамическом массиве элементов типа `char`. Поскольку с ростом строки выполняется перераспределение памяти для массива, он не представляется возможным кандидатом для использования простого аллокатора блоков фиксированного размера из предыдущего раздела. Но иногда можно преодолеть даже это ограничение. Разработчик, который знает, какой максимальный размер строки может быть в программе, может создать аллокатор, который всегда выделяет блоки фиксированного максимального размера. Это очень распространенная ситуация, поскольку количество приложений со строками из миллиона символов на удивление мало.

В примере 13.11 приведен частичный листинг аллокатора блоков фиксированного размера для строк.

Пример 13.11. Аллокатор блоков фиксированного размера для строк

```
template <typename T> class NewAllocator {
public:
    ...
    pointer allocate(
        size_type /* Количество */,
        typename std::allocator<void>::const_pointer = nullptr) {
        return reinterpret_cast<pointer>
                (string_memory_manager.allocate(512));
    }
    void deallocate(pointer p, size_type) {
    ::operator delete(p);
    }
};
```

Важной особенностью этого аллокатора является то, что `allocate()` полностью игнорирует запрашиваемый размер и возвращает блок фиксированного размера.

Производительность аллокатора для строк

Я протестировал аллокатор с помощью версии функции `remove_ctrl()` из примера 4.1. Эта функция неэффективно использует `std::string`, создавая множество временных строк. В примере 13.12 приведена модифицированная функция.

Пример 13.12. Версия `remove_ctrl()` с использованием аллокатора блоков фиксированного размера

```
typedef std::basic_string<
        char,
        std::char_traits<char>,
        StatelessStringAllocator<char>> fixed_block_string;

fixed_block_string remove_ctrl_fixed_block(std::string s) {
    fixed_block_string result;
    for (size_t i = 0; i<s.length(); ++i) {
        if (s[i] >= 0x20)
            result = result + s[i];
    }
    return result;
}
```

Хронометраж исходной функции `remove_ctrl()` показал время выполнения, равное 2693 мс. Усовершенствованная версия из примера 13.12 выполнилась за 1124 мс, т.е. в 2,4 раза быстрее. Это значительное повышение производительности, но, как мы видели в главе 4, "Оптимизация использования строк", другие оптимизации оказываются еще лучше.

Написание пользовательского диспетчера памяти или аллокатора может быть эффективным, но приносит меньше пользы, чем оптимизации, которые полностью удаляют вызовы диспетчера памяти.

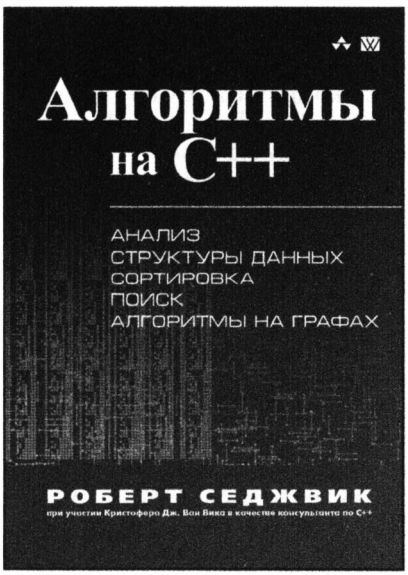
Резюме

- *Имеется множество более плодотворных мест для поиска возможностей улучшения производительности, чем диспетчер памяти.*
- *Улучшения производительности программы в целом, вызванные заменой диспетчера памяти по умолчанию, варьировались в некоторых крупных программах с открытым исходным кодом от незначительного до 30%.*
- *Диспетчеры памяти для запросов блоков одинакового размера легки в написании и эффективны в работе.*
- *Все запросы выделения памяти для экземпляров определенного класса запрашивают одинаковое количество байтов.*
- *`operator new()` может быть переопределен на уровне класса.*
- *Классы контейнеров стандартной библиотеки `std::list`, `std::map`, `std::multimap`, `std::set` и `std::multiset` создают структуру данных из множества одинаковых узлов.*
- *Контейнеры стандартной библиотеки принимают аргумент `Allocator`, обеспечивающий возможность настройки управления памятью, аналогичную предоставляемой оператором `new` для конкретного класса.*
- *Написание пользовательского диспетчера памяти или аллокатора может быть эффективным, но приносит меньше пользы, чем оптимизации, которые полностью удаляют вызовы диспетчера памяти.*

Предметный указатель

АЛГОРИТМЫ НА C++
АНАЛИЗ, СТРУКТУРЫ ДАННЫХ, СОРТИРОВКА, ПОИСК, АЛГОРИТМЫ НА ГРАФАХ

Роберт Седжвик



www.williamspublishing.com

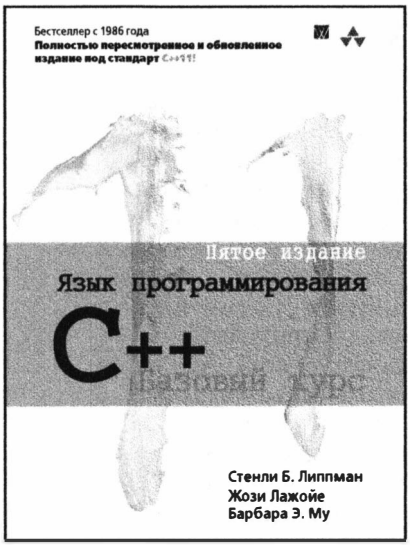
Эта классическая книга удачно сочетает в себе теорию и практику, что делает ее популярной у программистов на протяжении многих лет. Кристофер Ван Вик и Седжвик разработали новые лаконичные реализации на C++, которые естественным и наглядным образом описывают методы и могут применяться в реальных приложениях. Каждая часть содержит новые алгоритмы и реализации, усовершенствованные описания и диаграммы, а также множество новых упражнений для лучшего усвоения материала. Акцент на АТД расширяет диапазон применения программ и лучше соотносится с современными средами объектно-ориентированного программирования. Книга предназначена для широкого круга разработчиков и студентов.

ISBN 978-5-8459-2070-6

в продаже

ЯЗЫК ПРОГРАММИРОВАНИЯ C++
БАЗОВЫЙ КУРС
ПЯТОЕ ИЗДАНИЕ

СТЕНЛИ Б. ЛИППМАН,
ЖОЗИ ЛАЖОЙЕ,
БАРБАРА Э. МУ



www.williamspublishing.com

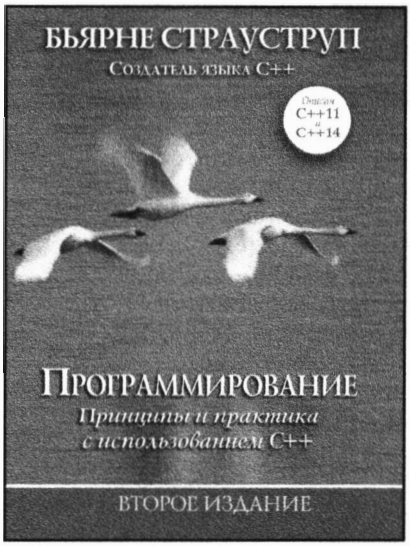
Полностью измененный и обновленный под недавно вышедший стандарт C++11, этот известный исчерпывающий вводный курс по C++ поможет вам быстро изучить язык и использовать его весьма эффективными и передовыми способами. Книга с самого начала знакомит читателя со стандартной библиотекой C++, ее наиболее популярными функциями и средствами, позволяя создавать полезные программы, еще не овладев всеми подробностями языка. Большинство примеров книги было пересмотрено так, чтобы использовать новые средства языка и продемонстрировать их наилучшие способы применения. Эта книга — не только проверенное руководство для новичков C++, она содержит также авторитетное обсуждение базовых концепций и методик языка C++ и является ценным ресурсом для опытных программистов, особенно желающих побыстрей узнать об усовершенствованиях C++11.

ISBN 978-5-8459-1839-0 **в продаже**

ПРОГРАММИРОВАНИЕ.
ПРИНЦИПЫ И ПРАКТИКА С ИСПОЛЬЗОВАНИЕМ C++

ВТОРОЕ ИЗДАНИЕ

Бьярне Страуструп



www.williamspublishing.com

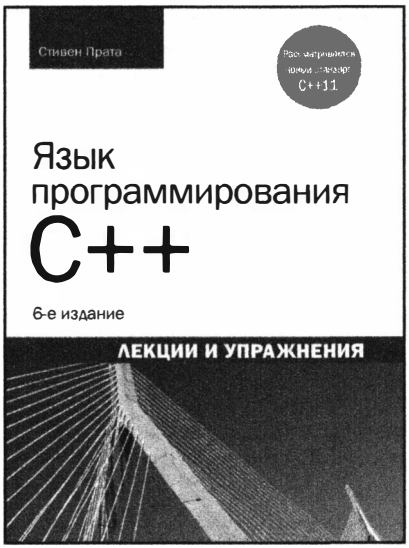
Эта книга — учебник по программированию. Несмотря на то что его автор — создатель языка C++, книга не посвящена этому языку; он играет в большей степени иллюстративную роль. Книга задумана как вводный курс по программированию с примерами программных решений на языке C++ и описывает широкий круг понятий 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 C++ на современные методы программирования.

ISBN 978-5-8459-1949-6 **в продаже**

ЯЗЫК ПРОГРАММИРОВАНИЯ C++
ЛЕКЦИИ И УПРАЖНЕНИЯ
6-Е ИЗДАНИЕ

Стивен Прата



www.williamspublishing.com

Книга представляет собой тщательно проверенный, качественно составленный полноценный учебник по одной из ключевых тем для программистов и разработчиков. Эта классическая работа по вычислительной технике обучает принципам программирования, среди которых структурированный код и нисходящее проектирование, а также использованию классов, наследования, шаблонов, исключений, лямбда-выражений, интеллектуальных указателей и семантики переноса. Автор и преподаватель Стивен Прата создал поучительное, ясное и строгое введение в C++. Фундаментальные концепции программирования излагаются вместе с подробными сведениями о языке C++. Множество коротких практических примеров иллюстрируют одну или две концепции за раз, стимулируя читателей осваивать новые темы за счет непосредственной их проверки на практике.

ISBN 978-5-8459-2048-5

в продаже

ЭФФЕКТИВНЫЙ И СОВРЕМЕННЫЙ C++

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

Скотт Мейерс

www.williamspublishing.com

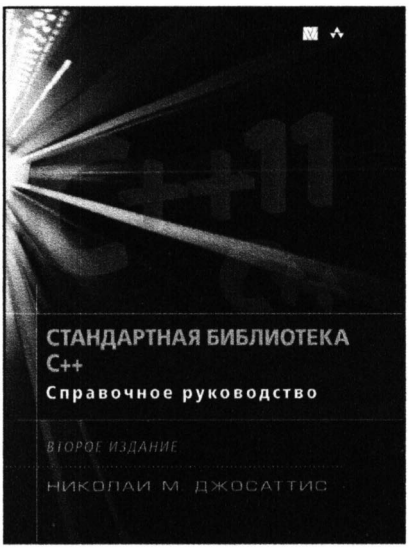
В этой книге отражен бесценный опыт ее автора как программиста на C++. Глобальные изменения в языке программирования C++, приведшие к появлению стандартов C++11/14, приводят к необходимости изучения C++ если не заново, то по крайней мере как очень сильно изменившегося языка программирования.
Пройти путь изучения и освоения современного C++ вам поможет книга Скотта Мейерса, показывающая наиболее интересные места языка и предупреждающая о возможных проблемах и ловушках.
Хотя эта книга в первую очередь предназначена для энтузиастов и профессионалов, она достойна места на полке любого программиста — как профессионала, так и зеленого новичка.

ISBN 978-5-8459-2000-3 **в продаже**

СТАНДАРТНАЯ БИБЛИОТЕКА C++: СПРАВОЧНОЕ РУКОВОДСТВО

ВТОРОЕ ИЗДАНИЕ

Николаи М. Джосаттис



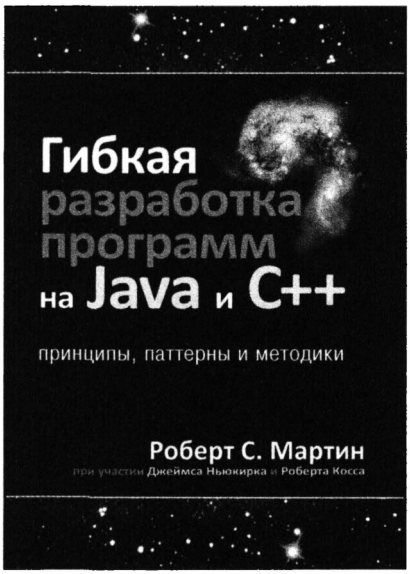
www.williamspublishing.com

В этой книге содержится полное описание библиотеки с учетом нового стандарта C++11. Читатели найдут в ней исчерпывающее описание каждого компонента библиотеки, сложные концепции и тонкости практического программирования, точные сигнатуры и определения наиболее важных классов и функций, а также многочисленные примеры работоспособных программ. Основное внимание уделяется стандартной библиотеке шаблонов (STL), в частности контейнерам, итераторам, функциональным объектам и алгоритмам. Справочник представляет собой настольную книгу всех программистов на C++.

ISBN 978-5-8459-1837-6 в продаже

ГИБКАЯ РАЗРАБОТКА ПРОГРАММ НА JAVA И C++: ПРИНЦИПЫ, ПАТТЕРНЫ И МЕТОДИКИ

Роберт Мартин, при участии Джеймса Ньюкирка и Роберта Косса



www.dialektika.com

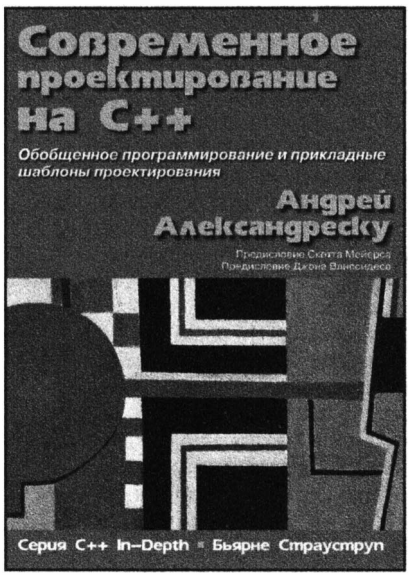
Будучи написанной разработчиками для разработчиков, книга содержит уникальный набор актуальных методов разработки программного обеспечения. В ней рассматриваются объектно-ориентированное проектирование, UML, паттерны, приемы гибкого и экстремального программирования, а также приводится детальное описание полного процесса проектирования для многократно используемых программ на C++ и Java. С применением практического подхода к решению задач в книге показано, как разрабатывать объектно-ориентированное приложение — от ранних этапов анализа и низкоуровневого проектирования до этапа реализации. Читатели ознакомятся с мыслями разработчика — здесь представлены ошибки, тупики и творческие идеи, которые возникают в процессе проектирования программного обеспечения. В книге раскрываются такие темы, как статика и динамика, принципы проектирования с использованием классов, управление сложностью, принципы проектирования с применением пакетов, анализ и проектирование, паттерны и пересечение парадигм.

ISBN 978-5-9908462-8-9

в продаже

Современное проектирование на C++

Андрей Александреску



В книге изложена новая технология программирования, представляющая собой сплав обобщенного программирования, метапрограммирования шаблонов и объектно-ориентированного программирования на C++. Настраиваемые компоненты, созданные автором, высоко подняли уровень абстракции, наделив язык C++ чертами языка спецификации проектирования, сохранив всю его мощь и выразительность.

В книге изложены способы реализации основных шаблонов проектирования. Разработанные компоненты воплощены в библиотеке Loki, которую можно загрузить с Web-страницы автора.

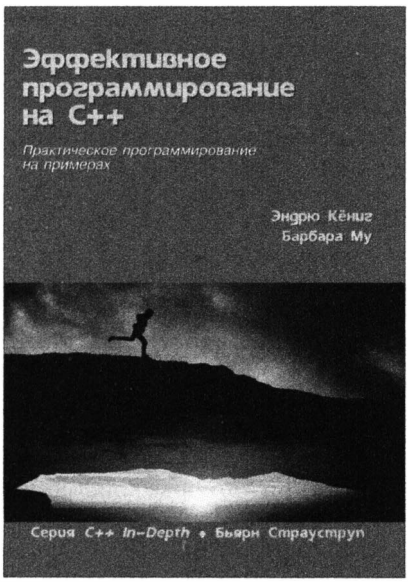
Книга предназначена для опытных программистов на C++.

www.williamspublishing.com

ISBN 978-5-8459-1940-3 **в продаже**

ЭФФЕКТИВНОЕ ПРОГРАММИРОВАНИЕ НА C++

Эндрю Кёниг,
Барбара Му



Эта книга, в первую очередь, предназначена для тех, кому хотелось бы быстро научиться писать настоящие программы на языке C++. Зачастую новички в C++ пытаются освоить язык чисто механически, даже не попытавшись узнать, как можно эффективно применить его к решению каждодневных проблем.
Цель данной книги — научить программированию на C++, а не просто изложить средства языка, поэтому она полезна не только для новичков, но и для тех, кто уже знаком с C++ и хочет использовать этот язык в более натуральном, естественном стиле.

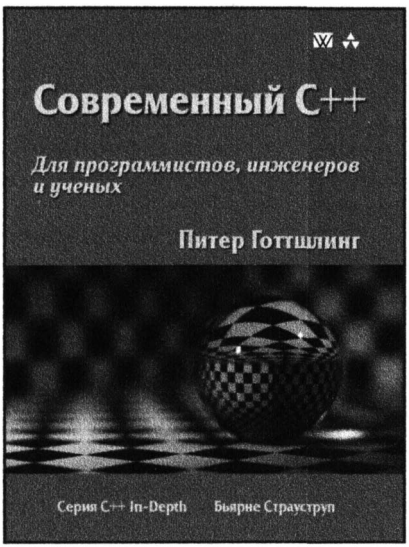
www.williamspublishing.com

ISBN 978-5-8459-2056-0 **в продаже**

СОВРЕМЕННЫЙ C++

Для программистов, инженеров и ученых

Питер Готтшлинг



www.williamspublishing.com

Перед вами книга для тех, кто нуждается в быстром освоении передовых возможностей C++. В ней описаны мощные возможности стандарта C++14, наиболее полезные для научных и инженерных приложений. Книга не предполагает у читателя наличия опыта программирования на C++ или иных языках программирования. Читатели узнают, как воспользоваться преимуществами мощных библиотек, доступных для программистов C++: стандартной библиотеки шаблонов (STL) и научных библиотек для решения задач линейной алгебры, арифметики, дифференциальных уравнений и построения графиков. На протяжении всей книги автор демонстрирует, как писать программы ясно и выразительно, используя объектно-ориентированное, обобщенное и метапрограммирование, параллелизм и процедурные технологии.

ISBN 978-5-8459-2095-9 **в продаже**

O'REILLY®

Оптимизация программ на C++

В современном быстром мире производительность программы является для клиентов таким же важным свойством, как и ее функциональные возможности. В данном практическом руководстве изложены основные принципы производительности, которые позволяют разработчикам оптимизировать программы на языке C++. Вы узнаете, как писать код, который воплощает наилучшие практики проектирования C++, работает быстрее и потребляет меньше ресурсов на любом компьютере — будь то часы, телефон, рабочая станция, суперкомпьютер или охватывающая весь земной шар сеть серверов.

Автор книги на нескольких примерах запущенного кода демонстрирует, как применять описанные принципы для постепенного улучшения существующих программ, чтобы привести их в соответствие с самыми высокими требованиями заказчика в отношении быстродействия и пропускной способности. Вы по достоинству оцените советы, приведенные в этой книге, когда услышите от коллеги: "Не может быть! Кто и как сумел это сделать?"

- Обнаружение узких мест программы с помощью профилировщика и программных таймеров
- Проведение экспериментов по измерению повышения производительности в связи с изменением кода
- Оптимизация использования динамически выделяемой памяти
- Повышение производительности циклов и функций
- Ускорение обработки строк
- Применение эффективных алгоритмов и шаблонов оптимизации
- Сильные и слабые стороны контейнеров C++
- Оптимизирующий взгляд на поиск и сортировку
- Эффективное использование потоков ввода-вывода C++
- Эффективное использование многопоточности C++

"Рог изобилия полезных советов — своевременных, иногда анекдотичных и всегда в точку. Справочник, показывающий новое лицо C++".

Джерри Тан,
старший программист
в *The Depository Trust & Clearing Corporation*

Курт Гантерот, программист более чем с 35-летним стажем, четверть века занимается разработкой высокопроизводительного кода на C++. Разрабатывал программы для Windows, Linux и встраиваемых устройств. Живет в Сиэттле, штат Вашингтон.

ДИАЛЕКТИКА
www.dialektika.com

Twitter: @oreillymedia
facebook.com/oreilly

ISBN 978-5-9908910-6-7