Федор Г. Пикус

Идиомы и паттерны проектирования в современном C++

Fedor G. Pikus

Hands-On Design Patterns with C++

Solve common C++ problems with modern design patterns and build robust applications

Packt

BIRMINGHAM - MUMBAI

Федор Г. Пикус

Идиомы и паттерны проектирования в современном C++

Применение современных паттернов проектирования к решению типичных задач на C++ для построения надежных приложений

Москва, 2020

УДК 004.4
ББК 32.973.202-018.2
П32

Пикус Ф. Г.
П32 Идиомы и паттерны проектирования в современном C++ / пер. с анг. А. А. Слинкина. – М.: ДМК Пресс, 2020. – 452 с.: ил.

ISBN 978-5-97060-786-2

В книге акцент сделан на паттерны проектирования, которые отвечают естественным нуждам программиста на C++, а также паттернам, выигрывающим от уникальных особенностей C++, в частности, обощенного программирования. Вооруженные знанием этих паттернов, вы будете тратить меньше времени на поиск решения конкретной задачи и познакомитесь с решениями, доставшимися тяжким опытом других разработчиков, их достоинствами и недостатками.

Издание предназначено программистам на C++, хорошо владеющих средствами и синтаксисом языка.

УДК 004.4
ББК 32.973.202-018.2

Authorized Russian translation of the English edition of Hands-On Design Patterns with C++ ISBN 9781788832564 © 2019 Packt Publishing.

ISBN 978-1-78883-256-4 (анг.)
ISBN 978-5-97060-786-2 (рус.)

Содержание

Об авторе

Федор Г. Пикус – главный конструктор в проектном отделе компании Mentor Graphics (подразделение Siemens), он отвечает за перспективное техническое планирование линейки продуктов Calibre, проектирование архитектуры программного обеспечения и исследование новых технологий. Ранее работал старшим инженером-программистом в Google и главным архитектором ПО в Mentor Graphics. Федор – признанный эксперт по высокопроизводительным вычислениям и C++. Он представлял свои работы на конференциях CPPCon, SD West, DesignCon и в журналах по разработке ПО, также является автором издательства O'Reilly. Федор – обладатель более 25 патентов и автор свыше 100 статей и докладов на конференциях по физике, автоматизации проектирования, электронике, проектированию ПО и C++.

Эта книга не появилась бы на свет без поддержки моей жены Галины, которая заставляла меня двигаться дальше в минуты сомнений в собственных силах. Спасибо моим сыновьям Аарону и Бенджамину за энтузиазм и моему коту Пушку который разрешал использовать свою подстилку в качестве моего ноутбука

О рецензенте

Кэйл Данлэп (Cale Dunlap) начал писать код на разных языках еще в старших классах, когда в 1999 году разработал свою первую моду для видеоигры Half-Life. В 2002 году он стал соразработчиком более-менее популярной моды Firearms для Half-Life и в конечном итоге способствовал переносу этой моды в ядро игры Firearms-Source. Получил профессиональный диплом по компьютерным информационным системам, а затем степень бакалавра по программированию игр и имитационному моделированию. С 2005 года работал программистом в небольших компаниях, участвуя в разработке различных программ, начиная с веб-приложений и заканчивая моделированием в интересах военных. В настоящее время работает старшим разработчиком в креативном агентстве Column Five в городе Оранж Каунти, штат Калифорния.

Спасибо моей невесте Элизабет, нашему сыну Мэйсону и всем остальным членам семьи, которые поддерживали меня, когда я писал свою первую рецензию на книгу

Предисловие

Еще одна книга по паттернам проектирования в C++? Зачем и почему именно сейчас? Разве написано еще не все, что можно сказать о паттернах?

Есть несколько причин для написания еще одной книги по *паттернам проектирования*, но прежде всего эта книга о C++ – не о *паттернах проектирования* в C++, а о паттернах проектирования *в C++*, и это различие в акцентах очень важно. C++ обладает всеми возможностями традиционного объектно-ориентированного языка, поэтому на нем можно реализовать все классические объектно-ориентированные паттерны, например Фабрику и Стратегию. Некоторые из них рассматриваются в этой книге. Но мощь C++ в полной мере раскрывается при использовании его средств обобщенного программирования. Напомним, что паттерн проектирования – это как часто встречающаяся задача проектирования, так и ее общепринятое решение, и обе эти грани одинаково важны. Понятно, что при появлении новых инструментов открывается возможность для нового решения. Со временем сообщество выбирает из этих решений наиболее предпочтительное, и тогда появляется на свет новый вариант старого паттерна проектирования – задача та же, но предпочтительное решение иное. Однако расширение возможностей также раздвигает границы – коль скоро в нашем распоряжении появляются новые инструменты, возникают и новые задачи проектирования.

В этой книге наше внимание будет обращено на те паттерны проектирования, для которых C++ может привнести нечто существенное хотя бы в одну из граней паттерна. С одной стороны, существуют паттерны, например Посетитель, для которых средства обобщенного программирования C++ позволяют предложить лучшее решение. Оно стало возможным благодаря новой функциональности, появившейся в последних версиях языка, от C++11 до C++17. С другой стороны, обобщенное программирование по-прежнему остается программированием (только выполнение программы производится на этапе компиляции), программирование нуждается в проектировании, а в проектировании возникают типичные проблемы, не так уж сильно отличающиеся от проблем традиционного программирования. Поэтому у многих традиционных паттернов есть близнецы или, по крайней мере, близкие родственники в обобщенном программировании, и именно они будут интересовать нас в этой книге. Характерный пример – паттерн Стратегия, который в обобщенном программировании больше известен под названием Политика (Policy). Наконец, в таком сложном языке, как C++, неизбежно присутствуют собственные идиосинкразии, которые часто приводят к специфическим для C++ проблемам, для которых имеются типичные, или *стандартные*, решения. Эти идиомы C++ не вполне заслуживают называться паттернами, но тоже рассматриваются в данной книге.

Итак, для написания этой книги было три основные причины:

- рассмотреть специфичные для C++ решения общих *классических* паттернов проектирования;
- продемонстрировать специфичные для C++ варианты паттернов, появляющиеся, когда старые задачи проектирования возникают в новом окружении обобщенного программирования;
- показать, как видоизменяются паттерны по мере эволюции языка.

Предполагаемая аудитория

Эта книга адресована программистам на C++, которые хотят почерпнуть из *коллективной мудрости сообщества* – от признанно хороших решений до часто встречающихся проблем проектирования. Можно сказать и по-другому: эта книга открывает для программиста возможность учиться на чужих ошибках.

Это не *учебник C++*; предполагается, что целевая аудитория состоит в основном из программистов, хорошо владеющих средствами и синтаксисом языка и интересующихся тем, как и почему эти средства следует использовать. Однако книга будет полезна и тем программистам, которые хотят больше узнать о C++, но предпочитают учиться на конкретных практических примерах (таким читателям я рекомендую держать под рукой какой-нибудь справочник по C++). Наконец, я надеюсь, что программисты, желающие узнать, не просто что нового появилось в версиях C++11, C++14 и C++17, а для чего эти новшества можно использовать, тоже найдут эту книгу интересной.

Структура книги

В главе 1 «Введение в наследование и полиморфизм» приводится краткое введение в объектно-ориентированные средства C++. Эта глава – не столько справочник по объектно-ориентированному программированию на C++, сколько описание аспектов языка, наиболее важных для последующих глав.

В главе 2 «Шаблоны классов и функций» кратко описываются средства обобщенного программирования в C++ – шаблоны классов, шаблоны функций и лямбда-выражения. Здесь рассмотрены конкретизации и специализации шаблонов, а также выведение аргументов и разрешение перегрузки шаблонной функции. И тут закладывается фундамент для более сложных применений шаблонов в последующих главах.

В главе 3 «Владение памятью» описываются современные идиоматические способы выражения различных видов владения памятью в C++. Это набор соглашений или идиом – компилятор не проверяет выполнение этих правил, но программистам проще понимать друг друга, если все пользуются общим словарем идиом.

В главе 4 «Обмен – от простого к нетривиальному» исследуется одна из основополагающих операций C++ – обмен двух значений. У этой операции на

удивление сложные взаимодействия с другими средствами C++, и они тоже обсуждаются здесь.

Глава 5 «Все о захвате ресурсов как инициализации» посвящена детальному разбору одной из фундаментальных концепций C++ – управлению ресурсами. Здесь вводится, пожалуй, самая популярная идиома C++, RAII (захват ресурса есть инициализация).

В главе 6 «Что такое стирание типа» обсуждается техника, которая существовала в C++ давно, но лишь с принятием стандарта C++11 завоевала популярность и приобрела важность. Механизм стирания типа позволяет писать абстрактные программы, в которых некоторые типы не упоминаются явно.

В главе 7 «SFINAE и управление разрешением перегрузки» рассматривается идиома C++ SFINAE, которая, с одной стороны, является важной составной частью механизма шаблонов в C++ и в этом смысле прозрачна для программиста, а с другой – для ее целенаправленного применения требуется ясное понимание тонкостей шаблонов.

В главе 8 «Рекурсивные шаблоны» описывается заковыристый паттерн, в котором достоинства объектно-ориентированного программирования сочетаются с гибкостью шаблонов. Объясняется идея шаблона и рассказывается, как правильно применять его для решения практических задач. Предполагается, что читатель будет готов распознать этот паттерн в последующих главах.

В главе 9 «Именованные аргументы и сцепление методов» рассматривается необычная техника вызова функций в C++ с использованием именованных аргументов вместо позиционных. Это еще одна идиома, которая неявно используется в каждой программе на C++, тогда как ее явное целенаправленное применение требует некоторых размышлений.

Глава 10 «Оптимизация локального буфера» – единственная в этой книге, целиком посвященная производительности. Производительность и эффективность – критически важные аспекты, учитываемые в каждом проектном решении, оказывающем влияние на сам язык, – ни одно языковое средство не включается в стандарт без всестороннего обсуждения с точки зрения эффективности. Поэтому неудивительно, что целая глава посвящена широко распространенной идиоме, призванной повысить производительность программ на C++.

В главе 11 «Охрана области видимости» описывается старый паттерн C++, который в последних версиях изменился почти до неузнаваемости. Речь идет о паттерне, который позволяет без труда писать безопасный относительно исключений и вообще безопасный относительно ошибок код на C++.

В главе 12 «Фабрика друзей» описывается старый паттерн, который нашел новые применения в современном C++. Он применяется для порождения функций, ассоциированных с шаблонами, например арифметических операторов для каждого порождаемого по шаблону типа.

В главе 13 «Виртуальные конструкторы и фабрики» рассматривается еще один классический объектно-ориентированный паттерн в C++ – Фабрика. По-

путно показано, как добиться видимости полиморфного поведения от конструкторов C++, хотя они и не могут быть виртуальными.

В главе 14 «Паттерн Шаблонный метод и идиома невиртуального интерфейса» описывается интересный гибрид классического объектно-ориентированного паттерна, шаблона и идиомы, специфичной только для C++. В совокупности получается паттерн, который описывает оптимальное использование виртуальных функций в C++.

В главе 15 «Одиночка – классический паттерн ООП» рассказано еще об одном классическом объектно-ориентированном паттерне, Одиночка, в контексте C++. Обсуждается, когда разумно применять этот паттерн, а когда следует его избегать. Демонстрируется несколько распространенных реализаций Одиночки.

Глава 16 «Проектирование на основе политик» посвящена одной из жемчужин проектирования в C++ – паттерну Политика (больше известному под названием Стратегия). Он применяется на этапе компиляции, т. е. является не объектно-ориентированным паттерном, а паттерном обобщенного программирования.

В главе 17 «Адаптеры и декораторы» обсуждаются два широко используемых и тесно связанных паттерна в контексте C++. Рассматривается их применение как в объектно-ориентированном, так и в обобщенном коде.

Глава 18 «Посетитель и множественная диспетчеризация» завершает галерею классических объектно-ориентированных паттернов неувядаемым паттерном Посетитель. Сначала объясняется сам паттерн, а затем рассматривается, как современный C++ позволяет реализовать его проще, надежнее и устойчивее к ошибкам.

Что необходимо для чтения этой книги

Для выполнения примеров из этой книги вам понадобится компьютер с операционной системой Windows, Linux или macOS (программы на C++ можно собирать даже на таком маленьком компьютере, как Raspberry Pi). Также понадобится современный компилятор C++, например GCC, Clang, Visual Studio или еще какой-то поддерживающий язык на уровне стандарта C++17. Необходимо также уметь работать на базовом уровне с GitHub и Git, чтобы клонировать проект, содержащий примеры.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте **www.dmkpress.com** или **www.дмк.рф** на странице с описанием соответствующей книги.

Код примеров из этой книги размещен также на сайте GitHub по адресу **https://github.com/PacktPublishing/Hands-On-Design-Patterns-with-CPP**. Все обновления выкладываются в репозиторий на GitHub.

В разделе **https://github.com/PacktPublishing/** есть и другие пакеты кода для нашего обширного каталога книг и видео. Не пропустите!

Обозначения и графические выделения

В этой книге применяется ряд соглашений о наборе текста.

`CodeInText`: код в тексте, имена таблиц базы данных, папок и файлов, расширения имен файлов, пути к файлам, данные, вводимые пользователем, и адреса в Твиттере. Например: «`overload_set` – *шаблон класса с переменным числом аргументов*».

Отдельно стоящие фрагменты кода набраны так:

```
template <typename T>
T increment(T x) { return x + 1; }
```

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте **www.dmkpress.com**, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**; при этом укажите название книги в теме письма.

Вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и MITP очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты **dmkpress@gmail.com**.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Глава 1

Введение в наследование и полиморфизм

C++ – прежде всего объектно-ориентированный язык, и объекты – фундаментальные строительные блоки программы на C++. Для описания связей и взаимодействий между различными частями программной системы, для определения и реализации интерфейсов между компонентами и для организации данных и кода применяются иерархии классов. И хотя эта книга – не учебник по C++, цель настоящей главы – сообщить читателю достаточно информации о тех касающихся классов и наследования языковых средствах, которые будут использоваться в последующих главах. Мы не будем пытаться полностью описать все возможности C++ для работы с классами, а лишь дадим введения в понятия и конструкции языка, которые нам понадобятся.

В этой главе рассматриваются следующие вопросы:

- что такое классы и какую роль они играют в C++?
- что такое иерархии классов и как в C++ используется наследование?
- что такое полиморфизм времени выполнения и как он применяется в C++?

Классы и объекты

Объектно-ориентированное программирование – это способ структурировать программу, объединив алгоритм и данные, которыми он оперирует, в единую сущность, именуемую **объектом**. Большинство объектно-ориентированных языков, в том числе и C++, основано на классах. Класс – это определение объекта, он описывает алгоритм и данные, формат объекта и связи с другими классами. Объект – это конкретный экземпляр класса, т. е. переменная. У объекта есть адрес, по которому он расположен в памяти. Класс – это тип, определенный пользователем. Вообще говоря, по определению, предоставленному классом, можно создать сколь угодно много объектов (некоторые классы ограничивают количество своих объектов, но это исключение, а не правило).

В C++ данные, составляющие класс, организуются в виде набора данных-членов, т. е. переменных различных типов. Алгоритмы реализованы в виде

функций – методов класса. Язык не требует, чтобы данные-члены класса были как-то связаны с реализацией его методов, но одним из признаков правильного проектирования является хорошая инкапсуляция данных в классах и ограниченное взаимодействие методов с внешними данными.

Идея инкапсуляции является центральной для классов в C++ – язык позволяет управлять тем, какие данные-члены и методы открыты (public), т. е. видимы извне класса, а какие закрыты (private), т. е. являются внутренними для класса. В хорошо спроектированном классе большая часть данных-членов, или даже все они, закрыты, а для выражения открытого интерфейса класса, т. е. того, что он делает, нужны только открытые методы. Этот открытый интерфейс можно уподобить контракту – проектировщик класса обещает, что класс будет предоставлять определенные возможности и операции. Закрытые данные и методы класса – часть его реализации, они могут изменяться при условии, что открытый интерфейс, т. е. заключенный нами контракт, остается неизменным. Например, следующий класс представляет рациональное число и поддерживает операцию инкремента, что и выражено в его открытом интерфейсе:

```
class Rational {
    public:
    Rational& operator+=(const Rational& rhs);
};
```

Хорошо спроектированный класс не раскрывает больше деталей реализации, чем необходимо его открытому интерфейсу. Реализация не является частью контракта, хотя документированный интерфейс может налагать на нее некоторые ограничения. Например, если мы обещаем, что числитель и знаменатель рационального числа не имеют общих множителей, то операция сложения должна включать шаг их сокращения. Для этого очень пригодилась бы закрытая функция-член, которую могли бы вызывать другие операции, но клиенту класса вызывать ее никогда не пришлось бы, потому что любое рациональное число уже сделано неприводимым до передачи вызывающей программе:

```
class Rational {
    public:
    Rational& operator+=(const Rational& rhs);
    private:
    long n_;     // числитель
    long d_;     // знаменатель
    void reduce();
};
```

```
Rational& Rational::operator+=(const Rational& rhs) {
    n_ = n_*rhs.d_ + rhs.n_*d_;
    d_ = d_*rhs.d_;
    reduce();
    return *this;
}
Rational a, b;
a += b;
```

Методам класса разрешен специальный доступ к данным-членам – они могут обращаться к закрытым данным классам. Отметим различие между классом и объектом: `operator+=()` – метод класса `Rational`, но вызывается от имени объекта `a`. Однако этот метод имеет также доступ к закрытым данным объекта `b`, поскольку `a` и `b` – объекты одного класса. Если функция-член ссылается на член класса по имени, без указания дополнительных квалификаторов, значит, она обращается к члену того объекта, от имени которого вызвана (мы можем указать это явно, написав `this->n_` и `this->d_`). Для доступа к членам другого объекта того же класса необходимо добавить указатель или ссылку на этот объект, но больше он ничем не ограничен – в отличие от случая, когда запрашивается доступ к закрытому члену из функции, не являющей членом класса.

Кстати говоря, C++ поддерживает и структуры в стиле языка C. Но в C++ структура – не просто агрегат данных-членов, она может иметь методы, модификаторы доступа `public` и `private` и все остальное, что есть в классах. С точки зрения языка, единственное различие между классом и структурой состоит в том, что все члены и методы класса по умолчанию закрыты, а в структуре они по умолчанию открыты. Если не считать этого нюанса, использовать структуры или классы – вопрос соглашения; традиционно ключевое слово `struct` применяется для описания структур в стиле C (т. е. таких, которые были бы допустимы в программе на C) и *почти* в стиле C, например структуры, в которую добавлен только конструктор. Конечно, эта граница подвижна и определяется стилем и практикой кодирования, принятыми в конкретном проекте или команде.

Помимо уже описанных методов и данных-членов, C++ поддерживает статические данные и методы. Статический метод похож на обычную функцию, не являющуюся членом, – он не вызывается от имени конкретного объекта, и единственный способ предоставить ему доступ к объекту, вне зависимости от типа, – передать объект в качестве аргумента. Однако, в отличие от свободной функции, не являющейся членом, статический метод сохраняет привилегированный доступ к закрытым данным класса.

Уже сами по себе классы – полезный способ сгруппировать алгоритмы с данными, которыми они манипулируют, и ограничить доступ к некоторым данным. Но свои богатейшие объектно-ориентированные возможности классы C++ получают благодаря наследованию и возникающим на его основе иерархиям классов.

Наследование и иерархии классов

Иерархии классов в C++ играют двоякую роль. С одной стороны, они позволяют выразить отношения между объектами, а с другой – строить сложные типы как композиции простых. То и другое достигается при помощи наследования.

Концепция наследования является центральной для использования классов и объектов в C++. Наследование позволяет определять новые классы как расширения существующих. Производный класс, наследующий базовому, со-

держит в той или иной форме все данные и алгоритмы, присутствующие в базовом классе, и добавляет свои собственные. В C++ важно различать два основных типа наследования: открытое и закрытое.

В случае открытого наследования наследуется интерфейс класса. Наследуется и его реализация – данные-члены базового класса являются также членами производного. Но именно наследование интерфейса – отличительная черта открытого наследования; это означает, что частью открытого интерфейса производного класса являются все открытые функции-члены базового.

Напомним, что открытый интерфейс подобен контракту – мы обещаем клиентам класса, что он будет поддерживать определенные операции, сохранять некоторые инварианты и подчиняться специфицированным ограничениям. Открыто наследуя базовому классу, мы связываем производный класс тем же контрактом (и, возможно, расширяем его, если решим определить дополнительные открытые интерфейсы). Поскольку производный класс соблюдает интерфейс базового класса, мы вправе использовать производный класс всюду, где допустим базовый; возможно, мы не сможем воспользоваться расширениями интерфейса (код ожидает получить базовый класс и не знает ни о каких расширениях), но интерфейс и ограничения базового класса остаются в силе.

Часто эту мысль формулируют в виде *принципа «является»* – экземпляр производного класса является также экземпляром базового класса. Однако способ интерпретации отношения *является* в C++ интуитивно не вполне очевиден. Например, является ли квадрат прямоугольником? Если да, то мы можем произвести класс `Square` от класса `Rectangle`:

```
class Rectangle {
    public:
    double Length() const { return length_; }
    double Width() const { return width_; }
    ...
    private:
    double l_;
    double w_;
};
class Square : public Rectangle {
    ...
};
```

Сразу видно, что здесь не все в порядке – в производном классе два члена, задающих измерения, тогда как в действительности нужен лишь один. Необходимо как-то гарантировать, что их значения одинаковы. Вроде бы ничего страшного – интерфейс класса `Rectangle` допускает любые положительные значения длины и ширины, а класс `Square` налагает дополнительные ограничения. Но на самом деле все гораздо хуже – контракт класса `Rectangle` разрешает пользователю задать разные измерения. Это даже можно выразить явно:

```
class Rectangle {
    public:
```

```
    void Scale(double sl, double sw) { // масштабировать измерения
        length_ *= sl;
        width_ *= sw;
    }
    ...
};
```

Итак, у нас имеется открытый метод, который позволяет изменить отношение сторон прямоугольника. Как и любой открытый метод, он наследуется производными классами, а значит, и классом `Square`. Более того, используя открытое наследование, мы утверждаем, что объект `Square` можно использовать всюду, где может встречаться объект `Rectangle`, даже не зная, что на самом деле это `Square`. Очевидно, что выполнить такое обещание невозможно – если клиент нашей иерархии классов попытается изменить отношение сторон квадрата, мы вынуждены будем ему отказать. Мы могли бы игнорировать такой вызов или сообщить об ошибке во время выполнения. Но в обоих случаях будет нарушен контракт базового класса. Выход только один – в C++ квадрат не является прямоугольником. Отметим, что и прямоугольник не является квадратом, поскольку контакт интерфейса `Square` может содержать такие гарантии, которые невозможно удовлетворить для `Rectangle`.

Точно так же пингвин не является птицей в C++, если интерфейс птицы включает умение летать. В таких случаях правильное проектирование обычно подразумевает наличие более абстрактного базового класса `Bird`, не дающего гарантий, который не способен поддержать хотя бы один производный класс (в частности, объект `Bird` не гарантирует умения летать). Затем можно создать промежуточные классы, скажем `FlyingBird` и `FlightlessBird`, которые наследуют общему базовому классу и сами служат базовыми для более конкретных классов, скажем `Eagle` или `Penguin`. Отсюда следует вынести важный урок – является ли пингвин птицей, в C++ зависит от того, как определить, что такое птица, или, в терминах языка, от открытого интерфейса класса `Bird`.

Поскольку открытое наследование подразумевает отношение *является*, язык допускает широкий спектр преобразований между ссылками и указателями на различные классы, принадлежащие одной иерархии. Прежде всего имеется неявное преобразование указателя на производный класс в указатель на базовый класс (и то же самое для ссылок):

```
class Base { ... };
class Derived : public Base { ... };
Derived* d = new Derived;
Base* b = d;   // неявное преобразование
```

Это преобразование всегда допустимо, потому что экземпляр производного класса является также экземпляром базового класса. Обратное преобразование тоже возможно, но оно должно быть явным:

```
Base* b = new Derived;    // *b в действительности имеет тип Derived
Derived* d = b;           // неявное, поэтому не компилируется
Derived* d = static_cast<Derived*>(b); // явное преобразование
```

Это преобразование не может быть неявным, потому что оно допустимо лишь тогда, когда указатель на базовый класс в действительности указывает на объект производного класса (в противном случае поведение не определено). Таким образом, программист должен с помощью статического приведения явно указать, что по какой-то причине – в силу логики программы, предварительной проверки или еще почему-то – известно, что это преобразование допустимо. Если вы не уверены в допустимости преобразования, то безопаснее попробовать его без риска неопределенного поведения; как это сделать, мы узнаем в следующем разделе.

В C++ существует также закрытое наследование. В этом случае производный класс не расширяет открытый интерфейс базового – все методы базового класса становятся закрытыми в производном. Открытый интерфейс должен быть определен производным классом с чистого листа. Не предполагается, что объект производного класса можно использовать вместо объекта базового. Единственное, что производный класс получает от базового, – детали реализации, т. е. может использовать его методы и данные-члены для реализации собственных алгоритмов. Поэтому говорят, что закрытое наследование реализует отношение *содержит* – внутри производного объекта находится экземпляр базового класса.

Следовательно, связь закрыто унаследованного класса со своим базовым похожа на связь класса с его данными-членами. Последняя техника реализации называется композицией – объект составлен из произвольного числа других объектов, которые рассматриваются как его данные-члены. В отсутствие веских причин поступить иначе композицию следует предпочесть закрытому наследованию. А когда все-таки может понадобиться закрытое наследование? Есть несколько случаев. Во-первых, бывает, что производному классу нужно раскрыть какие-то открытые функции-члены базового класса с помощью объявления `using`:

```
class Container : private std::vector<int> {
    public:
    using std::vector<int>::size;
    ...
};
```

Хоть и редко, но это бывает полезно и эквивалентно встроенной переадресующей функции:

```
class Container {
    private:
    std::vector<int> v_;
    public:
    size_t size() const { return v_.size(); }
    ...
};
```

Во-вторых, указатель или ссылку на производный объект можно преобразовать в указатель или ссылку на базовый объект, но только внутри функции-чле-

на производного класса. Опять-таки эквивалентную функциональность можно получить с помощью композиции, взяв адрес члена данных. До сих пор мы не увидели ни одной убедительной причины использовать закрытое наследование, и действительно в общем случае рекомендуют предпочесть композицию. Но вот следующие две причины более важны и могут служить достаточным обоснованием для использования закрытого наследования.

Одна из них связана с размером составных и производных объектов. Часто бывает, что базовые классы предоставляют только методы, но не данные-члены. В таких классах нет собственных данных, поэтому их объекты не занимают места в памяти. Но в C++ у них обязан быть ненулевой размер, поскольку требуется, чтобы любые два объекта или переменные имели уникальные и различающиеся адреса. Обычно если две переменные объявлены подряд, то адрес второй равен адресу первой плюс размер первой:

```
int x;    // хранится по адресу 0xffff0000, размер равен 4
int y;    // хранится по адресу 0xffff0004
```

Чтобы не обрабатывать объекты нулевого размера специальным образом, C++ назначает пустому объекту размер 1. Если такой объект используется как член класса, то он занимает по меньшей мере 1 байт (из-за требований выравнивания следующего члена в памяти это значение может оказаться больше). Эта память расходуется напрасно, она ни для чего не используется. С другой стороны, если пустой класс используется в качестве базового, то базовая часть объекта не обязана иметь ненулевой размер. Размер всего объекта производного класса должен быть больше нуля, но адреса производного объекта, его базового объекта и первого члена данных могут совпадать. Таким образом, в C++ разрешается не выделять память для пустого базового класса, пусть даже оператор `sizeof()` возвращает для этого класса 1. Хотя такая оптимизация пустого базового класса допустима, она необязательна и рассматривается именно как оптимизация. Тем не менее большинство современных компиляторов ее выполняет:

```
class Empty {
    public:
    void useful_function();
};
class Derived : private Empty {
    int i;
};    // sizeof(Derived) == 4
class Composed {
    int i;
    Empty e;
};    // sizeof(Composed) == 8
```

Если мы создаем много производных объектов, то экономия памяти вследствие оптимизации пустого базового класса может быть значительной.

Вторая причина использовать закрытое наследование связана с виртуальными функциями и объясняется в следующем разделе.

Полиморфизм и виртуальные функции

Обсуждая открытое наследование, мы упомянули, что производный объект можно использовать всюду, где ожидается базовый. Даже при таком требовании часто бывает полезно знать фактический тип объекта, т. е. тот тип, который был указан при его создании:

```
Derived d;
Base& b = d;
...
b.some_method();  // в действительности b - объект класса Derived
```

Метод `some_method()` – часть открытого интерфейса класса `Base` и должен присутствовать также в классе `Derived`. Но в пределах гибкости, допускаемой контрактом интерфейса базового класса, он может делать что-то иное. Например, выше мы уже встречали иерархию пернатых для представления птиц и, в частности, птиц, умеющих летать. Предполагается, что в классе `FlyingBird` имеется метод `fly()` и что любой конкретный класс птиц, производный от него, должен поддерживать способность к полету. Но орлы летают не так, как грифы, поэтому реализация метода `fly()` в двух производных классах, `Eagle` и `Vulture`, может быть разной. Любой код, работающий с произвольными объектами типа `FlyingBird`, может вызвать метод `fly()`, но результат будет зависеть от фактического типа объекта.

В C++ эта функциональность реализуется посредством виртуальных функций. Открытая виртуальная функция должна быть объявлена в базовом классе:

```
class FlyingBird : public Bird {
    public:
    virtual void fly(double speed, double direction) {
        ... переместить птицу в заданном направлении с указанной
        скоростью ...
    }
    ...
};
```

Производный класс наследует объявление и реализацию этой функции. Реализация должна соответствовать объявлению и соблюдать подразумеваемый им контракт. Если эта реализация отвечает нуждам производного класса, то больше ничего делать не нужно. Но при желании производный класс может переопределить реализацию из базового класса:

```
class Vulture : public FlyingBird {
    public:
    virtual void fly(double speed, double direction) {
        ... переместить птицу, но реализовать переутомление, если
        скорость слишком велика...
    }
};
```

Когда вызывается виртуальная функция, исполняющая система C++ должна определить истинный тип объекта, поскольку эта информация обычно неизвестна на этапе компиляции:

```
void hunt(FlyingBird& b) {
    b.fly(...); // может быт как Vulture, так и Eagle
    ...
};
Eagle e;
hunt(e); // Сейчас b в hunt() имеет тип Eagle, поэтому вызывается FlyingBird::fly()
Vulture v;
hunt(v); // Сейчас b в hunt() имеет тип Vulture, поэтому вызывается Vulture::fly()
```

Техника программирования, при которой некий код работает с произвольным числом базовых объектов и вызывает одни и те же методы, но результат зависит от фактических типов этих объектов, называется полиморфизмом времени выполнения, а объекты, поддерживающие эту технику, – **полиморфными**. В C++ полиморфный объект должен иметь хотя бы одну виртуальную функцию, и только те части его интерфейса, в которых используются виртуальные функции, являются полиморфными.

Из этого объяснения должно быть ясно, что объявления виртуальной функции и любого ее переопределенного варианта должны быть одинаковы. Действительно, программист вызывает функцию от имени базового объекта, но исполняется функция, реализованная в производном классе. Это возможно, только если типы аргументов и возвращаемого значения в точности совпадают (есть, правда, одно исключение – если виртуальная функция в базовом классе возвращает указатель или ссылку на объект некоторого типа, то переопределенная функция может возвращать указатель или ссылку на объект производного от него типа).

Распространенный частный случай полиморфной иерархии – когда в базовом классе нет хорошей реализации виртуальной функции по умолчанию. Например, все летающие птицы умеют летать, но летают они с разной скоростью, поэтому нет причины выбрать какую-то одну скорость в качестве значения по умолчанию. В C++ мы можем отказаться предоставлять реализацию виртуальной функции в базовом классе. Такие функции называются чисто виртуальными, а базовый класс, содержащий хотя бы одну чисто виртуальную функцию, называется абстрактным:

```
class FlyingBirt {
public:
    virtual void fly(...) = 0; // чисто виртуальная функция
};
```

Абстрактный базовый класс определяет только интерфейс; реализовать его – задача конкретного производного класса. Если базовый класс содержит чисто виртуальную функцию, то любой производный от него класс, экземпляры которого создает программа, должен предоставить ее реализацию. Иными словами, объект абстрактного базового класса создать нельзя. Однако в про-

грамме может быть определен указатель или ссылка на объект базового класса. В действительности он указывает на объект производного класса, но оперировать им можно через интерфейс базового.

Уместно сделать несколько замечаний о синтаксисе C++. При переопределении виртуальной функции повторять ключевое слово `virtual` необязательно. Если в базовом классе определена виртуальная функция с таким же именем и типами аргументов, то функция в производном классе всегда будет виртуальной и будет переопределять функцию из базового класса. Отметим, что если типы аргументов различаются, то функция в производном классе ничего не переопределяет, а маскирует имя функции из базового класса. Это может приводить к тонким ошибкам, когда программист собирался переопределить функцию из базового класса, но неправильно скопировал ее объявление:

```
class Eagle : public FlyingBird {
    public:
    virtual void fly(int speed, double direction);
};
```

Здесь типы аргументов немного различаются. Функция `Eagle::fly()` также виртуальная, но не переопределяет `FlyingBird::fly()`. Если последняя является чисто виртуальной функцией, то компилятор выдаст ошибку, потому что любая чисто виртуальная функция должна быть реализована в производном классе. Но если у `FlyingBird::fly()` имеется реализация по умолчанию, то компилятор не сочтет это ошибкой. В C++11 имеется очень полезное средство, которое упрощает поиск подобных ошибок, – любую функцию, которая, по задумке программиста, должна переопределять виртуальную функцию из базового класса, можно объявить с ключевым словом `override`:

```
class Eagle : public FlyingBird {
    public:
    void fly(int speed, double direction) override;
};
```

Ключевое слово `virtual` по-прежнему необязательно, но если в классе `FlyingBird` нет виртуальной функции, которую можно было бы переопределить указанным образом, то код не откомпилируется.

Чаще всего виртуальные функции используются в иерархиях с открытым наследованием – поскольку любой объект производного класса является также объектом базового класса (отношение *является*), программа часто может оперировать коллекцией производных объектов так, будто все они имеют один тип, а переопределенные виртуальные функции гарантируют, что каждый объект будет обрабатываться, как должно:

```
void MakeLoudBoom(std::vector<FlyingBird*> birds) {
    for (auto bird : birds) {
        bird->fly(...); // действие одно, результаты разные
    }
}
```

Но виртуальные функции можно использовать и совместно с закрытым наследованием. Такое употребление не столь прямолинейно (и встречается гораздо реже) – в конце концов, к закрыто унаследованному объекту нельзя обратиться по указателю на базовый класс (закрытый базовый класс иногда называют *недоступной базой*, и попытка привести указатель на производный класс к типу указателя на базовый заканчивается неудачей). Однако существует один контекст, в котором такое приведение разрешено, а именно внутри функции-члена производного класса. Ниже показан способ вызвать виртуальную функцию из закрыто унаследованного базового класса:

```
class Base {
    public:
    virtual void f() { std::cout << "Base::f()" << std::endl; }
    void g() { f(); }
};
class Derived : private Base {
    virtual void f() { std::cout << "Derived::f()" << std::endl; }
    void h() { g(); }
};
Derived d;
d.h(); // печатается "Derived::f()"
```

Любой открытый метод класса `Base` становится закрытым в классе `Derived`, поэтому напрямую мы его вызвать не можем. Но его можно вызвать из другого метода класса `Derived`, например из открытого метода `h()`. Затем мы можем вызвать `f()` напрямую из `h()`, но это ничего не доказывает – было бы неудивительно, если бы `Derived::h()` вызывала `Derived::f()`. Однако же мы вызываем функцию `Base::f()`, унаследованную от класса `Base`. Внутри этой функции мы находимся в классе `Base` – ее тело могло быть написано и откомпилировано задолго до того, как был реализован класс `Derived`. И тем не менее в этом контексте переопределение виртуальной функции работает правильно – вызывается именно `Derived::f()`, как если бы наследование было открытым.

В предыдущем разделе мы рекомендовали использовать композицию, а не закрытое наследование, если нет веских причин поступить иначе. Но толком реализовать такое поведение с помощью композиции невозможно, поэтому если необходима функциональность виртуальной функции, то ничего не остается, как прибегнуть к закрытому наследованию.

Класс, обладающий виртуальными методами, должен записывать свой тип в любой объект, иначе во время выполнения было бы невозможно узнать, каким был тип объекта в момент конструирования, после того как указатель на него преобразован в указатель на базовый класс и информация об исходном типе потеряна. Такое хранение информации о типе обходится не даром, оно занимает место, поэтому полиморфный объект всегда больше объекта с такими же данными-членами, но без виртуальных методов (обычно на размер одного указателя). Дополнительный размер не зависит от количества виртуальных функций в классе; если есть хотя бы одна, то информацию о типе надо хранить. Вспомним теперь, что указатель на базовый класс можно преобра-

зовать в указатель на производный, но только если известен правильный тип производного класса. Статическое приведение не позволяет проверить, правильны ли наши знания. Для неполиморфных классов (классов без виртуальных функций) лучшего способа не существует; если исходный тип потерян, то восстановить его невозможно. Но для полиморфных объектов тип хранится в самом объекте, поэтому должен быть способ воспользоваться этой информацией для проверки правильности наших предположений об истинном типе производного объекта. И такой способ есть – оператор динамического приведения `dynamic_cast`:

```
class Base { ... };
class Derived : public Base { ... };
Base* b1 = new Derived;   // действительно производный
Base* b2 = new Base;      // непроизводный
Derived* d1 = dynamic_cast<Derived*>(b1); // правильно
Derived* d2 = dynamic_cast<Derived*>(b2); // d2 == nullptr
```

Динамическое приведение не сообщает нам истинный тип объекта, но позволяет задать вопрос «*Правда ли, что истинный тип* `Derived`?». Если наша догадка верна, то приведение завершается успешно и возвращается указатель на производный объект. Если же истинный тип иной, то приведение не проходит, и мы получаем нулевой указатель. Динамическое приведение работает и для ссылок, но с одним отличием – нет такой вещи, как *нулевая ссылка*. Функция, возвращающая ссылку, должна вернуть ссылку на какой-то существующий объект. Но оператор динамического приведения не может вернуть ссылку на объект, если запрошенный тип не совпадает с истинным. Единственная альтернатива – возбудить исключение.

До сих пор мы ограничивались только одним базовым классом. Действительно, об иерархиях классов гораздо проще рассуждать, представляя их в виде деревьев, в которых базовый класс расположен в корне, а классы, производные от него, – на ветвях. Но язык C++ не налагает такого ограничения. Далее мы расскажем о наследовании от нескольких базовых классов.

Множественное наследование

В C++ класс может наследовать нескольким базовым классам. Возвращаясь к птицам, заметим, что летающие птицы имеют много общего друг с другом, но кое-что и с другими летающими животными, а именно способность летать. Поскольку способность к полету присуща не только птицам, имеет смысл перенести данные и алгоритмы, относящиеся к обработке полета, в отдельный базовый класс. Но при этом нельзя отрицать, что орел – птица. Мы могли бы выразить эту связь, используя два базовых класса в объявлении класса `Eagle`:

```
class Eagle : public Bird, public FlyingAnimal { ... };
```

В данном случае наследование обоим базовым классам открытое, т. е. производный класс наследует оба интерфейса и должен соблюдать оба контракта.

Что произойдет, если в обоих интерфейсах определен метод с одним и тем же именем? Если этот метод не виртуальный, то попытка вызвать его в производном классе неоднозначна, и программа не компилируется. Если же метод виртуальный и переопределен в производном классе, то неоднозначности не возникает, поскольку вызывается метод производного класса. Кроме того, `Eagle` теперь является одновременно `Bird` и `FlyingAnimal`:

```
Eagle* e = new Eagle;
Bird* b = e;
FlyingAnimal* f = e;
```

Оба преобразования указателя на производный класс в указатель на базовый класс допустимы. Обратные преобразования следует выполнять явно, применяя статическое или динамическое приведение. Есть еще одно интересное преобразование: если имеется указатель на объект класса `FlyingAnimal`, который является также объектом класса `Bird`, то можно ли преобразовать указатель из одного типа в другой? Да, можно, посредством динамического приведения:

```
Bird* b = new Eagle; // также FlyingAnimal
FlyingAnimal* f = dynamic_cast<FlyingAnimal*>(b);
```

При использовании в таком контексте динамическое приведение иногда называют **перекрестным приведением** (cross-cast) – приведение производится не вверх и не вниз по иерархии (между производным и базовым классом), а поперек иерархии – между классами, находящимися в разных ветвях дерева.

Множественное наследование в C++ часто не любят и поносят. Большинство таких рекомендаций устарело и восходит к тому времени, когда компиляторы реализовывали множественное наследование плохо и неэффективно. Для современных компиляторов тут нет никакой проблемы. Нередко можно услышать, что из-за множественного наследования иерархию классов труднее понять и обсуждать. Пожалуй, было бы точнее сказать, что труднее спроектировать хорошую иерархию множественного наследования, которая точно отражает связи между различными свойствами, и что плохо спроектированную иерархию действительно трудно понять.

Все эти соображения в основном относятся к иерархиям с открытым наследованием. Множественное наследование может быть и закрытым. Но причин использовать закрытое множественное наследование вместо композиции еще меньше, чем в случае одиночного наследования. Впрочем, оптимизация пустого базового класса применима и тогда, когда пустых базовых классов несколько, поэтому она остается основанием для использования закрытого наследования:

```
class Empty1 {};
class Empty2 {};
class Derived : private Empty1, private Empty2 {
    int i;
}; // sizeof(Derived) == 4
```

```
class Composed {
    int i;
    Empty1 e1;
    Empty2 e2;
}; // sizeof(Composed) == 8
```

Множественное наследование может оказаться особенно эффективным, если базовый класс представляет систему, объединяющую несколько не связанных между собой и непересекающихся атрибутов. Мы столкнемся с такими случаями в этой книге, когда будем изучать различные паттерны проектирования и их представления в C++.

Резюме

Хотя эта глава ни в коем случае не является справочником по классам и объектам, в ней все же вводятся и объясняются концепции, которыми читатель должен владеть, если хочет понять примеры и пояснения в последующих главах. Поскольку нас интересует представление паттернов проектирования в языке C++, эта глава была посвящена правильному применению классов и наследования. Особое внимание мы уделили тому, какие отношения выражаются с помощью различных средств C++, т. к. именно эти средства мы будем использовать для выражения связей и взаимодействий между различными компонентами, образующими паттерн проектирования.

В следующей главе мы точно так же рассмотрим шаблоны C++, без которых невозможно понять основной материал книги.

Вопросы

- В чем важность объектов в C++?
- Какое отношение выражает открытое наследование?
- Какое отношение выражает закрытое наследование?
- Что такое полиморфный объект?

Для дальнейшего чтения

Для получения дополнительной информации о материале этой главы обратитесь к следующим книгам:

- **C++ Fundamentals**: https://www.packtpub.com/application-development/cfundamentals;
- **C++ Data Structures and Algorithms**: https://www.packtpub.com/application-development/c-data-structures-and-algorithms;
- **Mastering C++ Programming**: https://www.packtpub.com/application-development/mastering-c-programming;
- **Beginning C++ Programming**: https://www.packtpub.com/application-development/beginning-c-programming.

Глава 2

Шаблоны классов и функций

Средства программирования шаблонов в C++ – большая и сложная тема, и есть немало книг, посвященных исключительно этим средствам. В этой книге мы будем использовать многие продвинутые возможности обобщенного программирования на C++. Как же подготовить читателя к пониманию этих языковых конструкций, которые будут встречаться сплошь и рядом? В этой главе мы примем неформальный подход – вместо точных определений продемонстрируем использование шаблонов на примерах и объясним, для чего предназначены различные языковые средства. Если вы почувствуете, что знаний не хватает, советуем углубить свое понимание, прочитав одну или несколько книг, целиком посвященных синтаксису и семантике языка C++. Разумеется, читателя, жаждущего получить точное формальное описание, мы направляем к стандарту C++ или к какому-нибудь полному справочнику.

В этой главе рассматриваются следующие вопросы:

- шаблоны в C++;
- шаблоны классов и функций;
- конкретизации шаблонов;
- специализации шаблонов;
- перегрузка шаблонных функций;
- шаблоны с переменным числом аргументов;
- лямбда-выражения.

Шаблоны в C++

Одно из самых больших достоинств C++ – поддержка обобщенного программирования. В обобщенном программировании алгоритмы и структуры данных записываются в терминах обобщенных типов, которые будут заданы позже. Это дает возможность реализовать функцию или класс один раз, а затем конкретизировать ее для разных типов. Шаблоны – это средство C++, позволяющее определять классы и функции для обобщенных типов. C++ поддерживает три вида шаблонов: функций, классов и переменных.

Шаблоны функций

Шаблоны функций – это обобщенные функции. В отличие от обычной функции, для шаблонной функции не объявляются типы аргументов. Вместо этого в роли типов выступают параметры шаблона:

```
template <typename T>
T increment(T x) { return x + 1; }
```

Эта шаблонная функция используется для увеличения на единицу значения любого типа, для которого сложение с единицей – допустимая операция:

```
increment(5);     // T типа int, возвращается 6
increment(4.2);   // T типа double, возвращается 5.2
char c[10];
increment(c);     // T типа char*, возвращается &c[1]
```

У большинства шаблонных функций имеются ограничения на типы, которые могут быть указаны в параметрах шаблона. Например, наша функция `increment()` требует, чтобы для типа `x` было допустимо выражение `x + 1`. Если это не так, то попытка конкретизировать шаблон завершится неудачно, с довольно многословным сообщением компилятора.

Шаблонами могут быть как свободные функции, так и функции-члены класса, но не виртуальные функции. Обобщенные типы можно использовать не только для объявления параметров функции, но и для объявления любых переменных в теле функции:

```
template <typename T>
T sum(T from, T to, T step) {
    T res = from;
    while ((from += step) < to) { res += from; }
    return res;
}
```

Позже мы еще встретимся с шаблонами функций, а пока поговорим о шаблонах классов.

Шаблоны классов

Шаблоны классов – это классы, в которых используются обобщенные типы, обычно в объявлениях данных-членов, но иногда и в объявлениях методов и их локальных переменных:

```
template <typename T>
class ArrayOf2 {
    public:
    T& operator[](size_t i) { return a_[i]; }
    const T& operator[](size_t i) const { return a_[i]; }
    T sum() const { return a_[0] + a_[1]; }
    private:
    T a_[2];
};
```

Этот класс реализован один раз, после чего его можно использовать для определения массива из двух элементов любого типа:

```
ArrayOf2<int> i;
i[0] = 1; i[1] = 5;
std::cout << i.sum(); // 6

ArrayOf2<double> x;
x[0] = -3.5; x[1] = 4;
std::cout << x.sum(); // 0.5

ArrayOf2<char*> c;
char s[] = "Hello";
c[0] = s; c[1] = s + 2;
```

Особое внимание обратите на последний пример – быть может, вы думали, что шаблон `ArrayOf2` нельзя конкретизировать типом `char*`, ведь в этом шаблоне есть метод `sum()`, который не должен компилироваться, если тип `a_[0]` и `a_[1]` – указатель. Однако пример компилируется, поскольку метод шаблона класса не обязан быть допустимым, до тех пор пока мы не попытаемся его использовать. Если мы не будем вызывать `c.sum()`, то никогда и не обнаружится, что этот метод не компилируется, и программа может считаться корректной.

Шаблоны переменных

И последний вид шаблона в C++ – шаблон переменной, появившийся в версии C++14. Такой шаблон позволяет определять переменную обобщенного типа:

```
template <typename T>
constexpr T pi =
T(3.1415926535897932384626433832795028841971693993751058209749445923078 1L);
pi<float>;   // 3.141592
pi<double>;  // 3.141592653589793
```

Шаблоны переменных в этой книге не используются, поэтому говорить о них мы больше не будем.

Параметры шаблонов, не являющиеся типами

Обычно в роли параметров шаблонов выступают типы, но C++ допускает также несколько видов параметров, не являющихся типами:

```
template <typename T, size_t N> class Array {
    public:
    T& operator[](size_t i) {
        if (i >= N) throw std::out_of_range("Bad index");
        return data_[i];
    private:
    T data_[N];
};
Array<int, 5> a;     // правильно
cin >> a[0];
Array<int, a[0]> b;  // ошибка
```

Это шаблон с двумя параметрами, первый является типом, второй – нет. Второй параметр – значение типа `size_t`, определяющее размер массива; преимущество такого шаблона над встроенным массивом в стиле C заключается в возможности выполнить проверку выхода за границу. В стандартной библиотеке C++ имеется шаблон класса `std::array`, в реальных программах следует использовать именно его, а не изобретать собственный тип массива, но это простой пример, в котором легко разобраться.

Параметры-нетипы, используемые для конкретизации шаблона, должны быть константами, известными во время компиляции, т. е. `constexpr`-значениями; последняя строка в предыдущем примере неправильна, потому что значение `a[0]` неизвестно, пока программа не прочтет его во время выполнения. Числовые параметры шаблонов когда-то были очень популярны в C++, потому что позволяют выполнять сложные вычисления на этапе компиляции, но в последних версиях стандарта того же эффекта можно добиться с помощью `constexpr`-функций, которые читать гораздо легче.

Стоит также упомянуть второй вид параметров-нетипов – *шаблонные параметры шаблона*, т. е. параметры шаблона, которые сами являются шаблонами. Они понадобятся нам в последующих главах. При подстановке такого параметра шаблона указывается не имя класса, а имя целого шаблона. Ниже приведен шаблон функции с двумя *шаблонными параметрами шаблона*:

```
template <template <typename> class Out_container,
          template <typename> class In_container,
          typename T>
Out_container<T> resequence(const In_container<T>& in_container) {
    Out_container<T> out_container;
    for (auto x : in_container) {
        out_container.push_back(x);
    }
    return out_container;
}
```

Эта функция принимает произвольный контейнер и возвращает другой контейнер, описываемый иным шаблоном, но конкретизируемый таким же типом. При этом значения копируются из первого контейнера во второй:

```
std::vector<int> v { 1, 2, 3, 4, 5 };
auto d = resequence<std::deque>(v); // deque с элементами 1, 2, 3, 4, 5
```

Шаблоны – это в некотором роде рецепт порождения кода. Далее мы увидим, как по этим рецептам приготовить фактический, допускающий выполнение код.

Конкретизация шаблона

Имя шаблона – это не тип, его нельзя использовать для объявления переменной или вызова функции. Чтобы создать тип или функцию, шаблон необходи-

мо конкретизировать. Как правило, шаблоны конкретизируются неявно в момент использования. Снова начнем с шаблонов функций.

Шаблоны функций

Чтобы воспользоваться шаблоном функции для порождения функции, мы должны указать типы, подставляемые вместо всех параметров-типов шаблона. Типы можно задать явно:

```
template <typename T>
T half(T x) { return x/2; }
int i = half<int>(5);
```

При этом шаблон функции `half` конкретизируется типом `int`. Тип задан явно, но мы могли бы вызвать функцию с аргументом другого типа, при условии что он допускает преобразование в запрошенный тип:

```
double x = half<double>(5);
```

Здесь аргумент имеет тип `int`, но, поскольку мы воспользовались конкретизацией `half<double>`, возвращаемое значение имеет тип `double`. Целое значение `5` неявно преобразуется в `double`.

Хотя любой шаблон функции можно конкретизировать, задав все параметры-типы, так поступают редко. Чаще всего при конкретизации шаблонов функций применяется автоматическое выведение типов. Рассмотрим пример:

```
auto x = half(8);    // int
auto y = half(1.5);  // double
```

Тип шаблона можно вывести, зная лишь аргументы шаблонной функции, – компилятор попытается подобрать тип параметра `T`, так чтобы он соответствовал типу аргумента функции. В нашем случае шаблон функции имеет аргумент `x` типа `T`. При любом вызове этой функции необходимо задать какое-то значение этого аргумента, и это значение должно иметь некоторый тип. Компилятор заключает, что `T` должен совпадать с этим типом. В первом из показанных выше вызовов аргумент равен `5`, т. е. имеет тип `int`. Нет ничего лучше, чем предположить, что `T` в данной конкретизации шаблон должен быть равен `int`. Аналогично для второго вызова можно заключить, что `T` должен быть равен `double`.

После выведения компилятор производит подстановку типа: все прочие упоминания `T` заменяются выведенным типом; в нашем случае `T` встречается еще в одном месте – в качестве типа возвращаемого значения.

Выведение аргументов шаблона широко применяется, когда тип нелегко определить вручную:

```
long x = ...;
unsigned int y = ...;
auto x = half(y + z);
```

Здесь компилятор заключает, что тип `T` должен быть таким же, как тип выражения `y + z` is (это `long`, но благодаря выведению аргументов шаблона мы

не обязаны указывать это явно, а выведенный тип будет *следовать* за типом аргумента, даже если впоследствии мы изменим типы y и z). Рассмотрим пример:

```
template <typename U> auto f(U);
half(f(5));
```

Мы выводим тип U, так чтобы он соответствовал типу значения, возвращаемого шаблонной функцией f(), когда она получает аргумент типа int (конечно, перед тем как вызывать шаблонную функцию f(), нужно предоставить ее определение, но нам необязательно копаться в заголовочных файлах, где определена f(), поскольку компилятор сам выведет правильный тип).

Вывести можно только типы, используемые при объявлении аргументов функции. Никто не требует, чтобы все параметры-типы шаблона присутствовали в списке аргументов, но те параметры, которые вывести невозможно, должны быть указаны явно при вызове:

```
template <typename U, typename V>
U half(V x) { return x/2; }
auto y = half<double>(8);
```

Здесь первый параметр-тип шаблона указан явно, т. е. U – это double, а V оказывается равным int в результате выведения.

Иногда компилятор не может вывести параметры-типы шаблонов, даже если они встречаются в объявлениях аргументов:

```
template <typename T>
T Max(T x, T y) { return (x > y) ? x : y; }
auto x = Max(7L, 11);
```

Здесь мы можем из первого аргумента вывести, что T должен быть равен long, но из второго аргумента следует, что T должен быть равен int. Программисты часто удивляются, почему в этом случае не выводится тип long, ведь если всюду подставить long вместо T, то второй аргумент будет неявно преобразован, и функция откомпилируется. Так почему же не выводится *более широкий* тип? Потому что компилятор не пытается найти тип, для которого возможны все преобразования аргументов, поскольку обычно таких типов несколько. В нашем примере T мог бы быть равен double или unsigned long, и функция по-прежнему осталась бы корректной. Если тип можно вывести, исходя из нескольких аргументов, то результаты любого выведения должны быть одинаковы, иначе конкретизация шаблона считается неоднозначной.

Выведение типа не всегда сводится к использованию типа аргумента в качестве параметра-типа. В объявлении аргумента может быть указан тип более сложный, чем сам параметр-тип:

```
template <typename T>
T decrement(T* p) { return --(*p); }
int i = 7;
decrement(&i);   // i == 6
```

Здесь типом аргумента является *указатель* на `int`, но результатом выведения типа `T` будет `int`. Выведение типов может быть сколь угодно сложным процессом, лишь бы оно давало однозначный результат:

```
template <typename T>
T first(const std::vector<T>& v) { return v[0]; }
std::vector<int> v{11, 25, 67};
first(v);    // T совпадает с int, возвращается 11
```

Здесь аргументом является конкретизация другого шаблона, `std::vector`, и мы должны вывести тип параметра шаблона из типа, который был использован при создании этой конкретизации вектора.

Как мы видели, если тип можно вывести более чем из одного аргумента функции, то результаты всех выведений должны быть одинаковы. С другой стороны, один аргумент можно использовать для выведения нескольких типов:

```
template <typename U, typename V>
std::pair<V, U> swap12(const std::pair<U, V>& x) {
    return std::pair<V, U>(x.second, x.first);
}
swap12(std::make_pair(7, 4.2);   // пара 4.2, 7
```

Здесь мы выводим два типа, `U` и `V`, из одного аргумента, а затем используем эти два типа для образования нового типа, `std::pair<V, U>`. Этот пример излишне многословен, мы могли бы воспользоваться еще несколькими средствами C++, чтобы сделать его более компактным и простым для сопровождения. Во-первых, в стандарте уже есть функция, которая выводит типы аргументов и использует их для объявления пары, и мы даже ей пользовались – `std::make_pair()`. Во-вторых, тип возвращаемого функцией значения можно вывести из выражения в предложении `return` (эта возможность появилась в C++14). Правила такого выведения похожи на правила выведения типов аргументов шаблона. После этих упрощений пример принимает такой вид:

```
template <typename U, typename V>
auto swap12(const std::pair<U, V>& x) {
    return std::make_pair(x.second, x.first);
}
```

Заметим, что типы `U` и `V` больше не используются явно. Но эта функция все равно должна быть шаблоном, поскольку она оперирует обобщенным типом, а именно парой двух типов, неизвестных до момента конкретизации функции. Однако мы могли бы использовать всего один параметр шаблона, который будет обозначать тип аргумента:

```
template <typename T>
auto swap12(const T& x) {
    return std::make_pair(x.second, x.first);
```

Между этими двумя вариантами есть важное различие – для второго шаблона функции выведение типа успешно завершается при любом вызове с одним аргументом, вне зависимости от типа этого аргумента. Если тип аргумента не

std::pair и, вообще, если аргумент не является ни классом, ни структурой либо не содержит данных-членов first и second, то выведение все равно будет успешным, но подстановка типа завершится неудачно. С другой стороны, для первой версии даже не рассматриваются аргументы, не являющиеся парой каких-то типов. Но для любого аргумента типа std::pair пара типов выводится, и при подстановке не должно возникнуть проблем.

Шаблоны функций-членов очень похожи на шаблоны свободных функций, и их аргументы выводятся аналогично. Шаблоны функций-членов могут использоваться в классах и в шаблонах классов, это тема следующего раздела.

Шаблоны классов

Конкретизация шаблонов классов похожа на конкретизацию шаблонов функций – при использовании шаблона для создания типа шаблон неявно конкретизируется. Чтобы воспользоваться шаблоном класса, необходимо задать типы параметров шаблона:

```
template <typename N, typename D>
class Ratio {
    public:
    Ratio() : num_(), denom_() {}
    Ratio(const N& num, const D& denom) : num_(num), denom_(denom) {}
    explicit operator double() const {
        return double(num_)/double(denom_);
    }
    private:
    N num_;
    D denom_;
};
Ratio<int, double> r;
```

Определение переменной r1 неявно конкретизирует шаблон класса Ratio типами int и double. Также конкретизируется конструктор этого класса по умолчанию. Второй конструктор в коде не используется и не конкретизируется. Именно эта особенность шаблонов классов – при конкретизации шаблона конкретизируются все данные-члены, но методы не конкретизируются, если не используются, – позволяет писать шаблоны классов, в которых для некоторых типов компилируются не все методы. Если мы воспользуемся вторым конструктором для инициализации значений Ratio, то этот конструктор будет конкретизирован и должен быть допустим для заданных типов:

```
Ratio<int, double> r(5, 0.1);
```

В C++17 эти конструкторы можно использовать для выведения типов шаблона класса по аргументам конструктора:

```
Ratio r(5, 0.1);
```

Разумеется, это работает, только если у конструктора достаточно аргументов для выведения типов. Например, объект Ratio, конструируемый по умолчанию,

должен конкретизироваться явно заданными типами, иначе их просто невозможно будет вывести. До выхода стандарта C++17 для конструирования объектов, тип которых можно было вывести из аргументов, часто применялся вспомогательный шаблон функции. По аналогии с рассмотренным выше шаблоном `std::make_pair()` можно было бы написать функцию `make_ratio`, делающую то же самое, что делает механизм выведения типов из аргументов конструктора в C++17:

```
template <typename N, typename D>
Ratio<N, D> make_ratio(const N& num, const D& denom) {
    return { num, denom };
}
auto r(make_ratio(5, 0.1));
```

Если возможно, следует предпочесть способ выведения аргументов шаблона, появившийся в C++17: он не требует написания отдельной функции, которая, по существу, дублирует конструктор класса, и не нуждается в дополнительном вызове копирующего или перемещающего конструктора для инициализации объекта (хотя на практике большинство компиляторов все равно убирает этот вызов в процессе оптимизации).

Если шаблон используется для порождения типа, то он конкретизируется неявно. Но шаблоны функций и классов можно конкретизировать и явно. При этом шаблон конкретизируется, но не используется:

```
template class Ratio<long, long>;
template Ratio<long, long> make_ratio(const long&, const long&);
```

Явная конкретизация редко бывает необходима и в этой книге больше не встретится.

До сих пор мы видели, как шаблоны классов позволяют объявлять обобщенные классы, т. е. классы, которые можно конкретизировать разными типами. Пока что все такие классы отличались только типами, а код для них генерировался одинаковый. Но это не всегда желательно, иногда разные типы следует обрабатывать по-разному.

Предположим, к примеру, что требуется представить не только отношение двух чисел, хранящихся в самом объекте `Ratio`, но и отношение двух чисел, хранящихся где-то в другом месте, – с помощью объекта `Ratio`, содержащего указатели на них. Очевидно, что если в объекте хранятся указатели на числитель и знаменатель, то некоторые методы объекта `Ratio`, в т. ч. оператор преобразования в тип `double`, необходимо реализовывать по-другому. В C++ для этого служит специализация шаблона, которую мы и рассмотрим ниже.

Специализация шаблона

Специализация шаблона позволяет генерировать другой код для некоторых типов, т. е. при подстановке различных типов генерируется не одинаковый, а совершенно разный код. В C++ есть два вида специализации шаблона: явная, или полная, и частичная. Начнем с первой.

Явная специализация

В случае явной специализации определяется специальная версия шаблона для определенного набора типов. При этом все обобщенные типы заменяются конкретными. Поскольку явная специализация не является обобщенным классом или функцией, то она и не должна конкретизироваться впоследствии. По этой причине ее иногда называют **полной специализацией**. Если произведена подстановка всех обобщенных типов, то ничего обобщенного не остается. Явную специализацию не следует путать с явной конкретизацией шаблона – хотя в обоих случаях создается конкретизация шаблона заданным набором аргументов-типов, механизм явной конкретизации создает конкретный экземпляр обобщенного кода, в котором вместо обобщенных типов поставлены конкретные. С другой стороны, явная специализация создает экземпляр функции или класса с тем же именем, но подменяет реализацию, так что результирующий код может оказаться совершенно другим. Это различие проще понять на примере.

Начнем с шаблона класса. Предположим, что если числитель и знаменатель `Ratio` имеют тип `double`, то мы хотим вычислить отношение и хранить его в виде одного числа. Обобщенный код `Ratio` должен остаться, но для одного конкретного набора типов код должен выглядеть совершенно иначе. Это позволяет сделать явная специализация:

```
template <>
class Ratio<double, double> {
    public:
    Ratio() : value_() {}
    template <typename N, typename D>
    Ratio(const N& num, const D& denom) :
        value_(double(num)/double(denom)) {}
    explicit operator double() const { return value_; }
    private:
    double value_;
};
```

Оба параметра шаблона заданы как `double`. Реализация класса не имеет ничего общего с обобщенной версией – вместо двух данных-членов есть только один, оператор преобразования просто возвращает значение, а конструктор вычисляет отношение числителя и знаменателя. Да и конструктор-то не тот – вместо нешаблонного конструктора `Ratio(const double&, const double&)`, который должен был бы появиться в обобщенной версии, конкретизированной двумя аргументами типа `double`, мы включили шаблонный конструктор, который принимает два аргумента произвольных типов, допускающих преобразование в `double`.

Иногда не требуется специализировать весь шаблон класса, поскольку большая часть обобщенного кода годится. Но хотелось бы изменить реализацию одной или нескольких функций-членов. Мы можем явно специализировать функцию-член следующим образом:

```
template <>
Ratio<float, float>::operator double() const { return num_/denom_; }
```

Шаблонные функции тоже можно специализировать явно. И снова, в отличие от явной конкретизации, мы должны написать тело функции, которое можем реализовать как угодно:

```
template <typename T>
T do_something(T x) { return ++x; }
template <>
double do_something<double>(double x) { return x/2; }

do_something(3);    // 4
do_something(3.0);  // 1.5
```

Однако мы не можем изменить количество или типы аргументов либо возвращаемого значения, они должны совпадать с результатом подстановки обобщенных типов, так что следующий код не откомпилируется:

```
template <>
long do_something<int>(int x) { return x*x; }
```

Явная специализация должна быть объявлена раньше первого использования шаблона, которое вызвало бы неявную конкретизацию обобщенного шаблона теми же типами. Это и понятно – в результате неявной конкретизации был бы создан класс или функция с таким же именем и типами, что при явной специализации. Таким образом, мы получили бы в программе два варианта одного и того же класса или функции, а это нарушает правило одного определения, и программа получается некорректной.

Явная специализация полезна, когда имеется один или несколько типов, для которых шаблон должен вести себя совершенно по-другому. Однако это не решает проблему с отношением указателей – нам требуется специализация, которая была бы *до некоторой степени обобщенной*, т. е. была бы применима к указателям на любые типы, а не ко всем типам вообще. Для этого предназначена частичная специализация, которую мы рассмотрим ниже.

Частичная специализация

Мы подошли к по-настоящему интересной части программирования шаблонов в C++ – частичной специализации шаблонов. Частично специализированный шаблон класса остается обобщенным кодом, но *менее обобщенным*, чем исходный. В простейшей форме частичной специализации некоторые обобщенные типы заменяются конкретными, а остальные остаются обобщенными:

```
template <typename N, typename D>
class Ratio {
.....
};

template <typename D>
class Ratio<double, D> {
```

```
  public:
  Ratio() : value_() {}
      Ratio(const double& num, const D& denom) : value_(num/double(denom)) {}
  explicit operator double() const { return value_; }
  private:
  double value_;
};
```

Здесь мы преобразуем Ratio в значение типа double, если числитель имеет тип double, вне зависимости от типа знаменателя. Для одного шаблона можно определить несколько частичных специализаций. Например, можно также написать специализацию для случая, когда знаменатель имеет тип double, а числитель произвольный:

```
template <typename N>
class Ratio<N, double> {
  public:
  Ratio() : value_() {}
  Ratio(const N& num, const double& denom) : value_(double(num)/denom) {}
  explicit operator double() const { return value_; }
  private:
  double value_;
};
```

В момент конкретизации шаблона выбирается наилучшая специализация для заданного набора типов. В нашем случае если ни числитель, ни знаменатель не принадлежат типу double, то конкретизируется общий шаблон, т. к. другого выбора нет. Если числитель имеет тип double, то первая специализация лучше общего шаблона (более специфична). Если знаменатель имеет тип double, то лучше вторая специализация. Но что, если оба члена дроби имеют тип double? В таком случае обе частичные специализации одинаково хороши, ни одна не лучше другой. Эта ситуация считается неоднозначной, и конкретизация завершается ошибкой. Отметим, что не проходит только эта конкретизация, Ratio<double, double>, а сам факт определения двух специализаций не считается ошибкой (по крайней мере, синтаксической). Ошибкой будет запрос конкретизации, который не может быть однозначно удовлетворен выбором наилучшей специализации. Чтобы любая конкретизация нашего шаблона завершалась успешно, мы должны устранить эту неоднозначность, а единственный способ сделать это – предоставить еще более узкую специализацию, которую компилятор предпочел бы обеим предыдущим. В нашем случае вариант только один – полная специализация Ratio<double, double>:

```
template <>
class Ratio<double, double> {
    public:
    Ratio() : value_() {}
    template <typename N, typename D>
    Ratio(const N& num, const D& denom) :
        value_(double(num)/double(denom)) {}
    explicit operator double() const { return value_; }
```

```
    private:
    double value_;
};
```

Теперь тот факт, что частичные специализации неоднозначны для конкретизации `Ratio<double, double>`, уже не играет роли, т. к. у нас есть версия шаблона, более специфичная, чем обе, она и будет выбрана.

Частичные специализации не обязаны полностью специфицировать часть обобщенных типов. То есть мы можем оставить все типы обобщенными, но наложить на них некоторые ограничения. Например, мы по-прежнему хотим иметь специализацию, в которой числитель и знаменатель – указатели. Это могут быть указатели на что угодно, поэтому типы остаются обобщенными, но *менее обобщенными*, чем произвольные типы в общем шаблоне:

```
template <typename N, typename D>
class Ratio<N*, D*> {
    public:
    Ratio(N* num, D* denom) : num_(num), denom_(denom) {}
    explicit operator double() const {
        return double(*num_)/double(*denom_);
    }
    private:
    N* const num_;
    D* const denom_;
};
int i = 5; double x = 10;
auto r(make_ratio(&i, &x));// Ratio<int*, double*>
double(r);          // 0.5
x = 2.5;
double(r);          // 2
```

В этой частичной специализации по-прежнему имеется два обобщенных типа, но оба они указательные, `N*` и `D*`, где `N` и `D` – произвольные типы. Реализация абсолютно не похожа на общий шаблон. Если производится конкретизация двумя указательными типами, то эта частичная специализация оказывается *более специфичной*, чем общий шаблон, и считается лучшим соответствием. Отметим, что в нашем примере знаменатель имеет тип `double`. Тогда почему не рассматривается частичная специализация с типом знаменателя `double`? Потому что, хотя знаменатель и имеет тип `double`, технически, с точки зрения логики программы, это `double*`, совершенно другой тип, для которого у нас нет специализации.

Чтобы определить специализацию, сначала должен быть объявлен общий шаблон. Но определять его необязательно, можно специализировать шаблон, для которого общей реализации не существует. Для этого мы должны написать опережающее объявление общего шаблона, а затем определить все нужные нам специализации:

```
template <typename T> class Value; // Declaration
template <typename T> class Value<T*> {
```

```
    public:
    explicit Value(T* p) : v_(*p) {}
    private:
    T v_;
};
template <typename T> class Value<T&> {
    public:
    explicit Value(T& p) : v_(p) {}
    private:
    T v_;
};

int i = 5;
int* p = &i;
int& r = i;

Value<int*> v1(p); // специализация для T*
Value<int&> v2(r); // специализация для T&
```

Здесь у нас нет общего шаблона `Value`, но есть специализации для всех указательных и ссылочных типов. При попытке конкретизировать шаблон каким-то другим типом, например `int`, мы получим сообщение об ошибке, в котором говорится, что тип `Value<int>` неполный; это то же самое, что попытаться определить объект, имея только опережающее объявление класса.

Пока что мы видели лишь примеры частичной специализации шаблонов классов. В отличие от предшествующего обсуждения полной специализации, здесь не было приведено ни одного примера специализации функций. И это не случайно – в C++ не существует такой вещи, как частичная специализация шаблона функции. То, что иногда по ошибке так называют, на самом деле является перегрузкой шаблонных функций. С другой стороны, перегрузка шаблонных функций – вещь весьма сложная, и знать о ней полезно, поэтому мы посвятим ей следующий раздел.

Перегрузка шаблонных функций

Мы привыкли к перегрузке обычных функций и методов класса, когда имеется несколько функций с одинаковым именем, но разными типами параметров. Вызывается тот вариант функции, для которого типы параметров лучше всего соответствуют фактическим аргументам, как показано в следующем примере:

```
void whatami(int x) { std::cout << x << " типа int" << std::endl; }
void whatami(long x) { std::cout << x << " типа long" << std::endl; }
whatami(5);    // 5 типа int
whatami(5.0);  // ошибка компиляции
```

Если аргументы точно соответствуют параметрам одного из перегруженных вариантов функции, то этот вариант и вызывается. В противном случае компилятор рассматривает возможность преобразовать типы параметров существующих функций. Если для какой-то функции имеется *наилучшее* преобразование, то она и вызывается. Иначе вызов считается неоднозначным,

как в последней строке примера выше. Точное определение того, что значит *наилучшее* преобразование, можно найти в стандарте. В общем случае *самыми дешевыми* считаются такие преобразования, как добавление const или удаление ссылки; вслед за ними идут преобразования между встроенными типами, преобразование указателя на производный класс в указатель на базовый класс и т. д. Если у функции несколько аргументов, то для каждого аргумента выбранного варианта должно существовать наилучшее преобразование. Никакого *голосования* не проводится – если из трех аргументов функции два точно соответствуют первому перегруженному варианту, а третий точно соответствует второму варианту, то даже если оставшиеся аргументы можно неявно преобразовать в типы соответственных параметров, все равно вызов считается неоднозначным.

Наличие шаблонов заметно усложняет процесс разрешения перегрузки. Помимо нешаблонных функций, может быть определено несколько шаблонов функций с тем же именем и, возможно, с таким же количеством аргументов. Все они являются кандидатами на разрешение вызова перегруженной функции, но шаблоны функций могут порождать функции с разными типами параметров. И как в таком случае решить, каково фактическое множество перегруженных функций? Точные правила еще сложнее, чем для нешаблонных функций, но основная идея такова: если существует нешаблонная функция, которая почти идеально соответствует фактическим аргументам, то она и выбирается. Конечно, в стандарте используются более точные термины, чем *почти идеально*, но *тривиальные* преобразования, в частности добавление const, попадают в эту категорию – мы получаем их *даром*. Если такой функции нет, то компилятор пытается конкретизировать все шаблоны функций с тем же именем, что у вызываемой, стремясь получить почти идеальное соответствие, для чего применяет выведение аргументов шаблона. Если был конкретизирован ровно один шаблон, то вызывается функция, получающаяся в результате этой конкретизации. В противном случае разрешение перегрузки продолжается обычным образом среди нешаблонных функций.

Это сильно упрощенное описание весьма сложного процесса, но следует отметить два важных момента. Во-первых, если шаблонная и нешаблонная функции одинаково хорошо соответствуют вызову, то выбирается нешаблонная, а во-вторых, компилятор не пытается конкретизировать шаблоны функций в нечто такое, что можно было бы преобразовать в нужные нам типы. После выведения типов аргументов шаблонная функция должна точно соответствовать вызову, иначе она не рассматривается. Добавим в предыдущий пример шаблон:

```
template <typename T>
void whatami(T* x) { std::cout << x << " указатель" << std::endl; }

int i = 5;
whatami(i);  // 5 типа int
whatami(&c); // 0x???? указатель
```

Здесь все выглядит похоже на частичную специализацию шаблона функции. Но на самом деле ничего подобного – это всего лишь шаблон функции, и нет никакого общего шаблона, специализацией которого этот шаблон мог бы стать. Мы имеем просто шаблон функции, для которого параметр-тип выводится из тех же аргументов, но по другим правилам. Тип шаблона можно вывести, если аргумент является указателем на что-нибудь. Сюда входит и указатель на const – T мог бы быть константным типом, поэтому если мы вызовем whatami(ptr), где ptr имеет тип const int*, то первый перегруженный вариант шаблона является точным совпадением, в котором T – это const int. Если выведение успешно, то функция, порожденная шаблоном, т. е. результат конкретизации шаблона, добавляется во множество перегруженных вариантов.

Для аргумента типа int* это единственный пригодный вариант, поэтому он и вызывается. Но что произойдет, если вызову соответствует два (или более) шаблона функций и обе конкретизации – допустимые варианты перегрузки? Давайте добавим еще один шаблон:

```
template <typename T>
void whatami<T&& x> { std::cout << "что-то странное" << std::endl; }
class C { ..... };
C c;
whatami(&c); // 0x???? указатель
whatami(c);  // что-то странное
```

Этот шаблон функции принимает аргументы по универсальной ссылке, поэтому он может быть конкретизирован для любого вызова whatami() с одним аргументом. С первым вызовом, whatami(&c), все просто – подходит только последний вариант с параметром типа T&&. Не существует преобразования из с в указатель или целое число. Но вот со вторым вызовом ситуация сложнее – мы имеем не одну, а две конкретизации шаблона, точно соответствующие вызову, без каких-либо преобразований. Тогда почему эта перегрузка не считается неоднозначной? Потому что правила разрешения перегруженных шаблонов функций отличаются от правил для нешаблонных функций и напоминают правила выбора частичной специализации шаблона класса (и это еще одна причина, по которой перегрузку шаблонов функций часто путают с частичной специализацией). Лучшим соответствием считается более специфичный шаблон.

В нашем случае более специфичен первый шаблон – он может принимать любые указатели, но только указатели. Второй шаблон готов принять вообще любой аргумент, так что в каждом случае, когда первый шаблон – потенциальное соответствие, то же справедливо и для второго, однако обратное неверно. Если более специфичный шаблон можно использовать для порождения функции, являющейся допустимым перегруженным вариантом, то он и будет выбран. В противном случае мы вынуждены обратиться к более общему шаблону.

Очень общие шаблонные функции во множестве перегруженных вариантов иногда приводят к неожиданным результатам. Пусть имеются такие три перегруженных варианта для int, double и произвольного типа:

```
void whatami(int x) { std::cout << x << " типа int" << std::endl; }
void whatami(double x) { std::cout << x << " типа double" << std::endl; }
template <typename T>
    void whatami<T&& x> { std::cout << "что-то странное" << std::endl; }
int i = 5;
float x = 4.2;
whatami(i);   // i типа int
whatami(x);   // что-то странное
```

В первом вызове аргумент имеет тип `int`, так что вариант `whatami(int)` является точным совпадением. Для второго вызова был бы выбран вариант `whatami(double)`, если бы не было перегруженного шаблона. Дело в том, что преобразование `float` в `double` неявное (как и преобразование `float` в `int`, но преобразование в `double` предпочтительнее), однако это все же преобразование, поэтому когда шаблон функции конкретизируется в точное совпадение `whatami(double&&)`, оно оказывается наилучшим соответствием и выбирается в качестве результата разрешения.

Наконец, существует еще один вид функций, играющий особую роль в разрешении перегрузки, – функции с переменным числом аргументов.

В объявлении такой функции вместо аргументов указывается многоточие ..., ее можно вызывать с любым количеством аргументов любых типов (примером может служить функция `printf`). Такая функция – последняя надежда разрешить перегрузку, она вызывается, лишь если никаких других вариантов нет:

```
void whatami(...) {
    std::cout << "что угодно" << std::endl;
}
```

Коль скоро имеется перегруженный вариант `whatami(T&& x)`, функция с переменным числом аргументов никогда не станет предпочтительным вариантом, по крайней мере не для вызовов `whatami()` с одним аргументом. Но если такого шаблона нет, то `whatami(...)` вызывается для любого аргумента, не являющегося ни числом, ни указателем. Функции с переменным числом аргументов существовали еще в языке C, и их не надо путать с шаблонами с переменным числом аргументов, появившимися в стандарте C++11. О них-то мы и поговорим в следующем разделе.

Шаблоны с переменным числом аргументов

Быть может, самое серьезное различие между обобщенным программированием на C и C++ – безопасность относительно типов. На C можно написать обобщенный код, убедительным примером тому служит стандартная функция `qsort()`. Она может сортировать значения любого типа, передаваемые по указателю типа `void*`, который на самом деле является указателем на что угодно. Конечно, программист должен знать истинный тип и привести указатель к правильному типу. В обобщенной программе на C++ типы либо задаются

явно, либо выводятся в момент конкретизации, а система типов для обобщенных типов такая же строгая, как для обычных. Если нам была нужна функция с заранее неизвестным числом аргументов, то до выхода C++11 единственным способом оставались функции с переменным числом аргументов, доставшиеся в наследство от старого доброго C. Для таких функций компилятор понятия не имеет о типах аргументов; программист должен сам корректно распаковать переданные аргументы, сколько бы их ни было.

В C++11 появился современный эквивалент функции с переменным числом аргументов – шаблон с переменным числом аргументов. Теперь можно объявить обобщенную функцию с любым числом аргументов:

```
template <typename ... T>
auto sum(const T& ... x);
```

Эта функция принимает один или более аргументов (возможно, разных типов) и вычисляет их сумму. Определить тип возвращаемого значения нелегко, но, по счастью, мы можем поручить это компилятору – нужно только объявить возвращаемый тип auto. А как фактически реализовать функцию, суммирующую неизвестное число значений, типы которых мы не можем назвать даже обобщенно? В C++17 это просто, потому что в нем есть выражение свертки:

```
template <typename ... T>
auto sum(const T& ... x) {
    return (x + ...);
}
sum(5, 7, 3);     // 15, int
sum(5, 7L, 3);    // 15, long
sum(5, 7L, 2.9);  // 14.9, double
```

В C++14, как и в C++17, когда выражения свертки недостаточно (а они полезны далеко не во всех контекстах, а в основном тогда, когда аргументы комбинируются посредством бинарных или унарных операторов), стандартная техника реализации сводится к рекурсии, повсеместно используемой в программировании шаблонов:

```
template <typename T1>
auto sum(const T& x1) {
    return x1;
}
template <typename T1, typename ... T>
auto sum(const T1& x1, const T& ... x) {
    return x1 + sum(x ...);
}
```

Первый перегруженный вариант (не частичная специализация!) относится к функции sum() с одним аргументом произвольного типа. Возвращается само переданное значение. Второй вариант относится к случаю, когда аргументов несколько, – тогда первый аргумент явно складывается с суммой остальных. Рекурсия продолжается, пока не останется один аргумент, в этот момент вы-

зывается первый перегруженный вариант, и рекурсия останавливается. Это стандартная техника раскрутки пакетов параметров в шаблонах с переменным числом аргументов, в этой книге мы еще не раз встретимся с ней. Компилятор встраивает рекурсивные вызовы функций и генерирует линейный код, складывающий все аргументы.

Шаблоны классов также могут иметь переменное число аргументов-типов, так что по ним можно создавать классы с переменным числом объектов разных типов. Объявление похоже на объявление шаблона функции. Например, давайте построим шаблон класса `Group`, который может хранить произвольное количество объектов разных типов и возвращать правильный объект в случае преобразования к одному из хранимых типов:

```
template <typename ... T>
struct Group;
```

Обычная реализация таких шаблонов тоже рекурсивна, с использованием глубоко вложенного наследования, хотя иногда возможна и нерекурсивная реализация. Одну такую мы рассмотрим в следующем разделе. Рекурсия должна остановиться, когда останется только один тип. Это делается с помощью частичной специализации, поэтому мы оставим показанный выше общий шаблон в качестве объявления без реализации и определим специализацию с одним параметром-типом:

```
template <typename T1>
struct Group<T1> {
    T1 t1_;
    Group() = default;
    explicit Group(const T1& t1) : t1_(t1) {}
    explicit Group(T1&& t1) : t1_(std::move(t1)) {}
    explicit operator const T1&() const { return t1_; }
    explicit operator T1&() { return t1_; }
};
```

В этом классе хранится значение одного типа `T1`, он инициализирует его путем копирования или перемещения и возвращает ссылку на него, когда производится преобразование в тип `T1`. Специализация для произвольного количества параметров-типов содержит значение первого типа в качестве члена данных, вместе с соответствующими методами инициализации и преобразования, и наследует шаблону класса `Group` с остальными типами:

```
template <typename T1, typename ... T>
struct Group<T1, T ...> : Group<T ...> {
    T1 t1_;
    Group() = default;
    explicit Group(const T1& t1, T&& ... t) :
        Group<T ...>(std::forward<T>(t) ...), t1_(t1) {}
    explicit Group(T1&& t1, T&& ... t) :
        Group<T ...>(std::forward<T>(t) ...), t1_(std::move(t1)) {}
    explicit operator const T1&() const { return t1_; }
```

```
    explicit operator T1&() { return t1_; }
};
```

Любой тип, содержащийся в классе `Group`, можно инициализировать одним из двух способов: копированием или перемещением. К счастью, нам не нужно выписывать конструкторы для каждой комбинации операций копирования и перемещения. Вместо этого мы завели два варианта конструктора для двух способов инициализации первого аргумента (того, который хранится в специализации), а для остальных используем идеальную передачу.

Теперь шаблон класса `Group` можно использовать для хранения значений разных типов (но в нем нельзя хранить несколько значений одного типа, потому что попытка извлечь этот тип была бы неоднозначной).

```
Group<int, long> g(3, 5);
int(g);   // 3
long(t);  // 5
```

Не слишком удобно выписывать все групповые типы явно и следить за тем, чтобы они соответствовали типам аргументов. Обычно для решения этой проблемы применяют вспомогательный шаблон функции (естественно, с переменным числом аргументов), чтобы можно было воспользоваться выведением аргументов шаблона:

```
template <typename ... T>
auto makeGroup(T&& ... t) {
    return Group<T ...>(std::forward<T>(t) ...);
}
auto g = makeGroup(3, 2.2, std::string("xyz"));
int(g);           // 3
double(g);        // 2.2
std::string(g);   // "xyz"
```

Отметим, что в стандартной библиотеке C++ имеется шаблон класса `std::tuple`, предлагающий гораздо более полную и разностороннюю версию нашего шаблона `Group`.

Шаблоны с переменным числом аргументов, особенно в сочетании с идеальной передачей, чрезвычайно полезны для написания очень общих шаблонов классов. Например, вектор может содержать объекты произвольного типа, и, для того чтобы конструировать такие объекты на месте, а не копировать, должны присутствовать конструкторы с разным числом аргументов. При написании шаблона вектора заранее неизвестно, сколько аргументов понадобится для инициализации объектов, составляющих вектор, поэтому необходимо использовать шаблон с переменным числом аргументов (и действительно, конструкторы `std::vector`, создающие вектор на месте, например `emplace_back`, являются таковыми).

Нельзя не упомянуть еще об одной конструкции C++, напоминающей шаблон и обладающей чертами одновременно класса и функции, – лямбда-выражении. Ей посвящен следующий раздел.

Лямбда-выражения

В C++ синтаксис обычной функции расширен концепцией *вызываемой сущности* – чего-то такого, что можно вызывать так же, как функцию. Примерами вызываемых сущностей являются функции (естественно), указатели на функции и объекты классов, содержащих оператор `operator()`, которые еще называют **функторами**:

```
void f(int i);
struct G {
    void operator()(int i);
};
f(5);         // функция
G g; g(5);    // функтор
```

Часто бывает полезно определить вызываемую сущность в локальном контексте, прямо в том месте, где она используется. Например, для сортировки последовательности объектов иногда необходима пользовательская функция сравнения. Ее можно определить в виде обыкновенной функции:

```
bool compare(int i, int j) { return i < j; }
void do_work() {
    std::vector<int> v;
    .....
    std::sort(v.begin(), v.end(), compare);
}
```

Но в C++ не разрешается определять функции внутри функций, поэтому наша функция `compare()` по необходимости будет определена где-то далеко от места использования. Если она используется всего один раз, то такое разделение неудобно и неблагоприятно сказывается на понятности и удобстве сопровождения кода.

Это ограничение можно обойти. Пусть функции внутри функций объявлять нельзя, но можно объявить класс и сделать его вызываемой сущностью:

```
void do_work() {
    std::vector<int> v;
    .....
    struct compare {
        bool operator()(int i, int j) const { return i < j; }
    };
    std::sort(v.begin(), v.end(), compare());
}
```

Это компактный и локальный способ, но слишком многословный. Нам ни к чему имя этого класса, и нужен всего один его экземпляр. В C++11 имеется гораздо более симпатичная возможность – лямбда-выражение:

```
void do_work() {
    std::vector<int> v;
    .....
```

```
    auto compare = [](int i, int j) { return i < j; };
    std::sort(v.begin(), v.end(), compare);
}
```

Компактнее не придумаешь. Мы могли бы задать тип возвращаемого значения, но обычно компилятор способен его вывести. Лямбда-выражение создает объект, поэтому у него есть тип, но этот тип генерируется компилятором, поэтому в объявлении объекта должно присутствовать слово auto.

Лямбда-выражения – объекты, поэтому могут иметь данные-члены. Конечно, у локального вызываемого класса тоже могут быть данные-члены. Обычно они инициализируются локальными переменными в объемлющей области видимости:

```
void do_work() {
    std::vector<double> v;
    .....
    struct compare_with_tolerance {
        const double tolerance;
        explicit compare_with_tolerance(double tol) :
            tolerance(tol) {}
        bool operator()(double x, double y) const {
            return x < y && std::abs(x - y) > tolerance;
        }
    };
    double tolerance = 0.01;
    std::sort(v.begin(), v.end(), compare_with_tolerance(tolerance));
}
```

Но это снова излишне многословный способ сделать простую вещь. Переменная tolerance упоминается трижды: как член данных, как аргумент конструктора и в списке инициализации членов. Лямбда-выражение упрощает и этот код, поскольку может захватывать локальные переменные. В локальных классах не разрешено ссылаться на переменные в объемлющей области видимости иначе как путем передачи их в аргументах конструктора, но для лямбда-выражений компилятор автоматически генерирует конструктор, который захватывает все локальные переменные, упоминаемые в теле выражения:

```
void do_work() {
    std::vector<double> v;
    .....
    double tolerance = 0.01;
    auto compare_with_tolerance = [=](auto x, auto y) {
        return x < y && std::abs(x - y) > tolerance;
    }
    std::sort(v.begin(), v.end(), compare_with_tolerance);
}
```

Здесь имя tolerance внутри лямбда-выражения ссылается на локальную переменную с таким же именем. Переменная захватывается по значению, что следует из конструкции захвата [=] в определении лямбда-выражения (можно было бы осуществить захват по ссылке, написав вместо этого [&]). Кроме того,

можно не изменять типы аргументов лямбда-выражения с int (как в предыдущем примере) на double, а просто объявить их как auto, что, по существу, превращает функцию-член operator() лямбда-выражения в шаблон (эта возможность появилась в C++14).

Лямбда-выражения чаще всего используются как локальные функции. Однако в действительности они не функции, а вызываемые объекты, и потому им недостает важной черты функций – возможности перегрузки. Последний прием, с которым мы познакомимся в этом разделе, – обход этого ограничения с целью создания набора перегруженных лямбда-выражений.

Сразу согласимся – перегружать вызываемые объекты действительно невозможно. С другой стороны, очень просто завести несколько перегруженных методов operator() в одном классе – методы допускают перегрузку, как любые функции. Конечно, метод operator() объекта лямбда-выражения генерируется компилятором, а не объявляется нами, поэтому невозможно заставить компилятор сгенерировать несколько вариантов operator() в одном лямбда-выражении. Но у классов есть свои преимущества, и главное из них – возможность наследования. Лямбда-выражения – это объекты, их типами являются классы, и этим классам можно унаследовать. Если класс открыто наследует базовому классу, то все открытые методы базового класса становятся открытыми методами производного. Если класс открыто наследует нескольким базовым классам (множественное наследование), то его открытый интерфейс состоит из всех открытых методов всех базовых классов. Если в этом наборе есть несколько одноименных методов, то они оказываются перегруженными, и применяются обычные правила разрешения перегрузки (в частности, можно создать неоднозначный набор перегруженных вариантов, и тогда программа не откомпилируется).

Таким образом, нам необходимо создать класс, который будет автоматически наследовать любому количеству базовых классов. И совсем недавно мы познакомились с подходящим инструментом – шаблонами с переменным числом аргументов. Из предыдущего раздела мы знаем, что обычный способ обхода произвольного количества параметров такого шаблона – рекурсия:

```
template <typename ... F> struct overload_set;

template <typename F1>
struct overload_set<F1> : public F1 {
    overload_set(F1&& f1) : F1(std::move(f1)) {}
    overload_set(const F1& f1) : F1(f1) {}
    using F1::operator();
};

template <typename F1, typename ... F>
    struct overload_set<F1, F ...> : public F1, public overload_set<F ...>
    {
        overload_set(F1&& f1, F&& ... f) :
          F1(std::move(f1)), overload_set<F ...>(std::forward<F>(f) ...) {}
        overload_set(const F1& f1, F&& ... f) :
```

```
          F1(f1), overload_set<F ...>(std::forward<F>(f) ...) {}
        using F1::operator();
    };

    template <typename ... F>
    auto overload(F&& ... f) {
        return overload_set<F ...>(std::forward<F>(f) ...);
}
```

`overload_set` – это шаблон класса с переменным числом аргументов. Общий шаблон необходимо объявить до специализации, но определения у него нет. Первой определяется специализация с одним лямбда-выражением – класс `overload_set` наследует лямбда-выражению и добавляет свой `operator()` в его открытый интерфейс. Специализация для N лямбда-выражений (N > 1) наследует первой специализации и классу `overload_set`, сконструированному из остальных N – 1 лямбда-выражений. Наконец, имеется вспомогательная функция, которая конструирует множество перегруженных вариантов из любого количества лямбда-выражений, – в нашем случае это необходимость, а не просто удобство, поскольку мы не можем явно задать типы лямбда-выражений, а вынуждены поручить их выведение шаблону функции. Теперь можно построить множество перегруженных вариантов из любого количества лямбда-выражений:

```
int i = 5;
double d = 7.3;
auto l = overload(
    [](int* i) { std::cout << "i=" << *i << std::endl; },
    [](double* d) { std::cout << "d=" << *d << std::endl; }
);
l(&i);   // i=5
l(&d);   // d=5.3
```

Это решение не идеально, потому что плохо справляется с неоднозначными перегрузками. В C++17 можно поступить лучше, и тут нам предоставляется шанс продемонстрировать альтернативный способ работы с пакетом параметров, не нуждающийся в рекурсии. Вот версия для C++17:

```
template <typename ... F>
struct overload_set : public F ... {
    overload_set(F&& ... f) : F(std::forward<F>(f)) ... {}
    using F::operator() ...; // C++17
};
template <typename ... F>
auto overload(F&& ... f) {
    return overload_set<F ...>(std::forward<F>(f) ...);
}
```

Шаблон с переменным числом аргументов теперь не полагается на частичные специализации, а напрямую наследует пакету параметров (эта часть реализации работает и в C++14, но для объявления `using` нужна версия C++17). Вспомогательный шаблон функции тот же самый – он выводит типы лямб-

да-выражений и конструирует объект из конкретизации overload_set этими типами. Сами лямбда-выражения передаются базовым классам посредством идеальной передачи и там используются для инициализации всех базовых объектов для объектов overload_set (лямбда-выражения допускают перемещение). Без рекурсии и частичной специализации шаблон получается гораздо компактнее и понятнее. Используется он так же, как предыдущая версия overload_set, но лучше обрабатывает почти неоднозначные перегрузки.

Как все это применяется на практике, мы увидим в последующих главах, когда потребуется написать фрагмент кода и присоединить его к объекту для последующего выполнения.

Резюме

Шаблоны, шаблоны с переменным числом аргументов и лямбда-выражения – все это мощные средства C++, позволяющие упростить написание программ, но изобилующие сложными деталями. Цель примеров в этой главе – подготовить читателя к восприятию последующих глав, где эти приемы будут использоваться для реализации паттернов проектирования, классических и новых, средствами современного языка C++. Читателей, желающих в полной мере освоить искусство использования этих сложных и мощных инструментов, мы отсылаем к другим книгам, посвященным изучению этих предметов; некоторые из них перечислены в конце главы.

Теперь читатель готов к изучению общеупотребительных идиом C++, начиная с тех, что выражают владение памятью.

Вопросы

- В чем разница между типом и шаблоном?
- Какие виды шаблонов имеются в C++?
- Какие виды параметров могут быть у шаблонов C++?
- В чем разница между специализацией и конкретизацией шаблона?
- Как осуществляется доступ к пакету параметров шаблона с переменным числом аргументов?
- Для чего применяются лямбда-выражения?

Для дальнейшего чтения

- **C++ Fundamentals**: https://www.packtpub.com/application-development/cfundamentals.
- **C++ Data Structures and Algorithms**: https://www.packtpub.com/application-development/c-data-structures-and-algorithms.
- **Mastering C++ Programming**: https://www.packtpub.com/application-development/mastering-c-programming.

Глава 3

Владение памятью

Владение памятью – одна из самых распространенных проблем в программах на C++. Многие проблемы такого рода сводятся к неправильным предположениям о том, какая часть кода или сущность владеет определенной областью памяти. Отсюда утечки памяти, доступ к невыделенной памяти, чрезмерное использование памяти и другие ошибки, которые трудно отлаживать. В современном C++ имеется набор идиом, которые в совокупности дают программисту возможность более отчетливо выражать свои намерения в части владения памятью. А это, в свою очередь, намного упрощает написание кода, который правильно выделяет память, обращается к ней и освобождает ее.

В этой главе рассматриваются следующие вопросы:

- что такое владение памятью;
- каковы характеристики хорошо спроектированного владения ресурсами;
- когда и как мы должны быть безразличны к владению ресурсами;
- как выражается монопольное владение памятью в C++;
- как выражается совместное владение памятью в C++;
- какова стоимость различных языковых конструкций, относящихся к владению памятью.

Технические требования

Базовые рекомендации по использованию C++ можно найти по адресу **https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md**.

Библиотеку поддержки базовых рекомендаций по применению C++ можно найти по адресу **https://github.com/Microsoft/GSL**.

Что такое владение памятью?

В C++ термин **владение памятью** относится к сущности, которая отвечает за обеспечение времени жизни определенной области выделенной памяти. На практике мы редко говорим о владении памятью как таковой. Обычно управлению подвергается владение и время жизни объектов, расположенных в об-

ласти памяти, а под владением памятью понимается *владение объектом*. Эта концепция владения памятью тесно связана с *владением ресурсами*. Во-первых, память – ресурс. Это не единственный ресурс, которым может управлять программа, но, безусловно, самый часто используемый. Во-вторых, в C++ управление ресурсами перепоручается объектам, владеющим ими. Таким образом, задача управления ресурсами сводится к задаче управления объектами-владельцами, а это, как мы только что выяснили, и есть то, что мы понимаем под владением памятью. В этом контексте владение памятью означает владение не только памятью, а неправильное управление владением может приводить к утечкам, ошибкам при подсчете или потере ресурсов, управляемых программой: памяти, мьютексов, файлов, описателей баз данных, видео с котиками, бронирований мест в самолете или ядерных боеголовок.

Правильно спроектированное владение памятью

Как должно выглядеть правильно спроектированное владение памятью? Наивный ответ, который сразу приходит в голову, таков: в каждой точке программы понятно, кто какими объектами владеет. Однако это требование чересчур ограничительно – большая часть программы никак не связана с владением ресурсами, в т. ч. памятью, а просто использует ресурсы. При написании такого кода достаточно знать, что некая функция или класс не владеет памятью. Знание о том, кто чем занимается, совершенно не относится к делу:

```
struct MyValues { long a, b, c, d; }
    void Reset(MyValues* v) { // безразлично, кто владеет v,
                              // главное - что не мы
    v->a = v->b = v->c = v->d = 0;
}
```

Тогда, может быть, такая формулировка: в каждой точке программы понятно, кто владеет данным объектом или что владение не изменяется? Это лучше, поскольку большая часть программы относится ко второй категории. Но все равно слишком ограничительно – когда мы принимаем владение объектом, обычно не важно, кто владел им раньше:

```
class A {
    public:
    A(std::vector<int>&& v) :
        v_(std::move(v)) {} // передача владения все равно от кого
    private:
    std::vector<int> v_;   // теперь владеем мы
};
```

Аналогично идея совместного владения (выражаемая посредством класса `std::shared_ptr` с подсчетом ссылок) заключается в том, что нам ни к чему знать, кто еще владеет объектом:

```
class A {
    public:
```

```
    A(std::shared_ptr<std::vector<int>> v) : v_(v) {}
            // не знаем и знать не хотим, кто владеет v
    private:
    std::shared_ptr<std::vector<int>> v_;
            // разделяем владение с любым числом владельцев
};
```

Для более точного описания хорошо спроектированного владения памятью недостаточно одного предложения. Ниже перечислены общие признаки хорошей практики владения памятью:

- если некоторая функция или класс никак не изменяет владение памятью, то это должно быть понятно каждому клиенту, а также автору этой функции или класса;
- если некоторая функция или класс принимает монопольное владение некоторыми переданными ей (ему) объектами, то это должно быть понятно клиенту (мы предполагаем, что автор уже знает об этом, потому что он писал код);
- если некоторая функция или класс разделяет владение переданным ей (ему) объектом, то это должно быть понятно клиенту (или человеку, читающему код клиента);
- для любого созданного объекта в любой точке, где он используется, понятно, должен код удалить объект или нет.

Плохо спроектированное владение памятью

Как правильное владение памятью не удается описать одной фразой, а приходится охарактеризовывать набором критериев, так и порочные практики владения памятью можно опознать по их типичным проявлениям. Вообще говоря, если хороший проект позволяет легко понять, владеет данный код ресурсом или нет, то плохой требует дополнительной информации, не выводимой из контекста. Например, кто владеет объектом, возвращенным следующей функцией `MakeWidget()`?

```
Widget* w = MakeWidget();
```

Ожидается ли, что клиент удалит виджет, когда в нем отпадет необходимость? Если да, то как его следует удалять? Если мы решим удалить виджет, но сделаем это неправильно, например вызовем оператор `delete` для виджета, который не был выделен оператором `new`, то обязательно повредим память. В лучшем случае программа просто «грохнется». Вот другой пример:

```
WidgetFactory WF;
Widget* w = WF.MakeAnother();
```

Владеет ли фабрика созданными ей виджетами? Будут ли они удалены вместе с удалением объекта фабрики? И ожидает ли этого клиент? Если мы решим, что фабрика, вероятно, знает, что насоздавала, и в должное время удалит эти объекты, то может произойти утечка памяти (или еще что-нибудь похуже, если эти объекты владеют какими-то ресурсами). Следующий пример:

```
Widget* w = MakeWidget();
Widget* w1 = Transmogrify(w);
```

Принимает ли `Transmogrify()` владение виджетом? Жив ли еще виджет `w`, после того как `Transmogrify()` закончила работу с ним? Если виджет удален, а `Transmogrify()` создала новый «могрифицированный» виджет `w1`, то мы имеем висячий указатель. А если виджет не удален, а мы предположили, что он удален, то налицо утечка памяти.

Чтобы вы не думали, будто все случаи плохого управления памятью можно опознать по наличию простых указателей, приведем пример неудачного подхода, который часто встречается как непродуманная реакция на проблемы, вызванные использованием простых указателей:

```
void Double(std::shared_ptr<std::vector<int>> v) {
    for (auto& x : *v) {
        x *= 2;
    }
};
    ...
std::shared_ptr<std::vector<int>> v(...);
Double(v);
    ...
```

Функция `Double()` объявляет в своем интерфейсе, что принимает владение вектором. Однако это совершенно неоправданно – у `Double()` нет никаких причин владеть своим аргументом, она всего лишь модифицирует вектор, переданный ей вызывающей программой. Мы с полным основанием можем ожидать, что вектором владеет вызывающая функция (или даже расположенная выше в стеке вызовов) и что вектор по-прежнему существует после возврата управления из `Double()`, – ведь вызывающая программа хотела, чтобы мы удвоили элементы вектора, вероятно рассчитывая, что потом сможет с ними что-то сделать.

Этот список, пусть и не полный, демонстрирует круг проблем, с которыми можно столкнуться при небрежном подходе к управлению владением памятью.

Выражение владения памятью в C++

На протяжении всей истории языка C++ эволюционировали его подходы к выражению владения памятью. Одни и те же синтаксические конструкции в разное время наделялись различной семантикой. Отчасти движущей силой эволюции были добавляемые в язык новые средства (трудно говорить о совместном владении памятью, когда нет разделяемых указателей). С другой стороны, большая часть средств управления памятью, добавленных в стандарте C++11 и позже, не была ни новыми идеями, ни новыми концепциями. Понятие разделяемого указателя существовало уже давно. Поддержка со стороны языка упрощает реализацию (а включение разделяемого указателя в стандарт-

ную библиотеку делает большинство сторонних реализаций ненужным), но вообще-то разделяемые указатели использовались в C++ задолго до включения их в стандарт C++11. Более важным изменением стала эволюция понимания в сообществе C++, а также появление общепринятых практик и идиом. Именно с точки зрения набора соглашений и семантики, связываемой с различными синтаксическими конструкциями, мы и можем говорить о сложившейся практике управления памятью как о паттерне проектирования в языке C++. А теперь рассмотрим способы выражения различных видов владения памятью.

Выражения невладения

Начнем с самого распространенного вида владения памятью. Большая часть программы ничего не выделяет, не освобождает, не конструирует и не удаляет. Она просто что-то делает с объектами, которые были созданы кем-то ранее и будут кем-то удалены в будущем. Как выразить тот факт, что функция собирается работать с объектом, но не будет пытаться удалить его или, наоборот, продлить время его жизни по завершении самой функции?

Очень просто, и любой программист на C++ делал это много раз:

```
void Transmogrify(Widged* w) {      // я не буду удалять w
    ...
}
void MustTransmogrify(Widget& w) {  // и я не буду
    ...
}
class WidgetProcessor {
    public:
    WidgetProcessor(Widget* w) : w_(w) {}
    ...
    Widget* w_;                      // я не владею w
};
```

Доступ к объекту без передачи владения следует предоставлять с помощью простых указателей и ссылок. Да, даже в C++ 17 со всеми его интеллектуальными указателями есть место для простых указателей. Более того, большинство указателей в программе простые, т. е. ничем не владеющие.

Сейчас можно с полным основанием возразить, что приведенный только что пример рекомендованной практики предоставления доступа без владения выглядит в точности так же, как один из приведенных ранее примеров порочной практики. Различие в контексте – в хорошо спроектированной программе с помощью простых указателей и ссылок предоставляется только доступ без владения. Фактическое владение всегда выражается иначе. Таким образом, понятно, что всякий раз, как встречается простой указатель, функция или класс не собирается никаким способом изменять владение объектом. Конечно, это может приводить к недоразумениям, когда приходится перерабатывать в соответствии с современной практикой старый унаследованный код, где простые указатели используются повсюду. Для ясности рекомендуется изменять такой

код частями, четко указывая переходы между кодом, следующим и не следующим современным рекомендациям.

Здесь же стоит обсудить вопрос об использовании указателей и ссылок. Синтаксически ссылка – это по существу указатель, который не может принимать значения `NULL` (или `nullptr`) и не может быть оставлен неинициализированным. Соблазнительно принять соглашение о том, что любой указатель, передаваемый функции, может быть равен `NULL` и, следовательно, должен проверяться и что функция, которая отвергает нулевые указатели, должна принимать не указатель, а ссылку. Это хорошее соглашение, и оно широко применяется, но все же недостаточно широко, чтобы считаться общепринятым паттерном проектирования. Быть может, признавая этот факт, в библиотеке C++ Core Guidelines предложена альтернатива для выражения ненулевых указателей – `not_null<T*>`. Заметим, что это соглашение – не часть языка, но может быть реализовано в стандарте C++ без расширения языка.

Выражение монопольного владения

Второй по распространенности вид владения – это монопольное владение, когда код создает объект, а затем удаляет его. Задача удаления никому не делегируется, поскольку расширение времени жизни объекта не допускается. Такой вид владения памятью настолько распространен, что мы пользуемся им, даже не задумываясь об этом:

```
void Work() {
    Widget w;
    Transmogrify(w);
    Draw(w);
}
```

Все локальные (стековые) переменные выражают монопольное владение памятью! Заметим, что в этом контексте владение не означает, что больше никому не разрешается модифицировать объект. Это просто означает, что создатель виджета `w` – в данном случае функция `Work()` – решает удалить его; удаление завершится успешно (никто не удалил объект ранее), и объект действительно будет удален (никто не попытается оставить объект в живых после его выхода из области видимости).

Это самый старый способ конструирования объекта в C++, и по сей день он остается самым лучшим. Если стековой переменной достаточно для ваших нужд, используйте ее. C++11 предлагает еще один способ выразить монопольное владение, он применяется в основном в тех случаях, когда объект нельзя создать в стеке, а приходится создавать в куче. Выделение памяти из кучи часто производится, когда владение совместное или передается – ведь объект, созданный в стеке, неизбежно будет удален по выходе из объемлющей области видимости, тут ничего не поделаешь. Если же нам нужно сохранить объект на более длительное время, то память для него нужно выделить где-то еще. Но сейчас мы говорим о монопольном владении – таком, которое не разделяется

и никому не передается. Единственная причина создавать подобные объекты в куче заключается в том, что размер или тип объекта неизвестен на этапе компиляции. Обычно так бывает с полиморфными объектами – создается производный объект, но используется указатель на базовый класс. Теперь у нас есть способ выразить монопольное владение такими объектами с помощью класса `std::unique_ptr`:

```
class FancyWidget : public Widget { ... };
std::unique_ptr<Widget> w(new FancyWidget);
```

А что, если объект создается способом, более сложным, чем `operator new`, и требуется фабричная функция? Этот тип владения рассматривается ниже.

Выражение передачи монопольного владения

В предыдущем примере новый объект был создан и тут же связан с уникальным указателем, `std::unique_ptr`, который гарантирует монопольное владение. Клиентский код выглядит точно так же, если объект создается фабрикой:

```
std::unique_ptr<Widget> w(WidgetFactory());
```

Но что должна вернуть фабричная функция? Конечно, она могла бы вернуть простой указатель, `Widget*`. Ведь именно это и возвращает оператор `new`. Но тогда открывается путь к некорректному использованию `WidgetFactory` – например, вместо запоминания возвращенного простого указателя в уникальном указателе мы могли бы передать его функции, допустим `Transmogrify`, которая принимает простой указатель по той причине, что не хочет связываться с владением им. Теперь никто не владеет виджетом, и дело кончится утечкой памяти. В идеале `WidgetFactory` должна быть написана так, чтобы вызывающая сторона была вынуждена принять владение возвращенным объектом.

В действительности нам здесь нужна передача владения – `WidgetFactory`, безусловно, является монопольным владельцем сконструированного ей объекта, но в какой-то момент она должна передать его другому, тоже монопольному, владельцу. Соответствующий код очень прост:

```
std::unique_ptr<Widget> WidgetFactory() {
    Widget* new_w = new Widget;
        ...
    return std::unique_ptr<Widget>(new_w);
}
std::unique_ptr<Widget> w(WidgetFactory());
```

Работает это точно так, как мы и хотели, но почему? Разве уникальный указатель не обеспечивает монопольного владения? Да, обеспечивает, но этот объект также допускает перемещение. При перемещении содержимого уникального указателя в другой уникальный указатель передается владение объектом; исходный указатель остается в состоянии «перемещен из» (его уничтожение не приводит к удалению каких-либо объектов). Что хорошего в этой идиоме? Очевидно, она выражает – и это проверяется во время компиляции – тот факт,

что фабрика ожидает от вызывающей стороны принятия монопольного (или разделяемого) владения объектом. Например, следующий код, который оставил бы новый виджет без владельца, не компилируется:

```
void Transmogrify(Widget* w);
Transmogrify(WidgetFactory());
```

А как же вызвать `Transmogrify()` для виджета, после того как мы должным образом передали владение? По-прежнему с помощью простого указателя:

```
std::unique_ptr<Widget> w(WidgetFactory());
Transmogrify(&*w); // или w.get() - то же самое, предоставляет доступ без владения
```

А как насчет стековых переменных? Можно ли передать кому-то монопольное владение до уничтожения переменной? Это несколько сложнее – память для объекта выделена в стеке и будет освобождена, поэтому не обойтись без копирования. Объем копирования зависит от того, допускает ли объект перемещение. В общем случае перемещение передает владение от старого объекта (перемещенного) новому (тому, куда он был перемещен). Это можно использовать для возвращаемых значений, но чаще применяется для передачи аргументов функциям, которые принимают монопольное владение. При объявлении таких функций следует указывать, что параметры передаются по ссылке на r-значение `T&&`:

```
void Consume(Widget&& w) { auto my_w = std::move(w); ... }
Widget w, w1;
Consume(std::move(w)); // нет больше w - он теперь в состоянии "перемещен из"
Consume(w1);           // не компилируется, нужно согласие на перемещение
```

Заметим, что вызывающая сторона должна явно отказаться от владения, обернув аргумент вызовом `std::move`. Это одно из преимуществ идиомы, иначе вызов с передачей владения выглядел бы точно так же, как обычный вызов.

Выражение совместного владения

И напоследок мы рассмотрим совместное владение, когда несколько сущностей владеют объектом на равных основаниях. Но прежде сделаем предупреждение – совместное владение часто используют неправильно или чрезмерно увлекаются им. Рассмотрим предыдущий пример, где функции передавался разделяемый указатель на объект, владеть которым ей не нужно. Возникает искушение поручить механизму подсчета ссылок заботу о владении объектами и *не думать об удалении*. Однако зачастую это признак плохого проектирования. В большинстве систем на каком-то уровне ясно, кто владеет ресурсами, и это следует отразить в проекте управления ресурсами. Желание *не думать об удалении* остается законным; явное удаление объектов должно производиться редко, но автоматическое удаление не требует совместного владения, достаточно ясно выраженного пожелания (уникальные указатели, данные-члены и контейнеры тоже обеспечивают автоматическое удаление).

При всем при том существуют случаи, когда совместное владение оправдано. Чаще всего это случается на низком уровне, внутри таких структур данных, как списки, деревья и т. п. Элементом данных могут владеть другие узлы той же структуры данных, на него может указывать несколько итераторов и, возможно, какие-то временные переменные внутри функций-членов структуры, которые работают со всей структурой или ее частью (например, в случае перебалансировки дерева). Кто владеет всей структурой данных, в хорошо продуманном проекте обычно не вызывает сомнений. Но владение отдельными узлами или элементами данных может быть по-настоящему совместным в том смысле, что все владельцы равны, нет ни привилегированных владельцев, ни главного.

В C++ понятие совместного владения выражается с помощью разделяемого указателя, `std::shared_ptr`:

```
struct ListNode {
    T data;
    std::shared_ptr<ListNode> next, prev;
};
class ListIterator {
    ...
    std::shared_ptr<ListNode> node_p;
};
```

Преимущество такого подхода состоит в том, что элемент списка, исключенный из списка, остается живым, пока к нему существует путь доступа через итератор. Класс `std::list` устроен иначе и таких гарантий не дает. Однако в некоторых приложениях, например в потокобезопасном списке, этот дизайн вполне допустим. Заметим, что для этого конкретного приложения понадобились бы также атомарные разделяемые указатели, которые станут доступны только в C++20 (впрочем, вы можете написать такой класс самостоятельно даже на C++11).

А что сказать о функциях, принимающих разделяемые указатели в качестве параметров? В программе, которая соблюдает рекомендации по владению памятью, такая функция извещает вызывающую сторону о том, что намеревается принять частичное владение, которое продлится дольше, чем сам вызов функции, – будет создана копия разделяемого указателя. В конкурентном контексте она может также указать, что собирается защитить объект от удаления другим потоком, по крайней мере на время своей работы.

У совместного владения есть несколько недостатков, о которых следует помнить. Самый известный – проклятие разделяемых указателей, а именно циклическая зависимость. Если два объекта с разделяемыми указателями указывают друг на друга, то вся пара остается *активной* неопределенно долго. C++ предлагает решение в виде слабого указателя, `std::weak_ptr`, – аналога разделяемого указателя, обеспечивающего безопасный указатель на объект, который, возможно, уже удален. Если в вышеупомянутой паре объектов используется один разделяемый и один слабый указатель, то циклическая зависимость разрывается.

Проблема циклической зависимости реальна, но чаще она возникает в программах, спроектированных так, что совместное владение используется, чтобы скрыть более существенную проблему нечетко определенного владения ресурсами. Однако у совместного владения есть и другие недостатки. Производительность разделяемого указателя ниже, чем простого. С другой стороны, уникальный указатель может работать так же эффективно, как простой (`std::unique_ptr` так и работает). При первом создании разделяемого указателя выделяется дополнительная память для счетчика ссылок.

В C++11 есть функция `std::make_shared`, которая объединяет сам объект со счетчиком ссылок, но это означает, что объект создан, имея в виду совместное владение (часто фабрика возвращает уникальные указатели, некоторые из которых затем преобразуются в разделяемые). Копирование или удаление разделяемого указателя должно сопровождаться соответственно увеличением или уменьшением счетчика ссылок. Разделяемые указатели зачастую имеет смысл использовать в конкурентных структурах данных, где, по крайней мере на нижнем уровне, понятие владения может быть размытым, поскольку одновременно может производиться несколько операций доступа к одному объекту. Однако спроектировать разделяемый указатель, который был бы потокобезопасным во всех контекстах, нелегко и влечет за собой дополнительные накладные расходы во время выполнения.

Резюме

В C++ владение памятью – на самом деле всего лишь синоним владения объектом, а это, в свою очередь, способ управления произвольными ресурсами – владением и доступом к ним. Мы рассмотрели современные идиомы, разработанные сообществом C++ для выражения различных типов владения памятью. C++ позволяет программисту выразить монопольное или совместное владение памятью. Не менее важно выражение идеи *невладения* в коде, безразличном к тому, кто владеет ресурсами. Мы узнали о практиках и атрибутах владения ресурсами в хорошо спроектированной программе.

Теперь мы располагаем идиоматическим языком для ясного выражения того, какая программная сущность владеет каждым объектом или ресурсом. В следующей главе рассматривается идиома простейшей операции над ресурсами: обмена.

Вопросы

- Почему так важно четко выражать, кто владеет памятью в программе?
- Каковы типичные проблемы, возникающие из-за нечеткого указания владельца памяти?
- Какие виды владения памятью можно выразить на C++?
- Как писать функции и классы, не владеющие памятью?

- Почему монопольное владение памятью предпочтительнее совместного?
- Как выразить монопольное владение памятью в C++?
- Как выразить совместное владение памятью в C++?
- Каковы потенциальные недостатки совместного владения памятью?

Для дальнейшего чтения

- *C++ – From Beginner to Expert* [видео], Аркадиуш Влодарчик (Arkadiusz Wlodarczyk): https://www.packtpub.com/application-development/c-beginner-expert-video.
- *C++ Data Structures and Algorithms*, Висну Ангорро (Wisnu Anggoro): https://www.packtpub.com/application-development/c-data-structures-and-algorithms.
- *Expert C++ Programming*, Джеганатан Сваминатан (Jeganathan Swaminathan), Майя Пош (Maya Posch) и Яцек Галовиц (Jacek Galowicz): https://www.packtpub.com/application-development/expert-c-programming.

Глава 4

От простого к нетривиальному

Мы начнем исследование основных идиом C++ с очень простой, даже скромной, операции – `swap` (обмен). Имеется в виду, что два объекта меняются местами – после обмена первый объект сохраняет свое имя, но во всех остальных отношениях выглядит так, как выглядел второй объект. И наоборот. Эта операция настолько фундаментальна для классов C++, что в стандарт включен выполняющий ее шаблон `std::swap`. Но будьте покойны – C++ ухитряется превратить даже такую простую операцию, как обмен, в сложную материю с тонкими нюансами.

В этой главе рассматриваются следующие вопросы:

- как обмен используется в стандартной библиотеке C++;
- где применяется обмен;
- как писать безопасный относительно исключений код с помощью обмена;
- как правильно реализовать обмен в собственных типах;
- как правильно обменять переменные произвольного типа.

Технические требования

Пример кода: **https://github.com/PacktPublishing/Hands-On-Design-Patterns-with-CPP/tree/master/Chapter04**.

Библиотека поддержки базовых рекомендаций по использованию C++ (GSL): **https://github.com/Microsoft/GSL**.

Обмен и стандартная библиотека шаблонов

Операция обмена широко используется в стандартной библиотеке C++. Все контейнеры, имеющиеся в **стандартной библиотеке шаблонов (STL)**, предоставляют возможность обмена, а кроме того, существует свободный шаблон функции `std::swap`. Эта операция встречается и в алгоритмах STL. Стандартную

библиотеку можно рассматривать как образец для реализации пользовательских функций, напоминающих стандартные. Поэтому мы начнем изучение операции обмена с обзора функциональности, предлагаемой стандартом.

Обмен и контейнеры STL

Концептуально обмен эквивалентен следующей операции:

```
template <typename T> void swap(T& x, T& y) {
    T tmp(x);
    x = y;
    y = tmp;
}
```

После вызова `swap()` содержимое объектов `x` и `y` переставлено местами. Однако это, пожалуй, худший способ фактической реализации обмена. Первая и самая очевидная проблема заключается в том, что эта реализация без необходимости копирует оба объекта (на самом деле выполняется три операции копирования). Время выполнения такой операции пропорционально размеру типа `T`. В случае контейнера STL размером будет размер контейнера, а не типа его элемента:

```
void swap(std::vector<int>& x, std::vector<int>& y) {
    std::vector<int> tmp(x);
    x = y;
    y = tmp;
}
```

Этот код компилируется и в большинстве случаев даже делает то, что надо. Однако каждый элемент вектора при этом копируется несколько раз. Вторая проблема – тот факт, что временно выделяются ресурсы. Так, в процессе обмена мы создаем третий вектор, который потребляет столько же памяти, сколько каждый из обмениваемых векторов. Это выделение памяти представляется излишним, поскольку в конце операции мы имеем столько же данных, сколько в начале, изменились только имена, по которым мы обращаемся к данным. И последняя проблема наивной реализации обнаруживается, стоит задуматься о том, что произойдет, если вышеупомянутое выделение памяти завершится неудачно.

Операция обмена в целом, которая должна быть простой и безотказной, поскольку меняются лишь имена для обращения к элементам вектора, завершается ошибкой выделения памяти. Но это не единственное место, где может произойти ошибка, – копирующий конструктор и оператор присваивания также могут возбуждать исключения.

Все STL-контейнеры, включая `std::vector`, гарантируют возможность обмена за постоянное время. Добиться этого довольно просто, если принять во внимание, что сами объекты STL-контейнеров содержат только указатели на данные плюс некоторую информацию о состоянии, например размер объекта. Чтобы обменять два контейнера, нам нужно просто обменять указатели (и, конечно, остальные части состояния) – элементы контейнера останутся там же,

где и были, в динамически выделенной памяти; их не нужно копировать, даже обращаться-то к ним нет надобности. Для реализации обмена необходимо только обменять указатели, размеры и другие переменные состояния (в настоящей реализации STL класс контейнера, например вектор, не состоит непосредственно из данных встроенных типов, к примеру указателей, а имеет один или несколько данных-членов типа класса, которые, в свою очередь, состоят из указателей и других встроенных типов).

Поскольку к указателям и другим данным-членам вектора нет открытого доступа, операция обмена должна быть реализована в виде функции-члена контейнера или объявлена дружественной. В STL принят первый подход – во всех STL-контейнерах имеется функция-член `swap()`, которая обменивает данный объект с другим объектом того же типа.

Реализация посредством обмена указателей косвенно решает и остальные две проблемы. Во-первых, поскольку обмениваются только данные-члены контейнеров, не производится никакого выделения памяти. Во-вторых, копирование указателей и прочих встроенных типов не может привести к исключению, поэтому и вся операция обмена не возбуждает исключений (и не может завершиться какой-то другой ошибкой).

Но описанная только что простая и непротиворечивая картина – не вся правда. Первое, и самое простое, осложнение относится к контейнерам, параметризованным не только типом элемента, но и каким-то вызываемым объектом. Например, контейнер `std::map` принимает факультативную функцию для сравнения элементов отображения; по умолчанию это `std::less`. Такие вызываемые объекты необходимо хранить вместе с контейнером. Поскольку они вызываются очень часто, из соображений производительности крайне желательно хранить их в той же выделенной области памяти, что и сам объект контейнера, и действительно они являются данными-членами класса контейнера.

Однако у этой оптимизации есть цена – теперь для обмена двух контейнеров нужно переставить местами и функции сравнения, т. е. сами объекты, а не указатели на них. Функции сравнения реализуются клиентом библиотеки, поэтому нет гарантии, что их вообще можно обменять. Что уж говорить о гарантии отсутствия исключений!

Поэтому для `std::map` стандарт дает следующую гарантию – чтобы отображение допускало обмен, должны допускать обмен вызываемые объекты. Кроме того, обмен двух отображений не возбуждает исключения, если этого не делает обмен объектов сравнения, а любое исключение, возбуждаемое обменом этих объектов, распространяется наружу из `std::map`. Все сказанное неприменимо к таким контейнерам, как `std::vector`, которые не пользуются вызываемыми объектами, т. е. обмен этих контейнеров не возбуждает исключения (по крайней мере, нам об этом неизвестно).

Второе осложнение в согласованном и естественном поведении обмена связано с распределителями памяти, и вот его-то разрешить трудно. Рассмотрим проблему – два обмениваемых контейнера по необходимости должны иметь

распределители одного типа, но это необязательно один и тот же объект. Память для элементов каждого контейнера выделяется его распределителем и должна им же и освобождаться. После обмена первый контейнер владеет элементами из второго контейнера и в конечном итоге должен освободить занятую ими память. Это можно корректно сделать только с помощью распределителя из первого контейнера, поэтому распределители тоже должны обмениваться.

До выхода C++11 стандарты C++ полностью игнорировали эту проблему и декларировали, что любой из двух распределителей объектов одного типа должен уметь освобождать память, выделенную другим. Если это так, то обменивать распределители вообще не нужно. Если нет, то мы уже нарушили стандарт и оказались на территории неопределенного поведения. C++11 разрешает распределителям иметь нетривиальное состояние, которое, следовательно, тоже необходимо обменять. Но объекты распределителей не обязаны допускать обмен. Стандарт решает эту проблему следующим образом: если для некоторого класса распределителя `allocator_type` существует класс характеристик `trait`, в котором среди прочего определено свойство `std::allocator_traits<allocator_type>::propagate_on_container_swap::value`, и если его значение равно `true`, то распределители обмениваются с помощью неквалифицированного обращения к свободной функции `swap`, т. е. просто вызовом `swap(allocator1, allocator2)` (что этот вызов делает, описано в следующем разделе). Если это значение не равно `true`, то распределители не обмениваются вовсе, и оба контейнерных объекта должны использовать один и тот же распределитель. Если это не так, то мы вновь возвращаемся к неопределенному поведению. В C++17 это формализовано несколько сильнее – функции-члены `swap()` STL-контейнеров объявляются с условным спецификатором `noexcept()` с такими же ограничениями.

Из запрета обмениваемым контейнерам возбуждать исключения, по крайней мере когда распределители не применяются и контейнер не пользуется вызываемыми объектами или эти объекты не возбуждают исключений, вытекает довольно тонкое ограничение на реализацию контейнера – нельзя использовать оптимизацию локального буфера.

Об этой оптимизации мы будем подробно говорить в главе 10, но в двух словах идея заключается в том, чтобы избежать динамического выделения памяти для контейнеров с очень малым числом элементов, например коротких строк, путем определения буфера внутри самого класса контейнера. Однако такая оптимизация в общем случае несовместима с концепцией не возбуждающего исключения обмена, потому что элементы внутри контейнерного объекта больше не получится обменять, просто обменяв указатели, а придется производить копирование между контейнерами.

Свободная функция swap

В стандарте описана также шаблонная функция `std::swap()`. До выхода C++11 она была объявлена в заголовке `<algorithm>`, а в C++11 перемещена в `<utility>`. Вот ее объявление:

```
template <typename T>
    void swap (T& a, T& b);
template <typename T, size_t N>
    void swap(T (&a)[N], T (&b)[N]); // Since C++11
```

Перегрузка для массивов была добавлена в C++11. В стандарте C++20 в объявления обоих вариантов еще добавлено ключевое слово `constexpr`. Для STL-контейнеров `std::swap()` вызывает функцию-член `swap()`. Как мы увидим в следующем разделе, поведение `swap()` можно настроить и для других типов, но если не предпринимать никаких усилий, то используется реализация по умолчанию. Эта реализация выполняет обмен с использованием временного объекта. До C++11 временный объект конструировался копированием, а обмен производился с помощью двух присваиваний, как было показано в предыдущем разделе. Тип должен быть копируемым (т. е. обладать копирующим конструктором и копирующим оператором присваивания), в противном случае `std::swap()` не откомпилируется. В C++11 `std::swap()` переопределена, так что используется конструирование перемещением и перемещающее присваивание. Как обычно, если класс копируемый, но операции перемещения в нем не объявлены, то используются копирующий конструктор и копирующий оператор присваивания. Заметим, что если в классе объявлены операции копирования, а операции перемещения объявлены удаленными, то автоматический откат к копированию не производится – тип этого класса не допускает перемещения, и `std::swap()` для него не компилируется.

Поскольку копирование объекта, вообще говоря, может возбуждать исключение, обмен двух объектов, для которых пользователем не реализовано специальное поведение функции обмена, также может возбуждать исключение. Операции перемещения обычно не возбуждают исключений, и в C++11 если объект обладает перемещающими конструктором и оператором присваивания, которые не возбуждают исключений, то `std::swap()` также предоставляет гарантию отсутствия исключений. Это поведение формализовано в C++17 с помощью условной спецификации `noexcept()`.

Обмен как в стандарте

Из приведенного выше обзора реализации обмена в стандартной библиотеке мы можем вывести следующие рекомендации:

- классы, поддерживающие обмен, должны реализовать функцию-член `swap()`, так чтобы она выполняла операцию за постоянное время;
- для всех типов, допускающих обмен, должна быть предоставлена также свободная функция `swap()`;
- обмен двух объектов не должен возбуждать исключений или еще каким-то образом отказывать.

Последняя рекомендация менее категорична, выполнить ее не всегда возможно. Вообще говоря, если в классе имеются операции перемещения, которые не возбуждают исключений, то возможна также не возбуждающая исключений

реализация обмена. Отметим еще, что многие гарантии безопасности относительно исключений, в частности предоставляемые стандартной библиотекой, требуют, чтобы операции перемещения и обмена не возбуждали исключений.

Когда и для чего использовать обмен

Что такого важного в функциональности обмена, что она заслуживает целой главы? И если уж на то пошло, зачем вообще использовать обмен? Почему нельзя и дальше обращаться к объекту по его исходному имени? В основном это связано с безопасностью относительно исключений, и именно поэтому мы не устаем отмечать, когда операция обмена может, а когда не может возбуждать исключения.

Обмен и безопасность относительно исключений

Самое важное применение обмена в C++ – написание кода, безопасного относительно исключений и вообще ошибок. Суть проблемы в том, что в программе, безопасной относительно исключений, возникновение исключения никогда не должно оставлять программу в неопределенном состоянии. Это относится не только к исключениям, но и к любым ошибкам. Отметим, что ошибка не обязательно должна обрабатываться средствами механизма исключений – например, возврат кода ошибки из функции тоже должен быть обработан, так чтобы не создавать неопределенного поведения. В частности, если операция приводит к ошибке, то все захваченные ей к этому моменту ресурсы должны быть освобождены. Часто желательна еще более строгая гарантия – любая операция либо завершается успешно, либо откатывает все выполненные действия.

Рассмотрим пример, когда ко всем элементам вектора применяется некоторое преобразование и результаты сохраняются в новом векторе:

```
class C; // тип элементов
C transmogrify(C x) { return C(...); } // некоторая операция над C
void transmogrify(const std::vector<C>& in, std::vector<C>& out) {
    out.resize(0);
    out.reserve(in.size());
    for (const auto& x : in) {
        out.push_back(transmogrify(x));
    }
}
```

Здесь мы возвращаем вектор в выходном параметре (в C++17 можно было бы возвратить значение и рассчитывать на устранение копирования (copy elision), но в предыдущих версиях стандарта устранение копирования не гарантируется. Сначала вектор опустошается, а его размер делается таким же, как у входного вектора. Любые данные, находившиеся в векторе `out`, могут быть стерты. Заметим, что `reserve()` вызывается, чтобы избежать повторных операций выделения памяти для растущего вектора.

Этот код правильно работает, если только не возникнет ошибок, т. е. исключений. Но это не гарантируется. Во-первых, `reserve()` выделяет память, что может закончиться неудачно. Если так и случится, то функция `transmogrify()` выйдет по исключению, и выходной вектор останется пустым, поскольку вызов `resize(0)` уже выполнен. Исходное содержимое выходного вектора потеряно, а взамен ничего не записано. Во-вторых, на любой итерации цикла обхода элементов вектора может возникнуть исключение. Его может возбудить копирующий конструктор нового элемента выходного вектора или само преобразование. В любом случае цикл будет прерван. STL гарантирует, что выходной вектор не останется в неопределенном состоянии, даже если копирующий конструктор внутри `push_back()` завершится неудачно, – новый элемент не будет создан *частично*, и размер вектора не увеличится. Однако уже сохраненные элементы останутся в выходном векторе (а те элементы, что хранились в нем раньше, пропадут). Возможно, нас это не устраивает – нет ничего неестественного в том, чтобы потребовать, чтобы операция `transmogrify()` либо завершилась успешно и была применена ко всему вектору, либо «грохнулась» и не изменила ничего.

Ключом к такой реализации, безопасной относительно исключений, является обмен:

```
void transmogrify(const std::vector<C>& in, std::vector<C>& out) {
    std::vector<C> tmp;
    tmp.reserve(in.size());
    for (const auto& x : in) {
        tmp.push_back(transmogrify(x));
    }
    out.swap(tmp); // не должна возбуждать исключений!
}
```

В этом примере мы изменили код, так что он работает с временным вектором на протяжении всего преобразования. Отметим, что в типичном случае переданный выходной вектор пуст, поэтому потребление памяти программой не увеличивается. Если в выходном векторе были какие-то данные, то как старые, так и новые данные будут находиться в памяти до конца работы функции. Это необходимо для гарантии того, что старые данные не будут удалены, если новые не удалось вычислить полностью. При желании эту гарантию можно обменять на пониженное потребление памяти и очищать выходной вектор в начале функции (с другой стороны, вызывающая сторона, согласная на такой компромисс, может просто опустошить вектор перед вызовом `transmogrify()`).

Если в любой точке выполнения функции `transmogrify()`, вплоть до последней строки, возникнет исключение, то временный вектор будет удален, как и любая другая локальная переменная, выделенная в стеке (см. главу 5). Последняя строка – ключ к безопасности относительно исключений, она обменивает содержимое выходного вектора и временного. Если эта строка может возбудить исключение, то вся наша работа пойдет насмарку – обмен не выполнен, и выходной вектор остался в неопределенном состоянии, поскольку мы не

знаем, какая часть операции обмена завершилась, перед тем как возникло исключение. Но если обмен не возбуждает исключений, как в случае `std::vector`, то коль скоро управление достигло последней строки, вся операция `transmogrify()` завершилась успешно, и вызывающей стороне возвращен результат. А что случилось со старым содержимым выходного вектора? Теперь им владеет временный вектор, который будет неявно удален в следующей строке (по достижении закрывающей скобки). В предположении, что деструктор класса `C` соблюдает рекомендации C++ и не возбуждает исключений – а иное стало бы приглашением в загробный мир неопределенного поведения, – вся наша функция безопасна относительно исключений.

Иногда эту идиому называют **копирование и обмен**, и, пожалуй, она дает самый простой способ реализовать операцию с семантикой фиксации или отката, или строгую гарантию безопасности относительно исключений. Ключ к применению этой идиомы – возможность обменять объекты чисто, не возбуждая исключений.

Другие распространенные идиомы обмена

Существует еще несколько общеупотребительных приемов, опирающихся на обмен, но ни один из них не важен так, как применение обмена для обеспечения безопасности относительно исключений.

Начнем с очень простого способа сбросить контейнер или любой другой объект, допускающий обмен, в состояние после конструирования по умолчанию:

```
C c = ....;       // объект, содержащий какие-то данные
{
    C tmp;
    c.swap(tmp);  // теперь с пуст
} // старого с больше нет
```

Заметим, что этот код явно создает пустой объект специально для того, чтобы обменяться с ним, а кроме того, введена дополнительная область видимости (пара фигурных скобок), чтобы объект был удален как можно скорее. Можно поступить еще лучше, используя безымянный временный объект для обмена:

```
C c = ....;    // объект, содержащий какие-то данные
C().swap(c);   // временный объект создан и удален
```

Здесь временный объект в одной строке создается и удаляется – и уносит с собой старое содержимое объекта `c`. Отметим, что порядок обмениваемых объектов очень важен – функция-член `swap()` вызывается от имени временного объекта. Попытка сделать наоборот не компилируется:

```
C c = ....;    // объект, содержащий какие-то данные
c.swap(C());   // близко к цели, но не компилируется
```

Дело в том, что функция-член `swap()` принимает аргумент по неконстантной ссылке `C&`, а неконстантные ссылки не могут связываться с временными объектами (и вообще с `r`-значениями). По той же причине свободную функ-

цию swap() нельзя использовать для обмена объекта с временным объектом, поэтому если в классе нет функции-члена swap(), то придется создавать явный именованный объект.

Более общая форма этой идиомы используется, когда нужно применить преобразования к исходному объекту, не изменяя его имя в программе. Предположим, что в нашей программе имеется вектор, к которому требуется применить показанную выше функцию transmogrify(), но новый вектор создавать мы не хотим. Вместо этого мы хотим и дальше использовать исходный вектор (по крайней мере, его имя), но с новыми данными. Следующая идиома позволяет элегантно достичь желаемого результата:

```
std::vector<C> vec;
... // записать данные в вектор
{
    std::vector<C> tmp;
    transmogrify(vec, tmp); // tmp содержит результат
    swap(vec, tmp);         // теперь результат содержит vec!
}                           // а теперь старый vec уничтожен
... // продолжаем использовать vec, но уже с новыми данными
```

Эту конструкцию можно повторять сколько угодно раз, заменяя содержимое объекта без введения новых имен в программу. Сравните с более традиционным, в духе C, способом, в котором обмен не используется:

```
std::vector<C> vec;
...                       // записать данные в вектор
std::vector<C> vec1;
transmogrify(vec, vec1);  // начиная с этого момента, используем vec1!
std::vector<C> vec2;
    transmogrify_other(vec1, vec2); // а теперь должны использовать vec2!
```

Отметим, что старые имена vec и vec1 по-прежнему доступны, после того как новые данные вычислены. Было бы легко по ошибке использовать в последующем коде vec вместо vec1. А продемонстрированная выше техника позволяет не засорять программы новыми именами переменных.

Как правильно реализовать и использовать обмен

Мы видели, как функциональность обмена реализована в стандартной библиотеке и какие требования предъявляются к реализации. Теперь посмотрим, как правильно поддержать обмен для своих типов.

Реализация обмена

Мы видели, что все STL-контейнеры и многие другие типы из стандартной библиотеки (например, std::thread) предоставляют функцию-член swap(). Поступать именно так необязательно, но это самый простой способ реализовать операцию обмена, которой необходим доступ к закрытым данным класса,

а кроме того, единственный способ обменять объект с временным объектом того же типа. Правильное объявление функции-члена `swap()` выглядит следующим образом:

```
class C {
    public:
    void swap(C& rhs) noexcept;
};
```

Разумеется, спецификацию `noexcept` следует включать, только если действительно можно дать гарантию отсутствия исключений; в некоторых случаях она должна быть условной, зависящей от свойств других типов.

Как следует реализовать обмен? Есть несколько способов. Для многих классов можно просто обменивать данные-члены один за другим. Это делегирует задачу обмена объектов их типам, и если все типы следуют этому образцу, то в конечном итоге все сведется к обмену встроенных типов, из которых все и состоит. Если вы наперед знаете, что в классе члена данных имеется функция-член `swap()`, то можете вызвать ее. В противном случае придется вызывать свободную функцию обмена. Скорее всего, это окажется конкретизация шаблона `std::swap()`, но вы не должны вызывать ее по этому имени по причинам, которые будут объяснены в следующем разделе. Вместо этого следует ввести имя в объемлющую область видимости и вызвать `swap()` без квалификатора `std::`:

```
#include <utility> // <algorithm> до выхода C++11
...
class C {
    public:
    void swap(C& rhs) noexcept {
        using std::swap;  // Вводит std::swap в эту область видимости
        v_.swap(rhs.v_);
        swap(i_, rhs.i_); // вызывается std::swap
    }
    ...
    private:
    std::vector<int> v_;
    int i_;
};
```

Существует идиома реализации, очень дружественная к обмену, она называется **идиома pimpl**, или **описатель-тело**. Ее основное применение – минимизировать количество зависимостей на этапе компиляции и избежать раскрытия реализации класса в заголовочном файле. Смысл идиомы в том, что все объявление класса в заголовочном файле состоит из необходимых открытых функций-членов плюс единственный указатель на настоящую реализацию. Реализация и тела функций-членов находятся в `C`-файле. Член, содержащий **указатель на реализацию** (**p**ointer to **impl**ementation), часто называют `p_impl` или `pimpl`, отсюда и название идиомы. Обмен объектов класса, реализованного с помощью идиомы pimpl, не сложнее обмена двух указателей:

```
// В заголовке C.h:
class C_impl;     // опережающее объявление
class C {
    public:
    void swap(C& rhs) noexcept {
        swap(pimpl_, rhs.pimpl_);
    }
    void f(...);  // только объявление
    ...
    private:
    C_impl* pimpl_;
};
// В C-файле:
class C_impl {
    ... настоящая реализация ...
};
void C::f(...) { pimpl_->f(...); } // настоящая реализация C::f()
```

С функцией-членом swap мы разобрались. Но что, если кто-то вызывает свободную функцию swap() в своих типах? Если код написан, как показано выше, то это приведет к вызову стандартной реализации std::swap(), если она видима (например, благодаря объявлению using std::swap), т. е. реализации, в которой используются операции copy или move:

```
class C {
    public:
    void swap(C& rhs) noexcept;
};
...
C c1(...), c2(...);
swap(c1, c2);   // либо не компилируется, либо вызывается std::swap
```

Очевидно, что мы также обязаны поддержать свободную функцию swap(). Ее легко можно было бы объявить сразу после объявления класса. Однако следует принять во внимание, что произойдет, если класс объявлен не в глобальной области видимости, а в некотором пространстве имен:

```
namespace N {
class C {
    public:
    void swap(C& rhs) noexcept;
};
void swap(C& lhs, C& rhs) noexcept { lhs.swap(rhs); }
}
...
N::C c1(...), c2(...);
swap(c1, c2);   // вызывается свободная функция N::swap()
```

Неквалифицированное обращение к swap() приводит к вызову свободной функции swap() в пространстве имен N, которая, в свою очередь, вызывает функцию-член swap() от имени одного из аргументов (по соглашению, приня-

тому в стандартной библиотеке, вызывается `lhs.swap()`). Заметим, однако, что мы вызывали не `N::swap()`, а просто `swap()`. Вне пространства имен `N` и в отсутствие директивы `using namespace N;` неквалифицированный вызов обычно не разрешается вызовом функции внутри пространства имен. Однако в этом случае происходит именно так в силу особенности стандарта, которая называется **поиском, зависящим от аргументов** (Argument-Dependent Lookup – **ADL**), или **поиском Кёнига**. Механизм ADL добавляет во множество разрешения перегрузки все функции, объявленные в тех областях видимости, где объявлены аргументы функции.

В нашем случае компилятор видит аргументы `c1` и `c2` функции `swap(...)` и понимает, что они имеют тип `N::C`, – еще даже до того, как выяснит, к чему относится имя `swap`. Поскольку аргументы объявлены в пространстве имен `N`, все функции, объявленные в этом пространстве имен, добавляются во множество разрешения перегрузки, вследствие чего функция `N::swap` становится видимой.

Если в типе имеется функция-член `swap()`, то самый простой способ реализовать свободную функцию `swap()` – вызвать функцию-член. Однако наличие функции-члена необязательно; если было принято решение не поддерживать функцию-член `swap()`, то свободная функция `swap()` должна иметь доступ к закрытым данным класса. Она должна быть объявлена дружественной:

```
class C {
    friend void swap(C& rhs) noexcept;
};
void swap(C& lhs, C& rhs) noexcept {
    ... обменять данные-члены C ...
}
```

Можно также определить встраиваемую реализацию функции `swap()`, не давая отдельного определения:

```
class C {
    friend void swap(C& lhs, C& rhs) noexcept {
        ... обменять данные-члены C ...
    }
};
```

Это особенно удобно, когда имеется не один класс, а шаблон класса. Этот паттерн мы рассмотрим подробнее в главе 11.

Часто забывают об одной детали реализации – обмене с собой, т. е. вызове `swap(x, x)` или, в случае функции-члена, `x.swap(x)`. Корректно ли определена эта операция, и если да, то что она делает? Ответ, похоже, такой: она корректно определена или, по крайней мере, должна быть таковой в стандартах C++03 и C++11 (и более поздних), но не делает ничего, т. е. она не изменяет объект (хотя, быть может, затраты на ее работу ненулевые). Пользовательская реализация обмена либо должна естественно обеспечивать безопасность обмена с собой, либо явно проверять это. Если обмен реализован в терминах копирующего или перемещающего присваивания, то важно отметить, что стандарт требует

от копирующего присваивания безопасности присваивания себе же, тогда как перемещающее присваивание может изменять объект, но должно оставить его в допустимом состоянии, которое называется «**перемещен из**» (moved-from) (в этом состоянии объекту все еще можно что-то присвоить).

Правильное использование обмена

До сих пор мы говорили то о вызове функции-члена `swap()`, то о вызове свободной функции `swap()`, то о явно квалифицированной операции `std::swap()` без всякой системы или обоснования. Пора навести порядок в этом вопросе.

Прежде всего всегда безопасно и правильно вызывать функцию-член `swap()`, если вы знаете, что она существует. Неизвестность чаще всего сопряжена с написанием кода шаблонов – имея дело с конкретными типами, мы обычно знаем, какой интерфейс они предоставляют. Это оставляет открытым всего один вопрос: должны ли мы указывать префикс `std::` при вызове свободной функции `swap()`?

Рассмотрим, что произойдет, если мы так сделаем:

```
namespace N {
class C {
    public:
    void swap(C& rhs) noexcept;
};
void swap(C& lhs, C& rhs) noexcept { lhs.swap(rhs); }
}
...
N::C c1(...), c2(...);
std::swap(c1, c2);  // вызывается std::swap()
swap(c1, c2);       // вызывается N::swap()
```

Отметим, что поиск, зависящий от аргументов, не применяется к квалифицированным именам, поэтому обращение к `std::swap()` по-прежнему вызывает конкретизацию шаблона `swap` из заголовочного файла `<utility>` стандартной библиотеки. По этой причине рекомендуется никогда не вызывать `std::swap()` явно, а вводить эту перегрузку в текущую область видимости с помощью объявления `using`, после чего вызывать `swap` без квалификации:

```
using std::swap;   // делает std::swap() доступной
swap(c1, c2);      // вызывается N::swap(), если она существует,
                   // в противном случае std::swap()
```

К сожалению, полностью квалифицированные вызовы `std::swap()` часто встречаются в программах. Чтобы защититься от такого кода и гарантировать, что при любых обстоятельствах будет вызвана ваша реализация `swap`, можно конкретизировать шаблон `std::swap()` своим типом:

```
namespace std {
    void swap(N::C& lhs, N::C& rhs) noexcept { lhs.swap(rhs); }
}
```

Вообще говоря, стандарт запрещает объявлять собственные функции или классы в зарезервированном пространстве имен std::. Однако в стандарте явно сделано исключение для некоторых шаблонов функций (и std::swap() – одна из них). При наличии такой специализации обращение к std::swap() вызовет именно ее, а та уже переадресует вызов нашей реализации. Заметим, что недостаточно конкретизировать шаблон std::swap(), поскольку такая конкретизация не участвует в поиске, зависящем от аргументов. Если не предоставлено другой свободной функции swap, то мы имеем обратную проблему:

```
using std::swap;   // делает std::swap() доступной
std::swap(c1, c2); // вызывается наша перегрузка std::swap()
swap(c1, c2);      // вызывается std::swap() по умолчанию
```

Теперь неквалифицированное обращение приводит к вызову реализации std::swap() по умолчанию – той, в которой используются перемещающие конструкторы и операторы присваивания. Чтобы гарантировать правильную обработку любого обращения к swap, необходимо реализовать как свободную функцию swap(), так и явную специализацию шаблона std::swap() (разумеется, они могут и даже должны делегировать работу одной и той же реализации). Наконец, заметим, что стандарт разрешает расширять пространство имен std:: конкретизациями шаблонов, но не разрешает добавлять новые перегрузки шаблонов. Поэтому если вместо конкретного типа мы имеем шаблон класса, то не сможем специализировать для него std::swap; такой код, по всей вероятности, откомпилируется, но стандарт не гарантирует, что будет выбран желательный перегруженный вариант (технически это неопределенное поведение, так что стандарт вообще ничего не гарантирует). Уже по этой причине следует избегать прямого обращения к std::swap.

Резюме

Операция обмена в C++ используется для реализации нескольких важных паттернов. Самый важный их них – реализация безопасных относительно исключений транзакций с помощью идиомы копирования и обмена. Все контейнеры в стандартной библиотеке и большинство других классов STL предоставляют функцию-член swap, быструю и, если возможно, не возбуждающую исключений. Пользовательские типы, которые нуждаются в поддержке обмена, должны следовать такому же принципу. Заметим, однако, что реализация функции swap, не возбуждающей исключений, обычно требует дополнительного уровня косвенности и идет вразрез с несколькими паттернами оптимизации. Помимо функции-члена swap, мы рассмотрели реализацию и использование свободной функции swap. Учитывая, что std::swap всегда доступна и может вызываться от имени любых объектов, допускающих копирование или перемещение, программист должен также позаботиться о реализации свободной функции swap, если для данного типа существует лучший способ обмена (в частности, любой

тип с функцией-членом swap должен предоставить также перегрузку свободной функции, которая будет вызывать функцию-член).

Наконец, хотя предпочтительно вызывать свободную функцию swap без префикса std::, это правило не соблюдается настолько часто, что следует подумать о явной специализации шаблона std::swap.

В следующей главе мы обсудим одну из самых популярных и мощных идиом C++ – механизм управления ресурсами.

Вопросы

- Что делает операция обмена?
- Как обмен используется в программах, безопасных относительно исключений?
- Почему функция swap не должна возбуждать исключений?
- Какую реализацию swap следует предпочесть: в виде функции-члена или свободной функции?
- Как обмен реализован в классах из стандартной библиотеки?
- Почему свободную функцию swap следует вызывать без квалификатора std::?

Глава 5

Все о захвате ресурсов как инициализации

Управление ресурсами – наверное, вторая по частоте вещь, которой занимается программа, – после вычислений. Но из того, что это делается часто, вовсе не следует, что это открыто взгляду, – некоторые языки скрывают от пользователя большую часть, а то и весь механизм управления ресурсами. Однако тот факт, что он скрыт, не означает, что его нет.

Любая программа должна использовать память, а память – это ресурс. Программа была бы бесполезна, если бы никогда не взаимодействовала с внешним миром, ну хотя бы путем вывода на печать, а каналы ввода-вывода (файлы, сокеты и т. д.) – тоже ресурсы.

Эту главу мы начнем с ответа на следующие вопросы:

- что считается ресурсом в программе на C++;
- каковы ключевые проблемы управления ресурсами в C++.

Затем мы введем концепцию захвата ресурсов как инициализации (Resource Acquisition is Initialization – RAII) и объясним, как она помогает управлять ресурсами в C++, ответив на следующие вопросы:

- каков стандартный подход к управлению ресурсами в C++ (RAII);
- как RAII решает проблемы управления ресурсами.

И закончим эту главу обсуждением следствий и возможных проблем, связанных с использованием RAII, дав ответы на такие вопросы:

- какие предосторожности необходимо соблюдать при написании RAII-объектов;
- каковы последствия использования RAII для управления ресурсами.

C++ со своей философией абстракций с нулевыми издержками не скрывает ресурсы и управление ими на уровне ядра языка. Но не следует путать сокрытие ресурсов с управлением ими.

Технические требования

Система автономного тестирования Google Test: **https://github.com/google/googletest**.

Библиотека Google Benchmark: **https://github.com/google/benchmark**.

Примеры кода: **https://github.com/PacktPublishing/Hands-On-Design-Patterns-with-CPP/tree/master/Chapter05**.

Управление ресурсами в C++

Все программы оперируют ресурсами и должны управлять ими. Конечно, самым распространенным ресурсом является память. Поэтому вам часто доводилось читать об **управлении памятью** в C++. Но на самом деле ресурсом может быть чуть ли не что угодно. Есть много программ, написанных специально для управления реальными, физически осязаемыми ресурсами или более эфемерными (но от того не менее ценными) цифровыми. Денежные средства на банковских счетах, места в самолете, запчасти для автомобилей и собранные автомобили, даже упаковки молока – в современном мире для всего, что следует подсчитывать и учитывать, есть программа, которая этим занимается. Но даже в программе, которая производит чистые вычисления, могут быть разнообразные и сложные ресурсы, если только программа не отвергает все и всяческие абстракции и не оперирует голыми числами. Например, в физической программе имитационного моделирования ресурсами могут быть частицы.

У всех этих ресурсов есть одна общая вещь – их необходимо учитывать. Они не должны исчезать без следа, и программа не должна доставать из воздуха ресурс, которого в действительности не существует. Часто бывает необходим конкретный экземпляр ресурса – вы же не хотите, чтобы плата за неизвестно кем сделанную покупку списывалась с вашего счета; в данном случае важно, с каким экземпляром ресурса иметь дело. Таким образом, главным при оценке различных подходов к управлению ресурсами является корректность – насколько хорошо дизайн гарантирует правильное управление ресурсами, легко ли допустить ошибку и насколько трудно будет ее найти? А раз так, то неудивительно, что при рассмотрении примеров кода в этой главе мы будем пользоваться каркасом тестирования.

Установка библиотеки эталонного микротестирования

Нас будет интересовать эффективность выделения памяти и небольшие фрагменты кода, содержащие такое выделение. Подходящим инструментом для измерения производительности небольших фрагментов кода является эталонный микротест. Средств такого рода существует много, в этой книге мы будем пользоваться библиотекой Google Benchmark. Чтобы следить за примерами, вам понадобится сначала скачать и установить библиотеку (следуйте инструкциям в файле `Readme.md`). Затем можно будет откомпилировать и выполнить примеры. Можете собрать демонстрационные примеры, прилагаемые к библиотеке, чтобы посмотреть, как эталонный тест собирается конкретно в вашей системе. Например, на компьютере под управлением Linux команда сборки и выполнения теста `malloc1.C` может выглядеть так:

```
$CXX malloc1.C -I. -I$GBENCH_DIR/include -g -O4 -Wall -Wextra -Werror \
  -pedantic --std=c++14 $GBENCH_DIR/lib/libbenchmark.a -lpthread -lrt \
  -lm -o malloc1 && ./malloc1
```

Здесь `$CXX` – ваш компилятор C++, например `g++` или `g++-6`, а `$GBENCH_DIR` – каталог, в который установлена библиотека `benchmark`.

Установка Google Test

Мы будем тестировать корректность очень небольших фрагментов кода. С одной стороны, это просто, потому что каждый фрагмент иллюстрирует одну концепцию или идею. С другой стороны, даже в крупной программной системе за управление ресурсами отвечают небольшие блоки кода. Они могут комбинироваться и образовывать весьма сложный диспетчер ресурсов, но каждый блок выполняет определенную функцию и тестопригоден. Для таких случаев подходит каркас автономного тестирования. Таких каркасов много, выбирать есть из чего; в этой книге мы будем использовать каркас Google Test. Чтобы следить за примерами, понадобится сначала скачать и установить библиотеку (следуйте инструкциям в файле `README`). После установки можно будет компилировать и выполнять примеры. Можете собрать демонстрационные тесты, прилагаемые к библиотеке, чтобы посмотреть, как программа компонуется с Google Test конкретно в вашей системе. Например, на компьютере под управлением Linux команда сборки и выполнения теста `memory1.C` может выглядеть так:

```
$CXX memory1.C -I. -I$GTEST_DIR/include -g -O0 -I. -Wall -Wextra \
  -Werror -pedantic --std=c++14 $GTEST_DIR/lib/libgtest.a \
  $GTEST_DIR/lib/libgtest_main.a -lpthread -lrt -lm -o memory1 && \
  ./memory1
```

Здесь `$CXX` – ваш компилятор C++, например `g++` или `g++-6`, а `$GTEST_DIR` – каталог, в который установлена библиотека Google Test.

Подсчет ресурсов

Каркас автономного тестирования, в частности Google Test, позволяет выполнить код и проверить, совпадает ли результат с ожидаемым. Проверяемый результат может содержать произвольные переменные или выражения, к которым дает доступ тестовая программа. Это определение не распространяется, например, на объем занятой памяти. Поэтому если мы хотим проверить, что ресурсы не пропадают, то должны подсчитывать их.

В следующей простой тестовой фикстуре используется специальный класс ресурса вместо, скажем, ключевого слова `int`. Этот класс оснащен средствами измерения, которые подсчитывают, сколько объектов данного типа было создано и сколько из них живо в настоящий момент:

```
struct object_counter {
    static int count;
```

```
    static int all_count;
    object_counter() { ++count; ++all_count; }
    ~object_counter() { --count; }
};
```

Теперь можно протестировать, правильно ли наша программа управляет ресурсами:

```
TEST(Scoped_ptr, Construct) {
    object_counter::all_count = object_counter::count = 0;
    object_counter* p = new object_counter;
    EXPECT_EQ(1, object_counter::count);
    EXPECT_EQ(1, object_counter::all_count);
    delete p;
    EXPECT_EQ(0, object_counter::count);
    EXPECT_EQ(1, object_counter::all_count);
}
```

В Google Test любой тест реализован в виде **тестовой фикстуры**. Существует несколько типов фикстур; простейшая – автономная тестовая функция, как в примере выше. Выполнение этой простой тестовой программы говорит, что тест прошел:

```
[----------] 1 test from Memory
[ RUN      ] Memory.AcquireRelease
[       OK ] Memory.AcquireRelease (0 ms)
[----------] 1 test from Memory (0 ms total)
```

Для проверки ожидаемых результатов служат макросы `EXPECT_*`, информация обо всех непрошедших тестах выводится на экран. Этот тест проверяет, что после создания и удаления экземпляра типа `object_counter` не остается ни одного объекта и что был сконструирован ровно один объект.

Опасности ручного управления ресурсами

C++ позволяет управлять ресурсами почти на аппаратном уровне, и кто-то где-то действительно должен управлять ими на этом уровне. Последнее справедливо для любого языка, даже высокоуровневого, который не раскрывает таких деталей программистам. Но *где-то* – необязательно в вашей программе! Прежде чем приступить к изучению решений и инструментов, предлагаемых C++ для управления ресурсами, давайте разберемся, какие проблемы возникают, если этими инструментами не пользоваться.

Ручное управление ресурсами чревато ошибками

Первая и самая очевидная опасность ручного управления ресурсами, когда захват и освобождение каждого ресурса требуют явного вызова функции, заключается в том, что легко забыть о необходимости освобождения. Рассмотрим пример:

```
{
    object_counter* p = new object_counter;
        ... еще много строк кода ...
} // Где-то здесь мы, кажется, что-то хотели сделать. Забыл...
```

И вот вам утечка ресурса (в данном случае объекта `object_counter`). Если бы мы сделали такое в автономном тесте, то он бы не прошел:

```
TEST(Memory, Leak1) {
    object_counter::all_count = object_counter::count = 0;
    object_counter* p = new object_counter;
    EXPECT_EQ(1, object_counter::count);
    EXPECT_EQ(1, object_counter::all_count);
    //delete p; // Забыли это сделать
    EXPECT_EQ(0, object_counter::count); // Тест не проходит!
    EXPECT_EQ(1, object_counter::all_count);
}
```

Каркас показывает нам непрошедшие тесты и местоположение отказа:

```
[----------] 2 tests from Memory
[ RUN      ] Memory.AcquireRelease
[       OK ] Memory.AcquireRelease (0 ms)
[ RUN      ] Memory.Leak1
memory.C:33: Failure
Expected equality of these values:
  0
  object_counter::count
    Which is: 1
[  FAILED  ] Memory.Leak1 (0 ms)
[----------] 2 tests from Memory (0 ms total)
```

В реальной программе искать такие ошибки гораздо труднее. Отладчики и контролеры памяти помогают в поиске утечек памяти, но они требуют, чтобы программа реально выполнила дефектный код, поэтому зависят от тестового покрытия.

Утечки ресурсов тоже могут быть гораздо более тонкими и трудными для обнаружения. Рассмотрим следующий код, в котором мы не забыли освободить ресурс:

```
bool process(... some parameters ... ) {
    object_counter* p = new object_counter;
    ... еще много строк кода ...
    delete p;    // ага, вспомнили!
return true;     // все хорошо
}
```

В процессе сопровождения программы была найдена потенциальная ошибка и добавлена соответствующая проверка:

```
bool process(... some parameters ... ) {
    object_counter* p = new object_counter;
      ... еще много строк кода ...
    if (!success) return false; // ошибка, продолжение невозможно
      ... еще строки кода ...
```

```
    delete p;     // все еще тут
    return true;  // все хорошо
}
```

Мы только что внесли тонкую ошибку – теперь ресурсы утекают, только если промежуточное вычисление завершилось аномально и произошел досрочный выход из функции. Если аномалия случается редко, то эта ошибка ускользнет от всех тестов, даже если процесс тестирования регулярно запускает контроль памяти. А допустить такую ошибку проще простого, поскольку изменение может находиться далеко и от конструирования, и от удаления объекта, и ничто в промежуточном контексте не подскажет программисту, что ресурс следует освободить.

Альтернативой утечке ресурса в данном случае является его освобождение. Заметим, что это ведет к дублированию кода:

```
bool process(... some parameters ... ) {
    object_counter* p = new object_counter;
      ... еще много строк кода ...
    if (!success) {
        delete p; return false;   // ошибка, продолжение невозможно
    }
      ... еще строки кода ...
    delete p;     // все еще тут
    return true;  // все хорошо
}
```

Как и при любом дублировании кода, нас подстерегает опасность расхождения. Предположим, что на очередном витке улучшения кода потребовалось несколько объектов `object_counter`, и была выделена память для массива таких объектов:

```
bool process(... some parameters ... ) {
object_counter* p = new object_counter[10]; // теперь это массив
      ... еще много строк кода ...
    if (!success) {
        delete p; return false;   // осталось удаление скаляра
    }
      ... еще строки кода ...
    delete [] p;   // парное удаление массива
    return true;   // все хорошо
}
```

Если оператор `new` изменяется с целью выделения памяти для массива, то нужно изменить и оператор `delete`. Естественно, мы ожидаем найти его в конце функции. А кто знал, что есть еще один посередке? Даже если программист не забыл о ресурсах, в процессе развития и усложнения программы ручное управление становится слишком уж уязвимым для ошибок. А ведь не все ресурсы такие безобидные, как счетчик объектов. Рассмотрим следующий код, в котором выполняется некоторое конкурентное вычисление, поэтому необходимо захватывать и освобождать блокировки-мьютексы. Заме-

тим, что термины **захватить** (acquire) и **освободить** (release), применяемые к блокировкам, наводят на мысль, что блокировка – это некоторый вид ресурса (ресурсом здесь является монопольный доступ к данным, защищенным блокировкой):

```
std::mutex m1, m2, m3;
bool process_concurrently(... some parameters ... ) {
    m1.lock();
    m2.lock();
    ... в этой секции нужны обе блокировки ...
    if (!success) {
        m1.unlock();
        m2.unlock();
        return false;
    } // обе блокировки освобождены
    ... код ...
    m2.unlock(); // монопольный доступ, охраняемый этим мьютексом, больше не нужен,
                 // но m1 еще нужен
    m3.lock();
    if (!success) {
        m1.unlock();
        return false;
    } // разблокировать m2 здесь не нужно
    ... код ...
    m1.unlock(); m3.unlock();
    return true;
}
```

В этом коде есть и дублирование, и расхождение. А еще ошибка – интересно, сможете ли вы ее найти (подсказка – посчитайте, сколько раз разблокирован `m3` и сколько раз встречается предложение `return` после его разблокировки). По мере того как количество ресурсов увеличивается, а их сложность возрастает, такие ошибки встречаются все чаще.

Управление ресурсами и безопасность относительно исключений

Вспомните код в начале предыдущего размера – тот, про который мы сказали, что он правилен, где мы не забыли освободить ресурс. Рассмотрим следующий код:

```
bool process(... some parameters ... ) {
    object_counter* p = new object_counter;
    ... еще много строк кода ...
    delete p;
    return true; // все хорошо
}
```

У меня для вас плохие новости – этот код тоже может оказаться неправильным. Если в коде, который не показан, возникнет исключение, то `delete p` никогда не выполнится:

```
bool process(... some parameters ... ) {
    object_counter* p = new object_counter;
    ... еще много строк кода ...
    if (!success) // исключительные обстоятельства, продолжение невозможно
        throw process_exception();
    ... еще строки кода ...
    delete p;     // ничего этого не будет, если возбуждено исключение!
    return true;
}
```

Это очень похоже на проблему досрочного возврата, только еще хуже – исключение может возбудить любой код, вызываемый из функции `process()`. Исключение может быть даже добавлено позже в некоторый код, который вызывает `process()`, при этом сама эта функция не изменится. Работала-работала, а потом в один прекрасный день перестала.

Если мы не изменим подход к управлению ресурсами, то единственное решение – воспользоваться блоками `try…catch`:

```
bool process(... some parameters ... ) {
    object_counter* p = new object_counter;
    try {
        ... еще много строк кода ...
    if (!success) // исключительные обстоятельства, продолжение невозможно
        throw process_exception();
    ... еще строки кода ...
    } catch ( ... ) {
        delete p; // для случая исключения
    }
    delete p;     // для нормального случая
    return true;
}
```

Очевидная проблема здесь – очередное дублирование кода, а также расползание блоков `try…catch` буквально по всей программе. Хуже того, этот подход не масштабируется на случай управления несколькими ресурсами или хотя бы на случай управления чем-то более сложным, чем один захват с последующим освобождением:

```
std::mutex m;
bool process(... some parameters ... ) {
    m.lock();
    object_counter* p = new object_counter;
                    // Проблема #1: конструктор может возбудить исключение
    try {
        ... еще много строк кода ...
        m.unlock();  // конец критической секции
        ... еще строки кода ...
    } catch ( ... ) {
        delete p;    // OK, это всегда нужно
        m.unlock();  // а это нужно? Может быть: зависит от того, где возникло исключение!
        throw;       // повторно возбудить исключение, чтобы его мог обработать клиент
```

```
    }
    delete p;        // для нормального случая, разблокировать мьютекс не нужно
    return true;
}
```

Теперь мы даже не можем решить, нужно ли в блоке `catch` освобождать мьютекс или нет, – все зависит от того, возникло исключение до или после операции `unlock()` в нормальном потоке управления без исключений. Кроме того, конструктор `object_counter` тоже мог бы возбудить исключение (не тот простенький, что мы видели до сих пор, а более сложный, в который он мог бы со временем эволюционировать). Это случилось бы вне блока `try...catch`, и мьютекс никогда не был бы разблокирован.

Теперь должно быть ясно, что нам необходимо совершенно другое решение проблемы управления ресурсами, заплатами тут не обойдешься. В следующем разделе мы обсудим паттерн, который стал **золотым стандартом** управления ресурсами в C++.

Идиома RAII

В предыдущем разделе мы видели, что бессистемные попытки управления ресурсами постепенно становятся ненадежными, подвержены ошибкам и в конце концов не приводят к желаемому результату. Нам необходимо, чтобы любой захват ресурса гарантированно сопровождался его освобождением и чтобы эти два действия происходили соответственно до и после секции кода, в которой ресурс используется. В C++ такой способ обрамления участка кода парой действий известен как паттерн Обрамляющее выполнение (Execute Around).

Дополнительные сведения см. в статье Kevlin Henney «C++ Patterns – Executing Around Sequences» по адресу **http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/ExecutingAroundSequences.pdf**.

В контексте применения к управлению ресурсами этот паттерн более широко известен под названием «**захват ресурса есть инициализация**» (**RAII**).

RAII в двух словах

Основная идея RAII очень проста – в C++ существует один вид функций, который гарантированно вызывается автоматически, и это деструктор объекта, созданного в стеке, или объекта, являющегося членом данных другого объекта (в последнем случае эта гарантия действует, только если уничтожается сам объемлющий объект). Если бы мы могли связать освобождение ресурса с деструктором такого объекта, то про освобождение никак нельзя было бы забыть или по ошибке пропустить его. Понятно, что если освобождение ресурса осуществляется деструктором, то захват должен осуществляться конструктором в процессе инициализации объекта. Отсюда и расшифровка акронима RAII.

Посмотрим, как это работает в простейшем случае выделения памяти оператором new. Прежде всего нам нужен класс, который можно инициализировать указателем на вновь созданный объект и деструктор которого удаляет этот объект:

```
template <typename T>
class raii {
    public:
    explicit raii(T* p) : p_(p) {}
    ~raii() { delete p_; }
    private:
    T* p_;
};
```

Теперь очень легко гарантировать, что удаление никогда не будет позабыто, а убедиться в том, что это работает, как и ожидалось, поможет тест с использованием object_counter:

```
TEST(RAII, AcquireRelease) {
    object_counter::all_count = object_counter::count = 0;
    {
        raii<object_counter> p(new object_counter);
        EXPECT_EQ(1, object_counter::count);
        EXPECT_EQ(1, object_counter::all_count);
    } // нет нужды в delete p, это делается автоматически
    EXPECT_EQ(0, object_counter::count);
    EXPECT_EQ(1, object_counter::all_count);
}
```

Заметим, что в C++17 тип шаблона класса выводится из конструктора, и мы можем просто написать:

```
raii p(new object_counter);
```

Разумеется, хотелось бы использовать новый объект не только для того, чтобы создавать и удалять его, поэтому было бы неплохо иметь доступ к указателю, хранящемуся в RAII-объекте. Нет никаких причин предоставлять такой доступ способом, отличным от синтаксиса стандартного указателя, поэтому наш RAII-объект сам становится своего рода указателем:

```
template <typename T>
class scoped_ptr {
    public:
    explicit scoped_ptr(T* p) : p_(p) {}
    ~scoped_ptr() { delete p_; }
    T* operator->() { return p_; }
    const T* operator->() const { return p_; }
    T& operator*() { return *p_; }
    const T& operator*() const { return *p_; }
    private:
    T* p_;
};
```

Этот указатель можно использовать для автоматического удаления объекта, на который он указывает, при выходе из области видимости (scope – отсюда и название scoped_ptr):

```
TEST(Scoped_ptr, AcquireRelease) {
    object_counter::all_count = object_counter::count = 0;
    {
        scoped_ptr<object_counter> p(new object_counter);
        EXPECT_EQ(1, object_counter::count);
        EXPECT_EQ(1, object_counter::all_count);
    }
    EXPECT_EQ(0, object_counter::count);
    EXPECT_EQ(1, object_counter::all_count);
}
```

Деструктор вызывается, когда происходит выход из области видимости scoped_ptr. И не важно, как произошел выход – посредством досрочного возврата из функции, в результате выполнения предложения break или continue в цикле либо вследствие исключения – любой вариант обрабатывается одинаково, и утечки не происходит. Конечно, это можно подтвердить тестами:

```
TEST(Scoped_ptr, EarlyReturnNoLeak) {
    object_counter::all_count = object_counter::count = 0;
    do {
        scoped_ptr<object_counter> p(new object_counter);
        break;
    } while (false);
    EXPECT_EQ(0, object_counter::count);
    EXPECT_EQ(1, object_counter::all_count);
}

TEST(Scoped_ptr, ThrowNoLeak) {
    object_counter::all_count = object_counter::count = 0;
    try {
        scoped_ptr<object_counter> p(new object_counter);
        throw 1;
    } catch ( ... ) {
    }
    EXPECT_EQ(0, object_counter::count);
    EXPECT_EQ(1, object_counter::all_count);
}
```

Все тесты проходят, что подтверждает отсутствие утечек:

```
[----------] 3 tests from Scoped_ptr
[ RUN      ] Scoped_ptr.AcquireRelease
[       OK ] Scoped_ptr.AcquireRelease (0 ms)
[ RUN      ] Scoped_ptr.EarlyReturnNoLeak
[       OK ] Scoped_ptr.EarlyReturnNoLeak (0 ms)
[ RUN      ] Scoped_ptr.ThrowNoLeak
[       OK ] Scoped_ptr.ThrowNoLeak (0 ms)
[----------] 3 tests from Scoped_ptr (0 ms total)
```

Объект scoped_ptr можно использовать и в качестве члена данных другого класса, который выделяет дополнительную память и должен освободить ее в момент уничтожения:

```
class A {
    public:
    A(object_counter* p) : p_(p) {}
    private:
    scoped_ptr<object_counter> p_;
};
```

Таким образом, нам не нужно удалять объект вручную в деструкторе класса A. Более того, если каждый член данных класса A так же заботится о себе, то в классе A даже не нужен явный деструктор.

Всякий, кто знаком с C++11, узнает в нашем классе scoped_ptr крайне упрощенный вариант стандартного класса std::unique_ptr, который можно использовать для той же цели. Как и следовало ожидать, стандартная реализация уникального указателя умеет гораздо больше – и не без причины. Некоторые из этих причин мы рассмотрим ниже в этой главе.

И последнее, на что стоит обратить внимание, – производительность. C++ стремится, чтобы все абстракции по возможности имели нулевые издержки. В данном случае мы обертываем простой указатель объектом интеллектуального указателя. Однако компилятору не нужно генерировать дополнительных машинных команд; обертка всего лишь понуждает компилятор сгенерировать код, который он все равно сгенерировал бы в правильно написанной программе. Мы можем с помощью простого эталонного теста подтвердить, что конструирование-удаление и разыменование нашего scoped_ptr (или std::unique_ptr, если на то пошло) занимают ровно столько же времени, сколько соответствующие операции с простым указателем. Например, следующий эталонный микротест (написанный с применением библиотеки Google Benchmark) сравнивает производительность разыменования всех трех типов указателя:

```
void BM_rawptr_dereference(benchmark::State& state) {
    int* p = new int;
    for (auto _ : state) {
        REPEAT(benchmark::DoNotOptimize(*p);)
    }
    delete p;
    state.SetItemsProcessed(32*state.iterations());
}
void BM_scoped_ptr_dereference(benchmark::State& state) {
    scoped_ptr<int> p(new int);
    for (auto _ : state) {
        REPEAT(benchmark::DoNotOptimize(*p);)
    }
    state.SetItemsProcessed(32*state.iterations());
}
void BM_unique_ptr_dereference(benchmark::State& state) {
```

```
    std::unique_ptr<int> p(new int);
    for (auto _ : state) {
        REPEAT(benchmark::DoNotOptimize(*p);)
    }
    state.SetItemsProcessed(32*state.iterations());
}
BENCHMARK(BM_rawptr_dereference);
BENCHMARK(BM_scoped_ptr_dereference);
BENCHMARK(BM_unique_ptr_dereference);
BENCHMARK_MAIN();
```

Этот тест показывает, что интеллектуальные указатели действительно не влекут за собой никаких накладных расходов:

```
-------------------------------------------------------------------
Benchmark                        Time           CPU Iterations
-------------------------------------------------------------------
BM_rawptr_dereference          10 ns         10 ns   70042145   3.09321G items/s
BM_scoped_ptr_dereference      10 ns         10 ns   72095679   3.11026G items/s
BM_unique_ptr_dereference      10 ns         10 ns   71510079   3.04359G items/s
```

Мы достаточно подробно рассмотрели применение RAII для управления памятью. Но программе на C++ приходится управлять и вести учет также другим ресурсам, поэтому пора расширить наше представление о RAII.

RAII для других ресурсов

В акрониме RAII речь идет о *ресурсах*, а не о *памяти*, и это не случайно – точно такой же подход применим и к другим ресурсам. Для каждого типа ресурсов нам нужен специальный объект, хотя благодаря обобщенному программированию и лямбда-выражениям объем кода можно уменьшить (подробнее об этом мы поговорим в главе 11). Ресурс захватывается в конструкторе и освобождается в деструкторе. Заметим, что есть две немного различающиеся разновидности RAII. Первую мы уже видели – собственно захват ресурса производится в момент инициализации, но вне конструктора RAII-объекта.

Конструктор просто запоминает описатель (например, указатель), который образовался в результате захвата. Именно так был устроен наш класс `scoped_ptr` – выделение памяти и конструирование объекта производились вне конструктора объекта `scoped_ptr`, но все-таки на стадии его инициализации. Вторая разновидность – когда конструктор RAII-объекта сам захватывает ресурс. Как это работает, разберем на примере RAII-объекта, управляющего мьютексами:

```
class mutex_guard {
    public:
    explicit mutex_guard(std::mutex& m) : m_(m) { m_.lock(); }
    ~mutex_guard() { m_.unlock(); }
    private:
    std::mutex& m_;
};
```

Здесь конструктор класса mutex_guard сам захватывает ресурс – в данном случае получает монопольный доступ к разделяемым данным, защищенный мьютексом. Деструктор освобождает этот ресурс. И снова паттерн делает невозможной **утечку** блокировок (т. е. выход из области видимости без освобождения мьютекса), например при возникновении исключения:

```
std::mutex m;
TEST(Scoped_ptr, ThrowNoLeak) {
    try {
        mutex_guard lg(m);
        EXPECT_FALSE(m.try_lock()); // ожидаем, что уже захвачен
        throw 1;
    } catch ( ... ) {
    }
    EXPECT_TRUE(m.try_lock());      // ожидаем, что освобожден
    m.unlock();                     // try_lock() захватывает, отменить
}
```

В этом тесте мы проверяем, захвачен ли мьютекс, вызывая функцию std::mutex::try_lock() – мы не можем вызвать lock(), если мьютекс уже захвачен, поскольку это приведет к взаимоблокировке. Вызов try_lock() позволяет проверить состояние мьютекса без риска взаимоблокировки (только надо не забыть освободить мьютекс, если try_lock() завершилась успешно).

Стандарт предлагает RAII-объект для блокировки мьютекса, std::lock_guard. Он используется примерно так же, только применим к мьютексам любого типа, имеющим функции-члены lock() и unlock().

Досрочное освобождение

Область видимости, образуемая телом функции или цикла, не всегда совпадает с желательным сроком удержания ресурса. Если мы не хотим захватывать ресурс в самом начале области видимости, то тут все просто – RAII-объект можно создать в любом месте, а не только в начале. Ресурс не захватывается, пока RAII-объект не будет сконструирован:

```
void process(...) {
    ... сделать все, что не требует монопольного доступа ...
    mutex_guard lg(m);      // теперь заблокировать
    ... работа с разделяемыми данными, защищенными мьютексом ...
} // здесь мьютекс освобождается
```

Однако освобождение все равно происходит в конце области видимости функции. А что, если мы хотим заблокировать мьютекс в коротком участке кода внутри функции? Самое простое решение – создать дополнительную область видимости:

```
void process(...) {
    ... сделать все, что не требует монопольного доступа ...
    {
        mutex_guard lg(m); // теперь заблокировать
```

```
        ... работа с разделяемыми данными, защищенными мьютексом ...
    } // здесь мьютекс освобождается
    ... продолжить работу в немонопольном режиме ...
}
```

Этот прием может показаться удивительным, если вы никогда не видели его раньше, но в C++ любую последовательность предложений можно заключить в фигурные скобки. При этом создается новая область видимости с собственными локальными переменными. В отличие от фигурных скобок, окружающих тело цикла или условного предложения, единственная цель такой области видимости – управление временем жизни этих локальных переменных. В программе, где RAII активно используется, бывает много таких областей видимости, окружающих переменные с различным временем жизни, более коротким, чем у всей функции или цикла. Это делает код понятнее, потому что сразу видно, что некоторые переменные не будут использоваться после определенного места, так что читателю не нужно просматривать остаток кода в поисках возможных упоминаний этих переменных. Кроме того, пользователь не сможет по ошибке добавить ссылку на такую переменную, если было ясно высказано намерение **ограничить** срок ее жизни и больше никогда не использовать.

А что, если ресурс можно освободить досрочно, но лишь при выполнении некоторых условий? Одна из возможностей – как и раньше, ограничить использование ресурса областью видимости и выйти из этой области, когда ресурс больше не нужен. Для выхода из области видимости было бы удобно воспользоваться предложением `break`. Обычно именно это делают с помощью цикла `do...once`:

```
void process(...) {
    ... сделать все, что не требует монопольного доступа ...
    do {                          // на самом деле это не цикл
        mutex_guard lg(m);        // теперь заблокировать
        ... работа с разделяемыми данными, защищенными мьютексом ...
        if (work_done) break;  // выйти из области видимости
        ... еще поработать с разделяемыми данными ...
    } while (false);              // здесь мьютекс освобождается
    ... продолжить работу в немонопольном режиме ...
}
```

Однако этот подход работает не всегда (мы можем захотеть освободить ресурсы, не уничтожая других локальных переменных в той же области видимости), к тому же код становится менее понятным из-за усложнения потока управления. Но не поддавайтесь порыву решить задачу путем динамического выделения RAII-объекта оператором `new`! Это сводит на нет саму идею RAII, поскольку теперь вы должны помнить о вызове оператора `delete`. Мы можем улучшить объекты управления ресурсами, добавив активируемое клиентом освобождение в дополнение к автоматическому освобождению в деструкторе. Нужно только позаботиться о том, чтобы один и тот же ресурс не был освобожден дважды. Рассмотрим следующий пример, в котором используется класс `scoped_ptr`:

```
template <typename T>
class scoped_ptr {
    public:
    explicit scoped_ptr(T* p) : p_(p) {}
    ~scoped_ptr() { delete p_; }
      ...
    void reset() {
        delete p_; p_ = nullptr;   // досрочно освободить ресурс
    }
    private:
    T* p_;
};
```

После вызова `reset()` объект, управляемый объектом `scoped_ptr`, удаляется, а указатель на него внутри `scoped_ptr` сбрасывается в ноль. Заметим, что нам не пришлось вводить дополнительное условие в деструктор, потому что удаление нулевого указателя разрешено стандартом – при этом не происходит ничего. Ресурс освобождается только один раз – либо явно вызовом `reset()`, либо неявно в конце области видимости, содержащей объект `scoped_ptr`.

В классе `mutex_guard` мы не можем по одному лишь мьютексу понять, состоялось досрочное освобождение или нет, поэтому необходим дополнительный член, который следит за этим:

```
class mutex_guard {
    public:
    explicit mutex_guard(std::mutex& m) :
        m_(m), must_unlock_(true) { m_.lock(); }
    ~mutex_guard() { if (must_unlock_) m_.unlock(); }
    void reset() { m_.unlock(); must_unlock_ = false; }
    private:
    std::mutex& m_;
    bool must_unlock_;
};
```

Теперь мы можем написать следующий тест, который проверяет, что мьютекс освобожден только один раз и в нужное время:

```
TEST(mutex_guard, Reset) {
    {
        mutex_guard lg(m);
        EXPECT_FALSE(m.try_lock());
        lg.reset();
        EXPECT_TRUE(m.try_lock()); m.unlock();
    }
    EXPECT_TRUE(m.try_lock()); m.unlock();
}
```

Стандартный класс `std::unique_ptr` поддерживает метод `reset()`, а класс `std::lock_guard` – нет, поэтому если вы захотите освободить мьютекс досрочно, то придется написать собственный защитный объект. По счастью, защита блокировки – довольно простой класс, но перед тем как приступать к его

написанию, прочитайте эту главу до конца, поскольку тут есть несколько подводных камней.

Отметим, что метод `reset()` класса `std::unique_ptr` на самом деле не только удаляет объект досрочно. Его можно использовать также для того, чтобы **сбросить** указатель, перенаправив его на новый объект и одновременно удалив старый. Работает это примерно так (настоящая реализация несколько сложнее из-за дополнительной функциональности уникального указателя):

```
template <typename T>
class scoped_ptr {
    public:
    explicit scoped_ptr(T* p) : p_(p) {}
    ~scoped_ptr() { delete p_; }
        ...
    void reset(T* p = nullptr) {
        delete p_; p_ = p; // сбросить указатель
    }
    private:
    T* p_;
};
```

Отметим, что этот код некорректен, если `scoped_pointer` сбрасывает сам себя (т. е. методу `reset()` передается то же значение, которое хранится в `p_`). Можно было бы проверить это условие и при его выполнении ничего не делать; отметим попутно, что стандарт не настаивает на подобной проверке для `std::unique_ptr`.

Аккуратная реализация RAII-объектов

Очевидно, что крайне важно, чтобы объекты управления ресурсами не портили ресурсы, которые они призваны защищать. К сожалению, простые RAII-объекты, которые мы писали до сих пор, имеют несколько вопиющих недостатков.

Первая проблема возникает, когда кто-то старается скопировать эти объекты. Каждый из рассмотренных в этой главе RAII-объектов отвечает за управление одним-единственным экземпляром своего ресурса, и тем не менее ничто не мешает нам скопировать данный объект:

```
scoped_ptr<object_counter> p(new object_counter);
scoped_ptr<object_counter> p1(p);
```

Этот код вызывает копирующий конструктор по умолчанию, который просто копирует объект побитово; в нашем случае в `object_counter` копируется указатель. Теперь мы имеем два RAII-объекта, каждый из которых управляет одним и тем же ресурсом. В конечном итоге будет вызвано два деструктора, и оба попытаются удалить один и тот же объект. Второе удаление является неопределенным поведением (если нам очень повезет, то программа в этом месте «грохнется»).

Присваивание RAII-объектов столь же проблематично:

```
scoped_ptr<object_counter> p(new object_counter);
scoped_ptr<object_counter> p1(new object_counter);
p = p1;
```

Оператор присваивания по умолчанию тоже копирует объект побитово. И снова два RAII-объекта удалят один и тот же управляемый объект. Не менее печален тот факт, что у нас нет ни одного RAII-объекта, который управлял бы вторым экземпляром `object_counter`: старый указатель, хранившийся в `p1`, потерян, а нового указателя на этот объект не появилось, так что удалить его нет никакой возможности.

С классом `mutex_guard` дело обстоит не лучше – попытка скопировать его порождает двух охранников, каждый их которых разблокирует один и тот же мьютекс. Вторая разблокировка будет произведена для мьютекса, который не заблокирован (по крайней мере, в вызывающем потоке), что, согласно стандарту, является неопределенным поведением. Впрочем, хотя бы присваивание объекту `mutex_guard` невозможно, потому что оператор присваивания по умолчанию не генерируется для объектов, содержащих члены-ссылки.

Вероятно, вы заметили, что проблема связана с копирующим конструктором по умолчанию и оператором присваивания по умолчанию. Означает ли это, что мы должны реализовать их самостоятельно? И что они должны делать? Для каждого сконструированного объекта деструктор должен вызываться только один раз: мьютекс нельзя разблокировать, после того как он уже разблокирован. Это наводит на мысль, что RAII-объекты вообще нельзя копировать и следует запретить копирование и присваивание:

```
template <typename T>
class scoped_ptr {
    public:
    explicit scoped_ptr(T* p) : p_(p) {}
    ~scoped_ptr() { delete p_; }
        ...
    private:
    T* p_;
    scoped_ptr(const scoped_ptr&) = delete;
    scoped_ptr& operator=(const scoped_ptr&) = delete;
};
```

Возможность пометки функций-членов признаком `delete` появилась в C++11, прежде нужно было объявить обе функции как `private`, но не определять их:

```
template <typename T>
class scoped_ptr {
    ...
    private:
    scoped_ptr(const scoped_ptr&); // {} отсутствуют - определения нет!
    scoped_ptr& operator=(const scoped_ptr&);
};
```

Существуют RAII-объекты, допускающие копирование. Это объекты управления, ведущие подсчет ссылок, т. е. подсчитывающие, сколько копий RAII-объекта существует для одного экземпляра управляемого ресурса. При удалении последнего RAII-объекта следует освободить сам ресурс. Более подробно мы обсуждали управление разделяемыми ресурсами в главе 3.

Для перемещающего конструктора и оператора присваивания действуют другие соображения. Перемещение объекта не нарушает предположения о том, что существует только один RAII-объект, владеющий данным ресурсом. Просто владельцем становится другой RAII-объект. Во многих случаях, например в случае охранника мьютексов, перемещать RAII-объект не имеет смысла (и действительно, в стандарте класс `std::lock_guard` не является перемещаемым). Перемещение уникального указателя возможно и осмысленно в некоторых контекстах, и этот вопрос также обсуждался в главе 3.

Однако для `scoped_pointer` перемещение было бы нежелательно, поскольку позволяет продлить время существования управляемого объекта за пределы области видимости, в которой он был создан. Отметим, что нам нет нужды удалять перемещающий конструктор или оператор присваивания, если мы уже удалили копирующие (хотя никакого вреда в этом тоже нет). С другой стороны, указатель `std::unique_ptr` – перемещаемый объект, а это значит, что использование его в роли интеллектуального указателя, защищающего область видимости, не дает такой же защиты, потому что ресурс можно вывести из этой области. Впрочем, если требуется указатель с ограниченной областью видимости, то можно очень просто приспособить для этой цели `std::unique_ptr` – нужно лишь объявить его объектом типа `const std::unique_ptr`:

```
std::unique_ptr<int> p;
{
    // Нельзя перемещать за пределы области видимости
    std::unique_ptr<int> q(new int);
    q = std::move(p);       // но именно это здесь и происходит

    // А это настоящий ограниченный указатель, его никуда не переместишь
    const std::unique_ptr<int> r(new int);
    q = std::move(r); // Does not compile
}
```

До сих пор мы защищали наши RAII-объекты от дублирования и утраты ресурсов. Но есть еще один вид ошибок при управлении ресурсами, который мы пока не рассматривали. Кажется очевидным, что ресурс следует освобождать способом, соответствующим его захвату. И тем не менее ничто не защищает наш объект `scoped_ptr` от такого несоответствия между конструированием и удалением:

```
scoped_ptr<int> p(new int[10]);
```

Проблема в том, что мы выделили память для нескольких объектов, применив вариант оператора `new` для массивов, значит, и удалять его следует опера-

тором delete для массивов, т. е. внутри деструктора scoped_ptr нужно вызывать delete [] p_, а не delete p_, как мы делали до сих пор.

Вообще, любой RAII-объект, который принимает на этапе инициализации описатель ресурса, а не захватывает ресурс сам (как mutex_guard), должен каким-то образом гарантировать, что ресурс будет освобожден **правильно** – способом, соответствующим его захвату. Очевидно, что в общем случае это невозможно. На самом деле это невозможно сделать автоматически даже в простом случае несоответствия выделения массива и удаления скаляра (в этом отношении std::unique_ptr ничем не лучше нашего scoped_ptr, хотя такие средства, как функция std::make_unique, уменьшают шанс сделать ошибку).

Вообще говоря, либо RAII-класс проектируется, так чтобы ресурсы удалялись одним определенным способом, либо пользователь должен указать, как освобождать ресурс. Первое, безусловно, проще и во многих случаях эффективнее. В частности, если RAII-класс сам захватывает ресурс, как, например, mutex_guard, то он, конечно, знает, как его освободить. Даже для scoped_ptr было бы нетрудно написать два варианта, scoped_ptr и scoped_array, причем второй будет предназначен для объектов, выделенных оператором new для массивов. Более общая версия RAII-класса параметризуется не только типом ресурса, но и вызываемым объектом для освобождения этого типа; обычно его ликвидатором (deleter). Ликвидатор может быть указателем на функцию, указателем на функцию-член или объектом, в котором определена функция operator(), – в общем, чем угодно, что можно вызывать как функцию. Отметим, что ликвидатор необходимо передать RAII-объекту в его конструкторе и сохранить внутри RAII-объекта, из-за чего размер объекта увеличивается. Кроме того, тип ликвидатора – это параметр шаблона RAII-класса, если только он не стерт из типа RAII (этот вопрос подробно рассматривается в главе 6).

Недостатки RAII

Честно говоря, у идиомы RAII нет существенных недостатков. Это самая распространенная идиома управления ресурсами в C++. Единственная проблема, о которой следует помнить, связана с исключениями. Освобождение ресурса, как и любая операция, может завершиться неудачно. В C++ обычный способ сигнализировать об ошибке – возбудить исключение. Если это нежелательно, то функция возвращает код ошибки. В случае RAII ни то, ни другое невозможно.

Легко понять, почему не годятся коды ошибок – деструктор вообще ничего не возвращает. Нельзя также записать код ошибки в какой-то член объекта, содержащий состояние, поскольку объект вот-вот будет уничтожен – и все его члены вместе с ним, как и любые локальные переменные в области видимости, где объявлен RAII-объект. Сохранить код ошибки для последующего анализа можно было бы только в какой-нибудь глобальной переменной или, по крайней мере, в переменной из объемлющей области видимости. Если ничего другого не остается, то можно поступить и так, но решение уж очень неэлегантное и чреватое ошибками. Это именно та проблема, какую C++ пытался решить,

вводя исключения, поскольку распространение кодов ошибок вручную слишком ненадежно.

Но если исключения – решение проблемы уведомления об ошибках в C++, то почему бы ими не воспользоваться и здесь? Обычный ответ – *потому что деструкторы не вправе возбуждать исключения*. Смысл передан правильно, но у этого ограничения есть дополнительные нюансы. Прежде всего до выхода C++11 деструкторы технически могли возбуждать исключения, но это исключение распространилось бы вверх по стеку и (хочется надеяться) в конечном итоге было бы перехвачено и обработано. В C++11 все деструкторы имеют квалификатор `noexcept`, если только не объявлены явно как `noexcept(false)`. Если функция, объявленная как `noexcept`, возбуждает исключение, программа немедленно завершается.

Так что в действительности деструкторы в C++11 не могут возбуждать исключения, если это явно не разрешено. Но что плохого в возбуждении исключения деструктором? Если деструктор выполняется, потому что объект был удален или потому что управление дошло до конца области видимости стекового объекта, то ничего страшного в этом нет. *Беда* случается, если управление не дошло до конца области видимости обычным порядком, а деструктор выполнился из-за того, что где-то уже возникло исключение. В C++ два исключения не могут распространяться одновременно. Если такое происходит, то программа немедленно завершается (заметим, что деструктор может возбудить и сам же перехватить исключение, проблемы не возникнет, коль скоро исключение не выходит за пределы деструктора). Разумеется, при написании программы заранее неизвестно, когда некоторая функция, вызванная кем-то в какой-то области видимости, возбудит исключение. Если при освобождении ресурса возникает исключение и RAII-объекту разрешено распространять это исключение за пределы своего деструктора, то программа аварийно завершится, если деструктор будет вызван в процессе обработки исключения. Единственный безопасный способ – никогда не выпускать исключения из деструктора. Это не означает, что функция, освобождающая ресурс, не может сама возбуждать исключения. Но если она это сделает, то деструктор RAII-объекта должен его перехватить:

```
class raii {
    ...
    ~raii() {
        try {
            release_resource(); // может возбуждать исключения
        } catch ( ... ) {
            ... обработать исключение, НЕ возбуждать заново ...
        }
    }
};
```

Так что мы опять остаемся без возможности сигнализировать об ошибке, произошедшей во время освобождения ресурса, – исключение было возбуждено, но мы вынуждены перехватить его и не дать распространиться.

Насколько серьезна эта проблема? Умеренно. Во-первых, освобождение памяти – а это ресурс, который чаще всего нуждается в управлении, – не возбуждает исключений вовсе. Обычно память освобождается не напрямую, а в результате удаления объекта. Но вспомним, что деструкторы не должны возбуждать исключений, чтобы весь процесс освобождения памяти посредством удаления объекта также не возбуждал исключений. В этот момент читатель в поисках контрпримера мог бы посмотреть, что говорит стандарт о ситуации, когда разблокировка мьютекса завершается неудачно (это вынудило бы деструктор класса `std::lock_guard` как-то обрабатывать ошибку). Ответ удивительный и назидательный – разблокировка мьютекса не может возбуждать исключений, но если происходит ошибка, то имеет место неопределенное поведение. Это не случайность – мьютекс проектировался для совместной работы с RAII-объектом. Это общий подход C++ к освобождению ресурса: если освобождение неудачно, то исключение не должно возбуждаться или, по крайней мере, ему не разрешено распространяться. Его можно перехватить и, например, запротоколировать, но вызывающая программа, вообще говоря, ничего не будет знать об ошибке, быть может, расплачиваясь за это неопределенным поведением.

Резюме

Изучив эту главу, читатель должен хорошо понимать опасности бессистемного подхода к управлению ресурсами. К счастью, мы познакомились с самой широко распространенной идиомой управления ресурсами в C++ – RAII. В соответствии с ней каждым ресурсом владеет некоторый объект. Конструирование (или инициализация) этого объекта приводит к захвату ресурса, а удаление объекта – к его освобождению. Мы видели, как использование RAII решает проблемы управления ресурсами – утечку ресурсов, непреднамеренное разделение ресурсов и неправильное их освобождение. Мы узнали об основах написания кода, безопасного относительно исключений, по крайней мере в том, что касается утечки ресурсов и других ошибок их обработки. Писать RAII-объекты достаточно просто, но следует помнить о нескольких подводных камнях. Наконец, мы рассмотрели осложнения, возникающие на стыке RAII с обработкой ошибок.

RAII – идиома управления ресурсами, но ее можно рассматривать и как способ абстрагирования: сложные ресурсы скрываются за простыми описателями. В следующей главе мы познакомимся еще с одной идиомой абстрагирования – стиранием типа: вместо сложных объектов мы будем скрывать сложные типы.

Вопросы

- Что понимается под *ресурсами*, которыми может управлять программа?
- Каковы основные проблемы управления ресурсами в программе на C++?
- Что такое RAII?

- Как RAII решает проблему утечки ресурсов?
- Как RAII решает проблему висячих описателей ресурсов?
- Какие RAII-объекты предоставляет стандартная библиотека C++?
- О каких предосторожностях следует помнить при написании RAII-объектов?
- Что происходит, когда освобождение ресурса завершается неудачно?

Для дальнейшего чтения

- https://www.packtpub.com/application-development/expert-c-programming.
- https://www.packtpub.com/application-development/c-data-structuresand-algorithms.
- https://www.packtpub.com/application-development/rapid-c-video.

Глава 6

Что такое стирание типа

Стирание типа многим кажется таинственной и загадочной техникой программирования. Это не исключительная особенность C++ (большинство пособий по стиранию типа написано для Java). Цель этой главы – сбросить покров тайны и объяснить, что такое стирание типа и как им пользоваться в C++.

В этой главе рассматриваются следующие вопросы:

- что такое стирание типа;
- как реализуется стирание типа;
- какие соображения – в части дизайна и производительности – следует принять во внимание при проектировании стирания типа.

Технические требования

Примеры кода: **https://github.com/PacktPublishing/Hands-On-Design-Patterns-with-CPP/tree/master/Chapter06**.

Библиотека Google Benchmark: **https://github.com/google/benchmark** (см. главу 5).

Что такое стирание типа?

Вообще говоря, стирание типа – это техника программирования, позволяющая исключить из программы явную информацию о типе. Это своего рода абстракция, гарантирующая, что программа явно не зависит от некоторых типов данных.

Это определение, хотя и абсолютно правильное, одновременно окружает стирание типа завесой тайны. А все из-за порочного круга – оно рождает призрачную надежду создать то, что, на первый взгляд, кажется невозможным: программу на строго типизированном языке, в которой не используются фактические типы. Как такое может быть? Ну конечно же, путем абстрагирования типа! А раз так, то и надежда, и тайна живут и не умирают.

Трудно вообразить программу, в которой типы используются без явного упоминания (по крайней мере, программу на C++; конечно, есть другие языки, где все типы становятся окончательно известны только на этапе выполнения).

Поэтому мы сначала продемонстрируем стирание типа на примере. Это должно дать интуитивное представление о нем, которое мы в последующих разделах разовьем и формализуем. Наша цель – повысить уровень абстракции; вместо написания кода, ориентированного на конкретный тип, или, быть может, нескольких его вариантов для разных типов мы сможем написать один вариант, который будет более абстрактным и станет выражать некую концепцию – например, вместо написания функции, интерфейс которой выражает концепцию *сортировки массива целых чисел*, мы напишем более абстрактную функцию *сортировки произвольного массива*.

Стирание типа на примере

Мы собираемся подробно объяснить, что такое стирание типа и как оно достигается в C++. Но сначала посмотрим, как выглядит программа, из которой исключена явная информация о типе.

Начнем с очень простого примера использования уникального указателя `std::unique_ptr`:

```
std::unique_ptr<int> p(new int(0));
```

Это владеющий указатель (см. главу 3) – сущность, содержащая этот указатель, например объект или области видимости функции, также контролирует время жизни памяти, выделенной под целое число, и отвечает за ее удаление. Самого удаления в коде не видно, оно происходит, когда удаляется указатель `p` (например, когда он покидает область видимости). Способ, которым достигается это удаление, тоже не виден явно – по умолчанию `std::unique_ptr` удаляет объект, которым владеет, с помощью оператора `delete`, а точнее посредством вызова функции `std::default_delete`, которая, в свою очередь, вызывает оператор `delete`. А что, если мы не хотим использовать стандартный оператор `delete`? Например, объекты могли быть выделены из нашей собственной кучи:

```
class MyHeap {
    public:
    ...
    void* allocate(size_t size);
    void deallocate(void* p);
    ...
};
void* operator new(size_t size, MyHeap* heap) {
    return heap->allocate(size);
}
```

Выделение не составляет проблемы благодаря перегруженному оператору `new`:

```
MyHeap heap;
std::unique_ptr<int> p(new(&heap) int(0));
```

В этой синтаксической конструкции вызывается функция `operator new` с двумя аргументами, первый из которых задает размер и добавляется компилято-

ром, а второй указывает на кучу. Поскольку мы объявили такой перегруженный вариант, то он и будет вызван и вернет память, выделенную из кучи. Но мы ничего не сделали, чтобы изменить способ удаления объекта. Будет вызвана обычная функция operator delete, и она попытается вернуть в глобальную кучу память, которая из нее не выделялась. Результатом, скорее всего, станет повреждение памяти и, возможно, аварийное завершение. Мы могли бы определить функцию operator delete с таким же дополнительным аргументом, но никакой пользы это не принесет – в отличие от оператора new, оператор delete не принимает никаких аргументов (часто в программах все равно встречается так определенная функция operator delete, и это правильно, но не имеет отношения ни к какой функции delete, которую можно увидеть в коде; она используется при раскрутке стека, когда конструктор возбуждает исключение).

Каким-то образом мы должны сообщить уникальному указателю, что этот конкретный объект следует удалять по-другому. Оказывается, что в шаблоне std::unique_ptr имеется второй аргумент. Обычно мы его не видим, потому что по умолчанию он равен std::default_delete, однако его можно изменить и указать свой объект-ликвидатор deleter, соответствующий механизму выделения. Интерфейс deleter очень прост – он должен быть вызываемой сущностью:

```
template <typename T> struct MyDeleter {
    void operator()(T* p);
};
```

Политика std::default_delete реализована именно так и попросту вызывает оператор delete для указателя p. А нашему объекту deleter понадобится нетривиальный конструктор, который будет сохранять указатель на кучу. Заметим, что хотя в общем случае ликвидатор deleter должен уметь удалять объект любого типа, допускающего выделение, он не обязан быть шаблоном класса. Подойдет и нешаблонный класс с шаблонной функцией-членом, при условии что данные-члены класса не зависят от типа удаляемого объекта. В нашем случае данные-члены зависят только от типа кучи, но не от типа того, что удаляется:

```
class MyDeleter {
    MyHeap* heap_;
    public:
    MyDeleter(MyHeap* heap) : heap_(heap) {}
    template <typename T> void operator()(T* p) {
        p->~T();
        heap_->deallocate(p);
    }
};
```

Ликвидатор должен выполнить эквиваленты обоих действий стандартной функции operator delete: вызвать деструктор удаляемого объекта и освободить память, выделенную этому объекту.

Теперь, имея подходящий ликвидатор, мы наконец можем воспользоваться объектом std::unique_ptr с собственной кучей:

```
MyHeap heap;
MyDeleter deleter(&heap);
std::unique_ptr<int, MyDeleter> p(new(&heap) int(0), deleter);
```

Заметим, что объекты deleter часто создаются прямо в точке выделения памяти:

```
MyHeap heap;
std::unique_ptr<int, MyDeleter> p(new(&heap) int(0), MyDeleter(&heap));
```

В любом случае deleter должен быть копирующим или перемещающим без возбуждения исключения; это означает, что в нем должен присутствовать копирующий или перемещающий конструктор, объявленный с квалификатором noexcept. Встроенные типы, например простые указатели, конечно же, являются копируемыми, и сгенерированный компилятором по умолчанию конструктор не возбуждает исключений. В любом агрегатном типе, содержащем данные-члены подобного типа, в частности в нашем объекте deleter, имеется конструктор по умолчанию, который также не возбуждает исключений (если, конечно, он не переопределен).

Заметим, что deleter – часть типа уникального указателя. Два уникальных указателя, владеющих объектами одного и того же типа, но имеющих разные ликвидаторы, – это объекты разных типов:

```
MyHeap heap;
std::unique_ptr<int, MyDeleter> p(new(&heap) int(0), MyDeleter(&heap));
std::unique_ptr<int> q(new int(0));
p = std::move(q);       // Не компилируется, типы p и q различны
```

И при конструировании уникального указателя следует задавать deleter подходящего типа:

```
std::unique_ptr<int> p(new(&heap) int(0), MyDeleter(&heap)); // Не компилируется
```

Попутно отметим, что два уникальных указателя разных типов в приведенном выше коде, p и q, хотя и не допускают присваивания, но допускают сравнение: конструкция p == q компилируется. Дело в том, что оператор сравнения на самом деле является шаблоном – он принимает два уникальных указателя разных типов и сравнивает скрываемые ими простые указатели (если и их типы различаются, то в сообщении компилятора об ошибке уникальный указатель, скорее всего, не будет упомянут вовсе, а будет сказано что-то о сравнении указателей на разные типы без приведения).

А теперь повторим тот же пример, но с разделяемым указателем std::shared_ptr. Сначала пусть разделяемый указатель указывает на объект, сконструированный обычной функцией operator new:

```
std::unique_ptr<int> p(new int(0));
std::shared_ptr<int> q(new int(0));
```

Для сравнения мы оставили также объявление уникального указателя. Оба уникальных указателя объявляются и конструируются в точности одинаково.

А в следующем фрагменте разделяемый указатель указывает на объект, выделенный в нашей куче heap:

```
MyHeap heap;
std::unique_ptr<int, MyDeleter> p(new(&heap) int(0), MyDeleter(&heap));
std::shared_ptr<int> q(new(&heap) int(0), MyDeleter(&heap));
```

Разница очевидна – разделяемый указатель, созданный с пользовательским ликвидатором deleter, имеет тот же тип, что и указатель с deleter по умолчанию! На самом деле все разделяемые указатели на int имеют один и тот же тип, std::shared_ptr<int>, – шаблон не принимает дополнительного аргумента. Поразмыслите об этом – deleter задается в конструкторе, но используется только в деструкторе, поэтому он должен храниться где-то внутри объекта интеллектуального указателя, пока не понадобится. Невозможно восстановить его впоследствии, если мы потеряем объект, переданный нам в момент конструирования. И std::shared_ptr, и std::unique_ptr должны сохранять объект deleter произвольного типа внутри самого объекта указателя. Но только в классе std::unique_ptr информация о deleter включена в тип. Класс std::shared_ptr одинаковый для всех типов ликвидаторов. Возвращаясь к самому началу этого раздела, мы можем сказать, что программа, использующая std::shared_ptr<int>, не имеет явной информации о типе ликвидатора. Этот тип стерт из программы. И вот как выглядит программа со стертым типом:

```
MyHeap heap;
{
    std::shared_ptr<int> p(      // deleter не присутствует в типе
        new(&heap) int(0),
        MyDeleter(&heap)         // deleter есть только в конструкторе
    );
    std::shared_ptr<int> q(p);   // типа deleter вообще нигде нет
    // void some_function(std::shared_ptr<int>) - deleter отсутствует
    some_function(p);            // p используется, deleter отсутствует
}                                // происходит удаление, вызывается MyDeleter
```

Итак, мы знаем, что делает стирание типа и как оно выглядит. Остался всего один вопрос – как оно работает?

Как стирание типа реализовано в C++?

Мы видели, как выглядит стирание типа в C++. Теперь мы понимаем, что имеется в виду, когда говорят, что программа явно не зависит от типа. Но загадка остается – в программе нет ни одного упоминания о типе, и тем не менее в нужный момент она вызывает операцию для типа, о котором ничего не знает. Как? Вот это мы сейчас и обсудим.

Очень старый способ стирания типа

Идея написания программы, в которой нет явной информации о типе, конечно, не нова. Она существовала задолго до появления объектно-ориентированного

программирования и понятия объекта. Рассмотрим следующую программу на C (здесь нет и следа C++):

```
int less(const void* a, const int* b) {
    return *(const int*)a - *(const int*)b;
}
int main() {
    int a[10] = { 1, 10, 2, 9, 3, 8, 4, 7, 5, 0 };
    qsort(a, sizeof(int), 10, less);
}
```

Вспомните объявления функции `qsort` в стандартной библиотеке C:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Заметим, что хотя мы используем ее для сортировки массива целых чисел, в самой функции `qsort` типы явно не упоминаются – массив, подлежащий сортировке, передается по указателю `void*`. И функция сравнения тоже принимает два указателя `void*` и не содержит явной информации о типах в своем объявлении. Конечно, в какой-то момент мы должны знать, как сравнивать реальные типы. В нашей программе на C указатели, которые теоретически могут указывать на что угодно, преобразуются в указатели на целые числа. Это действие, противоположное абстрагированию, называется **материализацией** (reification).

В языке C восстановление конкретных типов – обязанность программиста; наша функция сравнения `less()` в действительности сравнивает только целые числа, но понять это из интерфейса невозможно. И во время выполнения невозможно проверить, что в программе используются правильные типы, и уж, конечно, программа не может автоматически выбрать правильную операцию сравнения для истинного типа.

В объектно-ориентированных языках некоторые из этих ограничений можно снять, но не бесплатно.

Объектно-ориентированное стирание типа

В объектно-ориентированных языках мы постоянно имеем дело с абстрагированными типами. Любую программу, которая работает только с указателями (или ссылками) на базовый класс, тогда как конкретные производные классы неизвестны до момента выполнения, можно рассматривать как своего рода реализацию стирания типа. Этот подход особенно популярен в Java, но его можно реализовать и в C++ (хотя напомним, что «можно» не означает «должно»).

Чтобы воспользоваться объектно-ориентированным подходом, конкретные типы должны наследовать известному базовому классу:

```
class Object {};
class Int : public Object {
    int i_;
    public:
    explicit Int(int i) : i_(i) {}
};
```

И сразу же возникает проблема – мы хотим сортировать массив целых чисел, но встроенный тип не является классом и не может ничему наследовать. Поэтому никакие встроенные типы использовать нельзя, и примитивный тип int необходимо обернуть классом. Но это даже не половина дела; оперирование только указателями на базовый класс работает лишь в случае, когда типы полиморфны, т. е. мы должны использовать виртуальные функции, а наш класс должен содержать, помимо целого числа, еще и указатель на таблицу виртуальных функций:

```
class Object {
    public:
    int less(const Object* rhs) const = 0;
};
class Int : public Object {
    int i_;
    public:
    explicit Int(int i) : i_(i) {}
    int less(const Object* rhs) const override {
        return i_ - dynamic_cast<const Int*>(rhs)->i_;
    }
};
```

Рассмотрим следующий фрагмент кода:

```
Object *p, *q;
p->less(q);
```

Если мы напишем такой код, то программа во время выполнения будет автоматически вызывать правильную функцию less(), исходя из типа одного из сравниваемых объектов, в данном случае p. Тип второго объекта проверяется во время выполнения (хорошо написанная программа не должна «падать», если динамическое приведение не проходит, но может возбудить исключение). Эта функция сравнения включена в сам тип. Но мы могли бы также предоставить свободную функцию сравнения:

```
class Object {
    public:
    virtual ~Object() = 0;
};
inline Object::~Object() {};
class Int : public Object {
    int i_;
    public:
    explicit Int(int i) : i_(i) {}
    int Get() const { return i_; }
};
int less(const Object* a, const Object* b) {
    return dynamic_cast<const Int*>(a)->Get() -
           dynamic_cast<const Int*>(b)->Get();
}
```

Теперь функция сравнения преобразует указатели в ожидаемый тип (и снова, хорошо написанная программа должна проверять указатель на NULL после динамического приведения). Поскольку функция Get() теперь не виртуальная (и не может быть виртуальной, потому что базовый класс не может объявить *универсальный* тип возвращаемого значения), объект необходимо сделать полиморфным другими средствами. Деструктор полиморфного объекта часто должен быть виртуальным в любом случае (но см. более детальное обсуждение этого вопроса в главе, посвященной идиоме невиртуального интерфейса). В нашем случае он еще и чисто виртуальный, потому что мы не хотим, чтобы пользователь мог создавать объекты типа Object. Здесь есть одна необычная деталь – чисто виртуальная функция фактически имеет реализацию, и это обязательно, потому что деструктор производного класса в конце всегда вызывает деструктор базового класса.

Программная ошибка – вызов функции сравнения с некорректными типами, – которая в предыдущей программе на C привела бы к неопределенному поведению, теперь обрабатывается как исключение во время выполнения. Альтернативно мы могли бы заставить исполняющую систему вызывать за нас правильную функцию сравнения, но ценой связывания функции сравнения с типом объекта.

Наверное, сейчас у вас возник вопрос о производительности. Конечно, приходится расплачиваться повышенным потреблением памяти, и в нашем случае цена довольно высока – вместо 4-байтовых целых чисел мы теперь имеем объект, содержащий указатель и целое число, а вместе с выравниванием на границу памяти получается 16 байтов в 64-разрядной системе. Такое увеличение объема рабочей памяти вполне может привести к снижению производительности из-за неэффективного использования кешей. Кроме того, приходится платить за вызов виртуальной функции. Однако в случае программы на C функция и так вызывалась косвенно – по указателю. Это похоже на то, как в большинстве компиляторов реализованы вызовы виртуальных функций, и обращение к функции сравнения будет стоить столько же. Динамическое приведение типа и проверка указателя на NULL внутри функции сравнения еще немного увеличивают время работы.

Но на этом наши проблемы не заканчиваются. Вспомним, что мы хотели отсортировать массив целых чисел. Хотя указатель на объект типа Object может служить абстракцией указателя на производный класс вроде нашего Int, массив объектов типа Int ни в коем случае не является массивом объектов типа Object:

```
void sort(Object* a, ... );
Int a[10] = ...;
sort(a, ...);
```

Этот код, возможно, откомпилируется, потому что a имеет тип Int[], который приводится к Int*, а тот можно неявно преобразовать в Object*, но вряд ли это то, что нам нужно, т. к. невозможно продвинуть указатель на следующий элемент массива.

Вариантов у нас несколько, но все они малопривлекательны. Классический объектно-ориентированный способ – использовать массив указателей `Object*` и сортировать его следующим образом:

```
Int a[10] = ...;
Object* p[10];
for (int i = 0; i < 10; ++i) p[i] = a + i;
```

Недостаток состоит в том, что нужно выделять память для дополнительного массива указателей, а для доступа к фактическому значению потребуется двойная косвенная адресация.

Можно было бы добавить в нашу иерархию классов еще одну виртуальную функцию, `Next()`, которая возвращала бы указатель на следующий элемент массива:

```
class Object {
    public:
    virtual Object* Next() const = 0;
};
class Int : public Object {
    ...
    Int* Next() const override { return this + 1; }
};
```

Это удорожает обход массива, и к тому же у нас нет никакого способа проверить, на этапе компиляции или выполнения, что объект, от имени которого вызывается `Next()`, действительно является элементом массива.

Противоположность стиранию типа

Мы так далеко углубились в кроличью нору, пытаясь стереть все типы в программе на C++, что забыли, с чего все начиналось. Исходная проблема заключалась в том, что мы не хотели писать новую функцию сортировки для каждого типа. В языке C эта проблема решается стиранием типов и использованием указателей `void*`, но при этом вся ответственность за правильность кода перекладывается на программиста – ни компилятор, ни исполняющая система не смогут обнаружить, что типы аргументов не соответствуют вызываемой функции.

Попытка сделать все функции виртуальными дает очень громоздкое решение проблемы. В C++ есть куда более элегантный способ – шаблоны. Шаблонная функция в некотором смысле является противоположностью стиранию типа. С одной стороны, тип вот он – прямо в сигнатуре функции, всем виден. С другой стороны, тип может быть любым, и компилятор выберет правильный тип, исходя из аргументов:

```
int a[10] = {...};
std::sort(a, a+10, std::less<void>());  // C++14
std::sort(a, a+10, std::less<int>());   // До C++14
```

Мы можем даже предоставить собственную функцию сравнения, не задавая типы явно:

```
int a[10] = {...};
std::sort(a, a+10, [](auto x, auto y) { return x < y; }); // C++14
```

При переходе от выведения типа шаблона к выведению типа с помощью ключевого слова auto может сложиться впечатление, что в этой программе нет явной информации о типе. Но это только иллюзия – сами шаблоны не являются типами, типами являются их конкретизации, а в сигнатуре каждой из них информация о типе присутствует в полном объеме. Ничто не стерто, компилятор точно знает тип каждого объекта или функции, а при некотором усилии это может узнать и программист.

Во многих случаях этого псевдостирания типов по принципу «чего не вижу, того и не существует» вполне достаточно. Но это все же не решает проблему того, что различные объекты могут иметь разные типы, когда мы этого не хотим. Можно было бы скрыть типы наших уникальных указателей с разными ликвидаторами:

```
template <typename T, typename D>
std::unique_ptr<T, D> make_unique_ptr(T* p, D d) {
    return std::unique_ptr<T, D>(p, d);
}
auto p(make_unique_ptr(new int, d));
```

Теперь тип deleter автоматически выводится при конструировании уникального указателя и нигде не упоминается явно. Тем не менее два уникальных указателя с ликвидаторами разных типов сами имеют разные типы:

```
struct deleter1 { ... };
struct deleter2 { ... };
deleter1 d1; deleter2 d2;
auto p(make_unique_ptr(new int, d1));
auto q(make_unique_ptr(new int, d2));
```

Типы этих уникальных указателей кажутся одинаковыми, но на самом деле это не так; auto – не тип, истинным типом указателя p является unique_ptr<int, deleter1>, а типом указателя q – unique_ptr<int, deleter2>. Итак, мы вернулись к тому, с чего начали, – у нас есть замечательный способ писать код, не зависящий от типов, но если действительно требуется стереть, а не просто скрыть типы, то как это сделать, мы по-прежнему не знаем. Пора бы уже и узнать.

Стирание типа в C++

Давайте, наконец, посмотрим, как std::shared_ptr вершит свою магию. Мы продемонстрируем это на упрощенном примере интеллектуального указателя, в котором все внимание направлено на стирание типа. Вряд ли вы удивитесь, узнав, что это делается с помощью комбинации обобщенного и объектно-ориентированного программирования:

```
template <typename T>
class smartptr {
    struct deleter_base {
```

```
        virtual void apply(void*) = 0;
        virtual ~deleter_base() {}
    };
    template <typename Deleter>
    struct deleter : public deleter_base {
        deleter(Deleter d) : d_(d) {}
        virtual void apply(void* p) { d_(static_cast<T*>(p)); }
        Deleter d_;
    };
    public:
    template <typename Deleter>
        smartptr(T* p, Deleter d) :
            p_(p), d_(new deleter<Deleter>(d)) {}
    ~smartptr() { d_->apply(p_); delete d_; }
    T* operator->() { return p_; }
    const T* operator->() const { return p_; }
    private:
    T* p_;
    deleter_base* d_;
};
```

У шаблона `smartptr` всего один параметр-тип. Поскольку стертый тип не является частью типа интеллектуального указателя, его необходимо запомнить в каком-то другом объекте. В нашем случае этот объект – результат конкретизации вложенного шаблона `smartptr<T>::deleter`. Он создается конструктором, и это последняя точка в коде, где тип `deleter` явно присутствует. Но `smartptr` должен ссылаться на экземпляр `deleter` по указателю, тип которого не зависит от `deleter` (поскольку объект `smartptr` имеет один и тот же тип для всех ликвидаторов). Поэтому все экземпляры шаблона `deleter` наследуют одному и тому же базовому классу, `deleter_base`, а фактический `deleter` вызывается через виртуальную функцию. Конструктор является шаблоном, который выводит тип `deleter`, но сам этот тип всего лишь скрыт, т. к. является частью фактического объявления конкретизации этого шаблона. В самом классе интеллектуального указателя и, в частности, в его деструкторе, где ликвидатор и используется, тип `deleter` действительно стерт. Определение типа на этапе компиляции применяется, для того чтобы создать корректный по построению полиморфный объект, который сумеет определить тип `deleter` во время выполнения и выполнить нужное действие. Поэтому нам не нужно динамическое приведение, а вполне достаточно статического, которое работает только тогда, когда мы знаем истинный производный тип (а мы его знаем).

Эта реализация стирания типа используется еще в нескольких компонентах стандартной библиотеки. Сложность фактической реализации варьируется, но подход всегда один. Начнем с обобщенной функциональной обертки `std::function`. Тип объекта `std::function` говорит нам, как вызывается функция, но это и все – он ничего не говорит о том, что это: свободная функция, вызываемый объект, лямбда-выражение или еще что-то, что можно вызвать, поставив круглые скобки. Например, можно следующим образом создать вектор вызываемых объектов:

```
std::vector<std::function<void(int)>> v;
```

Любой элемент этого вектора можно вызвать, например, как `v[i](5)`, но это все, что мы можем узнать об этих элементах из сигнатуры типа. Фактический тип того, что вызывается, стерт.

Есть также класс `std::any`, в котором стирание типа достигло высшей точки (в C++17 и последующих версиях). Это класс, а не шаблон, но он может содержать значение любого типа:

```
std::any a(5);
int i = std::any_cast<int>(a);     // i == 5
std::any_cast<long>(a);            // исключение bad_any_cast
```

Разумеется, коль скоро тип неизвестен, `std::any` не может предоставлять никаких интерфейсов. В нем можно сохранить произвольное значение и получить его обратно, если мы знаем правильный тип (или можно запросить тип и получить в ответ объект `std::type_info`).

Может показаться, что стирание типа и объекты со стертым типом способны здорово упростить программирование, но в действительности злоупотреблять стиранием типа не стоит. Его недостатки обсуждаются в следующем разделе.

Когда использовать стирание типа, а когда избегать его

Мы уже видели пример, в котором стирание типа использовалось очень успешно, – разделяемый указатель `std::shared_ptr`. С другой стороны, авторы стандартной библиотеки не стали применять эту технику при реализации `std::unique_ptr`, и не потому, что это труднее (на самом деле наш простой шаблон `smartptr` гораздо ближе к `std::unique_ptr`, чем к `std::shared_ptr`, поскольку не занимается подсчетом ссылок).

Потом была безуспешная попытка стереть тип в функции сортировки и родственных ей, закончившаяся полной неразберихой, которую удалось разгрести раз и навсегда, позволив компилятору выводить правильные типы, а не стирать их. Можно уже сформулировать некоторые рекомендации о том, когда имеет смысл подумать о стирании типа. Следует иметь в виду двоякие соображения: проектирования и производительности.

Стирание типа и проектирование программ

Как показала печально окончившаяся попытка написать функцию сортировки с полным стиранием типа, не всегда эта техника делает программу проще. Даже введение стандартных классов со стертым типом, например `std::any`, не решает фундаментальную проблему – в какой-то момент необходимо выполнить действие со стертым типом. В этот момент у нас есть два варианта: либо мы знаем истинный тип и можем привести к нему (с помощью `dynamic_cast` или `any_cast`), либо должны работать с объектом через полиморфный интерфейс (виртуальные функции), для чего необходимо, чтобы все задействованные типы наследовали одному базовому классу, и в этом случае мы ограничены ин-

терфейсом этого класса. Второй вариант – обычное объектно-ориентированное программирование, которое, конечно, широко и успешно используется, но здесь применяется к проблеме, для которой не слишком пригодно, – стремление поступать так только для того, чтобы стереть некоторые типы, ведет к запутанным проектам (как мы поняли на собственном опыте). Первый вариант – по существу, современное воплощение старого трюка, связанного с применением `void*` в C, – программист должен знать истинный тип и использовать его для получения значения. Программа все равно будет работать неправильно, если программист допустит ошибку, но мы, по крайней мере, сможем отловить такие ошибки во время выполнения и сообщить о них.

У *универсальных* объектов со стертым типом, таких как `std::any`, есть применения, например безопасная замена `void*`. Часто они используются совместно с тегированными типами, когда фактический тип можно определить по какой-то информации во время выполнения. Например, тип `std::map<std::string, std::any>` может пригодиться для добавления динамических возможностей в программу на C++. Подробное описание таких применений выходит за рамки этой книги. У них есть общая черта – в объектах со стертым типом программа хранит какую-то идентифицирующую информацию, а программист должен по этим *хлебным крошкам* восстановить исходные типы.

С другой стороны, *ограниченное* стирание типа, как в `std::shared_ptr`, можно использовать, когда мы можем по построению гарантировать, благодаря применению шаблонов и выведению аргументов, что все стертые типы правильно восстанавливаются. В реализациях обычно применяется одна иерархия, используемая полиморфно, – вместо того чтобы спрашивать объект «какого ты на самом деле типа?» или говорить ему «я знаю твой настоящий тип, по крайней мере надеюсь», мы просто указываем ему – *выполни эту операцию, ты сам знаешь, как*. Это «ты сам знаешь, как» гарантировано самим способом конструирования объектов, который полностью контролируется реализацией. Еще одно отличие такого стирания типа, с точки зрения проектирования, заключается в том, что тип стирается только у объекта, с которым программа непосредственно взаимодействует, например `std::shared_ptr<int>`. *Более крупный* объект, являющийся разделяемым указателем со всеми его вспомогательными объектами, таки запоминает стертый тип в одной из своих частей. Именно это дает возможность объекту автоматически *делать правильную вещь*, не принуждая программиста высказывать утверждения или догадки о фактическом типе.

Очевидно, что не всякая программная система может получить преимущество от стирания типа, даже с точки зрения проектирования. Часто самый простой и прямой путь решения задачи – использовать явные типы или позволить компилятору вывести правильные типы. Так обстоит дело с функцией `std::sort`. Но должна быть еще какая-то причина, по которой в `std::unique_ptr` тип не стирается, ведь он же так похож на `std::shared_ptr`, где стирание имеет место. Вторая причина не пользоваться стиранием типа – возможное падение производительности. Но прежде чем рассуждать о производительности, мы должны научиться измерять ее.

Установка библиотеки эталонного микротестирования

Нас интересует эффективность очень небольших фрагментов кода, которые конструируют и удаляют объекты с помощью различных интеллектуальных указателей. Подходящим инструментом для измерения производительности небольших фрагментов кода является эталонный микротест. Средств такого рода существует много, в этой книге мы будем пользоваться библиотекой Google Benchmark. Чтобы следить за примерами, вам понадобится сначала скачать и установить библиотеку (следуйте инструкциям в файле `Readme.md`). Затем можно будет откомпилировать и выполнить примеры. Можете собрать демонстрационные примеры, прилагаемые к библиотеке, чтобы посмотреть, как эталонный тест собирается конкретно в вашей системе. Например, на компьютере под управлением Linux команда сборки и выполнения теста `smartptr.C` может выглядеть так:

```
$CXX smartptr.C smartptr_extra.C -I. -I$GBENCH_DIR/include -g -O4 -I. \
    -Wall -Wextra -Werror -pedantic --std=c++14 \
    $GBENCH_DIR/lib/libbenchmark.a -lpthread -lrt -lm -o smartptr && \
    ./smartptr
```

Здесь `$CXX` – ваш компилятор C++, например `g++` или `g++-6`, а `$GBENCH_DIR` – каталог, в который установлена библиотека `benchmark`.

Издержки стирания типа

Для любого эталонного теста нужен эталон – точка отсчета. В нашем случае таким эталоном будет простой указатель. Разумно предположить, что никакой интеллектуальный указатель не сможет превзойти простой в производительности и что у лучшего интеллектуального указателя накладные расходы будут нулевыми. Поэтому для начала измерим, сколько времени занимает конструирование и уничтожение объекта при использовании простого указателя:

```
struct deleter {
    template <typename T> void operator()(T* p) { delete p; }
};
void BM_rawptr(benchmark::State& state) {
    deleter d;
    for (auto _ : state) {
        int* p = new int(0);
        d(p);
    }
    state.SetItemsProcessed(state.iterations());
}
```

Абсолютные результаты теста, конечно, зависят от машины, на которой он выполнялся. Но нас интересуют относительные изменения, поэтому подойдет любая машина, лишь бы все измерения проводились на одной и той же.

Теперь можно проверить, что у std::unique_ptr накладные расходы действительно нулевые (если, конечно, мы конструируем и удаляем объекты одним и тем же способом):

```
void BM_uniqueptr(benchmark::State& state) {
    deleter d;
    for (auto _ : state) {
        std::unique_ptr<int, deleter> q(new int(0), d);
    }
    state.SetItemsProcessed(state.iterations());
}
```

Результат такой же, как для простого указателя, в пределах погрешности измерений:

```
-------------------------------------------------------
Benchmark             Time             CPU Iterations
-------------------------------------------------------
BM_rawptr            21 ns           21 ns    31178493   46.2325M items/s
```

Точно так же можно измерить производительность std::shared_ptr:

```
void BM_sharedptr(benchmark::State& state) {
    deleter d;
    for (auto _ : state) {
        std::shared_ptr<int> p(new int(0), d);
    }
    state.SetItemsProcessed(state.iterations());
}
```

Легко видеть, что разделяемый указатель гораздо медленнее:

```
-------------------------------------------------------
Benchmark             Time             CPU Iterations
-------------------------------------------------------
BM_uniqueptr         21 ns           21 ns    33681137   46.4407M items/s
```

Разумеется, причина не одна. Интеллектуальный указатель std::shared_ptr подсчитывает ссылки, а с этим связаны свои накладные расходы. Чтобы измерять только издержки стирания типа, следует реализовать уникальный указатель со стиранием типа. Но мы уже это сделали – это наш тип smartptr, который был продемонстрирован в разделе «Стирание типа в C++». Его функциональности достаточно для измерения производительности в том же эталонном тесте, что для других указателей:

```
void BM_smartptr_te(benchmark::State& state) {
    deleter d;
    for (auto _ : state) {
        smartptr_te<int> p(new int(0), d);
    }
    state.SetItemsProcessed(state.iterations());
}
```

Здесь `smartptr_te` означает версию интеллектуального указателя со стиранием типа (type-erased). Она немного быстрее `std::shared_ptr`, что подтверждает наше подозрение о наличии нескольких причин накладных расходов. Тем не менее отчетливо видно, что стирание типа уменьшает производительность нашего уникального указателя примерно вдвое:

```
-------------------------------------------------------
Benchmark             Time           CPU Iterations
-------------------------------------------------------
BM_sharedptr         51 ns         51 ns    12689804   18.7263M items/s
```

Однако существуют способы противостоять этой напасти и оптимизировать указатели (а также любые другие структуры данных) со стиранием типа. Основная причина замедления – дополнительное выделение памяти, которое производится при конструировании полиморфного объекта `smartptr::deleter`. Этого выделения можно избежать, по крайней мере иногда, если заранее выделить для таких объектов буфер в памяти. Детали и ограничения этой оптимизации обсуждаются в главе 10. Здесь же отметим лишь, что при удачной реализации она способна почти полностью (но не совсем) погасить накладные расходы на стирание типа:

```
-------------------------------------------------------
Benchmark             Time           CPU Iterations
-------------------------------------------------------
BM_smartptr_te       42 ns         42 ns    13659133   22.6096M items/s
```

Мы можем заключить, что стирание типа влечет за собой дополнительные накладные расходы (как минимум, лишний вызов виртуальной функции). Кроме того, почти всегда реализация со стиранием типа потребляет больше памяти, чем со статическим типом. Особенно велики накладные расходы, когда требуется выделение дополнительной памяти. Иногда возможно оптимизировать реализацию и сократить эти накладные расходы, но стирание типа всегда сопровождается некоторым снижением производительности.

Резюме

Надеюсь, что в этой главе мы сбросили покров тайны с техники программирования, известной как стирание типа. Мы показали, как можно написать программу, не выставляя напоказ всю информацию о типах, и объяснили причины, по которым это может оказаться желательным. Мы также продемонстрировали, что не для всякой задачи стирание типа необходимо. Во многих приложениях это средство попросту неуместно. И даже когда оно упрощает программу, следует взвесить, оправдывают ли выгоды от стирания типа падение производительности. Впрочем, если реализовать стирание типа эффективно, а использовать мудро, то эта техника может послужить созданию гораздо более простых и гибких интерфейсов.

В следующей главе мы сменим направление – оставим на некоторое время идиомы абстрагирования и перейдем к идиомам, которые облегчают сборку сложных взаимодействующих систем из шаблонных компонентов. Начнем с идиомы SFINAE.

Вопросы

- Что собой представляет стирание типа?
- Как стирание типа реализуется в C++?
- В чем разница между сокрытием типа за ключевым словом `auto` и его стиранием?
- Как материализуется конкретный тип, когда у программы возникает в нем необходимость?
- Каковы издержки стирания типа?

Глава 7

SFINAE и управление разрешением перегрузки

Идиома **Substitution Failure Is Not An Error** (SFINAE – «неудавшаяся подстановка – не ошибка»), изучаемая в этой главе, – одна из самых сложных с точки зрения используемых языковых средств. Поэтому она привлекает неумеренное внимание со стороны программистов на C++. Есть в ней что-то такое, что созвучно умонастроению типичного программиста на C++. Нормальный человек думает, что если ничего не сломалось, то и беспокоиться не о чем. А программист, особенно пишущий на C++, склонен думать, что раз еще не сломалось, значит, используется не на полную катушку. Просто остановимся на том, что потенциал SFINAE весьма велик.

В этой главе рассматриваются следующие вопросы:

- что такое перегрузка функции и разрешение перегрузки;
- что такое выведение и подстановка типов;
- что такое SFINAE и почему она так необходима в C++;
- как можно использовать SFINAE для написания безумно сложных и иногда полезных программ.

Технические требования

Примеры кода: **https://github.com/PacktPublishing/Hands-On-Design-Patterns-with-CPP/tree/master/Chapter07**.

Разрешение перегрузки и множество перегруженных вариантов

В этом разделе вы сможете проверить свое знание новейших и самых передовых добавлений в стандарт C++. Начнем с одного из самых фундаментальных понятий C++ – функций и их перегрузки.

Перегрузка функций в C++

Сама концепция перегрузки функций в C++ очень проста: несколько разных функций могут иметь одно и то же имя. Вот, собственно, и все – видя синтаксическую конструкцию, обозначающую вызов функции, например `f(x)`, компилятор знает, что функций с именем `f` может быть несколько. Если так оно и есть, то имеет место перегрузка и для выбора вызываемой функции нужно прибегнуть к разрешению перегрузки.

Начнем с простого примера:

```
void f(int i) { std::cout << "f(int)" << std::endl; }       // 1
void f(long i) { std::cout << "f(long)" << std::endl; }     // 2
void f(double i) { std::cout << "f(double)" << std::endl; } // 3
f(5);
f(5l);
f(5.0);
```

Здесь мы имеем три определения функции с именем `f` и три вызова функции. Заметим, что все сигнатуры функций различны (различаются типы параметров). Это требование – параметры перегруженных функций должны в чем-то различаться. Не может быть двух перегруженных вариантов, которые принимают в точности одинаковые параметры, а различаются только типом возвращаемого значения или телом. Заметим также, что хотя в этом примере речь идет о свободной функции, точно такие же правила применяются и к перегруженным функциям-членам, поэтому не будем специально рассказывать о них.

Но вернемся к нашему примеру. Какой вариант функции `f()` вызывается в каждой строчке? Чтобы это понять, нужно знать, как разрешается перегрузка функций в C++. Точные правила довольно сложны и в разных версиях стандарта различаются тонкими нюансами, но по большей части они спроектированы так, чтобы в наиболее распространенных случаях компилятор делал именно то, чего мы от него ожидаем. А ожидаем мы, что `f(5)` вызовет вариант, принимающий целый аргумент, поскольку `5` – литерал типа `int`. Компилятор так и делает. Аналогично литерал `5l` имеет тип `long`, поэтому `f(5l)` вызывает второй перегруженный вариант. Наконец, `5.0` – число с плавающей точкой, поэтому вызывается последний вариант.

Это было не сложно, правда? Но что, если тип аргумента не совпадает с типом параметра точно? Тогда компилятор вынужден рассматривать преобразования типа. Например, литерал `5.0` имеет тип `double`. Посмотрим, что произойдет, если вызвать `f()` с аргументом типа `float`:

```
f(5.0f);
```

Теперь мы должны преобразовать аргумент из типа `float` в один из типов `int`, `long` или `double`. На этот случай в стандарте есть строгие правила, но вряд ли вы удивитесь, узнав, что предпочтительным является преобразование в `double` и, следовательно, вызывается именно этот перегруженный вариант.

А что, если вызвать с другим целым типом, например unsigned int:

```
f(5u);
```

Теперь у нас два варианта действий: преобразовать unsigned int в signed int или в signed long. Можно предположить, что преобразование в long *безопаснее*, а значит, лучше, но стандарт считает эти два преобразования настолько близкими, что компилятор не может сделать выбор. Этот вызов не компилируется, поскольку разрешение перегрузки считается неоднозначным, о чем и будет сказано в сообщении об ошибке. Если вы столкнетесь с такой ошибкой в своем коде, то должны будете помочь компилятору – привести аргументы к типу, при котором разрешение станет однозначным. Обычно самый простой способ – привести к типу параметра в желаемом перегруженном варианте:

```
unsigned int i = 5u;
f(static_cast<int>(i));
```

До сих пор мы разбирали ситуации, в которых типы параметров различны, но количество одинаково. Конечно, если в разных объявлениях одноименных функций количество параметров различается, то будут рассматриваться только функции, принимающие требуемое число параметров. Ниже приведен пример двух функций с одинаковым именем, но разным числом параметров:

```
void f(int i) { std::cout << "f(int)" << std::endl; }                 // 1
void f(long i, long j) { std::cout << "f(long, long)" << std::endl; }  // 2

f(5.0, 7);
```

Здесь перегрузка разрешается очень просто – нам нужна функция, способная принять два аргумента, а такая всего одна. Оба аргумента нужно будет преобразовать в тип long. Но что, если есть несколько функций с одинаковым количеством параметров? Рассмотрим следующий пример:

```
void f(int i, int j) { std::cout << "f(int, int)" << std::endl; }     // 1
void f(long i, long j) { std::cout << "f(long, long)" << std::endl; } // 2
void f(double i) { std::cout << "f(double)" << std::endl; }           // 3

f(5, 5);     // 1
f(5l, 5l);   // 2
f(5, 5.0);   // 1
f(5, 5l);    // ?
```

Сначала очевидный случай: если типы всех аргументов точно совпадают с типами соответственных параметров хотя бы в одном перегруженном варианте, то он и вызывается. Дальше становится интереснее: если точного совпадения нет, то к каждому аргументу придется применять преобразования. Рассмотрим третий вызов, f(5, 5.0). Первый аргумент, типа int, точно соответствует первому перегруженному варианту, но при необходимости его можно было бы преобразовать в long. Второй аргумент, типа double, не соответствует ни одному варианту, но может быть преобразован в тип обоих. Первый вариант дает лучшее соответствие, поскольку требует меньшего числа преобразо-

ваний аргументов. Наконец, как насчет последней строки? Первый перегруженный вариант можно вызвать, при этом потребуется преобразовать второй аргумент. Второй тоже можно вызвать, и тоже с преобразованием второго аргумента. Мы снова имеем неоднозначную перегрузку, поэтому строка не компилируется. Заметим, что, вообще говоря, неверно, что перегрузка с меньшим числом преобразований всегда побеждает; в более сложных случаях неоднозначность может иметь место даже тогда, когда один вариант требует меньше преобразований, чем все остальные (общее правило таково: если существует перегруженный вариант, для которого преобразование каждого аргумента – лучшее из возможных, то он выбирается, в противном случае вызов неоднозначен).

Заметим, что третий перегруженный вариант вообще не рассматривался, потому что количество параметров в нем не соответствует ни одному вызову функции. Но не всегда все так просто – у функций могут быть аргументы по умолчанию, а это означает, что количество аргументов не обязано совпадать с количеством параметров.

Рассмотрим такой фрагмент кода:

```
void f(int i) { std::cout << "f(int)" << std::endl; }                 // 1
void f(long i, long j) { std::cout << "f(long, long)" << std::endl; } // 2
void f(double i, double j = 0) {                                      // 3
    std::cout << "f(double, double = 0)" << std::endl;
}

f(5);     // 1
f(5l, 5); // 2
f(5, 5);  // ?
f(5.0);   // 3
f(5.0f);  // 3
f(5l);    // ?
```

Теперь у нас три перегруженных варианта. Первый со вторым не спутаешь, потому что у них разное количество параметров. А вот третий вариант можно вызвать с одними или с двумя аргументами; в первом случае предполагается, что второй аргумент равен нулю. Первый вызов самый простой – один аргумент, тип которого точно совпадает с типом параметра в первом варианте. Второй вызов напоминает случай, который мы уже встречали: два аргумента, причем тип первого точно соответствует одному перегруженному варианту, а второй нуждается в преобразовании. При выборе альтернативного варианта понадобилось бы преобразовывать типы обоих аргументов, поэтому второе определение функции наилучшее. Третий вызов с двумя целыми аргументами кажется простым, но эта простота обманчива – имеется два перегруженных варианта, принимающих два аргумента, и в обоих случаях требуется преобразование обоих аргументов. Может показаться, что преобразование из `int` в `long` лучше, чем из `int` в `double`, но C++ так не думает. Этот вызов неоднозначен. В следующем вызове, `f(5.0)`, аргумент всего один, и его можно преобразовать в `int`, тип параметра в варианте с одним параметром. Но все равно лучшим остается

третий перегруженный вариант, в котором преобразования не нужны вовсе. В следующем вызове тип аргумента не `double`, а `float`. Преобразование в `double` лучше, чем в `int`, а задействование аргумента по умолчанию не считается преобразованием и не влечет *штрафа* при сравнении перегруженных вариантов. Последний вызов неоднозначен – оба преобразования, в `double` и в `int`, считаются равнозначными, поэтому первый и третий варианты одинаковы хороши. Второй вариант дает точное совпадение с типом первого параметра, но без второго аргумента вызвать его нельзя, поэтому он даже не рассматривается.

Пока что мы рассматривали только свободные функции в C++, но все вышесказанное в полной мере относится и к функциям-членам. Настало время включить в рассмотрение шаблонные функции.

Шаблонные функции

Помимо обычных функций, типы параметров которых известны, в C++ имеются также шаблонные функции. При вызове таких функций типы параметров выводятся из типов переданных аргументов. Шаблонные функции могут называться так же, как нешаблонные, и у нескольких шаблонных функций могут быть одинаковые имена. Поэтому необходимо разобраться, как производится разрешение перегрузки при наличии шаблонов.

Рассмотрим пример:

```
void f(int i) { std::cout << "f(int)" << std::endl; }                    // 1
void f(long i) { std::cout << "f(long)" << std::endl; }                  // 2
template <typename T> void f(T i) { std::cout << "f(T)" << std::endl; } // 3

f(5);    // 1
f(5l);   // 2
f(5.0);  // 3
```

Имя функции `f` может относиться к любой их трех функций, одна из которых является шаблонной. Наилучший вариант в каждом случае будет выбираться из этих трех возможностей. Множество функций, рассматриваемых при разрешении перегрузки для конкретного вызова, называется **множеством перегруженных вариантов** (overload set). Первый вызов `f()` точно соответствует первой нешаблонной функции во множестве перегруженных вариантов – аргумент имеет тип `int`, а сигнатура первой функции – `f(int)`. Если во множестве перегруженных вариантов найдено точное совпадение с нешаблонной функцией, то оно всегда считается наилучшим вариантом. Шаблонную функцию тоже можно конкретизировать, получив точное совпадение; процесс замены параметров шаблона конкретными типами называется подстановкой аргументов шаблона (или подстановкой типов). Если вместо параметра шаблона `T` подставить `int`, то мы получим еще одну функцию, точно соответствующую вызову. Однако точное совпадение с нешаблонной функцией считается лучшим вариантом перегрузки. Второй вызов обрабатывается аналогично, но он точно совпадает со второй функцией во множестве перегруженных вариантов, поэтому вызывается именно эта функция. В последнем вызове аргумент име-

ет тип double, который можно преобразовать в int или в long либо подставить вместо T, сделав конкретизацию шаблона точным совпадением. Поскольку точно совпадающей нешаблонной функции нет, то наилучшим перегруженным вариантом считается конкретизация шаблона, дающая точное совпадение.

Но что, если имеется несколько шаблонных функций, параметры которых можно подставить так, чтобы получить соответствие типам аргументов, переданных при вызове? Посмотрим:

```
void f(int i) { std::cout << "f(int)" << std::endl; } // 1
template <typename T> void f(T i) {                   // 2
    std::cout << "f(T)" << std::endl;
}
template <typename T> void f(T* i) {                  // 3
    std::cout << "f(T*)" << std::endl;
}

f(5);       // 1
f(5l);      // 2
int i = 0;
f(&i);      // 3
```

Первый вызов опять точно соответствует нешаблонной функции, так что разрешение на ней и останавливается. Второй вызов соответствует первому, нешаблонному, перегруженному варианту с преобразованием или второму варианту, в который вместо T подставляется тип long. Последний вариант не соответствует ни одному из этих вызовов – не существует подстановки, которая сделала бы параметр T* совпадающим с int или long. Однако последний вызов может соответствовать третьему варианту, если вместо T подставить int. Проблема в том, что он может соответствовать и второму варианту, если вместо T подставить int*. И какой же шаблон выбрать? Более специфичный – первый вариант, f(T), может соответствовать вызову любой функции с одним аргументом, а второй, f(T*), – только вызовам функций с аргументом-указателем. Более специфичный перегруженный вариант считается лучшим соответствием, он и выбирается. Это новая идея, относящаяся только к шаблонам, – вместо того чтобы выбирать лучшие преобразования (в общем случае чем их *меньше* или чем они *проще*, тем лучше), мы выбираем вариант, который *труднее* конкретизировать.

Наконец, существует еще один вид функций, который соответствует чуть ли не любому вызову функции с тем же именем, а именно функции с переменным числом аргументов:

```
void f(int i) { std::cout << "f(int)" << std::endl; } // 1
void f(...) { std::cout << "f(...)" << std::endl; }   // 2

f(5);       // 1
f(5l);      // 1
f(5.0);     // 1

struct A {};
A a;
f(a);       // 2
```

Первый перегруженный вариант годится для первых трех вызовов функций – он точно соответствует первому вызову, а для двух других существуют преобразования аргумента. Вторую функцию в этом примере можно вызывать с любыми аргументами любого типа. Это считается крайней мерой – если существует функция с конкретными параметрами, допускающими преобразование в типы аргументов, то компилятор предпочтет ее. Допустимы даже определенные пользователем преобразования, как в следующем примере:

```
struct B {
    operator int() const { return 0; }
};
B b;
f(b);  // 1
```

Функция `f(...)` с переменным числом аргументов будет вызвана, только если не существует никаких преобразований, позволяющих этого избежать.

Итак, теперь нам известен порядок разрешения перегрузки. Сначала выбирается нешаблонная функция, для которой типы параметров точно совпадают с типами переданных аргументов. Если во множестве перегруженных вариантов таковой не оказалось, то выбирается шаблонная функция, если вместо ее параметров можно подставить конкретные типы и получить точное соответствие вызову. Если таких шаблонных функций несколько, то предпочтение отдается более специфичной. Если подобрать шаблонную функцию не удается, то выбирается нешаблонная функция, при условии что ее аргументы можно преобразовать в типы параметров. Наконец, если ничего другого не остается, но существует функция с подходящим именем и переменным числом аргументов, то вызывается она. Заметим, что некоторые преобразования считаются *тривиальными* и подпадают под понятие *точного* соответствия, например преобразование из `T` в `const T`. На каждом шаге, если существует два или более подходящих варианта, ни один из которых не лучше другого, перегрузка считается неоднозначной, а программа – некорректной.

Процесс подстановки типов в шаблон функции – это то, что определяет окончательные типы параметров шаблона и степень их соответствия аргументам вызова функции. Этот процесс может приводить к неожиданным результатам, поэтому стоит рассмотреть его подробнее.

Подстановка типов в шаблонных функциях

Следует четко различать два шага конкретизации шаблона функции с целью установления соответствия с конкретным вызовом. Сначала типы параметров шаблона выводятся из типов аргументов (этот процесс называется выведением типов). После того как типы выведены, вместо всех типов параметров подставляются конкретные типы (этот процесс называется **подстановкой типов**). Различие становится более очевидным, если у функции несколько параметров.

Выведение и подстановка типов

Выведение и подстановка типов тесно связаны, но это не одно и то же. Выведение – это *высказывание гипотез*: какими должны быть типы параметров шаблона, чтобы получилось соответствие вызову? Конечно, компилятор на самом деле не гадает, а руководствуется точными правилами, прописанными в стандарте. Рассмотрим пример:

```
template <typename T> void f(T i, T* p) {
    std::cout << "f(T, T8)" << std::endl;
}

int i;
f(5, &i);   // T == int
f(5l, &i);  // ?
```

При рассмотрении первого вызова мы можем из первого аргумента вывести, что параметр шаблона `T` должен быть равен `int`. Таким образом, `int` подставляется вместо `T` в обоих параметрах функции. Шаблон конкретизируется как `f(int, int*)`, и это точно совпадает с типами аргументов. При рассмотрении второго вызова из первого аргумента можно вывести, что `T` должен быть равен `long`. Или же из второго аргумента можно было бы вывести, что `T` должен быть равен `int`. Из-за этой неоднозначности процесс выведения аргументов завершается неудачно. Если бы этот перегруженный вариант был единственным, то программа не откомпилировалась бы. Если существуют другие варианты, то они рассматриваются по очереди, и последней – функция с переменным числом аргументов `f(...)`. Отметим важную деталь: при выведении типов параметров шаблона преобразования не рассматриваются – выведение `int` в качестве `T` дало бы `f(int, int*)` для второго вызова, что подошло бы для вызова функции `f(long, int*)` с преобразованием первого аргумента. Однако этот вариант не рассматривается вовсе, и выведение типов признается неоднозначным.

Неоднозначность выведения можно разрешить, явно задав типы параметров шаблона, что устраняет необходимость в их выведении:

```
f<int>(5l, &i); // T == int
```

Теперь выведение типов вообще не производится: мы знаем, каков тип `T`, из вызова функции, поскольку он указан явно. Но подстановка типов все равно имеет место – первый параметр имеет тип `int`, а второй – тип `int*`. Функцию можно вызвать, если преобразовать первый аргумент. Можно также направить выведение в нужном направлении другим способом:

```
f<long>(5l, &i); // T == long
```

И снова выведение не требуется, поскольку тип `T` уже известен. Процесс подстановки происходит как обычно, и мы получаем `f(long, long*)`. Эту функцию нельзя вызвать, если второй аргумент имеет тип `int*`, поскольку преобразования из `int*` в `long*` не существует. Таким образом, программа не компилируется. Отметим, что, явно задавая типы, мы заодно указываем, что `f()` должна быть

шаблонной функцией. Нешаблонные перегруженные варианты f() больше не рассматриваются. С другой стороны, если существует более одной шаблонной функции f(), то такие варианты рассматриваются обычным порядком, только результат выведения типов уже предопределен явным заданием.

У шаблонных функций могут быть аргументы по умолчанию, как и у нешаблонных. Однако значения этих аргументов не принимаются во внимание при выведении типов (в C++11 параметры-типы шаблонных функций могут иметь значения по умолчанию, так что мы получаем некоторую альтернативу). Рассмотрим пример:

```
void f(int i, int j = 1) { std::cout << "f(int, int)" << std::endl; } // 1
template <typename T>
void f(T i, T* p = NULL) {std::cout << "f(T, T*)" << std::endl; }     // 2

int i;
f(5);    // 1
f(5l);   // 2
```

Первый вызов точно соответствует нешаблонной функции f(int, int), у которой второй аргумент имеет значение по умолчанию 1. Отметим, что с тем же успехом можно было бы объявить функцию в виде f(int i, int j = 1L), где значение по умолчанию имеет тип long. Тип аргумента по умолчанию не играет роли; если его можно преобразовать в заданный тип параметра, то он будет использоваться, иначе строка 1 программы не откомпилируется. Второй вызов точно соответствует шаблонной функции f(T, T*) при T == long и значении по умолчанию NULL второго аргумента. Опять-таки не важно, что тип этого значения – не long*.

Теперь мы понимаем разницу между выведением и подстановкой типов. Выведение типов может оказаться неоднозначным, если конкретные типы выводятся из разных аргументов. Если такое происходит, значит, вывести типы невозможно, и этой шаблонной функцией воспользоваться нельзя. Подстановка типов никогда не бывает неоднозначной – если мы знаем тип T, то можем просто подставить его во все места, где T встречается в определении функции. Этот процесс тоже может завершиться неудачно, но по другой причине.

Неудавшаяся подстановка

После того как типы параметров шаблона выведены, подстановка типов производится чисто механически:

```
template <typename T> T* f(T i, T& j) { j = 2*i; return new T(i); }
int i = 5, j = 7;
const int* p = f(i, j);
```

В этом примере тип T выводится из первого аргумента как int. Его можно вывести и из второго аргумента, тоже как int. Заметим, что тип возвращаемого значения при выведении типов не используется. Поскольку тип T выводится единственным образом, мы можем перейти к подстановке int вместо всех вхождений T в определение функции:

```
int* f(int i, int& j) { j = 2*i; return new int(i); }
```

Однако не все типы одинаковы, одни пользуются большей свободой, чем другие. Рассмотрим такой код:

```
template <typename T> void f(T i, typename T::t& j) {
    std::cout << "f(T, T::t)" << std::endl;
}
template <typename T> void f(T i, T j) {
    std::cout << "f(T, T)" << std::endl;
}

struct A {
    struct t { int i; };
    t i;
};

A a{5};
f(a, a.i);  // T == A
f(5, 7);    // T == int
```

При рассмотрении первого вызова компилятор выводит параметр шаблона `T` как тип `A` и из первого, и из второго аргумента; первый аргумент – это значение типа `A`, а второй – ссылка на значение вложенного типа `A::t`, который соответствует `T::t`, если придерживаться сделанного ранее вывода о том, что `T` – это `A`. Второй перегруженный вариант дает конфликтующие значения `T` при выведении из двух аргументов, поэтому его использовать нельзя. Итак, вызывается первый вариант.

Теперь присмотримся пристальнее ко второму вызову. Тип `T` выводится как `int` из первого аргумента для обоих перегруженных вариантов. Однако подстановка `int` вместо `T` приводит к странному результату во втором аргументе первого варианта – `int::t`. Конечно, такая конструкция не откомпилируется; `int` – не класс и не может иметь вложенных типов. На самом деле можно было бы ожидать, что первый перегруженный вариант шаблона не будет компилироваться ни для какого типа `T`, который не является классом или не может иметь вложенный тип с именем `t`. В самом деле, попытка подставить `int` вместо `T` в первую шаблонную функцию не удается из-за недопустимого типа второго аргумента. Однако эта неудача не означает, что вся программа не откомпилируется. Вместо этого она молчаливо игнорируется, и перегруженный вариант, оказавшийся некорректным, удаляется из множества перегруженных вариантов. Затем разрешение перегрузки продолжается как обычно. Конечно, может обнаружиться, что ни один перегруженный вариант не соответствует вызову функции, и тогда программа все-таки не откомпилируется, но в сообщении об ошибке не будет упоминаний о некорректном типе `int::t`, а будет просто сказано, что нет функций, которые можно было бы вызвать.

И снова важно различать неудавшееся выведение типов и неудавшуюся подстановку типов. Первое можно вообще исключить из рассмотрения:

```
f<int>(5, 7); // T == int
```

Теперь выведение не требуется, но подстановка `int` вместо `T` все равно должна произойти, и эта подстановка даст недопустимое выражение в первом перегруженном варианте. Как и раньше, из-за неудавшейся подстановки этот кандидат на роль `f()` исключается из множества перегруженных вариантов, и разрешение перегрузки продолжается для остальных кандидатов (на этот раз успешно). Обычно на этом нашим упражнениям в перегрузке и пришел бы конец: шаблон порождает некомпилируемый код, и вся программа тоже не компилируется. По счастью, C++ в этой конкретной ситуации более снисходителен и допускает специальное исключение, о котором необходимо знать.

Неудавшаяся подстановка – не ошибка

Правило, согласно которому неудавшаяся подстановка, дающая выражение, недопустимое для указанных или выведенных типов, не делает всю программу некорректной, имеет название: **неудавшаяся подстановка – не ошибка** (Substitution Failure Is Not An Error – SFINAE). Это правило критически важно для использования шаблонных функций в C++; без него было бы невозможно написать многие в остальных отношениях вполне допустимые программы. Рассмотрим следующий перегруженный шаблон, различающий обычные указатели и указатели на члены:

```
template <typename T> void f(T* i) {        // 1
    std::cout << "f(T*)" << std::endl;
}
template <typename T> void f(int T::* p) {  // 2
    std::cout << "f(T::*)" << std::endl;
}

struct A {
    int i;
};

A a;
f(&a.i);    // 1
f(&A::i);   // 2
```

Пока все хорошо – в первый раз функция вызывается с указателем на переменную `a.i`, и тип `T` выводится как `int`. Во втором случае функции передается указатель на член класса `A`, и `T` выводится как `A`. А теперь давайте вызовем `f()`, передав ей указатель на другой тип:

```
int i;
f(&i);   // 1
```

Первый перегруженный вариант по-прежнему работает, и это именно то, что мы хотим вызвать. Но второй вариант мало того что подходит хуже, так еще и недопустим – при попытке подставить `int` вместо `T` произошла бы синтаксическая ошибка. Компилятор замечает эту ошибку и молча игнорирует, как и сам перегруженный вариант.

Заметим, что правило SFINAE распространяется не только на недопустимые типы, например ссылки на несуществующие члены класса. Подстановка может завершиться неудачей по многим другим причинам:

```
template <size_t N> void f(char(*)[N % 2] = NULL) {       // 1
    std::cout << "N=" << N << " нечетно" << std::endl;
}
template <size_t N> void f(char(*)[1 - N % 2] = NULL) {   // 2
    std::cout << "N=" << N << " четно" << std::endl;
}

f<5>();
f<8>();
```

Здесь параметр шаблона – значение, а не тип. У нас есть два перегруженных шаблона, каждый из которых принимает указатель на массив символов, а выражения размера массива допустимы только при некоторых значениях `N`. Точнее, массив нулевого размера в C++ недопустим. Следовательно, первый вариант допустим, только если `N % 2` не равно нулю, т. е. для нечетных `N`. Аналогично второй вариант допустим только для четных `N`. Функции не переданы аргументы, т. е. мы намереваемся использовать аргументы по умолчанию. Оба перегруженных варианта были бы неоднозначны, если бы не тот факт, что для обоих вызовов один из вариантов отбрасывается из-за неудавшейся подстановки.

Предыдущий пример очень лаконичен. Точнее, выведение значения параметра шаблона, эквивалент выведения типов для числовых параметров, отключено в силу явного задания. Можно вернуть выведение, и все равно подстановка может завершиться неудачей в зависимости от того, допустимо выражение или нет:

```
template <typename T, size_t N = T::N>
void f(T t, char(*)[N % 2] = NULL) {
  std::cout << "N=" << N << " нечетно" << std::endl;
}
template <typename T, size_t N = T::N>
void f(T t, char(*)[1 - N % 2] = NULL) {
    std::cout << "N=" << N << " четно" << std::endl;
}

struct A {
    enum {N = 5};
};
struct B {
    enum {N = 8};
};

A a;
B b;
f(a);
f(b);
```

Теперь компилятор должен вывести тип из первого аргумента. Для первого вызова, `f(a)`, тип `A` выводится легко. Второй параметр шаблона, `N`, вывести невозможно, поэтому используется значение по умолчанию (тут мы попадаем на территорию C++11). После того как оба параметра шаблона выведены, мы переходим к подстановке – заменяем `T` на `A`, а `N` – на `5`. Эта подстановка не проходит для второго перегруженного варианта, но проходит для первого. Поскольку во множестве перегруженных вариантов остался только один кандидат, разрешение перегрузки успешно завершается. Аналогично второй вызов, `f(b)`, завершается выбором второго варианта.

Отметим, что SFINAE не защищает нас ни от каких синтаксических ошибок, которые могут произойти во время конкретизации шаблона. Например, даже если параметры шаблона выведены и аргументы шаблона подставлены, все равно может получиться некорректная шаблонная функция:

```
template <typename T> void f(T) { std::cout << sizeof(T::i) << std::endl; }
void f(...) { std::cout << "f(...)" << std::endl; }
f(0);
```

Этот фрагмент кода очень похож на рассмотренные выше, с одним исключением – мы не узнаем, что в перегруженном варианте шаблона предполагается, что тип `T` является классом и имеет член данных `T::i`, пока не исследуем тело функции. Но тогда уже будет слишком поздно, потому что разрешение перегрузки производится только на основании объявления функции – параметров, аргументов по умолчанию и типа возвращаемого значения (последнее не применяется для выведения типов или выбора наилучшего перегруженного варианта, но также подвергается подстановке типов и подчиняется правилу SFINAE). После того как шаблон конкретизирован и выбран механизмом разрешения перегрузки, никакие синтаксические ошибки, в частности недопустимое выражение в теле функции, уже не игнорируются. Такая неудавшаяся подстановка – самая настоящая ошибка. Полный перечень контекстов, в которых неудавшаяся подстановка рассматривается и не рассматривается как ошибка, приведен в стандарте и был значительно расширен в версии C++11.

Теперь, когда мы знаем, зачем правило SFINAE было добавлено в C++ и как оно применяется к разрешению перегрузки шаблонных функций, можно попытаться намеренно вызвать неудачу подстановки, чтобы повлиять на разрешение перегрузки.

Управление разрешением перегрузки

Правило SFINAE, согласно которому неудавшаяся подстановка не является ошибкой, необходимо было добавить в язык, просто для того чтобы сделать возможными некоторые узко определенные шаблонные функции. Но изобретательность программистов на C++ не знает границ, и потому SFINAE было переосмыслено и стало использоваться для ручного управления множеством

перегруженных вариантов посредством преднамеренного провоцирования неудавшейся подстановки.

Рассмотрим во всех подробностях, как SFINAE можно использовать, чтобы вышибить нежелательный перегруженный вариант. Заметим, что в большей части этой главы мы опираемся на стандарт C++11, а позже и на некоторые средства из C++14. Возможно, вы захотите почитать об этих недавних добавлениях в язык; в таком случае обратитесь к разделу «Для дальнейшего чтения» в конце этой главы.

Простое применение SFINAE

Начнем с очень простой задачи: имеется некоторый общий код, который мы хотим применять к объектам всех типов, кроме встроенных. А для целых и других встроенных типов мы написали специальную версию этого кода. Эту задачу можно решить, явно перечислив все встроенные типы в наборе перегруженных функций. И не забудем простые указатели, это тоже встроенные типы. И ссылки. И константные указатели. При некотором усилии этот подход можно довести до успешного конца. Но, пожалуй, проще каким-то образом проверить, является наш тип классом или нет. Нужно лишь найти нечто такое, чем классы обладают, а встроенные типы нет. Очевидное различие – указатели на члены. Объявление любой функции, в котором используется синтаксис указателя на член, будет отвергнуто на этапе подстановки, а нам остается предоставить перегруженный вариант «последней надежды», который перехватит все такие вызовы. И мы уже знаем, как это сделать:

```
template <typename T>
void f(int T::*) { std::cout << "T - класс" << std::endl; }
template <typename T>
void f(...) { std::cout << "T - не класс" << std::endl; }

struct A {
};

f<int>(0);
f<A>(0);
```

Выведение типа нас здесь не интересует. Мы задаем вопрос: «Является ли `A` классом? Является ли `int` классом?» В первом перегруженном варианте используется синтаксис указателя на член, поэтому подстановка завершается неудачно для любого экземпляра типа `T`, не являющегося классом. Но неудача подстановки – не ошибка (благодаря SFINAE!), и мы переходим к рассмотрению другого перегруженного варианта. Кроме того, интерес здесь представляет идея *дважды универсального* шаблона функции `f(...)` – она принимает аргументы любого типа, даже без шаблона. Но тогда зачем нужен шаблон? Конечно, для того чтобы при вызове с явно заданным типом, например `f<int>()`, эта функция рассматривалась как один из возможных перегруженных вариантов (напомним, что, задавая тип параметра шаблона, мы исключаем из рассмотрения все нешаблонные функции).

Если нам часто приходится проверять, является ли тип классом, то вряд ли мы захотим добавлять конструкцию SFINAE для каждой вызываемой функции. Предпочтительнее был бы специальный фрагмент кода, который проверяет, является ли тип классом, и устанавливает константу времени компиляции в `true` или `false`. Тогда ее значение можно было бы использовать совместно с различными приемами условной компиляции:

```
template <typename T> ??? test(int T::*);  // выбирается, если тип T - класс
template <typename T> ??? test(...);       // выбирается в противном случае
template <typename T>
struct is_class {
    ...
    static constexpr bool value = ???;  // сделать равным true, если T - класс
};
```

Теперь нужно постараться и сделать так, чтобы значение константы в нашем вспомогательном классе `is_class` зависело от перегруженного варианта, который был бы выбран при вызове функции `test<T>()`, но без фактического вызова (вызов производится во время выполнения, а нам нужна информация на этапе компиляции – обратите внимание, что мы даже не удосужились определить тела функций, поскольку не планируем их вызывать).

Последний кусочек, связывающий все это воедино, – контекст времени компиляции, в котором мы определяем, какие функции были бы вызваны. Один такой контекст дает оператор `sizeof` – компилятор обязан вычислять выражение `sizeof(T)` на этапе компиляции для любого типа `T`, а раз так, то если нам удастся вычислить разные значения в выражении `sizeof()`, то мы будем знать, какой тип был выбран в результате разрешения перегрузки. Различить две функции мы можем по типу возвращаемого значения, поэтому определим их так, чтобы эти типы были разного размера, и посмотрим, что получится:

```
template<typename T>
class is_class {
    // Выбирается, если тип C - класс
    template<typename C> static char test(int C::*);
    // Выбирается в противном случае
    template<typename C> static int test(...);
    public:
    static constexpr bool value = sizeof(test<T>(0)) == 1;
};

struct A {
};
std::cout << is_class<int>::value << std::endl;
std::cout << is_class<A>::value << std::endl;
```

Мы также скрыли функцию `test` внутри класса `is_class` – ни к чему засорять глобальное пространство имен именами функций, которые не предполагается вызывать. Заметим, что C++, строго говоря, не гарантирует, что размер `int` не совпадает с размером `char`. С этой и другими деталями, интересными для педантов, мы разберемся в следующем разделе.

До выхода стандарта C++11 и появления в нем ключевого слова constexpr мы вынуждены были использовать для достижения того же эффекта ключевое слово enum, и такие примеры до сих пор встречаются, так что важно уметь распознавать их:

```
template <typename T>
struct is_class {
    ...
    enum { value = sizeof(test<T>(NULL)) == 1 };
};
```

Но при работе с C++11 следует иметь в виду, что в нем определен стандартный тип для такого рода константных объектов этапа компиляции – integral_constant. Этот тип может принимать значения true и false, но, помимо этого, добавляет несколько деталей, которых ожидают другие классы STL, поэтому не надо изобретать велосипед:

```
namespace implementation {
    // Выбирается, если тип C - класс
    template<typename C> static char test(int C::*);
    // Выбирается в противном случае
    template<typename C> static int test(...);
}

template <class T>
struct is_class :
    std::integral_constant<bool, sizeof(implementation::test<T>(0)) ==
                                 sizeof(char)> {};

struct A {
};

static_assert(!is_class<int>::value, "int is a class?");
static_assert(is_class<A>::value, "A is not a class?");
```

Не нужно ждать выполнения программы и для того, чтобы проверить, сработала наша реализация is_class или нет, – предложение static_assert позволяет проверить это во время компиляции, а ведь именно на этом этапе мы и собираемся проверять результат is_class.

Конечно, в C++11 вообще нет причин писать показанный выше код, потому что стандарт предоставляет тип std::is_class, который используется и работает точно так, как наш (с тем отличием, что этот тип не считает объединения union классами). Однако, заново реализовав is_class, мы освоили применение SFINAE в простейшем контексте. Дальше будет труднее.

Продвинутое применение SFINAE

Далее рассмотрим задачу, которую просто сформулировать, но нелегко решить. Мы хотим написать обобщенный код, способный работать с контейнерным объектом произвольного типа T. В какой-то момент мы захотим отсортировать данные в этом контейнере. Предполагается, что если контейнер предоставляет

функцию-член T::sort(), то она и дает наилучший способ сортировки (автор контейнера, вероятно, знает, как организованы данные). Если такой функции-члена нет, но контейнер представляет собой последовательность с функциями-членами begin() и end(), то мы можем вызвать для этой последовательности функцию std::sort(). В противном случае мы не знаем, как сортировать данные, и программа не должна компилироваться.

Наивная попытка решить эту задачу могла бы выглядеть так:

```
template <typename T> void best_sort(T& x, bool use_member_sort) {
    if (use_member_sort) x.sort();
    else std::sort(x.begin(), x.end());
}
```

Проблема в том, что этот код не откомпилируется, если тип T не предоставляет функцию-член sort(), пусть даже мы не собираемся ее использовать. Может показаться, что ситуация безнадежна: чтобы вызвать x.sort() хотя бы для некоторых типов x, в программе где-то должен присутствовать код x.sort(). Но даже тогда код не будет компилироваться для типов x, в которых эта функция-член не определена. Однако выход все-таки существует – вообще говоря, шаблоны C++ не приводят к синтаксическим ошибкам, если не конкретизируются (это чрезмерное упрощение, и точные правила весьма сложны, но пока сойдет и так). Например, рассмотрим следующий код:

```
class Base {
    public:
    Base() : i_() {}
    virtual void increment(long v) { i_ += v; }
    private:
    long i_;
};

template <typename T>
class Derived : public T {
    public:
    Derived() : T(), j_() {}
    void increment(long v) { j_ += v; T::increment(v); }
    void multiply(long v) { j_ *= v; T::multiply(v); }
    private:
    long j_;
};

int main() {
    Derived<Base> d;
    d.increment(5);
}
```

Здесь мы имеем производный класс, наследующий параметру своего шаблона. Функции-члены increment() и multiply() производного класса вызывают соответствующие функции-члены базового класса. Но в базовом классе есть только функция-член increment()! Тем не менее код компилируется при условии, что мы нигде не вызываем Derived::multiply(). Потенциальная синтаксиче-

ческая ошибка не становится реальной, если только шаблон не генерирует действительно недопустимый код.

Заметим, что это работает, только если потенциально недопустимый код зависит от типа параметра шаблона и компилятор не может проверить его корректность до момента конкретизации шаблона (в нашем случае `Derived::multiply()` вообще никогда не конкретизируется). Если потенциально недопустимый код не зависит от параметра шаблона, то он уже не *потенциально недопустимый*, а просто недопустимый, поэтому отвергается:

```
template <typename T>
class Derived : public Base {
    public:
    Derived() : Base(), j_() {}
    void increment(long v) { j_ += v; Base::increment(v); }
    // multiply() недопустимо дл любого T:
    void multiply(long v) { j_ *= v; Base::multiply(v); }
    private:
    T j_;
};
```

Этот пример указывает нам возможный путь к успеху – коль скоро мы сумеем скрыть обращение к `x.sort()` в шаблоне, который не конкретизируется, если нет абсолютной уверенности в том, что код откомпилируется, то синтаксической ошибки не будет. Одно из мест, где можно скрыть код, – специализация шаблона класса. Нам нужно что-то в этом роде:

```
template <typename T> struct fast_sort_helper;

template <typename T>
struct fast_sort_helper<???> {       // специализация для сортировки функцией-членом
    static void fast_sort(T& x) {
        std::cout << "Сортирует T::sort" << std::endl;
        x.sort();
    }
};

template <typename T>
struct fast_sort_helper<???> {       // специализация для std::sort
    static void fast_sort(T& x) {
        std::cout << "Сортирует std::sort" << std::endl;
        std::sort(x.begin(), x.end());
    }
};
```

Здесь мы имеем общий шаблон для вспомогательного типа `fast_sort_helper` и две специализации, которые мы пока не знаем, как конкретизировать. Если бы мы могли конкретизировать только правильную специализацию, но не вторую, то результатом компиляции вызова `fast_sort_helper::fast_sort(x)` было бы либо `x.sort()`, либо `std::sort(x.begin(), x.end())`. Если ни одна специализация не конкретизируется, то наша программа не откомпилируется, потому что тип `fast_sort_helper` неполон.

Но теперь у нас на руках проблема курицы и яйца. Чтобы решить, какую специализацию конкретизировать, мы должны попытаться откомпилировать `x.sort()`, чтобы узнать, получится это или нет. Если не получилось, то следует избегать конкретизации первой специализации `fast_sort_helper` – именно для того, чтобы не компилировать `x.sort()`. Мы должны попытаться откомпилировать `x.sort()` в каком-то контексте, где неудача компиляции не будет перманентной. SFINAE как раз и дает нам такой контекст – мы должны объявить шаблонную функцию с аргументом, тип которого корректно определен, только если выражение `x.sort()` допустимо. В C++11 нет нужды ходить вокруг да около – если требуется, чтобы некоторый тип был допустим, то мы ссылаемся на него с помощью `decltype()`:

```
template <typename T> ? test_sort(decltype(&T::sort));
```

Мы только что объявили шаблонную функцию, которая приведет к неудавшейся подстановке, если в выведенном типе `T` отсутствует функция-член `sort()` (мы упомянули тип указателя на эту функцию-член). Мы не собираемся фактически вызывать эту функцию; она нужна нам только на этапе компиляции, чтобы сгенерировать правильную специализацию `fast_sort_helper`. Если подстановка завершится неудачно, то все-таки нужно, чтобы процесс разрешения перегрузки был успешным, поэтому необходим перегруженный вариант, который, однако, будет выбираться, только если альтернатив не осталось. Мы знаем, что функции с переменным числом аргументов выбираются в последнюю очередь. Чтобы узнать, какой перегруженный вариант был выбран, мы снова можем применить `sizeof` к типу возвращаемого функцией значения, для чего нужно, чтобы размеры этих типов были различны (поскольку мы не планируем их вызывать, то совершенно не важно, что именно мы будем возвращать, можно взять любой тип). В предыдущем разделе мы использовали для этой цели `char` и `int`, предполагая, что `sizeof(int)` больше `sizeof(char)`. Скорее всего, на любом реальном оборудовании так оно и есть, но стандарт этого не гарантирует. При небольшом усилии можно создать два типа, размеры которых гарантированно различны:

```
struct yes { char c; };
struct no { char c; yes c1; };
static_assert(sizeof(yes) != sizeof(no),
              "Возьмите что-нибудь другое для типов yes/no");

template <typename T> yes test_sort(decltype(&T::sort));
template <typename T> no test_sort(...);
```

Теперь точно так же, как делали это раньше в тесте `is_class`, мы можем сравнить размер типа, возвращаемого выбранным перегруженным вариантом, с размером типа `yes` и тем самым определить, имеется ли функция-член `T::sort` в произвольном типе `T`. Чтобы выбрать правильную специализацию шаблона `fast_sort_helper`, необходимо использовать этот размер в качестве параметра шаблона, а это означает, что в дополнение к параметру-типу `T` наш шаблон

fast_sort_helper должен принимать целочисленный параметр. Вот теперь мы готовы собрать все вместе:

```
struct yes { char c; };
struct no { char c; yes c1; };
static_assert(sizeof(yes) != sizeof(no),
              "Возьмите что-нибудь другое для типов yes/no");
template <typename T> yes test_sort(decltype(&T::sort));
template <typename T> no test_sort(...);

template <typename T, size_t s>
struct fast_sort_helper;                    // в общем случае имеем неполный тип
template <typename T>
struct fast_sort_helper<T, sizeof(yes)> { // специализация для yes
    static void fast_sort(T& x) {
        std::cout << "Sorting with T::sort" << std::endl;
        x.sort();                           // не компилируется, если не выбрана
    }
};

template <typename T>
void fast_sort(T& x) {
    fast_sort_helper<T, sizeof(test_sort<T>(NULL))>::fast_sort(x);
}

class A {
    public:
    void sort() {}
};

class C {
    public:
    void f() {}
};

A a; fast_sort(a);    // компилируется, вызывается a.sort()
C c; fast_sort(c);    // не компилируется
```

Мы объединили шаблон функции fast_sort(T&), необходимый для выведения типа аргумента, с шаблоном класса fast_sort_helper<T, s>. Этот шаблон класса позволяет воспользоваться частичными специализациями и скрыть потенциально недопустимый код внутри шаблона, который не конкретизируется, если нет уверенности в безошибочной компиляции.

Пока что мы решили только половину задачи – мы умеем определять, что нужно вызвать функцию-член sort(), но можно ли вызывать std::sort, мы не знаем. Чтобы это было возможно, в контейнере должны быть определены функции-члены begin() и end(). Конечно, тип данных должен еще допускать сравнение с помощью оператора operator<(), но это необходимо для любой сортировки (можно было бы также добавить поддержку произвольной функции сравнения, как делает std::sort). Вторую часть задачи можно решить так же, как первую, проверив наличие функций-членов begin() и end() с помощью двух аргументов типа указателя на член:

```
template <typename T> ??? test_sort(decltype(&T::begin),
decltype(&T::end));
```

Чтобы другой перегруженный вариант – тот, что проверяет наличие функции-члена `sort()`, – принимал участие в разрешении перегрузки наряду с этим, придется перейти от одного параметра к двум. Функция с переменным числом аргументов и так уже принимает любое число аргументов. Наконец, вопрос уже не сводится к «да» или «нет» – нам нужны три возвращаемых типа, чтобы различить контейнеры, предоставляющие функцию-член `sort()`, контейнеры, предоставляющие `begin()` и `end()`, и контейнеры, не предоставляющие ни того, ни другого:

```
struct have_sort { char c; };
struct have_range { char c; have_sort c1; };
struct have_nothing { char c; have_range c1; };
template <typename T> have_sort test_sort(decltype(&T::sort),
                                          decltype(&T::sort));
template <typename T> have_range test_sort(decltype(&T::begin),
                                           decltype(&T::end));
template <typename T> have_nothing test_sort(...);
template <typename T, size_t s> struct fast_sort_helper;

template <typename T>
struct fast_sort_helper<T, sizeof(have_sort)> {
    static void fast_sort(T& x) {
        std::cout << "Сортирует T::sort" << std::endl;
        x.sort();
    }
};

template <typename T>
struct fast_sort_helper<T, sizeof(have_range)> {
    static void fast_sort(T& x) {
        std::cout << "Сортирует std::sort" << std::endl;
        std::sort(x.begin(), x.end());
    }
};

template <typename T>
void fast_sort(T& x) {
    fast_sort_helper<T, sizeof(test_sort<T>(NULL, NULL))>::fast_sort(x);
}

class A {
    public:
    void sort() {}
};

class B {
    public:
    int* begin() { return i; }
    int* end() { return i + 10; }
    int i[10];
};
```

```
class C {
    public:
    void f() {}
};

A a; fast_sort(a);   // компилируется, используется a.sort()
B b; fast_sort(b);   // компилируется, используется std::sort(b.begin(), b.end())
C c; fast_sort(c);   // не компилируется
```

Вызов `fast_sort()` для контейнера, который нельзя отсортировать ни тем, ни другим способом, заканчивается загадочным сообщением о неполноте типа `fast_sort_helper`. Сообщение об ошибке можно было бы сделать более внятным, если предоставить специализацию для результата разрешения `have_nothing` и использовать в нем `static_assert`:

```
template <typename T>
struct fast_sort_helper<T, sizeof(have_nothing)> {
    static void fast_sort(T& x) {
        static_assert(sizeof(T) < 0, "Нет способа сортировки");
    }
};
```

Заметим, что мы вынуждены воспользоваться условием, зависящим от типа шаблона. Нельзя просто написать

```
static_assert(false, "Нет способа сортировки");
```

Это утверждение не выполняется безусловно, даже если специализация `fast_sort_helper` для результата `have_nothing` никогда не конкретизируется. Разумеется, выражение `sizeof(T) < 0` тоже всегда ложно, но компилятору не разрешено принимать решение заранее.

Только что описанное решение раскрывает детали реализации, в которой правило SFINAE используется для проверки ошибок компиляции в таком контексте, где они игнорируются, чтобы избежать тех же ошибок в контексте, где они были бы фатальны. Этот способ написания такого кода весьма поучительный, но далеко не самый компактный. C++11 предлагает несколько средств, намного упрощающих условную компиляцию. Одно из них, `std::enable_if`, – шаблон класса, параметризованный булевым значением, который активирует неудачную подстановку, если это значение равно `false` (его реализация могла бы напомнить наш шаблон класса `fast_sort_helper`). Обычно `std::enable_if` используется в типе возвращаемого значения перегруженной функции; если выражение равно `false`, то подстановка возвращаемого типа завершается неудачей, и функция исключается из множества перегруженных вариантов. Благодаря этому шаблону наше решение можно упростить:

```
struct have_sort { char c; };
struct have_range { char c; have_sort c1; };
struct have_nothing { char c; have_range c1; };
template <typename T> have_sort test_sort(decltype(&T::sort),
                                          decltype(&T::sort));
```

```
template <typename T> have_range test_sort(decltype(&T::begin),
                                           decltype(&T::end));
template <typename T> have_nothing test_sort(...);

template <typename T>
typename std::enable_if<sizeof(test_sort<T>(NULL, NULL)) ==
                        sizeof(have_sort)>::type fast_sort(T& x) {
    std::cout << "Сортирует T::sort" << std::endl;
    x.sort();
}
template <typename T>
typename std::enable_if<sizeof(test_sort<T>(NULL, NULL)) ==
                        sizeof(have_range)>::type fast_sort(T& x) {
    std::cout << "Сортирует std::sort" << std::endl;
    std::sort(x.begin(), x.end());
}
```

Здесь мы избавились от вспомогательного класса и его специализаций, а вместо этого напрямую исключаем один или оба перегруженных варианта `fast_sort()`. Как и раньше, если исключены оба варианта, то выдается невнятное сообщение об ошибке, но мы могли бы предоставить третий перегруженный вариант с помощью `static_assert`.

Однако мы еще не все сделали. Что будет, если контейнер предоставляет одновременно `sort()` и `begin()`/`end()`? Тогда у нас появляется два допустимых перегруженных варианта `test_sort()`, и программа перестает компилироваться из-за неоднозначного разрешения перегрузки. Мы могли бы попытаться сделать один вариант предпочтительнее другого, но это непросто и не обобщается на более сложные случаи. Вместо этого следует пересмотреть способ задания вопроса: «Обладает ли контейнер функцией-членом `sort()` или парой `begin()`/`end()`?» Наша проблема в том, как мы сформулировали проверяемое условие, – возможность ответа «есть то и другое» в вопросе даже не рассматривается, поэтому неудивительно, что мы испытываем затруднения с ответом. Вместо одного вопроса «или-или» мы должны задать два отдельных вопроса:

```
struct yes { char c; };
struct no { char c; yes c1; };
template <typename T> yes test_have_sort(decltype(&T::sort));
template <typename T> no test_have_sort(...);
template <typename T> yes test_have_range(decltype(&T::begin),
                                          decltype(&T::end));
template <typename T> no test_have_range(...);
```

Теперь мы должны явно решить, что делать, если ответ на оба вопроса – *да*. Эту проблему можно решить с помощью `std::enable_if` и не слишком сложных логических выражений. А можно вернуться к нашему вспомогательному шаблону класса, у которого теперь должно быть два дополнительных целочисленных параметра вместо одного, и рассмотреть все четыре возможные комбинации ответов да-нет на два наших вопроса:

```
template <typename T, bool have_sort, bool have_range> struct
fast_sort_helper;

template <typename T>
struct fast_sort_helper<T, true, true> {
    static void fast_sort(T& x) {
        std::cout << "Сортирует T::sort, std::sort игнорируется"
                  << std::endl;
        x.sort();
    }
};

template <typename T>
struct fast_sort_helper<T, true, false> {
    static void fast_sort(T& x) {
        std::cout << "Сортирует T::sort" << std::endl;
        x.sort();
    }
};

template <typename T>
struct fast_sort_helper<T, false, true> {
    static void fast_sort(T& x) {
        std::cout << "Сортирует std::sort" << std::endl;
        std::sort(x.begin(), x.end());
    }
};

template <typename T>
struct fast_sort_helper<T, false, false> {
    static void fast_sort(T& x) {
        static_assert(sizeof(T) < 0, "Нет способа сортировки");
    }
};

template <typename T>
void fast_sort(T& x) {
    fast_sort_helper<T,
        sizeof(test_have_sort<T>(NULL)) == sizeof(yes),
        sizeof(test_have_range<T>(NULL, NULL)) ==
            sizeof(yes)>::fast_sort(x);
}

class AB {
    public:
    void sort() {}
    int* begin() { return i; }
    int* end() { return i + 10; }
    int i[10];
};

AB ab; fast_sort(ab); // используется ab.sort(), наличие std::sort игнорируется
```

Какой способ выбрать – дело вкуса; `std::enable_if` – хорошо знакомая идиома, ясно выражающая намерения автора, но ошибки в логических выражениях,

которые должны быть взаимно исключительными и покрывать все возможные случаи, трудно отлаживать. Громоздкость и сложность кода зависят от конкретной задачи.

Попробуем применить нашу функцию fast_sort() к нескольким реальным контейнерам. Например, в контейнере std::list есть как функция-член sort(), так и пара begin()/end(), тогда как в std::vector функции-члена sort() нет.

```
std::list<int> l; std::vector<int> v;
... сохранить в контейнерах какие-то данные ...
fast_sort(l);   // должна быть вызвана l.sort()
fast_soft(v);   // должна быть вызвана std::sort
```

Результат довольно неожиданный – оба обращения к fast_sort() не компилируются. Если мы обрабатываем ответ «нет на оба вопроса» явно, то получаем статическое утверждение, предназначенное для контейнеров, не предоставляющих ни одного из нужных нам интерфейсов. Но как такое может быть? Мы точно знаем, что в std::list есть функция-член sort(), а в std::vector – функции begin() и end(). Проблема в том, что таких функций слишком много. Все они перегружены. Например, в std::list имеется два объявления sort(): обычная функция-член void sort() без аргументов и шаблонная функция-член void sort(Compare), принимающая объект сравнения, который должен использоваться вместо operator<(). Мы хотели вызвать sort() без аргументов, поэтому должны знать, будет этот вызов компилироваться или нет. Но задавали-то мы другой вопрос: «Существует ли функция T::sort?», а про аргументы не было сказано ни слова. И получили в ответ встречный вопрос: «А какую sort вы имели в виду?» В результате подстановка завершилась неудачей из-за неоднозначности (есть два возможных ответа, поэтому на вопрос нельзя ответить определенно). А мы неправильно интерпретировали неудавшуюся подстановку как признак того, что в типе вовсе нет функции-члена sort(). Чтобы исправить ситуацию, мы должны сузить вопрос – существует ли в типе функция-член sort() без аргументов? Для этого попытаемся привести указатель на функцию-член &T::sort к определенному типу, например void (T::*)(), т. е. к типу указателя на функцию-член без аргументов, возвращающую void. Если это получится, то функция-член желаемого типа существует, а остальные перегруженные варианты мы можем спокойно игнорировать. Нужно внести лишь небольшое изменение в тестовые функции SFINAE:

```
template <typename T> yes test_have_sort(
    decltype(static_cast<void (T::*)()>(&T::sort)));
template <typename T> yes test_have_range(
    decltype(static_cast<typename T::iterator (T::*)()>(&T::begin)),
    decltype(static_cast<typename T::iterator (T::*)()>(&T::end)));
```

Заметим, что для begin() и end() тоже необходимо указать возвращаемый тип (так же, как для sort(), разве что указать void проще). Стало быть, мы проверяем наличие функций begin() и end(), которые не принимают аргументов и возвращают значение вложенного типа T::iterator (перегруженный вариант, ранее

создавший нам проблемы с вектором, возвращает тип `T::const_iterator`). Теперь, если использовать `fast_sort()` для списка, то будет вызвана функция-член `sort()`, а `std::sort` для последовательности `begin()` [...] `end()` проигнорирована. Ее также можно вызвать для вектора, и тогда будет использована `std::sort` с неконстантными версиями `begin()` и `end()`, как и положено.

Во всех рассмотренных до сих пор примерах мы полностью управляли разрешением перегрузки – после исключения всех неудавшихся подстановок в результате применения SFINAE оставался только один перегруженный вариант. Возможно (но очень осторожно) сочетать ручное и автоматическое разрешение перегрузки. Например, можно было бы предоставить перегруженный вариант `fast_sort` для обычных С-массивов, которые можно сортировать функцией `std::sort()`, хотя у них нет функций-членов `begin()` и `end()`:

```
template <typename T, size_t N>
void fast_sort(T (&x)[N]) {
    std::sort(x, x + N);
}
```

Этот вариант предпочтительнее всех остальных и без применения SFINAE, поэтому мы имеем регулярную автоматическую перегрузку для обработки массивов, а для всего остального необходимо ручное управление множеством перегруженных вариантов с помощью SFINAE.

Задача решена, и наши тесты дают ожидаемые результаты. И все же осталось какое-то чувство неудовлетворенности от этой полной, с трудом одержанной победы. Мы никак не можем избавиться от смутного сомнения – не получилось ли так, что в попытке исключить все прочие перегруженные варианты функций-членов – те, которые мы не хотим вызывать ни в коем случае, – решение оказалось чрезмерно специфичным? Что, если единственная имеющаяся функция `sort()` возвращает не `void`? Возвращаемое значение можно проигнорировать, поэтому обращение к `x.sort()` компилироваться будет, а наш код – нет, поскольку мы проверяем только наличие функции-члена `sort()`, которая ничего не возвращает. Быть может, мы снова задаем не тот вопрос и наш подход к применению SFINAE следует пересмотреть?

Еще раз о продвинутом применении SFINAE

Рассмотренная в предыдущем разделе задача, вызов функции-члена `sort()`, если таковая существует, естественно подводит нас к вопросу: «Существует ли функция-член `sort()`?» Мы придумали два способа задать этот вопрос. Первый такой: «Имеется ли в классе функция-член с именем `sort` и каков тип указателя на нее?» Плохо, когда ответ имеет вид: «да, причем две, и тип указателя зависит от того, какая нужна». Второй способ задать вопрос: «Имеется ли в классе функция-член `void sort()`?» В этом случае плохо, когда ответ звучит так: «Нет, но функция, которую можно было бы вызвать взамен, имеется». Быть может, с самого начала следовало формулировать вопрос иначе: «Если я напишу `x.sort()`, то этот код откомпилируется?»

Чтобы снова не попасть в эту ловушку, рассмотрим другую задачу, где вынуждены атаковать подобную неоднозначность в лоб. Мы хотим написать функцию, которая будет умножать значение любого типа на заданное целое число:

```
template <typename T> ??? increase(const T& x, size_t n);
```

В общем случае `T` – пользовательский тип, в котором определены некоторые арифметические операторы. Есть много способов реализовать нужную нам операцию. Самое простое, когда существует функция `operator*`, которая принимает аргументы типа `T` и `size_t` (и необязательно возвращает значение типа `T`, так что мы пока не знаем, значение какого типа должна возвращать наша функция). Но даже тогда этот оператор мог бы быть функцией-членом класса `T` или свободной функцией. Если такого оператора не существует, то можно было бы попробовать написать `x *= n`, поскольку не исключено, что в типе `T` определен `operator*=`, который принимает целое число. Если и это не проходит, то можно было бы вычислить `x + x` и повторить сложение `n` раз, но только при условии, что в типе `T` определена функция-член `operator+`. Наконец, что, если ни один из этих способов не работает, но тип `T` можно преобразовать в другой тип, допускающий умножение на `n`? Тогда можно было бы пойти по этому пути. Ясно, однако, что было бы тщетно пытаться составить полный список вопросов вида «Существует ли такой оператор или такая функция-член?». Слишком много есть способов получить желаемую функциональность. А что, если вместо этого задать вопрос: «Является ли `x * n` допустимым выражением?» По счастью, в C++11 имеется оператор `decltype`, который именно это и позволяет сделать, – `decltype(x * n)` вернет тип результата умножения, если компилятор найдет способ вычислить это выражение. В противном случае выражение недопустимо, поэтому потенциальная ошибка обязательно произойдет во время подстановки типов, когда ее можно смело проигнорировать и продолжить поиски альтернативного способа вычисления результата. С другой стороны, если выражение `x * n` допустимо, то его тип, каким бы он ни был, вероятно, должен стать типом значения, возвращаемого функцией. К счастью, в C++11 можно заставить функцию вывести свой тип:

```
template <typename T>
auto increase(const T& x, size_t n) -> decltype(x * n)
{
    return x * n;
}
```

Эта функция откомпилируется, если выражение `x * n` допустимо, поскольку компилятор выведет его тип и подставит его вместо `auto`. Затем он начнет компилировать тело функции. Здесь ошибка компиляции была бы фатальной и неигнорируемой, так что очень хорошо, что мы заранее проверили выражение `x * n` и знаем, что оно не порождает ошибку.

А что будет, если подстановка окажется неудачной? Тогда нам нужен другой перегруженный вариант, иначе вся программа будет некорректной. Попробуем далее функцию `operator*=`:

```
template <typename T>
auto increase(const T& x, size_t n) -> decltype(T(x) *= n)
{
    T y(x);
    return y *= n;
}
```

Отметим, что мы не можем просто проверить, допустимо ли выражение `x *= n`. Во-первых, `x` передается по константной ссылке, а `operator*=` всегда модифицирует свою левую часть. Во-вторых, недостаточно, чтобы в типе `T` был определен `operator*=`. В теле нашей функции необходимо создать временный объект такого же типа, чтобы было, от имени чего вызывать `*=`. Поскольку мы решили создавать такой объект копирующим конструктором, он тоже должен присутствовать. В общем, необходимо, чтобы все выражение `T(x) *= n` было допустимым.

Имея эти два перегруженных варианта, мы охватили оба сценария умножения – лишь бы не одновременно! Снова мы столкнулись с проблемой чрезмерного богатства выбора – если оба перегруженных варианта допустимы, то мы не сможем выбрать какой-то один, и программа не откомпилируется. Но мы уже с этим боролись и победили – нужно либо сложное логическое выражение, либо вспомогательный шаблон класса, специализируемый несколькими булевыми значениями:

```
struct yes { char c; };
struct no { char c; yes c1; };

template <typename T>
auto have_star_equal(const T& x, size_t n) -> decltype(T(x) *= n, yes());
no have_star_equal(...);

template <typename T>
auto have_star(const T& x, size_t n) -> decltype(x * n, yes());
no have_star(...);

template <typename T, bool have_star_equal, bool have_star> struct
increase_helper;

template <typename T> struct increase_helper<T, true, true> {
    static auto f(const T& x, size_t n) {
        std::cout << "T *= n, игнорируется T * n" << std::endl;
        T y(x);
        return y *= n;
    }
};

template <typename T> struct increase_helper<T, true, false> {
    static auto f(const T& x, size_t n) {
        std::cout << "T *= n" << std::endl;
        T y(x);
        return y *= n;
    }
};
```

```
template <typename T> struct increase_helper<T, false, true> {
    static auto f(const T& x, size_t n) {
        std::cout << "T * n" << std::endl;
        return x * n;
    }
};

template <typename T> auto increase(const T& x, size_t n) {
    return increase_helper<T,
        sizeof(have_star_equal(x, n)) == sizeof(yes),
        sizeof(have_star(x, n)) == sizeof(yes)>::f(x, n);
}
```

Заметим, что способность использовать автоматический возвращаемый тип, не употребляя явный `decltype` после списка параметров, – черта, появившаяся в C++14; в C++11 нужно было бы написать так:

```
static auto f(const T& x, size_t n) -> decltype(x * n) { ... }
```

Это решение кажется похожим на приведенное в предыдущем разделе, и действительно мы пользуемся теми же самыми инструментами (почему вы и должны были узнать о них здесь). Ключевое различие кроется в проверочных функциях `have_star` и `have_star_equal` – вместо того чтобы проверять существование некоторого типа, например `void T::sort()`, мы проверяем допустимость выражения `x * n`. Заметим также, что хотя мы и хотим, чтобы выражение было допустимым, нам не нужно фактически возвращать его тип, у нас уже есть предлагаемый тип возвращаемого значения, `struct yes`. Именно поэтому в `decltype` мы видим несколько выражений, разделенных запятыми:

```
auto have_star_equal(const T& x, size_t n) -> decltype(T(x) *= n, yes());
```

Тип выводится из последнего выражения, но все предыдущие также должны быть допустимы, иначе подстановка закончится неудачно (к счастью, в контексте SFINAE).

Пока что мы рассмотрели только два способа увеличить `x`. Мы можем добавить и другие способы, например многократное сложение, если ничего другого не остается. Нужно только добавлять новые булевы параметры во вспомогательный шаблон. Кроме того, нам нужен более удачный способ справляться с растущим количеством комбинаций. Например, если самое простое выражение `x * n` работает, то до того, работает ли `x + x`, нам нет никакого дела. Иерархию логических решений можно упростить, воспользовавшись частичными специализациями шаблона:

```
// Общий случай, неполный тип
template <typename T, bool have_star, bool have_star_equal>
struct increase_helper;
// Использовать эту специализацию, если have_star равна true
template <typename T, bool have_star_equal>
struct increase_helper<T, true, have_star_equal> {
    static auto f(const T& x, size_t n) {
```

```
        std::cout << "T * n" << std::endl;
        return x * n;
    }
};
```

И наконец, хватит возиться с доморощенными типами yes и no – в C++11 есть два специальных типа: std::true_type и std::false_type. Не предполагается сравнивать их размер (на самом деле размер у них одинаковый), да это и ни к чему – в них имеются булевы члены данных value с квалификатором constexpr (константа времени выполнения), принимающие значения true и false соответственно. Собирая все вместе, приходим к окончательному решению:

```
template <typename T>
auto have_star(const T& x, size_t n) -> decltype(x * n, std::true_type());
std::false_type have_star(...);

template <typename T>
auto have_star_equal(const T& x, size_t n) -> decltype(T(x) *= n,
std::true_type());
std::false_type have_star_equal(...);

template <typename T>
auto have_plus(const T& x) -> decltype(x + x, std::true_type());
std::false_type have_plus(...);

template <typename T, bool have_star, bool have_star_equal, bool have_plus>
struct increase_helper;

template <typename T, bool have_star_equal, bool have_plus>
struct increase_helper<T, true, have_star, have_plus> {  // x * n работает,
    static auto f(const T& x, size_t n) {                // остальное нас не волнует
        std::cout << "T * n" << std::endl;
        return x * n;
    }
};

template <typename T, bool have_plus>
struct increase_helper<T, false, true, have_plus> {  // x * n не работает,
    static auto f(const T& x, size_t n) {            // но работает x *= n,
        std::cout << "T *= n" << std::endl;          // + нас не волнует
        T y(x);
        return y *= n;
    }
};
template <typename T>
struct increase_helper<T, false, false, true> {  // ничего не работает, кроме x + x
    static auto f(const T& x, size_t n) {
        std::cout << "T + T + ... + T" << std::endl;
        T y(x);
        for (size_t i = 1; i < n; ++i) y = y + y;
        return y;
    }
};
```

```
template <typename T> auto increase(const T& x, size_t n) {
    return increase_helper<T,
                           decltype(have_star_equal(x, n))::value,
                           decltype(have_star(x, n))::value,
                           decltype(have_plus(x))::value
                          >::f(x, n);
}
```

Этот код работает для любой комбинации операторов, реализующих операции *, *=, + и принимающих аргументы нужных нам типов, быть может, требующих преобразования. При этом бросается в глаза его сходство с кодом, изученным в предыдущем разделе, хотя мы решали совершенно другую задачу. Сталкиваясь с трафаретным кодом, который повторяется снова и снова, мы просто обязаны поискать более универсальное решение, допускающее повторное использование.

SFINAE без компромиссов

Наша цель – разработать повторно используемый каркас, который был бы пригоден для проверки любого выражения на допустимость и не приводил бы к синтаксической ошибке, если выражение недопустимо. Как и в предыдущем разделе, мы не хотим точно указывать способ вычисления выражения – с помощью конкретной функции, с использованием преобразования или еще как-то. Мы просто хотим предварительно откомпилировать выражение, которое собираемся использовать в теле функции, чтобы убедиться, что настоящая компиляция не приведет к ошибке, а если приведет, то конкретизировать какую-то другую шаблонную функцию. Нам нужен общий механизм `is_valid`, применимый к любому выражению.

Для достижения этой цели нам придется задействовать все изученные до сих пор приемы программирования шаблонов, включая SFINAE, `decltype` и выведение возвращаемого типа. Нам также понадобится одно из дополнительных средств, введенных в C++14, – полиморфные лямбда-выражения, которые станут удобным способом сохранить произвольное выражение. Если вы незнакомы с чем-то из вышеперечисленного, самое время обратиться к разделу «Для дальнейшего чтения» в конце этой главы.

Сейчас мы окончательно отринем всякие доморощенные навороты вроде типов `yes/no` и в полной мере воспользуемся тем, что дает нам стандарт C++14. Эти средства в сочетании с лаконичным языком лямбда-выражений позволяют записать решение удивительно компактно:

```
template <typename Lambda> struct is_valid_helper {
    template <typename LambdaArgs>
    constexpr auto test(int) ->
        decltype(std::declval<Lambda>()(std::declval<LambdaArgs>()),
                                        std::true_type())
    {
        return std::true_type();
    }
```

```
    template <typename LambdaArgs> constexpr std::false_type test(...) {
        return std::false_type();
    }

    template <typename LambdaArgs>
      constexpr auto operator()(const LambdaArgs&) {
          return this->test<LambdaArgs>(0);
      }
};

template <typename Lambda> constexpr auto is_valid(const Lambda&) {
    return is_valid_helper<Lambda>();
}
```

Прежде чем объяснять, как это работает, полезно посмотреть, как используется шаблон `is_valid`. Он служит для объявления *объектов проверки* произвольных выражений, как в следующем примере:

```
auto is_assignable = is_valid([](auto&& x) -> decltype(x = x) {});
void my_function( const A& a ) {
    static_assert(decltype(is_assignable(a))::value,
                  "A is not assignable");
}
```

Здесь мы проверяем выражение `x = x`, т. е. присваивание объекта другому объекту того же типа (тот факт, что в выражении объект присваивается сам себе, не играет решительно никакой роли – нам нужно лишь проверить, существует ли оператор присваивания, принимающий два аргумента того же типа, что `x`). Объект `is_assignable`, который мы определили с помощью `is_valid`, имеет тип `std::integral_constant`, поэтому в нем имеется член `value` с квалификатором `constexpr`, который принимает значение `true` или `false` в зависимости от того, допустимо выражение или нет. Эту константу времени выполнения можно использовать в утверждении времени выполнения, или для специализации шаблона, или для активации функции с помощью `std::enable_if`, или еще каким-то способом, в котором используются константы времени выполнения.

Подробное объяснение принципа работы `is_valid` потребовало бы погружения в стандарт C++14 и выходит за рамки этой книги. Однако мы можем остановиться на некоторых деталях, связывающих этот код с предложенными ранее решениями. Начнем с двух перегруженных вариантов `test()`, которые теперь скрыты внутри вспомогательной структуры, а не вынесены наружу в виде свободных функций. Это только улучшает инкапсуляцию, а никаких других выгод не несет. Вместо аргументов, специфичных для решаемой задачи, эти перегруженные функции принимают фиктивный целый аргумент, ноль, который не используется, но необходим для задания порядка перебора вариантов (вариант с переменным числом аргументов рассматривается в крайнем случае). SFINAE-проверка, которая может завершиться неудачно во время подстановки, – это, как и раньше, первое выражение внутри оператора `decltype` варианта `yes` (или `true`) функции `test()`. Теперь это выражение имеет вид `std::declval<Lambda>()(std::declval<LambdaArgs>())`, т. е. в нем используется

declval и производится попытка сконструировать ссылку на указанные типы Lambda и LambdaArgs (без использования конструктора по умолчанию, который может и отсутствовать). Lambda – это тип, взятый из параметра вспомогательной структуры. Тип LambdaArgs выводится шаблоном-членом operator() из его аргумента. А где же обращение к этому оператору? Запомните этот вопрос, мы скоро к нему вернемся. Шаблонная функция is_valid() конструирует по умолчанию и возвращает вызываемый объект (объект, содержащий функцию-член operator()); этот объект имеет тип is_valid_helper<Lambda>, где Lambda – тип лямбда-выражения в этом конкретном вызове is_valid(), в нашем случае – [](auto&& x) -> decltype(x = x) {}. Этот тип лямбда-выражения запоминается и используется для конкретизации вспомогательного шаблона, и это тот самый тип Lambda, который появляется в первом перегруженном варианте функции test(). Наконец, объект вызывается, и ему передается переменная; мы хотим проверить, допускает ли он присваивание. В нашем случае это аргумент a функции my_function() – именно здесь наконец вызывается функция operator(), которая выводит тип a и передает его перегруженному варианту test() для конструирования ссылки на этот тип, чтобы его можно было проверить для выражения внутри указанной лямбды; в нашем случае это выражение x + x, где вместо x подставлен a. Из-за обилия скобок этот последний фрагмент кода будет проще понять, если временно изменить имя функции-члена operator() на f() и посмотреть, где эта f() вылезает:

```
template <typename Lambda> struct is_valid_helper {
    ...
    // Вызываемый объект
    template <typename LambdaArgs> constexpr auto f(const LambdaArgs&) {
        return this->test<LambdaArgs>(0);
    }
};
    ...
// И сам вызов
static_assert(decltype(is_assignable.f(a))::value, "A is not assignable");
```

Хотя ничего плохого в именах вида f() нет, особого смысла в них тоже нет, а вызываемые объекты – очень распространенная идиома в C++.

Теперь, обзаведясь молотком на все случаи жизни, мы можем рассматривать различные задачи условной компиляции как гвозди. Например, почему бы не проверить, можно ли складывать объекты:

```
auto is_addable = is_valid([](auto&& x) -> decltype(x + x) {});
```

Или проверить, похож ли объект на указатель:

```
auto is_pointer = is_valid([](auto&& x) -> decltype(*x) {});
```

Последний пример очень поучителен. В нем проверяется, допустимо ли выражение *p. Объект p мог бы быть простым указателем, но также и любым интеллектуальным указателем, например std::shared_ptr.

Можно проверить, имеет ли объект доступный нам конструктор по умолчанию, хотя это не так просто – нельзя объявить в выражении переменную такого же типа, как x. Мы можем, например, проверить, допустимо ли выражение new X, где X – тип x, который можно получить с помощью decltype, но только надо иметь в виду, что decltype скрупулезно сохраняет все детали типа, включая и оператор ссылки (тип x, запомненный лямбда-выражением, включает оператор &&). Оператор ссылки следует изъять, как показано ниже:

```
auto is_default_constructible =
    is_valid([](auto&& x) ->
        decltype(new typename std::remove_reference<decltype(x)>::type)
        {});

auto is_destructible =
    is_valid([](auto&& x) -> d
        decltype(delete (&x)) {});
```

Проверка наличия деструктора проще – можно просто вызвать оператор delete, передав адрес объекта (отметим, что выполнять такой код в реальной программе категорически не рекомендуется, потому что он, скорее всего, освободит память, которую мы никогда не получали от выражения new, но здесь ничего не выполняется – все происходит на этапе компиляции).

Осталось преодолеть еще одно ограничение – до сих пор у наших лямбда-выражений был только один аргумент. Конечно, можно определить лямбда-выражение с несколькими аргументами:

```
auto is_addable2 = is_valid([](auto&& x, auto&&y) -> decltype(x + y) {});
```

Но этого недостаточно, потому что в нашей вспомогательной структуре мы собираемся вызывать это лямбда-выражение с одним аргументом – там есть место только для одного, LambdaArgs. И если вы еще не наелись C++14 досыта, так мы сейчас пальнем по задаче из последнего оставшегося тяжелого оружия – готовьте шаблоны с переменным числом аргументов!

```
template <typename Lambda> struct is_valid_helper {
    template <typename... LambdaArgs>
    constexpr auto test(int) ->
        decltype(std::declval<Lambda>()(std::declval<LambdaArgs>()...),
                 std::true_type())
    {
        return std::true_type();
    }

    template <typename... LambdaArgs>
    constexpr std::false_type test(...) {
        return std::false_type();
    }

    template <typename... LambdaArgs>
    constexpr auto operator()(const LambdaArgs& ...) {
        return this->test<LambdaArgs...>(0);
    }
};
```

```
template <typename Lambda> constexpr auto is_valid(const Lambda&) {
    return is_valid_helper<Lambda>();
}
```

Теперь можно объявлять объекты, проверяющие возможности, с любым числом аргументов, как, например, `is_addable2` выше.

В C++17 нет нужды явно задавать тип параметра вспомогательного шаблона внутри `is_valid`:

```
template <typename Lambda> constexpr auto is_valid(const Lambda&) {
    return is_valid_helper();    // C++17: выведение типа конструктора
}
```

Те из нас, кто до сих пор томится в темных веках, предшествующих C++11, должны сейчас чувствовать себя ущемленными. Ни лямбда-выражений, ни шаблонов с переменным числом аргументов, ни `decltype` – да есть ли хоть что-нибудь, что позволило бы использовать SFINAE? Как выясняется, есть, хотя компенсировать отсутствие шаблонов с переменным числом аргументов трудновато, и это заставляет нас явно объявлять `is_valid` для одного, двух и т. д. аргументов. Кроме того, не имея удобного контекста для применения SFINAE к типу возвращаемого значения и располагая гораздо более коротким перечнем контекстов SFINAE в C++03, нам придется подыскивать место, где можно проверить наши выражения, не опасаясь вызвать ошибку компиляции. Один такой контекст есть – в аргументах по умолчанию шаблонных функций, но нам нужно что-то такое, что вычисляется на этапе компиляции. Сначала покажем, как построить SFINAE-проверку, не используя никаких средств C++11:

```
struct valid_check_passes { char c; };
struct valid_check_fails { valid_check_passes c; char c1; };

struct addable2 {
    // Неиспользуемый последний аргумент, активирует SFINAE
    template <typename T1, typename T2>
    valid_check_passes operator()(
        const T1& x1,
        const T2& x2,
        char (*a)[sizeof(x1 + x2)] = NULL);
    // Перегруженный вариант с переменным числом аргументов
    template <typename T1, typename T2>
    valid_check_fails operator()(const T1& x1, const T2& x2, ...);
};

sizeof(addable2()(i, x, 0)) == // константа времени компиляции
    sizeof(valid_check_passes)
```

Константное выражение времени компиляции `sizeof(addable2()(i, x, 0)) == sizeof(valid_check_passes)` можно использовать в любом контексте, где требуется константа времени компиляции, например в целочисленном параметре шаблона для условной компиляции с помощью специализаций шаблонов.

Чтобы обернуть это средство в форму, более-менее допускающую повторное использование, мы вынуждены использовать макросы:

```
namespace IsValid {
struct check_passes { char c; };
struct check_fails { check_passes c; char c1; };
} // namespace IsValid

#define DEFINE_TEST2(NAME, EXPR) \
struct NAME { \
    template <typename T1, typename T2> \
    IsValid::check_passes operator()( \
        const T1& x1, \
        const T2& x2, \
        char (*a)[sizeof(EXPR)] = NULL); \
    template <typename T1, typename T2> \
    IsValid::check_fails operator()(const T1& x1, const T2& x2, ...); \
}

#define IS_VALID2(TEST, X1, X2) \
    sizeof(TEST()(X1, X2, 0)) == sizeof(IsValid::check_passes)
```

Заметим, что число аргументов (2) и их имена (x1 и x2) зашиты в код макросов. Теперь эти макросы можно использовать, чтобы определить проверку возможности сложения во время компиляции:

```
DEFINE_TEST2(addable2, x1 + x2);
int i, j;
IS_VALID2(addable2, i, j)   // результат вычисления во время компиляции равен true
IS_VALID2(addable2, &i, &j) // результат вычисления во время компиляции равен false
```

Нам придется определить такие макросы для любого числа аргументов, которое мы хотим поддержать. До выхода C++11 это было лучшее, что можно сделать.

Резюме

SFINAE – несколько экзотическое средство стандарта C++, в нем много сложных и тонких деталей. Хотя обычно оно упоминается в контексте *ручного управления разрешением перегрузки*, его основная цель не в том, чтобы гуру могли писать особо изощренный код, а в том, чтобы регулярное (автоматическое) разрешение перегрузки функционировало так, как нужно программисту. В этой роли SFINAE, как правило, работает точно так, как ожидается, без дополнительных усилий – на самом деле программисту в большинстве случаев даже не нужно знать о существовании этого средства. Чаще всего, когда вы пишете общий перегруженный шаблон и специальный вариант для указателей, вы ожидаете, что последний не будет вызываться для типов, не являющихся указателями. И вы ни на секунду не задумываетесь о том, что отвергнутый перегруженный вариант некорректно сформирован, – кому какое дело, ведь его же и не предполагается использовать. Но чтобы понять, что его не предполагается использовать, необходимо подставить тип, а это привело бы к появлению недопустимого кода. SFINAE разрывает этот порочный круг «яйцо или курица» – чтобы выяснить, что перегруженный вариант следует отвергнуть, мы должны

подставить типы, но это породит некомпилируемый код, что само по себе не оставило бы проблемы, поскольку этот вариант все равно был бы отвергнут, однако мы не узнаем этого, пока не подставим типы. И так по кругу...

Конечно, мы не стали бы писать несколько десятков страниц только для того, чтобы сказать, что компилятор волшебным образом делает то, что нужно, и вам не о чем беспокоиться. Более хитроумное применение SFINAE заключается в том, чтобы искусственно вызвать неудавшуюся подстановку и таким образом получить контроль над разрешением перегрузки, исключив некоторые перегруженные варианты. В этой главе мы узнали о *безопасных* контекстах для таких *временных* ошибок, которые в конечном итоге подавляются SFINAE. При разумном и аккуратном использовании эта техника позволяет анализировать и различать во время компиляции все, начиная от простых свойств различных типов (*является ли это классом?*) до сложных видов поведения, которые могут быть реализованы разнообразными языковыми средствами C++ (*можно ли как-то сложить эти два типа?*).

В следующей главе мы рассмотрим еще один продвинутый паттерн на основе шаблонов, который позволяет значительно расширить возможности иерархий классов в C++: наследование классов позволяет передавать информацию от базового класса к производному, а паттерн Рекурсивный шаблон делает прямо противоположное – уведомляет базовый класс о производном.

Вопросы

- Что такое множество перегруженных вариантов?
- Что такое разрешение перегрузки?
- Что такое выведение типов и подстановка типов?
- Что такое SFINAE?
- В каких контекстах потенциально недопустимый код не приводит к ошибке компиляции, если только он не понадобится в действительности?
- Как можно определить, какой перегруженный вариант был выбран, не вызывая его?
- Как SFINAE применяется для управления условной компиляцией?

Для дальнейшего чтения

- https://www.packtpub.com/application-development/c17-example.
- https://www.packtpub.com/application-development/getting-started-c17-programming-video.
- https://www.packtpub.com/application-development/mastering-c17-stl.
- https://www.packtpub.com/application-development/c17-stl-cookbook.

Глава 8

Рекурсивный шаблон

Мы уже знакомы с понятиями наследования, полиморфизма и виртуальной функции. Производный класс наследует базовому и модифицирует его поведение, переопределяя виртуальные функции. Все операции производятся от имени экземпляра базового класса – полиморфно. Если объект базового класса на самом деле является экземпляром производного, то вызываются переопределенные в нем виртуальные функции. Базовый класс ничего не знает о производном. Быть может, производного класса даже не существовало, когда базовый был написан и откомпилирован. Паттерн **Рекурсивный шаблон** (Curiously Recurring Template Pattern – **CRTP**) переворачивает эту благостную картину с ног на голову и выворачивает наизнанку.

В этой главе рассматриваются следующие вопросы:

- что такое CRTP;
- что такое статический полиморфизм и чем он отличается от динамического;
- каковы недостатки вызовов виртуальных функций и почему было бы предпочтительнее разрешать эти вызовы во время компиляции;
- какие еще применения есть у CRTP.

Технические требования

Библиотека Google Benchmark: `https://github.com/google/benchmark`.

Примеры кода: `https://github.com/PacktPublishing/Hands-On-Design-Patterns-with-CPP/tree/master/Chapter08`.

Укладываем CRTP в голове

Паттерн CRTP впервые был описан под таким именем Джеймсом Коплиеном (James Coplien) в 1995 году в статье, опубликованной в журнале «C++ Report». Это частный случай более общего ограниченного полиморфизма (см. *Peter S. Canning et al.* F-bounded polymorphism for object-oriented programming // Conference on Functional Programming Languages and Computer Architecture, 1989). Не являясь полноценной заменой виртуальным функциям, он все же предлагает

программисту на C++ похожий инструмент, который при определенных обстоятельствах обладает несколькими преимуществами.

Что не так с виртуальной функцией?

Прежде чем говорить о *лучшей* альтернативе виртуальной функции, надо разобраться, зачем мы вообще ищем ей замену. Что может не нравиться в виртуальных функциях?

Проблема в накладных расходах и, стало быть, в снижении производительности. Вызов виртуальной функции может обходиться в несколько раз дороже невиртуального вызова, особенно для простых функций, которые, не будь они виртуальными, компилятор мог бы встроить (напомним, что виртуальная функция никогда не встраивается). Измерить разницу позволяет библиотека эталонного микротестирования, идеальное средство для измерения производительности небольших фрагментов кода. Средств такого рода существует много, в этой книге мы будем пользоваться библиотекой Google Benchmark. Чтобы следить за примерами, вам понадобится сначала скачать и установить библиотеку (подробные инструкции приведены в главе 5). Затем можно будет откомпилировать и выполнить примеры.

Установив библиотеку эталонного микротестирования, мы можем измерить накладные расходы на вызов виртуальной функции. Будем сравнивать очень простую виртуальную функцию, содержащую минимум кода, с невиртуальной функцией, делающей то же самое. Вот код нашей виртуальной функции:

```
class B {
    public:
    B() : i_(0) {}
    virtual ~B() {}
    virtual void f(int i) = 0;
    int get() const { return i_; }
    protected:
    int i_;
};
class D : public B {
    public:
    void f(int i) { i_ += i; }
};
```

А вот ее невиртуальный эквивалент:

```
class A {
    public:
    A() : i_(0) {}
    void f(int i) { i_ += i; }
    int get() const { return i_; }
    protected:
    int i_;
};
```

Теперь вызовем обе из фикстуры микротеста и измерим время работы:

```
void BM_none(benchmark::State& state) {
    A* a = new A;
    int i = 0;
    for (auto _ : state) {
        a->f(++i);
    }
    benchmark::DoNotOptimize(a->get());
    delete a;
}
void BM_dynamic(benchmark::State& state) {
    B* b = new D;
    int i = 0;
    for (auto _ : state) {
        b->f(++i);
    }
    benchmark::DoNotOptimize(b->get());
    delete b;
}
```

Функция `benchmark::DoNotOptimize` не дает компилятору удалить из кода неиспользуемый объект и, как следствие, сделать весь набор вызовов функций ненужным. Обратите внимание на одну тонкость в измерении времени работы виртуальной функции; проще было бы написать код, в котором операторы `new` и `delete` не используются, а объект производного класса конструируется в стеке:

```
void BM_dynamic(benchmark::State& state) {
    D d;
    int i = 0;
    for (auto _ : state) {
        d.f(++i);
    }
    benchmark::DoNotOptimize(b->get());
}
```

Однако этот тест, скорее всего, даст такое же время, как для невиртуальной функции. Действительно, при таком вызове нет никаких накладных расходов, поскольку компилятор может вывести, что при вызове виртуальной функции `f()` всегда вызывается `D::f()` (ведь вызов производится не через указатель на базовый класс, а по ссылке на производный, поэтому чем же еще он может быть?). Хороший оптимизирующий компилятор «девиртуализирует» такой вызов, например сгенерирует прямое обращение к `D::f()` безо всякой косвенности и поиска в таблице виртуальных функций (*v-таблице*). Такой вызов можно даже встроить.

Еще одно возможное осложнение заключается в том, что оба микротеста, особенно в случае невиртуального вызова, могут оказаться слишком быстрыми – тело цикла вполне может занимать меньше времени, чем накладные расходы на организацию цикла. Это можно исправить, выполнив несколько вы-

зовов внутри тела цикла. Для этого воспользуемся копированием и вставкой или макросами препроцессора C++:

```
#define REPEAT2(x) x x
#define REPEAT4(x) REPEAT2(x) REPEAT2(x)
#define REPEAT8(x) REPEAT4(x) REPEAT4(x)
#define REPEAT16(x) REPEAT8(x) REPEAT8(x)
#define REPEAT32(x) REPEAT16(x) REPEAT16(x)
#define REPEAT(x) REPEAT32(x)
```

Теперь внутри цикла микротеста можно написать:

```
REPEAT(b->f(++i);)
```

Время одной итерации, сообщаемое тестом, сейчас относится к 32 вызовам функции. Хотя для сравнения двух вызовов это несущественно, иногда удобно, чтобы сам тест сообщал истинное количество вызовов в секунду. Для этого нужно добавить в конец фикстуры, после цикла, следующую строку:

```
state.SetItemsProcessed(32*state.iterations());
```

Вот теперь можно сравнить результаты обоих эталонных тестов:

```
-----------------------------------------------------
Benchmark           Time           CPU Iterations
-----------------------------------------------------
BM_none             6 ns          6 ns  112876558    5.2072G items/s
BM_dynamic         52 ns         52 ns   13330215   592.541M items/s
```

Мы видим, что вызов виртуальной функции почти в 10 раз дороже вызова невиртуальной. Заметим, что сравнение не совсем честное; виртуальный вызов обладает дополнительной функциональностью. Однако часть этой функциональности можно реализовать другими способами, без накладных расходов, снижающих производительность.

Введение в CRTP

Пора познакомиться с паттерном CRTP, который переворачивает все наши представления о наследовании:

```
template <typename D> class B {
    ...
};
class D : public B<D> {
    ...
};
```

Первое изменение заключается в том, что базовый класс теперь является шаблоном класса. Производный класс по-прежнему наследует базовому, но только его вполне определенной конкретизации – самому себе! Класс `B` конкретизирован классом `D`, а класс `D` наследует классу `B`, конкретизированному классом `D`, который наследует классу `B`, который... Вот вам и рекурсия в действии. Привыкайте, в этой главе она будет встречаться часто.

В чем же смысл этого умопомрачительного паттерна? Подумайте о том, что теперь базовый класс во время компиляции располагает информацией о производном. Следовательно, то, что раньше было виртуальной функцией, теперь можно связать с нужной функцией на этапе компиляции:

```
template <typename D> class B {
    public:
    B() : i_(0) {}
    void f(int i) { static_cast<D*>(this)->f(i); }
    int get() const { return i_; }
    protected:
    int i_;
};
class D : public B<D> {
    public:
    void f(int i) { i_ += i; }
};
```

Сам вызов по-прежнему можно производить через указатель на базовый класс:

```
B<D>* b = ...;
b->f(5);
```

Но здесь нет ни косвенности, ни накладных расходов на виртуальный вызов. Компилятор может проследить весь путь к фактически вызываемой функции и даже встроить ее.

```
void BM_static(benchmark::State& state) {
    B<D>* b = new D;
    int i = 0;
    for (auto _ : state) {
        REPEAT(b->f(++i);)
    }
    benchmark::DoNotOptimize(b->get());
    state.SetItemsProcessed(32*state.iterations());
}
```

Этот тест показывает, что вызов функции посредством CRTP занимает ровно столько времени, сколько вызов обычной функции:

```
-------------------------------------------------------
Benchmark             Time             CPU Iterations
-------------------------------------------------------
BM_none               6 ns           6 ns  114162238   5.26757G items/s
BM_dynamic           52 ns          52 ns   13353176   587.244M items/s
BM_static             6 ns           6 ns  119104189   5.16101G items/s
```

Основное ограничение CRTP заключается в том, что размер базового класса `B` не может зависеть от его параметра шаблона `D`. Вообще, шаблон класса `B` конкретизируется неполным типом `D`. Например, следующий код не компилируется:

```
template <typename D> class B {
    typedef typename D::T T;
```

```
    T* p_;
};
class D : public B<D> {
    typedef int T;
};
```

Осознание того, что этот код не компилируется, может вызвать шок, если учесть, насколько он похож на широко распространенные шаблоны, ссылающиеся на вложенные типы своих параметров. Рассмотрим, к примеру, следующий шаблон, который преобразует любой контейнер-последовательность, обладающий функциями `push_back()` и `pop_back()`, в стек:

```
template <typename C> class stack {
    C c_;
    public:
    typedef typename C::value_type value_type;
    void push(const valuetype& v) { c.push_back(v); }
    value_type pop() { value_type v = c.back(); c.pop_back(); return v; }
};
stack<std::vector<int>> s;
```

Заметим, что `typedef`, определяющий `value_type`, выглядит точно так же, как в предыдущем примере, где мы пытались объявить класс `B`. Так что же не так с классом `B`? С ним самим ничего. Он отлично откомпилировался бы в контексте, похожем на наш класс `stack`:

```
class A {
    public:
    typedef int T;
    T x_;
    };
B<A> b; // компилируется без проблем
```

Проблема не в классе `B`, а в том, как мы собираемся его использовать:

```
class D : public B<D> ...
```

В точке, где тип `B<D>` должен быть известен, тип `D` еще не объявлен. Так не может быть – для объявления класса `D` необходимо точно знать, что представляет собой базовый класс `B<D>`. Но если класс `D` еще не объявлен, то как компилятор знает, что идентификатор `D` вообще относится к классу? Ведь не можем же мы конкретизировать шаблон совершенно неизвестным типом. Ответ лежит где-то посередине – для класса `D` имеется опережающее объявление, как если бы в программе присутствовал такой код:

```
class A;
B<A> b; // теперь не компилируется
```

Некоторые шаблоны можно конкретизировать типами с опережающим объявлением, другие – нельзя. Точные правила можно, изрядно помучившись, извлечь из стандарта, но суть такова: все, что может повлиять на размер класса, должно быть объявлено полностью. Ссылка на тип, объявленный внутри не-

полного типа, например `typedef typename D::T T`, рассматривалась бы как опережающее объявление вложенного класса, а это тоже запрещено.

С другой стороны, тело функции-члена шаблона класса не конкретизируется до момента вызова. На самом деле при заданном параметре шаблона функция-член даже не компилируется, если она нигде не вызывается. Поэтому ссылки на производный класс, на его вложенные типы и на функции-члены внутри функций-членов базового класса вполне законны. Кроме того, поскольку тип производного класса рассматривается как объявленный опережающе внутри базового класса, мы можем объявлять ссылки и указатели на него. Вот очень распространенная переделка базового CRTP-класса, в которой применения статического приведения собраны в одном месте:

```
template <typename D> class B {
    ...
    void f(int i) { derived()->f(i); }
    D* derived() { return static_cast<D*>(this); }
    };
class D : public B<D> {
    ...
    void f(int i) { i_ += i; }
};
```

Объявление базового класса владеет указателем на неполный (опережающе объявленный) тип `D`. Работает это, как любой другой указатель на неполный тип; к моменту разыменования указателя тип должен стать полным. В нашем случае это происходит в теле функции-члена; как мы только что говорили, `B::f()` не компилируется, пока не будет вызвана из клиентского кода.

CRTP и статический полиморфизм

Поскольку CRTP позволяет замещать функции базового класса функциями производного, он реализует полиморфное поведение. Ключевое отличие в том, что происходит это на этапе компиляции, а не выполнения.

Полиморфизм времени компиляции

Как мы только что видели, CRTP позволяет производному классу модифицировать поведение базового:

```
template <typename D> class B {
    public:
    ...
    void f(int i) { static_cast<D*>(this)->f(i); }
    protected:
    int i_;
};
class D : public B<D> {
    public:
```

```
    void f(int i) { i_ += i; }
};
```

Когда вызывается метод базового класса `B::f()`, вызов переадресуется методу реального производного класса, точно так же, как в случае виртуальной функции. Конечно, чтобы в полной мере насладиться преимуществами этого полиморфизма, мы должны иметь возможность вызывать методы базового класса через указатель на базовый класс. Иначе мы просто вызываем методы производного класса, тип которого уже знаем:

```
D* d = ...;    // получить объект типа D
d->f(5);
B<D>* b = ...; // также должен быть объектом типа D
b->f(5);
```

Заметим, что вызов функции выглядит точно так же, как вызов любой виртуальной функции через указатель на базовый класс. В действительности вызывается функция `f()` из производного класса, `D::f()`. Однако имеется существенное отличие: фактический тип производного класса, `D`, должен быть известен во время компиляции – указатель на базовый класс имеет не тип `B*`, а `B<D>*`, откуда следует, что производный объект имеет тип `D`. На первый взгляд, в таком *полиморфизме*, когда программисту должен быть известен фактический тип, особого смысла нет. Но это потому, что мы не до конца осознали, что на самом деле означает *полиморфизм времени компиляции*. Достоинством виртуальной функции является то, что мы можем вызывать функции-члены типа, о существовании которого даже не знаем, и то же самое должно быть справедливо для *статического полиморфизма*, иначе он бесполезен.

А как написать функцию, которая должна компилироваться с параметрами неизвестного типа? С помощью шаблона функции, конечно:

```
template <typename D> void apply(B<D>* b, int& i) {
    b->f(++i);
}
```

Эта шаблонная функция может вызываться для любого указателя на базовый класс и автоматически выводит тип производного класса `D`. Теперь мы можем написать нечто, выглядящее как обычный полиморфный код:

```
B<D>* b = new D;   // 1
apply(b);          // 2
```

Заметим, что в первой строке объект необходимо конструировать, зная фактический тип. Но так бывает всегда, то же самое справедливо и для обычного полиморфизма времени выполнения на основе виртуальных функций:

```
void apply(B* b) { ... }
B* b = new D;           // 1
apply(b);               // 2
```

В обоих случаях во второй строке мы вызываем код, в котором требуется только знание базового класса.

Чисто виртуальная функция времени компиляции

А что было бы эквивалентом чисто виртуальной функции в этом сценарии? Чисто виртуальная функция должна быть реализована во всех производных классах. Класс, в котором объявлена или унаследована, но не переопределена чисто виртуальная функция, является абстрактным – ему можно наследовать, но нельзя создать его экземпляр.

Размышляя об эквиваленте чисто виртуальной функции для статического полиморфизма, мы приходим к выводу, что наша реализация CRTP страдает серьезной уязвимостью. Что, если мы забудем переопределить *виртуальную функцию времени компиляции* `f()` в одном из производных классов?

```
template <typename D> class B {
    public:
    ...
    void f(int i) { static_cast<D*>(this)->f(i); }
};
class D : public B<D> {
    // здесь нет f()!
};
...
B<D>* b = ...;
b->f(5);                // 1
```

Этот код компилируется без ошибок и предупреждений – в строке 1 мы вызываем функцию `B::f()`, которая, в свою очередь, вызывает `D::f()`. В классе `D` не объявлена собственная версия члена `f()`, поэтому вызывается унаследованная от базового класса. Конечно, это та самая функция-член `B::f()`, которую мы уже видели. Она снова вызывает `D::f()`, которая не что иное, как `B::f()` ..., и мы имеем бесконечный цикл.

Проблема здесь в том, что нас никто не заставляет переопределять функцию-член `f()` в производном классе, но если этого не сделать, то получается некорректная программа. Корень зла – в смешении интерфейса и реализации; объявление открытой функции-члена в базовом классе говорит, что во всех производных классах должна быть функция `void f(int)`, это часть их открытого интерфейса. Вариант этой функции в производном классе и дает фактическую реализацию. Мы будем рассматривать вопрос о разделении интерфейса и реализации в главе 14, а пока скажем лишь, что жизнь наша стала бы куда проще, если бы эти функции имели разные имена:

```
template <typename D> class B {
    public:
    ...
    void f(int i) { static_cast<D*>(this)->f_impl(i); }
};
class D : public B<D> {
    void f_impl(int i) { i_ += i; }
};
```

```
...
B<D>* b = ...;
b->f(5);
```

Что случится, если мы забудем реализовать функцию `D::f_impl()`? Код не откомпилируется, потому что в классе `D` нет такой функции-члена – ни своей, ни унаследованной. Таким образом, мы реализовали чисто виртуальную функцию времени компиляции! Заметим, что виртуальной является функция `f_impl()`, а не `f()`.

Итак, эта задача решена. А как написать обычную виртуальную функцию, имеющую реализацию по умолчанию, которая факультативно может быть переопределена? Если следовать тому же паттерну разделения интерфейса и реализации, то мы должны только предоставить реализацию по умолчанию для `B::f_impl()`:

```
template <typename D> class B {
    public:
    ...
    void f(int i) { static_cast<D*>(this)->f_impl(i); }
    void f_impl(int i) {}
};
class D : public B<D> {
    void f_impl(int i) { i_ += i; }
};
class D1 : public B<D1> {
    // Здесь нет f()
};
...
B<D>* b = ...;
b->f(5);          // вызывается D::f()
B<D1>* b1 = ...;
b1->f(5);         // вызывается B::f() по умолчанию
```

Деструкторы и полиморфное удаление

До сих пор мы сознательно избегали вопроса об удалении объектов, реализованных посредством CRTP, неким полиморфным способом. Если вы заново просмотрите представленный выше полный код, например фикстуру эталонного теста `BM_static`, то увидите, что мы либо вообще не удаляли объект, либо конструировали производный объект в стеке. Это связано с тем, что полиморфное удаление приводит к дополнительным осложнениям, с которыми мы только теперь готовы разобраться.

Прежде всего заметим, что во многих случаях полиморфное удаление – вообще не проблема. При создании любого объекта его фактический тип известен. Если код, конструирующий объект, владеет им и в конце концов удаляет, то вопрос «Каков тип удаленного объекта?» вообще никогда не встает. Аналогично, если объекты хранятся в контейнере, то они не удаляются через указатель или ссылку на базовый класс:

```
template <typename D> void apply(B<D>& b) { ... operate on b ... }
{
    std::vector<D> v;
    v.push_back(D(...)); // объекты создаются как D
    ...
    apply(v[0]);         // объекты обрабатываются как B&
}                        // объекты удаляются как D
```

Часто бывает, что при создании и удалении объектов их фактический тип известен, как в этом примере, и тогда никакого полиморфизма нет. В то же время код, обрабатывающий их между созданием и удалением, универсальный и работает с базовым типом, а стало быть, и с любым производным от него типом.

Но что, если нам действительно нужно удалить объект через указатель на базовый класс? Ну что ж, это нелегко. Для начала заметим, что просто вызов оператора `delete` делает совсем не то, что надо:

```
B<D>* b = new D;
...
delete b;
```

Этот код компилируется. Хуже того, даже компиляторы, которые обычно предупреждают, что в классе имеется виртуальная функция, но отсутствует виртуальный деструктор, в этом случае не выдают никаких предупреждений, поскольку виртуальных функций нет, а CRTP-полиморфизм компилятор не считает потенциальным источником проблем. Однако проблема существует и заключается в том, что вызывается только сам деструктор базового класса `B<D>`, а деструктор класса `D` так и не вызывается!

Может возникнуть искушение решить эту проблему так же, как для других *виртуальных функций времени компиляции*, т. е. привести к известному производному типу и вызвать нужную функцию-член производного класса:

```
template <typename D> class B {
    public:
    ~B() { static_cast<D*>(this)->~D(); }
};
```

Но, в отличие от обычных функций, эта попытка полиморфизма катастрофически некорректна, и не по одной, а сразу по двум причинам! Во-первых, в деструкторе базового класса фактический объект уже не принадлежит производному типу, и вызов функций-членов производного класса приводит к неопределенному поведению. Во-вторых, даже если это каким-то образом сработает, деструктор производного класса сделает свою работу, после чего вызовет деструктор базового класса – и мы получим бесконечный цикл.

У этой проблемы есть два решения. Первое – распространить полиморфизм времени компиляции на акт удаления так же, как это делается для любой другой операции, – с помощью шаблонной функции:

```
template <typename D> void destroy(B<D>* b) { delete static_cast<D*>(b); }
```

Здесь все корректно. Оператору delete передается указатель на фактический тип D, и вызывается правильный деструктор. Однако надо следить за тем, чтобы такие объекты удалялись только функцией destroy(), а не оператором delete.

Второе решение – все-таки сделать деструктор виртуальным. При этом возвращаются накладные расходы на вызов виртуальной функции, но только для деструктора. Ну и еще размер объекта увеличивается на размер указателя. Если ни один из этих двух источников снижения производительности вас не тревожит, то можно было бы использовать гибридный статическо-динамический полиморфизм, когда все вызовы виртуальных функций разрешаются во время компиляции, без накладных расходов, за исключением деструктора.

CRTP и управление доступом

Реализуя CRTP-классы, придется побеспокоиться о доступе – любой метод, который вы собираетесь вызывать, должен быть доступен. Либо метод должен быть открытым, либо у вызывающей стороны должны быть специальные права. Это несколько отличается от порядка вызова виртуальных функций – при обращении к виртуальной функции вызывающая сторона должна иметь доступ к функции-члену, поименованной в вызове. Например, для вызова функции базового класса B::f() требуется, чтобы либо B::f() была открытой, либо у вызывающей стороны был доступ к неоткрытым функциям-членам (другая функция-член класса B может вызывать B::f(), даже если та закрыта). А если B::f() виртуальная и переопределена в производном классе D, то во **время выполнения** фактически вызывается D::f(). Не требуется, чтобы D::f() была доступна в месте вызова, в частности D::f() может быть и закрытой.

Ситуация с CRTP-полиморфными вызовами иная. Все вызовы явно прописаны в коде, и вызывающая сторона должна иметь доступ к вызываемым функциям. Обычно это означает, что у базового класса должен быть доступ к функциям-членам производного класса. Рассмотрим пример из предыдущего раздела, только сделаем управление доступом явным:

```
template <typename D> class B {
    public:
    ...
    void f(int i) { static_cast<D*>(this)->f_impl(i); }
    private:
    void f_impl(int i) {}
};
class D : public B<D> {
    private:
    void f_impl(int i) { i_ += i; }
    friend class B<D>;
};
```

Здесь обе функции, B::f_impl() и D::f_impl(), закрыты в своих классах. У базового класса нет специального доступа к производному, и он не может вызывать его закрытые функции-члены. Если мы не готовы сделать закрытую функцию

D::f_impl() открытой для любой вызывающей стороны, то должны будем объявить базовый класс другом производного.

У противоположного действия тоже есть определенное преимущество. Создадим новый производный класс D1, в котором функция реализации f_impl() переопределена по-другому:

```
class D1 : public B<D> {
    private:
    void f_impl(int i) { i_ -= i; }
    friend class B<D1>;
};
```

В этом классе имеется тонкая ошибка – он произведен не от D1, как положено, а от старого класса D; такую ошибку легко допустить, создавая новый класс по старому шаблону. Ошибка обнаружится, если мы попытаемся использовать класс полиморфно:

```
B<D1>* b = new D1;
```

Этот код не компилируется, потому что B<D1> не является базовым классом для D1. Однако не всегда применение CRTP связано с полиморфными вызовами. В любом случае было бы лучше, если бы ошибка диагностировалась сразу при объявлении класса D1. Этого можно добиться, если сделать класс B как бы абстрактным, только в смысле статического полиморфизма. Для этого всего-то и нужно, что сделать конструктор класса B закрытым и объявить производный класс другом:

```
template <typename D> class B {
    int i_;
    B() : i_(0) {}
    friend D;
    public:
    void f(int i) { static_cast<D*>(this)->f_impl(i); }
    private:
    void f_impl(int i) {}
};
```

Обратите внимание на не вполне обычную форму объявления другом: friend D, а не friend class D. Именно так объявляется другом параметр шаблона. Теперь единственным типом, который может конструировать экземпляры класса B<D>, является этот производный класс, D, используемый как параметр шаблона, а ошибочный код class D1 : public B<D> больше не компилируется.

CRTP как паттерн делегирования

До сих пор мы использовали CRTP как аналог динамического полиморфизма на этапе компиляции. Сюда входили и похожие на виртуальные вызовы через указатель на базовый класс (разумеется, на этапе компиляции, с помощью шаблонной функции). Но это не единственный способ применения CRTP. На

самом деле чаще функция вызывается напрямую от имени объекта производного класса. Это фундаментальное различие – обычно открытое наследование выражает отношение *является*, т. е. производный объект является разновидностью базового. Интерфейс и общий код находятся в базовом классе, а производные классы переопределяют конкретную реализацию. Это отношение сохраняется и тогда, когда обращение к CRTP-объекту производится через указатель или ссылку на базовый класс. Такое использование CRTP иногда называют **статическим интерфейсом**.

Если производный объект используется напрямую, то ситуация кардинально меняется – базовый класс больше не определяет интерфейс, а производный является не только реализацией. Производный класс расширяет интерфейс базового, а базовый делегирует часть своего поведения производному.

Расширение интерфейса

Рассмотрим несколько примеров, когда CRTP применяется для делегирования поведения от базового класса производному.

Первый пример совсем простой – для любого класса, предоставляющего оператор `operator==()`, мы хотим автоматически реализовать оператор `operator!=()` как его инверсию:

```
template <typename D> struct not_equal {
    bool operator!=(const D& rhs) const {
        return !static_cast<const D*>(this)->operator==(rhs);
    }
};

class C : public not_equal<C> {
    int i_;
    public:
    C(int i) : i_(i) {}
    bool operator==(const C& rhs) const { return i_ == rhs.i_; }
};
```

Любой класс, наследующий таким образом `not_equal`, автоматически приобретает оператор *неравенства*, который гарантированно согласован с предоставленным оператором *равенства*. Внимательный читатель, возможно, обратил внимание на, мягко говоря, странный способ объявления операторов `==` и `!=`. Разве они не должны быть свободными функциями? В действительности обычно так и есть. Но стандарт этого не требует, и приведенный выше код технически правилен. Причина, по которой такие бинарные операторы, как `==`, `+` и т. д., обычно объявляются свободными функциями, связана с неявными преобразованиями типов. Если имеется сравнение `x == y` и оператор `operator==`, который его предположительно выполняет, вообще является функцией-членом, то он должен быть функцией-членом объекта `x`. Не какого-нибудь объекта, который можно неявно преобразовать в тип `x`, а самого `x` – эта функция-член вызывается от имени `x`. С другой стороны, объект `y` должен допускать неявное

преобразование в тип аргумента этого operator==, который обычно совпадает с типом x. Чтобы восстановить симметрию и разрешить неявные преобразования (если таковые определены) слева и справа от знака ==, мы должны объявить operator== как свободную функцию. Обычно такой функции необходим доступ к закрытым данным-членам класса, как в предыдущем примере, поэтому она должна быть объявлена другом. Собирая все сказанное воедино, мы приходим к такой альтернативной реализации:

```
template <typename D> struct not_equal {
    friend bool operator!=(const D& lhs, const D& rhs) {
        return !(lhs == rhs);
    }
};

class C : public not_equal<C> {
    int i_;
    public:
    C(int i) : i_(i) {}
    friend bool operator==(const C& lhs, const C& rhs) {
        return lhs.i_ == rhs.i_;
    }
};
```

Отметим, что эта реализация not_equal будет правильно работать, даже если производный класс предоставляет operator== в виде функции-члена (понимая все тонкости неявного преобразования, описанные выше).

Существует важное различие между таким использованием CRTP и тем, что мы видели раньше, – объект, который будет использоваться в программе, имеет тип C, и обращение к нему никогда не будет производиться через указатель на класс not_equal<C>. Последний не является полным интерфейсом чего бы то ни было, а представляет собой реализацию, в которой используется интерфейс, предоставляемый производным классом.

Похожий, но чуть более полный пример дает реестр объектов. Иногда желательно, зачастую для отладки, знать, сколько объектов определенного типа существует в данный момент, и, быть может, вести список таких объектов. Мы, безусловно, не хотим оснащать каждый класс механизмом реестра, поэтому следует перенести его в базовый класс. Но тогда возникает проблема: если имеется два производных класса, C и D, наследующих одному и тому же базовому классу B, то счетчик экземпляров B будет учитывать объекты классов C и D. И проблема не в том, что базовый класс не может определить истинный тип производного, – может, если готов нести издержки полиморфизма времени выполнения. Настоящая проблема в том, что в базовом классе только один счетчик (или столько, сколько мы в него зашили), тогда как количество производных классов не ограничено. Можно было бы реализовать очень сложное, дорогое и непереносимое решение на основе **информации о типе во время выполнения** (Run-Time Type Information – **RTTI**), например использовать typeid для определения имени класса и хранить отображение между именами

и счетчиками. Но на самом-то деле нам нужен один счетчик на каждый производный тип, а единственный способ добиться этого – сделать так, чтобы базовый класс знал о типе производного класса на этапе компиляции. И это вновь возвращает нас к CRTP:

```
template <typename D> class registry {
    public:
    static size_t count;
    static D* head;
    D* prev;
    D* next;
    protected:
    registry() {
        ++count;
        prev = nullptr;
        next = head;
        head = static_cast<D*>(this);
        if (next) next->prev = head;
    }
    registry(const registry&) {
        ++count;
        prev = nullptr;
        next = head;
        head = static_cast<D*>(this);
        if (next) next->prev = head;
    }
    ~registry() {
        --count;
        if (prev) prev->next = next;
        if (next) next->prev = prev;
        if (head == this) head = next;
    }
};
template <typename D> size_t registry<D>::count(0);
template <typename D> D* registry<D>::head(nullptr);
```

Мы объявили конструктор и деструктор защищенными, потому что не хотим, чтобы объекты, заносимые в реестр, создавались кем-то, кроме производных классов. Важно также не забыть про копирующий конструктор, потому что тот, что генерируется компилятором по умолчанию, не увеличивает счетчик и не обновляет список (а деструктор уменьшает счетчик, поэтому если не принять мер, счетчик станет отрицательным и переполнится). Для каждого производного класса `D` базовым классом является `registry<D>` – отдельный тип со своими статическими данными-членами, `count` и `head` (последний – указатель на начало списка активных объектов). Любой тип, которому нужно вести реестр активных объектов во время выполнения, должен всего лишь унаследовать `registry`:

```
class C : public registry<C> {
    int i_;
```

```
    public:
    C(int i) : i_(i) {}
};
```

Похожий пример, когда базовый класс должен знать тип производного класса и использовать его для объявления собственных членов, приведен в главе 9.

Еще один сценарий, в котором часто необходимо делегировать поведение производным классам, – задача о посещении. Вообще говоря, посетители – это объекты, которые вызываются для обработки коллекции объектов с данными и выполняют некоторое действие для каждого объекта по очереди. Часто строят иерархии посетителей, в которых производные классы уточняют или изменяют некоторые аспекты поведения базовых. В большинстве популярных реализаций посетителей используется динамический полиморфизм и вызовы виртуальных функций, но статический посетитель дает рассмотренное выше повышение производительности. Посетители обычно не вызываются полиморфно; можно просто создать посетителя и выполнить его. Однако базовый класс посетителя производит обращения к функциям-членам, которые на этапе компиляции могут быть диспетчеризованы производным классам, если в тех имеются подходящие переопределения. Рассмотрим обобщенного посетителя для коллекции животных:

```
struct Animal {
    public:
    enum Type { CAT, DOG, RAT };
    Animal(Type t, const char* n) : type(t), name(n) {}
    const Type type;
    const char* const name;
};

template <typename D> class GenericVisitor {
    public:
    template <typename it> void visit(it from, it to) {
        for (it i = from; i != to; ++i) {
            this->visit(*i);
        }
    }
    private:
    D& derived() { return *static_cast<D*>(this); }
    void visit(const Animal& animal) {
        switch (animal.type) {
            case Animal::CAT:
                derived().visit_cat(animal);
                break;
            case Animal::DOG:
                derived().visit_dog(animal);
                break;
            case Animal::RAT:
                derived().visit_rat(animal);
                break;
        }
```

```
    }
    void visit_cat(const Animal& animal) {
            std::cout << "Feed the cat " << animal.name << std::endl;
    }
    void visit_dog(const Animal& animal) {
        std::cout << "Wash the dog " << animal.name << std::endl;
    }
    void visit_rat(const Animal& animal) {
        std::cout << "Eeek!" << std::endl;
    }
    friend D;
    GenericVisitor() {}
};
```

Посещение дает ожидаемый результат:

```
Feed the cat Fluffy
Wash the dog Fido
Eeek!
```

Но мы не обязаны ограничиваться действиями по умолчанию, а можем переопределить действия для одного или нескольких типов животных:

```
class TrainerVisitor : public GenericVisitor<TrainerVisitor> {
    friend class GenericVisitor<TrainerVisitor>;
    void visit_dog(const Animal& animal) {
        std::cout << "Train the dog " << animal.name << std::endl;
    }
};

class FelineVisitor : public GenericVisitor<FelineVisitor> {
    friend class GenericVisitor<FelineVisitor>;
    void visit_cat(const Animal& animal) {
        std::cout << "Hiss at the cat " << animal.name << std::endl;
    }
    void visit_dog(const Animal& animal) {
        std::cout << "Hiss at the dog " << animal.name << std::endl;
    }
    void visit_rat(const Animal& animal) {
        std::cout << "Eat the rat " << animal.name << std::endl;
    }
};
```

Если наших животных задумает посетить дрессировщик собак, то мы будем использовать класс `TrainerVisitor`:

```
Feed the cat Fluffy
Train the dog Fido
Eeek!
```

Наконец, у посетителя-кошки есть свой набор действий:

```
Hiss at the cat Fluffy
Hiss at the dog Fido
Eat the rat Stinky
```

Гораздо подробнее мы поговорим о различных видах посетителей в главе 18.

Резюме

Мы изучили довольно хитроумный паттерн проектирования, сочетающий обе стороны C++: обобщенное программирование (шаблоны) и объектно-ориентированное программирование (наследование). В полном соответствии со своим названием паттерн Рекурсивный шаблон создает петлю, в которой производный класс наследует интерфейс и реализацию от базового класса, а базовый класс имеет доступ к интерфейсу производного через параметры шаблона. У CRTP два основных применения: настоящий статический полиморфизм, или *статический интерфейс*, когда доступ к объекту осуществляется преимущественно как к базовому типу, и расширение интерфейса, или делегирование, когда к объекту производного класса обращаются напрямую, но в реализации используется CRTP для предоставления общей функциональности.

В следующей главе мы познакомимся с идиомой, в которой используется только что рассмотренный паттерн. Эта идиома также изменяет привычный способ передачи параметров функциям – по порядку – и позволяет использовать независимые от порядка именованные аргументы. Хотите узнать, как? Читайте дальше!

Вопросы

- Насколько дорого обходится вызов виртуальной функции и почему?
- Почему у вызова аналогичной функции, разрешаемого во время компиляции, нет таких накладных расходов?
- Как реализовать вызовы полиморфных функций на этапе компиляции?
- Как использовать CRTP для расширения интерфейса базового класса?

Глава 9

Именованные аргументы и сцепление методов

В этой главе мы изучим решение очень актуальной для C++ проблемы – слишком большое количество аргументов. Нет, мы ведем речь не об аргументах, предъявляемых программистами в спорах между собой: ставить ли фигурную скобку в конце строки или в начале следующей (решения этой проблемы мы не знаем). Проблема в том, что в C++ встречаются функции, принимающие чересчур много аргументов. Если вам доводилось сопровождать большую систему на C++ в течение достаточно длительного времени, то вы наверняка наблюдали это – поначалу объявление функции совсем простое, но со временем, для поддержки новых возможностей, появляются все новые аргументы, часто имеющие значения по умолчанию.

В этой главе рассматриваются следующие вопросы:

- чем плохи длинные объявления функций;
- какие имеются альтернативы;
- в чем недостатки идиомы именованных аргументов;
- как можно обобщить идиому именованных аргументов.

Технические требования

Примеры кода: **https://github.com/PacktPublishing/Hands-On-Design-Patterns-with-CPP/tree/master/Chapter09**.

Библиотека Google Benchmark: **https://github.com/google/benchmark** (инструкции по установке см. в главе 5).

Проблема аргументов

Каждый, кому приходилось работать с достаточно большой системой, написанной на C++, в какой-то момент времени начинал добавлять аргументы в объявление функции. Чтобы не «сломать» существующий код, у нового аргумента часто бывает значение по умолчанию, совместимое с прежней функцио-

нальностью. В первый раз это работает на ура, во второй – нормально, а потом начинаешь пересчитывать аргументы при каждом вызове функции. Длинным объявлениям функций присущи и другие проблемы, и если мы хотим что-то улучшить, то надо бы сначала понять, в чем они состоят. Мы начнем этот раздел с углубленного анализа проблемы, а затем перейдем к решению.

Что плохого в большом количестве аргументов?

Не важно, был ли код, которому передается много аргументов, написан так с самого начала или *органически* эволюционировал, он все равно хрупкий и уязвимый для ошибок программиста. Основная проблема заключается в том, что обычно бывает много аргументов одного типа, и человек неправильно их подсчитывает. Рассмотрим проектирование игры в построение цивилизации – когда игрок создает новый город, конструируется соответствующий объект. Игрок должен выбрать, какие сооружения построить в городе, а игра предлагает варианты, исходя из доступных ресурсов:

```
class City {
    public:
    enum center_t { KEEP, PALACE, CITADEL };
    City(size_t number_of_buildings,
         size_t number_of_towers,
         size_t guard_strength,
         center_t center,
         bool with_forge,
         bool with_granary,
         bool has_fresh_water,
         bool is_coastal,
         bool has_forest);
    ...
};
```

Вроде бы мы все предусмотрели. В начале игры выделим каждому игроку город с донжоном, сторожевой башней, двумя зданиями и караульной ротой:

```
City Capital(2, 1, 1, City::KEEP, false, false, false, false);
```

Ошибку видите? Компилятор, к счастью, видит – недостаточно аргументов. Поскольку компилятор не позволит нам допустить здесь ошибку, то особой проблемы нет, мы просто добавим аргумент для параметра `has_forest`. Кроме того, предположим, что игра поместила город близ реки, так что теперь в нем есть вода:

```
City Capital(2, 1, 1, City::KEEP, false, true, false, false, false);
```

Это было просто... но! Город теперь стоит на реке, но у него нет питьевой воды (интересно, а что течет в этой реке?). Но, по крайней мере, горожане не будут голодать, поскольку они задаром получили амбар. Эту ошибку – передачу `true` не в том аргументе – придется искать в ходе отладки. Ко всему прочему, код слишком многословный, и нам, возможно, придется снова и снова

набивать одни и те же значения. А быть может, игра старается по умолчанию располагать города вблизи рек и лесов? Ладно, тогда сделаем так:

```
class City {
    public:
    enum center_t { KEEP, PALACE, CITADEL };
    City(size_t number_of_buildings,
         size_t number_of_towers,
         size_t guard_strength,
         center_t center,
         bool with_forge,
         bool with_granary,
         bool has_fresh_water = true,
         bool is_coastal = false,
         bool has_forest = true);
    ...
};
```

А теперь вернемся к первой попытке создать город – на этот раз код компилируется, хотя одного аргумента недостает, а мы и не в курсе, что посчитали аргументы неправильно. Игра пользуется огромным успехом, и в очередной версии мы завели новое классное здание – храм! Конечно, в конструктор нужно добавить еще один аргумент. Имеет смысл поместить его после `with_granary`, рядом с прочими зданиями и до особенностей ландшафта. Но тогда придется изменить все обращения к конструктору `City`. Хуже того, очень легко допустить ошибку, поскольку `false` в роли *нет храма* для программиста и для компилятора выглядит в точности так же, как `false` в роли *нет питьевой воды*. Новый аргумент необходимо вставить в нужное место, среди длинной череды похожих на него значений.

Понятное дело, старый код игры работает без храмов, поэтому нужны они только в новой версии. Всегда лучше не трогать существующий код без острой необходимости. Мы могли бы этого добиться, поместив новый аргумент в конец и снабдив его значением по умолчанию, тогда все прежние вызовы конструктора будут создавать в точности такой же город, как и раньше:

```
class City {
    public:
    enum center_t { KEEP, PALACE, CITADEL };
    City(size_t number_of_buildings,
         size_t number_of_towers,
         size_t guard_strength,
         center_t center,
         bool with_forge,
         bool with_granary,
         bool has_fresh_water = true,
         bool is_coastal = false,
         bool has_forest = true,
         bool with_temple = false);
         ...
};
```

Но теперь мимолетное удобство диктует нам дизайн интерфейса, которому суждена долгая жизнь. Параметры не собраны в логические группы, и в будущем это сулит еще больше ошибок. К тому же мы не достигли в полной мере цели не трогать код, не нуждающийся в изменении, – в следующей версии добавится новый ландшафт, пустыня, а вместе с ним и новый аргумент:

```
class City {
    public:
    enum center_t { KEEP, PALACE, CITADEL };
    City(size_t number_of_buildings,
         size_t number_of_towers,
         size_t guard_strength,
         center_t center,
         bool with_forge,
         bool with_granary,
         bool has_fresh_water = true,
         bool is_coastal = false,
         bool has_forest = true,
         bool with_temple = false,
         bool is_desert = false);
    ...
};
```

Один раз начав, мы должны будем снабжать значениями по умолчанию все новые аргументы, добавляемые в конец. Кроме того, чтобы создать город в пустыне, мы должны будет также указать, есть ли в нем храм. Логических причин для этого нет, но нам связывает руки процесс эволюции интерфейса. Ситуация усугубляется, если принять во внимание, что многие используемые нами типы допускают преобразование друг в друга:

```
City Capital(2, 1, false, City::KEEP, false, true, false, false, false);
```

Это предложение создает город без караульных рот, а не без того, что программист хотел убрать, задавая третий аргумент равным `false`. Даже типы `enum` не обеспечивают полной защиты. Вы, вероятно, обратили внимание, что все новые города обычно начинаются с донжона, поэтому разумно было бы включать его по умолчанию:

```
class City {
    public:
    enum center_t { KEEP, PALACE, CITADEL };
    City(size_t number_of_buildings,
         size_t number_of_towers,
         size_t guard_strength,
         center_t center = KEEP,
         bool with_forge = false,
         bool with_granary = false,
         bool has_fresh_water = true,
         bool is_coastal = false,
         bool has_forest = true,
         bool with_temple = false,
```

```
        bool is_desert = false);
    ...
};
```

Теперь нам не нужно набирать так много аргументов и, быть может, даже удастся избежать некоторых ошибок (если мы не записываем аргументы вообще, то не можем записать их в неправильном порядке). Зато появляется возможность новых:

```
City Capital(2, 1, City::CITADEL);
```

Двум только что нанятым нами караульным ротам (числовое значение `CITADEL` равно 2) будет тесновато в непритязательном донжоне (собирались-то мы изменить именно этот аргумент, да не получилось). Тип `enum class` в стандарте C++11 предлагает улучшенную защиту, потому что каждый такой класс перечисления – отдельный тип, не допускающий преобразования в целое число, но проблема в целом остается. Как мы видели, существует две проблемы, связанные с передачей большого количества значений функциям в виде отдельных аргументов. Во-первых, объявления получаются очень длинными, а вызовы функций подвержены ошибкам. Во-вторых, если требуется добавить значение или изменить тип параметра, то приходится модифицировать много кода. Решение обеих проблем существовало еще до появления C++ и использовалось в C: создавать агрегаты, т. е. структуры, для объединения многих значений в одном параметре.

Агрегатные параметры

Агрегатный параметр – это структура или класс, содержащий все значения аргументов, мы используем его, вместо того чтобы передавать аргументы по отдельности. Необязательно иметь один агрегат; например, наш конструктор города может принимать несколько структур: одну для всех свойств, связанных с ландшафтом, которые задает игра, другую – для свойств, которыми игрок управляет сам:

```
struct city_features_t {
    size_t number_of_buildings;
    size_t number_of_towers;
    size_t guard_strength;
    enum center_t { KEEP, PALACE, CITADEL };
    center_t center = KEEP;
    bool with_forge = false;
    bool with_granary = false;
    bool with_temple;
};
struct terrain_features_t {
    bool has_fresh_water;
    bool is_coastal;
    bool has_forest;
    bool is_desert;
```

```
};
class City {
    public:
    City(city_features_t city_features, terrain_features_t
        terrain_features);
...
};
```

У такого решения много преимуществ. Во-первых, присваивание значения аргументам производится явно, по имени, и очень хорошо видно в коде:

```
city_features_t city_features;
city_features.number_of_buildings = 2;
city_features.center = city_features::KEEP;
...
terrain_features_t terrain_features;
terrain_features.has_fresh_water = true;
...
City Capital(city_features, terrain_features);
```

Так гораздо проще понять, какое значение имеет каждый аргумент, и ошибки куда менее вероятны. Если нужно добавить новое свойство, то, как правило, дело сводится к добавлению еще одного члена в какой-то агрегатный тип. Изменить придется лишь код, имеющий непосредственное отношение к новому аргументу, а функции и классы, которые просто передают и перенаправляют аргументы, не нужно менять вовсе. Мы можем даже снабдить агрегатные типы конструкторами по умолчанию, которые будут присваивать всем аргументам значения по умолчанию:

```
struct terrain_features_t {
    bool has_fresh_water;
    bool is_coastal;
    bool has_forest;
    bool is_desert;
    terrain_features_t() :
        has_fresh_water(true),
        is_coastal(false),
        has_forest(true),
        is_desert(false)
    {}
};
```

В общем и целом это отличное решение проблемы функций с большим количеством параметров. Но у него есть один недостаток: агрегаты необходимо создавать явно и инициализировать построчно. Во многих случаях это нормально, особенно когда эти классы и структуры представляют переменные, которые мы собираемся хранить в течение длительного времени. Но если они используются только как контейнеры параметров, то код получается излишне многословным – хотя бы уже потому, что у агрегатной переменной должно быть имя. Нам это имя ни к чему, поскольку мы собираемся использовать его

всего один раз при вызове функции, однако придумать и записать его придется. Хорошо было бы просто обойтись временной переменной:

```
City Capital(city_features_t() ... как-то сюда попадают аргументы ... );
```

Оно бы и работало, если бы могли присвоить значения данным-членам. Можно было бы сделать это в конструкторе:

```
struct terrain_features_t {
    bool has_fresh_water;
    bool is_coastal;
    bool has_forest;
    bool is_desert;
    terrain_features_t(
        bool has_fresh_water,
        bool is_coastal,
        bool has_forest,
        bool is_desert
    ) :
        has_fresh_water(has_fresh_water),
        is_coastal(is_coastal),
        has_forest(has_forest),
        is_desert(is_desert)
    {}
};

City Capital(city_features_t(...),
terrain_features_t(true, false, false, true));
```

Это работает, но приводит к тому, с чего мы начали: к функции с длинным списком булевых аргументов, которые легко перепутать. Фундаментальная проблема заключается в том, что в C++ аргументы функций позиционные, а мы пытаемся придумать что-то, что позволило бы задавать аргументы по имени. Решение, предлагаемое агрегатными объектами, – скорее, побочный эффект, но если группировка значений в одном классе улучшает общий дизайн, то это обязательно нужно сделать. Однако если рассматривать это как попытку решить конкретно проблему именованных аргументов, без иных, более насущных причин группировать значения, то следует признать ее неудачной. Далее мы покажем, как преодолеть этот недостаток.

Именованные аргументы в C++

Мы видели, что объединение логически связанных значений в агрегатном объекте дает полезный побочный эффект: мы можем передавать значения функциям и обращаться к ним по имени, а не по позиции в длинном списке. Однако ключевое слово здесь – *логически связанные*, агрегирование значений только потому, что они оказались вместе в одном обращении к функции, приводит к созданию лишних объектов, для которых мы предпочли бы не изобретать имена. Нам нужен способ создавать временные агрегаты – и лучше бы без яв-

ных имен и объявлений. У этой задачи есть решение, и оно существовало в C++ уже давно; нужно только посмотреть на проблему свежим взглядом и под другим углом, что мы и собираемся сделать.

Сцепление методов

Сцепление методов – техника, которую C++ заимствовал у языка **Smalltalk**. Ее главная цель – избавиться от лишних локальных переменных. Вы уже пользовались сцеплением методов, быть может, не осознавая этого. Рассмотрим код, который вы наверняка писали много раз:

```
int i, j;
std::cout << i << j;
```

В последней строке дважды вызывается оператор вывода в поток <<. В первый раз он вызывается от имени объекта в левой части оператора, `std::cout`. А во второй раз – от имени какого объекта? Вообще говоря, синтаксически оператор – это просто способ вызвать функцию с именем `operator<<()`. Обычно данный конкретный оператор реализуется как свободная функция, но в классе `std::ostream` также имеется несколько перегруженных функций-членов, одна из которых принимает значения типа `int`. Таким образом, последняя строка – на самом деле вот что:

```
std::cout.operator(i).operator<<(j);
```

Второй вызов `operator<<()` производится от имени результата первого. Эквивалентный код на C++ выглядит так:

```
auto& out1 = std::cout.operator(i);
out1.operator<<(j);
```

Это и есть сцепление методов – вызов одного метода возвращает объект, от имени которого вызывается следующий метод. В случае `std::cout` функция-член `operator<<()` возвращает ссылку на сам объект. Кстати говоря, свободная функция `operator<<()` делает то же самое, только вместо неявного аргумента `this` ей в качестве первого аргумента явно передается объект потока (в C++20 различие станет гораздо менее заметным из-за универсального синтаксиса вызова).

Теперь можно использовать сцепление методов для устранения явно поименованного объекта аргумента.

Сцепление методов и именованные аргументы

Как мы уже видели, агрегатные объекты хороши, когда используются не только для хранения аргументов; если нам нужен объект, чтобы сохранить состояние системы в течение длительного времени, то можно построить его и заодно передать в качестве единственного аргумента функции, которой это состояние нужно. Проблема возникает, когда агрегаты создаются, только чтобы один раз вызвать функцию. С другой стороны, писать функции с большим количеством

аргументов тоже не хочется. В особенности это относится к функциям, для большинства аргументов которых оставлены значения по умолчанию, а изменено лишь несколько. Возвращаясь к нашей игре, предположим, что каждый день игрового времени обрабатывается вызовом функции.

Функция вызывается один раз за игровой день, чтобы перевести город в следующий день и обработать последствия различных случайных событий, сгенерированных игрой:

```
class City {
    ...
    void day(bool flood = false, bool fire = false,
             bool revolt = false, bool exotic_caravan = false,
             bool holy_vision = false, bool festival = false, ... );
    ...
};
```

За день может произойти много разных событий, но в течение одного дня редко происходит больше одного события. По умолчанию все аргументы равны false, но это ничем не помогает; у событий нет определенного порядка, поэтому если происходит праздник (festival), то придется задать и все предыдущие аргументы, пусть даже они имеют значения по умолчанию.

Агрегатный объект помог бы, да еще как, только его необходимо создать и поименовать:

```
class City {
    ...
    struct DayEvents {
        bool flood;
        bool fire;
        ...
        DayEvents() : flood(false), fire(false) ,,, {}
    };
    void day(DayEvents events);
    ...
};

City capital(...);
City::DayEvents events;
events.fire = true;
capital.day(events);
```

Мы хотели бы создать временный объект DayEvents только для вызова City::day(), но нужен способ задать его данные-члены. Тут-то и приходит на помощь сцепление методов:

```
class City {
    ...
    class DayEvents {
        public:
        DayEvents() : flood(false), fire(false) ,,, {}
        DayEvents& SetFlood() { flood = true; return *this; }
        DayEvents& SetFire() { fire = true; return *this; }
```

```
        ...
        private:
        friend City;
        bool flood;
        bool fire;
        ...
    };
    void day(DayEvents events);
    ...
};

City capital(...);
capital.day(City::DayEvents().SetFire());
```

Конструктор по умолчанию создает безымянный временный объект. От имени этот объекта вызывается метод `SetFire()`. Он модифицирует объект и возвращает ссылку на него же. Этот созданный и модифицированный временный объект передается функции `day()`, которая обрабатывает произошедшие за день события, выводит изображение города, объятого пламенем, воспроизводит звуки пожара и обновляет состояние города, отражая тот факт, что некоторые здания повреждены огнем.

Поскольку каждый из методов `Set()` возвращает ссылку на один и тот же объект, то мы можем вызывать в одной цепочке несколько методов, описывающих разные события. Ну и конечно, методы `Set()` могут принимать аргументы; например, `SetFire()` необязательно должен присваивать событию `fire` значение `true` вместо подразумеваемого по умолчанию `false`, можно было бы определить метод, который устанавливает любое значение флага пожара:

```
DayEvents& SetFire(bool value = true) { fire = value; return *this; }
```

Сегодня в нашем городе проходит ярмарка и одновременно большой праздник, поэтому король нанял дополнительную караульную роту вдобавок к двум уже расквартированным:

```
City capital(...);
capital.day(City::DayEvents().SetMarket().SetFestival().SetGuard(3));
```

Заметим, что нам не пришлось ничего задавать для событий, которые не происходили. Теперь мы получили настоящие именованные аргументы: при вызове функции аргументы передаются по имени в любом порядке, причем не требуется явно упоминать аргументы, значения которых мы не изменяем. Это и есть идиома именованных аргументов в C++. Вызов с именованными аргументами, конечно, более многословный, чем с позиционными, т. к. для каждого аргумента нужно явно указать имя. В этом и был смысл упражнения. С другой стороны, мы выигрываем, если имеется длинный список аргументов по умолчанию, которые изменять не нужно. Стоит остановиться на производительности – мы совершаем много дополнительных вызовов функций: конструктора и по одному вызову `Set()` для каждого именованного аргумента, а это, наверное, не бесплатно. Давайте точно выясним, сколько это стоит.

Производительность идиомы именованных аргументов

Без сомнения, вызов с именованными аргументами обходится дороже, потому что вызывается больше функций. С другой стороны, все эти вызовы очень простые, и если определены в заголовочном файле, где вся реализация видна компилятору, то почему бы компилятору не встроить все вызовы `Set()`, исключив тем самым лишние временные переменные. При наличии хорошего оптимизатора можно ожидать от идиомы именованных аргументов почти такой же производительности, как от именованного агрегатного объекта.

Подходящим инструментом для измерения производительности одного вызова функции является эталонный микротест. Мы используем для этой цели библиотеку Google Benchmark. Обычно эталонные тесты помещают в один файл, но нам понадобится еще один исходный файл, если мы хотим, чтобы функция была внешней, а не встраиваемой. С другой стороны, методы `Set()` просто-таки обязаны встраиваться, поэтому их следует определить в заголовочном файле. Второй исходный файл должен содержать определение функции, которую мы будем вызывать с именованными или позиционными аргументами. Оба файла объединяются на этапе компоновки:

```
$CXX named_args.C named_args_extra.C -I$GBENCH_DIR/include -g -O4 -I. \
    -Wall -Wextra -Werror -pedantic --std=c++14 \
    $GBENCH_DIR/lib/libbenchmark.a -lpthread -lrt -lm -o named_args
```

Мы можем сравнить позиционные аргументы, именованные аргументы и агрегат аргументов. Результат будет зависеть от типов и количества аргументов. Например, для функции с четырьмя булевыми аргументами можно сравнить следующие вызовы:

```
Positional p(true, false, true, false);          // позиционные аргументы

Named n(Named::Options().SetA(true).SetC(true)); // идиома именованных аргументов
Aggregate::Options options;
options.a = true;
options.c = true;
Aggregate a(options));                           // агрегатный объект
```

Измеренная производительность сильно зависит от компилятора и заданных параметров оптимизации. Так, следующие результаты получены для GCC6 с флагом -O3:

```
-------------------------------------------------------------
Benchmark                   Time           CPU Iterations
-------------------------------------------------------------
BM_positional_const        47 ns         47 ns   13459657   642.531M items/s
BM_named_const             40 ns         40 ns   17145954   767.657M items/s
BM_aggregate_const        182 ns        182 ns    3872255   167.293M items/s
```

Производительность падает для явно поименованного агрегатного объекта, который компилятор не смог оптимизировать «до полного уничтожения» (поскольку объект больше нигде не используется, такая оптимизация теоретиче-

ски возможна). Именованные и позиционные аргументы показывают схожую производительность, причем именованные даже слегка опережают (но не делайте из этого далеко идущих выводов, производительность вызовов функций сильно зависит от того, что еще происходит в программе, потому что аргументы передаются в регистрах, а доступность регистров определяется контекстом).

В нашем эталонном тесте значениями аргументов были константы времени компиляции. Это довольно типичный случай, особенно если аргументы задают какие-то параметры – очень часто в каждом месте вызова многие параметры статичны и неизменны (в других местах программы, где вызывается та же функция, значения могут быть другими, но в данной конкретной строке они известны уже во время компиляции). Например, если в нашей игре имеется специальная ветвь для обработки стихийных бедствий, то в основной ветви при моделировании дня флаги наводнения, пожара и прочих бедствий будут равны `false`. Однако не менее часто аргументы вычисляются во время выполнения. Как это сказывается на производительности? Напишем еще один эталонный тест, в котором значения аргументов будут извлекаться из вектора:

```
std::vector<int> v;            // заполнить v случайными значениями
size_t i = 0;

// ... Цикл тестирования ...
const bool a = v[i++];
const bool b = v[i++];
const bool c = v[i++];
const bool d = v[i++];
if (i == v.size()) i = 0;      // предполагаем, что v.size() % 4 == 0

Positional p(a, b, c, d);      // позиционные аргументы

Named n(Named::Options().SetA(a).SetC(b)
                        .SetC(c).SetD(d)); // идиома именованных аргументов

Aggregate::Options options;
options.a = a;
options.b = b;
options.c = c;
options.d = d;
Aggregate a(options));
```

Кстати говоря, не рекомендуется сокращать предыдущий код таким манером:

```
Positional p(v[i++], v[i++], v[i++], v[i++]);
```

Причина в том, что порядок вычисления аргументов не определен, поэтому неизвестно, какое выражение `i++` будет вычислено первым. Если вначале `i` равно `0`, то может быть вычислено как выражение `Positional(v[0], v[1], v[2], v[3])`, так и `Positional(v[3], v[2], v[1], v[0])` или еще какая-то перестановка.

При том же компиляторе и на том же оборудовании мы теперь получаем другие числа:

```
-------------------------------------------------------------
Benchmark                   Time           CPU Iterations
-------------------------------------------------------------
BM_positional_var          77 ns         77 ns      8873908   396.065M items/s
BM_named_var              211 ns        211 ns      2929368   144.322M items/s
BM_aggregate_var          200 ns        200 ns      3391551   152.224M items/s
```

Изучение результатов позволяет высказать догадку, что на этот раз компилятор не полностью устранил временный безымянный объект и сгенерировал код, похожий на явно поименованный локальный объект с аргументами. В общем случае результат работы оптимизатора трудно предсказать. Например, можно протестировать слегка отличающийся случай, когда функция принимает больше аргументов и только последний принимает значение, отличное от умалчиваемого:

```
-------------------------------------------------------------
Benchmark                   Time           CPU Iterations
-------------------------------------------------------------
BM_positional_const       115 ns        115 ns      5921750   265.832M items/s
BM_named_const             42 ns         42 ns     16754061    727.64M items/s
BM_aggregate_const        220 ns        220 ns      3147659   138.592M items/s
BM_positional_var         142 ns        142 ns      5894532   214.538M items/s
BM_named_var               49 ns         49 ns     13096441   623.279M items/s
BM_aggregate_var          258 ns        257 ns      3058525   118.651M items/s
```

Вообще говоря, вызов функции с большим количеством позиционных параметров занимает больше времени, и наш тест с позиционными аргументами отражает это. Время конструирования агрегатного объекта практически такое же, по крайней мере для небольших объектов, но наблюдаемое изменение связано с тем, что компилятору удалось оптимизировать безымянный временный объект с параметрами, устранив все его следы из кода.

Эталонный тест не дал убедительных результатов. Мы можем сказать, что идиома именованных аргументов работает не хуже явно поименованного агрегатного объекта. Если бы компилятор умел устранять безымянный временный объект, то результат был бы сравним или даже лучше, чем вызов, с позиционными аргументами, особенно когда аргументов много. Если оптимизация не производится, то вызов может оказаться немного медленнее. С другой стороны, во многих случаях производительность самого вызова функции некритична; например, в нашей игре объекты городов конструируются, только когда игрок строит город, всего несколько раз за игру. События, произошедшие за день, обрабатываются один раз за игровой день, который вряд ли занимает больше нескольких секунд реального времени, чтобы игрок мог в полной мере насладиться взаимодействием с игрой. Но функции, которые многократно вызываются в критическом с точки зрения быстродействия коде, должны по возможности встраиваться, и в этом случае мы вправе ожидать лучшей оптимизации передачи аргументов. В общем и целом можно заключить, что если скорость вызова конкретной функции некритична для работы программы, то о накладных расходах, связанных с именованными аргументами, можно не думать. А для критических вызовов производительность следует измерять для

каждого случая в отдельности, и может случиться, что именованные аргументы будут даже быстрее позиционных.

Сцепление методов в общем случае

Применение сцепления методов в C++ не ограничено передачей аргументов (мы уже видели еще одно, хотя и хорошо скрытое, применение – потоковый ввод-вывод). Чтобы использовать его в других контекстах, полезно рассмотреть более общие формы сцепления методов.

Сцепление и каскадирование методов

Термин *каскадирование методов* нечасто встречается в контексте C++, и не без причины – C++ его попросту не поддерживает. Под каскадированием методов понимается вызов последовательности методов для одного и того же объекта. Например, в языке *Dart*, где каскадирование поддерживается, можно написать:

```
var opt = Options();
opt.SetA()..SetB();
```

Здесь сначала вызывается метод `SetA()` от имени объекта `opt`, а потом метод `SetB()` от имени того же объекта. Этот код эквивалентен такому:

```
var opt = Options();
opt.SetA()
opt.SetB();
```

Минуточку, но не то же ли самое мы проделали в C++ с нашим объектом `options`? Так-то оно так, но мы упустили из виду одно важное различие. В случае сцепления методов следующий метод применяется к результату работы предыдущего. Вот как выглядят сцепленные методы в C++:

```
Options opt;
opt.SetA().SetB();
```

Этот сцепленный вызов эквивалентен такому коду:

```
Options opt;
Options& opt1 = opt.SetA();
Options& opt2 = opt1.SetB();
```

В C++ нет синтаксиса каскадирования, но код, эквивалентный каскаду, можно было бы написать так:

```
Options opt;
opt.SetA();
opt.SetB();
```

Однако это то же самое, что мы делали раньше, и короткая форма будет такой же:

```
Options opt;
opt.SetA().SetB();
```

В данном случае каскадирование возможно, потому что методы возвращают ссылку на свой же объект. Мы еще можем сказать, что и следующий код эквивалентен:

```
Options opt;
Options& opt1 = opt.SetA();
Options& opt2 = opt1.SetB();
```

Технически это правда. Но в силу способа написания методов имеется дополнительная гарантия, что `opt`, `opt1` и `opt2` ссылаются на один и тот же объект. Каскадирование методов всегда можно реализовать посредством сцепления, но это налагает ограничения на интерфейс, потому что все вызовы должны возвращать ссылку на `this`. Иногда для этой техники реализации используется несколько громоздкое название: **каскадирование путем сцепления посредством возврата себя** (cascading-by-chaining by returning self). В общем случае сцепление методов не ограничивается возвратом *себя*, т. е. ссылки на сам объект (`*this` в C++). Чего можно достичь с помощью более общего сцепления методов? Давайте посмотрим.

Сцепление методов в общем случае

Если сцепленный метод не возвращает ссылку на сам объект, то он должен вернуть новый объект. Обычно этот объект имеет такой же тип или, по крайней мере, тип, принадлежащий той же иерархии классов, если методы полиморфны. Например, рассмотрим класс, реализующий коллекцию данных. В нем имеется метод для фильтрации данных с помощью предиката (вызываемого объекта, метод `operator()` которого возвращает `true` или `false`). Также имеется метод для сортировки коллекции. Оба метода создают новый объект коллекции, оставляя исходный без изменения. Теперь мы можем оставить в коллекции только допустимые данные (в предположении, что имеется предикат `is_valid`) и создать отсортированную коллекцию допустимых данных:

```
Collection c;
... поместить данные в коллекцию ...
Collection valid_c = c.filter(is_valid);
Collection sorted_valid_c = valid_c.sort();
```

Промежуточный объект можно устранить, воспользовавшись сцеплением методов:

```
Collection c;
...
Collection sorted_valid_c = c.filter(is_valid).sort();
```

Из предыдущего раздела должно быть ясно, что это пример сцепления методов, причем более общий, чем тот, что мы видели раньше, – каждый метод возвращает объект одного и того же типа, но не один и тот же объект. В этом примере очень наглядно проявляется различие между сцеплением и каскадированием – каскад методов отфильтровал бы и отсортировал исходную коллекцию (в предположении, что мы решили поддерживать такие операции).

Сцепление методов в иерархиях классов

Когда сцепление методов применяется к иерархиям классов, возникает следующая проблема. Предположим, что наш метод `sort()` возвращает отсортированную коллекцию данных, являющуюся объектом типа `SortedCollection`, производного от класса `Collection`. К другому классу мы переходим, чтобы можно было поддержать эффективный поиск, поэтому класс `SortedCollection` располагает методом `search()`, которого нет в базовом классе. Мы по-прежнему можем использовать сцепление методов и даже вызывать методы базового класса для объектов производного, но при этом цепочка рвется:

```
class SortedCollection;
class Collection {
    public:
    Collection filter();
    SortedCollection sort(); // преобразует Collection в SortedCollection
};

class SortedCollection : public Collection {
    public:
    SortedCollection search();
    SortedCollection median();
};

SortedCollection Collection::sort() {
    SortedCollection sc;
    ... отсортировать коллекцию ...
    return sc;
}

Collection c;
auto c1 = c.sort()    // теперь это SortedCollection
          .search()  // требует SortedCollection и получает ее
          .filter()  // вызывается, но возвращает Collection
          .median(); // требует SortedCollection, получает Collection
```

Полиморфизм (виртуальные функции) здесь не помогает. Во-первых, пришлось бы определить виртуальные функции `search()` и `median()` в базовом классе, хотя мы не собирались поддерживать в нем эту функциональность, т. к. она свойственна только производному классу. И чисто виртуальными объявить их тоже нельзя, потому что мы используем `Collection` как конкретный класс, а любой класс с виртуальными функциями – абстрактный, и, стало быть, создать объекты такого класса невозможно. Можно было бы написать эти функции, так чтобы они завершали программу во время выполнения, но тогда мы просто переместим обнаружение программной ошибки – поиск в неотсортированной коллекции – с этапа компиляции на этап выполнения. Хуже того, это и не помогает даже:

```
class SortedCollection;
class Collection {
    public:
```

```
    Collection filter();
    SortedCollection sort();      // преобразует Collection в SortedCollection
    virtual SortedCollection median();
};

class SortedCollection : public Collection {
    public:
    SortedCollection search();
    SortedCollection median();
};

SortedCollection Collection::sort() {
    SortedCollection sc;
    ... отсортировать коллекцию ...
    return sc;
}

SortedCollection Collection::median() {
    cout << "Вызвана Collection::median!!!" << endl;
    abort();
    SortedCollection dummy;
    return dummy;             // все равно нужно что-то вернуть
}

Collection c;
auto c1 = c.sort()     // теперь это SortedCollection
          .search()    // требует SortedCollection и получает ее
          .filter()    // вызывается, но возвращает Collection
          .median();   // вызвана Collection::median!
```

Этот код не работает, потому что `Collection::filter` возвращает копию объекта, а не ссылку на него. А возвращаемый объект принадлежит базовому классу `Collection`. Будучи вызван от имени объекта `SortedCollection`, он «выдергивает» и возвращает часть, относящуюся к производному классу. Если вы думаете, что, сделав `filter()` тоже виртуальной и переопределив ее в производном классе, сможете решить проблему ценой переопределения всех функций базового класса, то вас поджидает еще один сюрприз – виртуальные функции должны возвращать одинаковые типы, а единственное исключение делается для *ковариантных возвращаемых типов*. Ссылки на базовый и производный классы – ковариантные типы, но сами классы, возвращаемые по значению, таковыми не являются.

Заметим, что этой проблемы не возникло бы, если бы мы возвращали ссылки на объекты. Однако функция-член может вернуть только ссылку на объект, от имени которого вызывалась; если создать в теле функции новый объект и вернуть ссылку на него, то она станет висячей ссылкой на временный объект, который удаляется в момент возврата из функции. Результат – неопределенное поведение (скорее всего, программа «грохнется»). С другой стороны, если всегда возвращать ссылку на исходный объект, то мы не сможем изменить его тип с базового класса на производный.

В C++ эта проблема решается использованием шаблонов и паттерна Рекурсивный шаблон. Этот паттерн настолько закрученный, что в его английском

названии даже присутствует слово *curious* (странный). В этой книге мы посвятили паттерну CRTP целую главу. К нашему случаю он применяется довольно прямолинейно – базовый класс должен вернуть правильный тип из своих функций-членов, но не может, потому что не знает, какой это тип. Решение – передать нужный тип базовому классу в виде параметра шаблона. Конечно, чтобы это работало, базовый класс должен быть шаблоном:

```
template <typename T>
class Collection {
    public:
    Collection() {}
    T filter();                         // "*this" на самом деле имеет тип T, а не A
    T sort() { T sc; ... return sc; } // создать новую отсортированную коллекцию
};

class SortedCollection : public Collection<SortedCollection> {
    public:
    SortedCollection search();
    SortedCollection median();
};

Collection<SortedCollection> c;
auto c1 = c.sort()     // теперь это SortedCollection
          .search()    // требует SortedCollection и получает ее
          .filter()    // вызывается, сохраняет SortedCollection
          .median();   // вызвана SortedCollection::median!
```

Это сложное решение. Оно работает, но сама его сложность наводит на мысль, что сцеплением методов не стоит злоупотреблять, а лучше вообще не использовать, если в середине цепочки объект должен изменить тип. И тому есть основательная причина – изменение типа объекта принципиально отличается от вызова последовательности методов. Это более важное событие, и, наверное, лучше сделать его явным, а новому объекту дать собственное имя.

Резюме

В очередной раз мы стали свидетелями того, как C++ по сути создает новый язык из существующего. В C++ нет именованных аргументов, только позиционные. Это часть языкового ядра. И тем не менее нам удалось расширить язык и добавить поддержку именованных аргументов вполне разумным способом с применением техники сцепления методов. Мы исследовали и другие применения сцепления методов, помимо идиомы именованных аргументов.

В следующей главе рассматривается единственная в этой книге идиома, связанная с производительностью. Выше мы несколько раз обсуждали, во что обходится выделение памяти и как это влияет на реализацию некоторых паттернов. В идиоме оптимизации локального буфера проблема решается в лоб – путем отказа от выделения памяти вообще.

Вопросы

- Почему наличие функций с большим количеством аргументов одного или родственных типов приводит к хрупкости кода?
- Как агрегатные объекты в качестве аргументов повышают удобство сопровождения и надежность кода?
- Что такое идиома именованных аргументов и чем она отличается от агрегатного объекта-аргумента?
- В чем разница между сцеплением и каскадированием методов?

Глава 10

Оптимизация локального буфера

Не все паттерны проектирования связаны с проектированием иерархий классов. Паттерн проектирования ПО – это наиболее общее и допускающее повторное использование решение часто встречающейся проблемы, а в C++ одна из самых типичных проблем – недостаточная производительность. Одной из основных причин низкой производительности является неэффективное управление памятью. Были разработаны различные паттерны для решения подобных проблем. Ниже мы рассмотрим один из них, который призван справиться с накладными расходами из-за частого выделения небольших блоков памяти.

В этой главе рассматриваются следующие вопросы:

- какие накладные расходы сопровождают выделение небольших блоков памяти и как их можно измерить;
- что такое оптимизация локального буфера, как она повышает производительность и как можно измерить улучшение;
- когда использование оптимизации локального буфера дает эффект;
- каковы возможные недостатки и ограничения оптимизации локального буфера.

Технические требования

Потребуется установить и сконфигурировать библиотеку Google Benchmark. Подробности приведены по адресу **https://github.com/google/benchmark** (инструкции по установке см. в главе 5).

Примеры кода: **https://github.com/PacktPublishing/Hands-On-Design-Patterns-with-CPP/tree/master/Chapter10**.

Издержки выделения небольших блоков памяти

Оптимизация локального буфера – это именно оптимизация. Это паттерн, ориентированный на повышение производительности, и потому мы должны пом-

нить о первом правиле настройки производительности – никогда не гадать. Производительность и эффект любой оптимизации необходимо измерять.

Стоимость выделения памяти

Поскольку мы исследуем издержки выделения памяти и способы их снижения, то первым делом нужно задаться вопросом, насколько дорого обходится выделение памяти. Никто ведь не хочет оптимизировать то, что и так работает настолько быстро, что в оптимизации не нуждается. Для ответа на этот вопрос можно воспользоваться библиотекой Google Benchmark (или любой другой библиотекой эталонного микротестирования). Простейший тест для измерения стоимости выделения памяти мог бы выглядеть так:

```
void BM_malloc(benchmark::State& state) {
    for (auto _ : state) {
        void* p = malloc(64);
        benchmark::DoNotOptimize(p);
    }
    state.SetItemsProcessed(state.iterations());
}
BENCHMARK(BM_malloc_free);
```

Обертка `benchmark::DoNotOptimize` не дает компилятору убрать из кода неиспользуемую переменную в процессе оптимизации. Увы, этот эксперимент, скорее всего, закончится плохо; библиотеке нужно прогнать тест много раз, зачастую миллионы, чтобы вычислить среднее время с достаточной точностью. Вполне может статься, что память закончится, прежде чем тест завершится. Исправить эту проблему просто – нужно освобождать выделенную память:

```
void BM_malloc_free(benchmark::State& state) {
    const size_t S = state.range(0);
    for (auto _ : state) {
        void* p = malloc(S);
        benchmark::DoNotOptimize(p);
        free(p);
    }
    state.SetItemsProcessed(state.iterations());
}
BENCHMARK(BM_malloc_free)->Arg(64);
```

Отметим, что теперь мы измеряем стоимость выделения и освобождения, что и отражено в новом имени функции. Такое изменение не так уж неразумно – любую выделенную память рано или поздно нужно будет освободить, так что в какой-то момент эту цену все равно придется заплатить. Мы внесли в эталонный тест еще одно изменение – возможность задавать размер блока выделяемой памяти. После прогона этого теста появится примерно такая картина:

Мы видим, что выделение и освобождение 64 байтов памяти на этой машине занимает примерно 18 наносекунд, т. е. получается около миллиона операций выделения-освобождения в секунду. Если вам любопытно, нет ли чего-то особенного в числе 64, можете изменить размер в аргументе теста или прогнать тест для целого диапазона размеров:

```
void BM_malloc_free(benchmark::State& state) {
    const size_t S = state.range(0);
    for (auto _ : state) {
        void* p = malloc(S);
        benchmark::DoNotOptimize(p);
        free(p);
    }
    state.SetItemsProcessed(state.iterations());
}
BENCHMARK(BM_malloc_free)->RangeMultiplier(2)->Range(32, 256);
```

Вы, возможно, обратили внимание, что до сих пор мы измеряли время, потраченное на самое первое выделение памяти в программе, т. к. до этого ничего не выделяли. Исполняющая система C++, наверное, выделяла какую-то память на этапе запуска программы, но все равно этот эталонный тест не слишком реалистичен. Поведение теста можно сделать более похожим на настоящее, если добавить выделение памяти в начале:

```
#define REPEAT2(x) x x
#define REPEAT4(x) REPEAT2(x) REPEAT2(x)
#define REPEAT8(x) REPEAT4(x) REPEAT4(x)
#define REPEAT16(x) REPEAT8(x) REPEAT8(x)
#define REPEAT32(x) REPEAT16(x) REPEAT16(x)
#define REPEAT(x) REPEAT32(x)

void BM_malloc_free(benchmark::State& state) {
    const size_t S = state.range(0);
    const size_t N = state.range(1);
    std::vector<void*> v(N);
    for (size_t i = 0; i < N; ++i) v[i] = malloc(S);
    for (auto _ : state) {
        REPEAT({
            void* p = malloc(S);
            benchmark::DoNotOptimize(p);
            free(p);
        });
    }
    state.SetItemsProcessed(32*state.iterations());
    for (size_t i = 0; i < N; ++i) free(v[i]);
}
BENCHMARK(BM_malloc_free)->RangeMultiplier(2)
                         ->Ranges({{32, 256}, {1<<15, 1<<15}});
```

Здесь мы N раз вызываем malloc до начала эталонного теста. Можно также менять размеры выделяемых на этой стадии блоков. Кроме того, мы повторили тело цикла 32 раза (с помощью макроса препроцессора), чтобы уменьшить

влияние самого цикла на результат измерения. Теперь тест сообщает время, затраченное на 32 операции выделения и освобождения, что не очень удобно, однако скорость выделения по-прежнему правильна, поскольку мы учли раскрутку цикла и умножили количество итераций на 32, когда задавали количество обработанных элементов (в Google Benchmark элементом может быть что угодно, а в конце теста выводится количество элементов в секунду; мы приняли за элемент пару выделение–освобождение).

Даже после всех этих модификаций и улучшений конечный результат оказывается близок к первоначальному – 54 млн выделений в секунду. Вроде бы очень быстро, всего-то 18 наносекунд на одну операцию. Вспомним, однако, что современные процессоры могут выполнить за это время десятки команд. Когда мы выделяем память мелкими блоками, вполне может статься, что время, потраченное на обработку каждого выделения, тоже мало, а накладные расходы нетривиальны. Это, конечно, пример догадки, от чего я вас предостерегал, поэтому подтвердим высказанное утверждение прямыми экспериментами.

Но сначала я хочу обсудить другую причину крайней неэффективности выделения памяти небольшими блоками. До сих пор мы рассматривали стоимость выделения памяти в одном потоке. Однако сегодня почти все программы, для которых важна производительность, являются конкурентными, и C++ поддерживает конкурентность и многопоточность. Посмотрим, как изменится стоимость, когда память выделяется сразу в нескольких потоках:

```
void BM_malloc_free(benchmark::State& state) {
    const size_t S = state.range(0);
    const size_t N = state.range(1);
    std::vector<void*> v(N);
    for (size_t i = 0; i < N; ++i) v[i] = malloc(S);
    for (auto _ : state) {
        REPEAT({
            void* p = malloc(S);
            benchmark::DoNotOptimize(p);
            free(p);
        });
    }
    state.SetItemsProcessed(32*state.iterations());
    for (size_t i = 0; i < N; ++i) free(v[i]);
}
BENCHMARK(BM_malloc_free)->RangeMultiplier(2)
                         ->Ranges({{32, 256}, {1<<15, 1<<15}})
                         ->ThreadRange(1, 2);
```

Результат существенно зависит от оборудования и версии `malloc` в системе. Кроме того, на больших компьютерах, где много процессоров, количество потоков может быть гораздо больше двух. Тем не менее тенденция понятна:

```
-------------------------------------------------------------
Benchmark                   Time           CPU Iterations
-------------------------------------------------------------
BM_malloc_free/64        18 ns         18 ns    39155519   54.0953M items/s
```

Печальная картина – стоимость выделения возросла в несколько раз при переходе от одного потока к двум (на более мощной машине произойдет аналогичное увеличение, только потоков будет больше двух). Похоже, системный распределитель памяти плохо уживается с конкурентностью. Существуют более эффективные распределители, которыми можно заменить `malloc()`, подразумеваемый по умолчанию, но у них есть свои недостатки. К тому же было бы лучше, если бы производительность нашей программы на C++ не зависела от конкретной нестандартной системной библиотеки. Нам необходим улучшенный способ выделения памяти. Познакомимся с ним.

Введение в оптимизацию локального буфера

Если программе нужно решить некоторую задачу, выполнив минимальный объем работы, то лучше всего вообще ничего не делать. Халява – это замечательно. Стало быть, самый быстрый способ выделить и освободить память – не делать этого вовсе. Оптимизация локального буфера как раз и есть способ получить кое-что задаром – в данном случае получить память, не понеся дополнительных расходов.

Основная идея

Чтобы понять суть оптимизации локального буфера, следует помнить, что выделение памяти – не изолированная операция. Обычно, когда требуется небольшой блок памяти, эта память выделяется в составе какой-то структуры данных. Рассмотрим, к примеру, очень простую строку символов:

```
class simple_string {
    public:
    simple_string() : s_() {}
    explicit simple_string(const char* s) : s_(strdup(s)) {}
    simple_string(const simple_string& s) : s_(strdup(s.s_)) {}
    simple_string& operator=(const char* s) {
        free(s_); s_ = strdup(s); return *this;
    }
    simple_string& operator=(const simple_string& s)
    {
        free(s_); s_ = strdup(s.s_); return *this;
    }
    bool operator==(const simple_string& rhs)
    const {
        return strcmp(s_, rhs.s_) == 0;
    }
    ~simple_string() { free(s_); }
    private:
    char* s_;
};
```

Строка выделяет для себя память с помощью функции `malloc()`, вызываемой из `strdup()`, а возвращает ее, обращаясь к `free()`. Чтобы быть хоть для чего-то по-

лезным, класс строки должен содержать еще много функций-членов, но и этого достаточно, дабы исследовать издержки выделения памяти. Выделение имеет место при каждом конструировании, копировании или присваивании строк. Точнее, всякий раз, как конструируется строка, происходит дополнительное выделение памяти; сам объект строки нужно где-то разместить – в стеке, если это локальная переменная, или в куче, если строка является частью динамически выделенной структуры данных. И кроме того, нужно выделить память для данных строки, а эта память всегда выделяется с помощью `malloc()`.

И вот теперь идея оптимизации локального буфера – а почему бы не сделать объект строки побольше, чтобы в нем можно было хранить его собственные данные? Тогда мы действительно получим кое-что задаром; память для объекта строки все равно нужно выделять, но дополнительную память для данных строки мы получаем «на халяву». Конечно, строка может быть произвольно длинной, и заранее неизвестно, насколько большим нужно сделать объект строки, чтобы он вместил любую строку, встречающуюся в программе. Впрочем, даже если бы мы это знали, было бы безрассудным транжирством всегда выделять для объекта память такого большого размера, какой бы коротенькой ни была строка.

Однако можно сделать одно наблюдение: чем длиннее строка, тем дольше времени занимает ее обработка (копирование, поиск, преобразование и т. п.).

Для очень длинных строк стоимость выделения будет мала по сравнению со стоимостью обработки. Наоборот, для коротких строк стоимость выделения будет значительна. Поэтому наибольший выигрыш мы получим, если будем сохранять в самом объекте короткие строки, а для строк, не помещающихся в объекте, будем выделять память динамически, как и раньше. Это и есть суть оптимизации локального буфера, которая в контексте строк еще называется **оптимизацией короткой строки**: объект (строка) содержит локальный буфер определенного размера, и любая строка, помещающаяся в этот буфер, хранится в самом объекте:

```
class small_string {
    public:
    small_string() : s_() {}
    explicit small_string(const char* s)
        : s_((strlen(s) + 1 < sizeof(buf_)) ? buf_ : strdup(s))
    {
        if (s_ == buf_) strncpy(buf_, s, sizeof(buf_));
    }
    small_string(const small_string& s)
        : s_((s.s_ == s.buf_) ? buf_ : strdup(s.s_))
    {
        if (s_ == buf_) memcpy(buf_, s.buf_, sizeof(buf_));
    }
    small_string& operator=(const char* s) {
        if (s_ != buf_) free(s_);
        s_ = (strlen(s) + 1 < sizeof(buf_)) ? buf_ : strdup(s);
        if (s_ == buf_) strncpy(buf_, s, sizeof(buf_));
```

```
        return *this;
    }
    small_string& operator=(const small_string& s) {
        if (s_ != buf_) free(s_);
        s_ = (s.s_ == s.buf_) ? buf_ : strdup(s.s_);
        if (s_ == buf_) memcpy(buf_, s.buf_, sizeof(buf_));
        return *this;
    }
    bool operator==(const small_string& rhs) const {
        return strcmp(s_, rhs.s_) == 0;
    }
    ~small_string() {
        if (s_ != buf_) free(s_);
    }
    private:
    char* s_;
    char buf_[16];
};
```

Здесь размер буфера задан статически – 16 символов, включая завершающий строку ноль. Память для любой строки длиннее 16 выделяется с помощью `malloc()`. В процессе присваивания или уничтожения объекта строки мы должны проверить, использовался внутренний буфер или было выполнено выделение памяти из кучи, и соответствующим образом освободить память, занятую строкой.

Эффект оптимизации локального буфера

Насколько класс `small_string` быстрее, чем `simple_string`? Это, конечно, зависит от того, что вы собираетесь делать. Для начала просто попробуем создать и удалить строки. Чтобы не набирать один и тот же тест дважды, воспользуемся шаблонным тестом:

```
template <typename T>
void BM_string_create_short(benchmark::State& state) {
    const char* s = "Simple string";
    for (auto _ : state) {
        REPEAT({ T S(s); benchmark::DoNotOptimize(S); })
    }
    state.SetItemsProcessed(32*state.iterations());
}
BENCHMARK_TEMPLATE1(BM_string_create_short, simple_string);
BENCHMARK_TEMPLATE1(BM_string_create_short, small_string);
```

Результат впечатляет:

```
-------------------------------------------------------------------------
Benchmark                                  Time           CPU Iterations
-------------------------------------------------------------------------
BM_malloc_free/32/32768/threads:1        649 ns        648 ns    1044207   47.1165M items/s
BM_malloc_free/32/32768/threads:2       1077 ns       2153 ns     327732   14.1718M items/s
```

А если прогнать тот же тест с несколькими потоками, то получается еще лучше:

```
-------------------------------------------------------------------------------
Benchmark                                      Time           CPU Iterations
-------------------------------------------------------------------------------
BM_string_create_short<simple_string>/threads:2      1117 ns       2233 ns     312608   13.6656M items/s
BM_string_create_short<small_string>/threads:2         15 ns         31 ns   22646526    997.64M items/s
```

Если обычное создание строк при работе в несколько потоков оказывается гораздо медленнее, то класс `small_string` мало того что не показывает никакого замедления, но еще и масштабируется почти идеально. Конечно, это был самый благоприятный сценарий для оптимизации короткой строки – во-первых, мы ничего не делали, кроме создания и удаления строк, т. е. как раз тех операций, которые оптимизировали, а во-вторых, поскольку строка – локальная переменная, память для нее выделяется в кадре стека, т. е. никаких дополнительных затрат на выделение нет.

Однако эта ситуация вовсе не уникальная; в конце концов, локальные переменные не редкость, а если строка является частью более крупной структуры данных, то за выделение памяти для этой структуры все равно придется платить, так что выделение чего-то еще заодно и без дополнительных затрат, по существу, обходится бесплатно.

Тем не менее маловероятно, что мы выделяем строки только для того, чтобы сразу же освободить их, поэтому стоит рассмотреть и стоимость других операций. Можно ожидать, что при копировании и присваивании строк выигрыш будет аналогичен, если, конечно, строки остаются короткими:

```
template <typename T>
void BM_string_copy_short(benchmark::State& state) {
    const T s("Simple string");
    for (auto _ : state) {
        REPEAT({ T S(s); benchmark::DoNotOptimize(S); })
    }
    state.SetItemsProcessed(32*state.iterations());
}

template <typename T>
void BM_string_assign_short(benchmark::State& state) {
    const T s("Simple string");
    T S;
    for (auto _ : state) {
        REPEAT({ benchmark::DoNotOptimize(S = s); })
    }
    state.SetItemsProcessed(32*state.iterations());
}

BENCHMARK_TEMPLATE1(BM_string_copy_short, simple_string);
BENCHMARK_TEMPLATE1(BM_string_copy_short, small_string);
BENCHMARK_TEMPLATE1(BM_string_assign_short, simple_string);
BENCHMARK_TEMPLATE1(BM_string_assign_short, small_string);
```

И действительно, наблюдается похожее кардинальное улучшение:

```
-------------------------------------------------------------------------------
Benchmark                                      Time           CPU Iterations
-------------------------------------------------------------------------------
BM_string_copy_short<simple_string>          706 ns        706 ns     971142   43.2178M items/s
BM_string_copy_short<small_string>            45 ns         45 ns   15521788   672.256M items/s
BM_string_assign_short<simple_string>        754 ns        753 ns     924908   40.5313M items/s
BM_string_assign_short<small_string>          51 ns         51 ns   13281364   596.836M items/s
```

Вероятно, придется также хотя бы один раз прочитать данные строки – для сравнения, для поиска подстроки или символа или для вычисления какого-то значения, зависящего от строки. Для таких операций мы не ожидаем похожего масштабирования, поскольку ни с выделением, ни с освобождением памяти они не связаны. Тогда спрашивается, стоит ли ожидать вообще каких-то улучшений?

Действительно, простой тест сравнения строк не показывает никакой разницы между двумя версиями класса. Чтобы заметить какой-то выигрыш, нужно создать много объектов строк и сравнить их:

```
template <typename T>
void BM_string_compare_short(benchmark::State& state) {
    const size_t N = state.range(0);
    const T s("Simple string");
    std::vector<T> v1, v2;
    ... записать в векторы строки ...
    for (auto _ : state) {
        for (size_t i = 0; i < N; ++i) {
            benchmark::DoNotOptimize(v1[i] == v2[i]);
        }
    }
    state.SetItemsProcessed(N*state.iterations());
}
#define ARG Arg(1<<22)
BENCHMARK_TEMPLATE1(BM_string_compare_short, simple_string)->ARG;
BENCHMARK_TEMPLATE1(BM_string_compare_short, small_string)->ARG;
```

Для небольших значений N (когда строк мало) оптимизация не дает почти никакого выигрыша. Но если нужно обработать много строк, то сравнение с оптимизацией короткой строки может дать двукратный выигрыш:

```
-------------------------------------------------------------------------------------------------
Benchmark                                                  Time           CPU Iterations
-------------------------------------------------------------------------------------------------
BM_string_compare_short<simple_string>/4194304     36388678 ns   36385972 ns         19   109.932M items/s
BM_string_compare_short<small_string>/4194304      14640789 ns   14639702 ns         55    273.23M items/s
```

Почему так происходит, коль скоро память вообще не выделяется? Этот эксперимент демонстрирует второе, очень важное преимущество оптимизации локального буфера – улучшенную локальность кеша. Прежде чем можно будет прочитать данные строки, необходимо обратиться к самому объекту строки, поскольку он содержит указатель на данные. Для обычной строки доступ к ее символам требует двух обращений к памяти по разным, вообще говоря, несвязанным адресам. С другой стороны, в случае оптимизированной строки данные находятся рядом с объектом, поэтому если объект находится в кеше, то там же будут и данные. А много разных строк нам нужно потому, что если строк мало, то все объекты строк вместе со своими данными постоянно находятся в кеше. Только тогда, когда суммарный размер строк превосходит размер кеша, начнет проявляться выигрыш в производительности. Ну а теперь поглядим на некоторые дополнительные оптимизации.

Дополнительные оптимизации

В реализованном нами классе `simple_string` имеется очевидная неэффективность – когда строка хранится в локальном буфере, нам не нужен указатель на данные. Мы и так знаем, где находятся данные, – в локальном буфере. Нам, конечно, нужно каким-то образом узнавать, находятся ли данные в локальном буфере или в памяти, выделенной из кучи, но использовать для этого целых 8 байтов (на 64-разрядной машине) совершенно необязательно. Да, указатель необходим при хранении более длинных строк, но когда строка короткая, эту память можно было бы использовать для буфера:

```
class small_string {
    ...
    private:
    union {
        char* s_;
        struct {
            char buf[15];
            char tag;
        } b_;
    };
};
```

Здесь последний байт используется как признак (`tag`), показывающий, хранится строка локально (`tag == 0`) или вне объекта (`tag == 1`). Заметим, что общий размер буфера по-прежнему составляет 16 символов: 15 для самой строки и 1 для признака, который играет также роль завершающего нуля, если строка занимает все 16 байтов (именно поэтому мы выбрали `tag == 0` как признак локального хранения, иначе пришлось бы использовать дополнительный байт). Указатель совмещается в памяти с первыми 8 байтами буфера символов. В этом примере мы решили оптимизировать общий размер памяти, занятой строкой; под строку по-прежнему отведен 16-байтовый буфер, как и в предыдущей версии, но сам объект занимает только 6 байтов, а не 24. Если бы мы хотели сохранить размер объекта, то могли бы увеличить размер буфера и локально хранить более длинные строки. Но, вообще говоря, выигрыш от оптимизации короткой строки уменьшается по мере увеличения длины строки. Оптимальная длина зависит от конкретного приложения и, разумеется, должна определяться прямыми измерениями на тестах.

Оптимизация локального буфера в общем случае

Оптимизацию локального буфера можно эффективно использовать далеко не только для коротких строк. Всякий раз, как необходимо выделение небольших блоков памяти, размер которых определяется во время выполнения, следует подумать об этой оптимизации. В этом разделе мы рассмотрим несколько таких структур данных.

Короткий вектор

Еще одна распространенная структура данных, в которой зачастую выгодна оптимизация локального буфера, – вектор. По сути своей вектор – это динамический сплошной массив элементов данных указанного типа (в этом смысле строка является вектором байтов, хотя наличие завершающего нуля придает строке специфику). В простом векторе, каковым является, например, класс `std::vector` из стандартной библиотеки C++, должны быть два члена: указатель на данные и размер данных:

```
class simple_vector {
    public:
    simple_vector() : n_(), p_() {}
    simple_vector(std::initializer_list<int> il)
        : n_(il.size()), p_(static_cast<int*>(malloc(sizeof(int)*n_)))
    {
        int* p = p_;
        for (auto x : il) *p++ = x;
    }
    ~simple_vector()
    {
        free(p_);
    }
    size_t size() const { return n_; }
    private:
    size_t n_;
    int* p_;
};
```

Обычно векторы определяются как шаблоны классов, как, например, `std::vector`, но мы упростили этот пример, ограничившись вектором целых чисел (преобразование этого класса в шаблон оставляем читателю в качестве упражнения, на применение оптимизации локального буфера это не влияет). Мы можем применить *оптимизацию короткого вектора* и сохранить данные вектора в самом объекте вектора, при условии что вектор достаточно мал:

```
class small_vector {
    public:
    small_vector() : n_(), p_() {}
    small_vector(std::initializer_list<int> il)
        : n_(il.size()),
          p_((n_ < sizeof(buf_)/sizeof(buf_[0])) ?
                 buf_ :
                 static_cast<int*>(malloc(sizeof(int)*n_)))
    {
        int* p = p_;
        for (auto x : il) *p++ = x;
    }
    ~small_vector() {
        if (p_ != buf_) free(p_);
    }
```

```
    private:
    size_t n_;
    int* p_;
    int buf_[16];
};
```

Мы можем и дальше оптимизировать вектор так же, как строку, и совместить локальный буфер с указателем. Правда, мы не сможем использовать последний байт как признак, потому что любой элемент вектора может иметь произвольное значение, а значение 0 ничем особенным не отличается. Однако размер вектора все равно нужно хранить, чтобы в любой момент можно было определить, используется локальный буфер или нет. Мы можем извлечь дополнительное преимущество из того факта, что если оптимизация локального буфера используется, то размер вектора не может быть слишком большим, поэтому нам не нужно поле типа `size_t` для его хранения:

```
class small_vector {
    public:
    small_vector() { short_.n = 0; }
    small_vector(std::initializer_list<int> il) {
        int* p;
        if (il.size() < sizeof(short_.buf)/sizeof(short_.buf[0])) {
            short_.n = il.size();
            p = short_.buf;
        } else {
            short_.n = UCHAR_MAX;
            long_.n = il.size();
            p = long_.p = static_cast<int*>(malloc(sizeof(int)*long_.n));
        }
        for (auto x : il) *p++ = x;
    }
    ~small_vector() {
        if (short_.n == UCHAR_MAX) free(long_.p);
    }
  private:
    union {
        struct {
            size_t n;
            int* p;
        } long_;
        struct {
            int buf[15];
            unsigned char n;
        } short_;
    };
};
```

Здесь мы сохраняем размер вектора в поле типа `size_t long_.n` или `unsigned char short_.n` в зависимости от того, используется локальный буфер или нет. Признаком внешнего буфера является совпадение короткого размера со значением `UCHAR_MAX` (т. е. 255). Поскольку эта величина больше размера локального

буфера, то такой признак интерпретируется однозначно (если бы в локальном буфере можно было хранить более 255 элементов, то тип `short_.n` нужно было бы сделать пошире).

Для измерения выигрыша в производительности, который дает оптимизация короткого вектора, можно использовать такой же эталонный тест, как для строк. В зависимости от фактического размера вектора ожидаемый выигрыш при создании и копировании векторов может составить до 10 раз, а при работе в несколько потоков еще больше. Конечно, таким же образом можно оптимизировать и другие структуры данных, когда в них хранится небольшой объем динамически выделенных данных.

Объекты со стертым типом и вызываемые объекты

Есть еще один тип приложений, совершенно из другой оперы, для которых оптимизация локального буфера может оказаться очень эффективной, – хранение вызываемых объектов, т. е. объектов, вызываемых как функции. Многие шаблонные классы предлагают возможность настройки поведения с помощью задания вызываемого объекта. Например, стандартный класс разделяемого указателя в C++, `std::shared_ptr`, позволяет задать ликвидатор. При вызове ликвидатору передается адрес удаляемого объекта, т. е. один аргумент типа `void*`. Это может быть указатель на функцию или объект-функтор (объект, в котором определен оператор `operator()`) – в общем, любой тип, который можно вызвать, передав указатель `p`, иначе говоря, любой тип, для которого компилируется выражение вызова функции `callable(p)`. Однако ликвидатор – это не просто тип, это объект, который задается во время выполнения и потому должен где-то храниться, чтобы разделяемый указатель мог его получить. Если бы ликвидатор был частью типа разделяемого указателя, то мы могли бы просто объявить член такого типа в объекте разделяемого указателя (или в случае `std::shared_ptr` в его объекте для подсчета ссылок, общем для всех копий разделяемого указателя). Это можно считать тривиальным применением оптимизации локального буфера, как в следующем примере интеллектуального указателя, который автоматически удаляет объект при выходе указателя из области видимости (как `std::unique_ptr`):

```
template <typename T, typename Deleter>
class smartptr {
    public:
    smartptr(T* p, Deleter d) : p_(p), d_(d) {}
    ~smartptr() { d_(p_); }
    T* operator->() { return p_; }
    const T* operator->() const { return p_; }
    private:
    T* p_;
    Deleter d_;
};
```

Но мы нацелились на более интересные вещи, и одна из них – случай объектов со стертым типом. Подробности рассматривались в главе, посвященной

стиранию типа, но суть в том, что для них вызываемый объект не является частью самого типа (тип вызываемого объекта *стерт* в объемлющем объекте). Вызываемая сущность хранится в полиморфном объекте, а для вызова объекта нужного типа во время выполнения используется виртуальная функция. Работа с самим полиморфным объектом осуществляется через указатель на базовый класс.

Теперь мы имеем проблему, которая в каком-то смысле похожа на задачу о коротком векторе, – нам необходимо сохранить некоторые данные, в данном случае вызываемый объект, тип которых, а следовательно, и размер статически неизвестны. Общее решение – динамически выделять память для таких объектов и обращаться к ним через указатель на базовый класс. Для ликвидатора, связанного с интеллектуальным указателем, мы могли бы поступить так:

```
template <typename T>
class smartptr_te {
    struct deleter_base {
        virtual void apply(void*) = 0;
        virtual ~deleter_base() {}
    };
    template <typename Deleter> struct deleter : public deleter_base {
        deleter(Deleter d) : d_(d) {}
        virtual void apply(void* p) { d_(static_cast<T*>(p)); }
        Deleter d_;
    };
    public:
    template <typename Deleter> smartptr_te(T* p, Deleter d)
        : p_(p), d_(new deleter<Deleter>(d))
    {}
    ~smartptr_te() { d_->apply(p_); delete d_; }
    T* operator->() { return p_; }
    const T* operator->() const { return p_; }
    private:
    T* p_;
    deleter_base* d_;
};
```

Заметим, что тип `Deleter` больше не является частью типа интеллектуального указателя, он *стерт*. Все интеллектуальные указатели на объект типа `T` имеют один и тот же тип `smartptr_te<T>` (здесь `te` означает *type-erased* – со стертым типом). Однако за такое синтаксическое удобство мы должны заплатить немалую цену – при создании каждого интеллектуального указателя приходится выделять дополнительную память. И насколько высока эта цена? Снова вспомним главное правило настройки производительности: «немалая» – это всего лишь догадка, которую надо подтвердить экспериментально, например таким эталонным тестом:

```
struct deleter {     // очень простой ликвидатор, соответствует оператору new
    template <typename T> void operator()(T* p) { delete p; }
};
```

```
void BM_smartptr(benchmark::State& state) {
    deleter d;
    for (auto _ : state) {
        smartptr<int, deleter> p(new int, d);
    }
    state.SetItemsProcessed(state.iterations());
}

void BM_smartptr_te(benchmark::State& state) {
    deleter d;
    for (auto _ : state) {
        smartptr_te<int> p(new int, d);
    }
    state.SetItemsProcessed(state.iterations());
}
BENCHMARK(BM_smartptr);
BENCHMARK(BM_smartptr_te);
```

Для интеллектуального указателя со статически определенным ликвидатором можно ожидать, что стоимость каждой итерации будет очень близка к стоимости вызова `malloc()` и `free()`, которую мы измеряли раньше:

```
-------------------------------------------------------------
Benchmark                   Time           CPU Iterations
-------------------------------------------------------------
BM_smartptr                21 ns         21 ns   34069972   45.7846M items/s
BM_smartptr_te             42 ns         42 ns   16832659   22.7958M items/s
```

Для интеллектуального указателя со стертым типом память выделяется не один раз, а два, поэтому время создания объекта-указателя удваивается. Кстати говоря, мы можем также измерить производительность простого указателя, и она должна быть такой же, как для интеллектуального, в пределах точности измерений (это требование было положено в основу проектирования стандартного класса `std::unique_ptr`).

Мы можем применить идею оптимизации локального буфера и здесь, и, вероятно, она даст даже больший эффект, чем для строк, ведь вызываемые объекты, как правило, невелики. Но твердо рассчитывать на это нельзя, поэтому нужно обработать также случай, когда вызываемый объект не помещается в локальный буфер:

```
template <typename T>
class smartptr_te_lb {
    struct deleter_base {
        virtual void apply(void*) = 0;
        virtual ~deleter_base() {}
    };
    template <typename Deleter> struct deleter : public deleter_base {
        deleter(Deleter d) : d_(d) {}
        virtual void apply(void* p) { d_(static_cast<T*>(p)); }
        Deleter d_;
    };
    public:
```

```
    template <typename Deleter> smartptr_te_lb(T* p, Deleter d) :
        p_(p),
        d_((sizeof(Deleter) > sizeof(buf_)) ? new deleter<Deleter>(d) :
                                              new (buf_)
    deleter<Deleter>(d))
        {}
    ~smartptr_te_lb() {
        d_->apply(p_);
        if (static_cast<void*>(d_) == static_cast<void*>(buf_)) {
            d_->~deleter_base();
        } else {
            delete d_;
        }
    }
    T* operator->() { return p_; }
    const T* operator->() const { return p_; }
    private:
    T* p_;
    deleter_base* d_;
    char buf_[16];
};
```

С помощью такого же эталонного теста, как и раньше, мы можем измерить производительность интеллектуального указателя с оптимизацией локального буфера. Конструирование и удаление интеллектуального указателя без стирания типа занимает 21 нс, со стиранием типа – 42 нс, а оптимизированного разделяемого указателя со стертым типом – всего 23 нс на той же машине. Небольшие накладные расходы связаны с проверкой – хранится ликвидатор внутри или снаружи объекта.

Оптимизация локального буфера в библиотеке C++

Следует отметить, что последнее применение оптимизации локального буфера, хранение вызываемых объектов со стертым типом, широко используется в стандартной библиотеке шаблонов C++. Например, в шаблоне `std::shared_ptr` имеется ликвидатор со стертым типом, и в большинстве его реализаций используется оптимизация локального буфера; конечно, ликвидатор хранится в объекте, подсчитывающем ссылки, а не в каждой копии разделяемого указателя. С другой стороны, в стандартном шаблоне `std::unique_pointer` стирание типа не применяется вовсе, чтобы избежать даже малейших накладных расходов, которые могли бы стать гораздо больше, если бы ликвидатор не поместился в локальный буфер.

При реализации объекта из стандартной библиотеки C++ с *бескомпромиссным стиранием типа*, `std::function`, обычно применяется локальный буфер для хранения небольших вызываемых объектов без затрат на дополнительное выделение памяти. Универсальный контейнер для объектов любого типа `std::any` (начиная с C++17) обычно также реализуется без динамического выделения памяти там, где это возможно.

Недостатки оптимизации локального буфера

У оптимизации локального буфера есть и недостатки. Самый очевидный – что размер объектов с локальным буфером больше, чем необходимо. Если типичные данные, хранящиеся в буфере, меньше выбранного размера буфера, то в каждом объекте часть памяти расходуется впустую, но это хотя бы окупается оптимизацией. Хуже, если мы сильно промахнулись с выбором размера буфера, так что данные чаще всего не помещаются в буфер и хранятся вне объекта. Тем не менее буфер в каждом объекте присутствует, и вся эта память потрачена зря. Понятно, что необходимо выбрать компромисс между объемом памяти, который мы готовы потратить впустую, и диапазоном размеров буфера, при которых оптимизация эффективна. Размер локального буфера следует тщательно подбирать для конкретного приложения.

Есть и более тонкое осложнение – данные, которые когда-то хранились вне объекта, теперь переместились внутрь. У этого действия есть несколько последствий, помимо выигрыша в производительности. Во-первых, каждая копия объекта содержит собственную копию данных, коль скоро она помещается в локальном буфере. Это исключает любые реализации с подсчетом ссылок на данные. Например, в реализации строк с **копированием при записи** (Copy-On-Write – COW), когда данные не копируются, до тех пор пока копии строк не различаются, использовать оптимизацию коротких строк невозможно.

Во-вторых, данные должны быть перемещены, если перемещается сам объект. Сравните с `std::vector`, который перемещается или обменивается по существу так же, как указатель, – указатель на данные перемещается, но сами данные остаются на месте. Нечто похожее справедливо для объекта, хранящегося в `std::any`. Можно отбросить это соображение как тривиальное; в конце концов, оптимизация локального буфера применяется в основном для данных небольшого размера, а стоимость их перемещения должна быть сравнима со стоимостью перемещения указателя. Но дело здесь не только в производительности – перемещение экземпляра `std::vector` (или `std::any`, если на то пошло) гарантированно не возбуждает исключений. Однако такие гарантии невозможно дать при перемещении произвольного объекта. Поэтому при реализации `std::any` можно применить оптимизацию локального буфера, только если содержащийся в нем объект обладает характеристикой `std::is_nothrow_move_constructible`. Но даже такой гарантии недостаточно для `std::vector`; в стандарте явно говорится, что перемещение, или обмен вектора, не должно делать недействительными итераторы, указывающие на любой его элемент. Очевидно, что это требование несовместимо с оптимизацией локального буфера, поскольку перемещение короткого вектора приведет к перемещению всех его элементов в другую область памяти. По этой причине многие высокоэффективные библиотеки предлагают специальный похожий на вектор контейнер, который поддерживает оптимизацию коротких векторов ценой отказа от гарантий «нерушимости» итераторов.

Резюме

Мы только что познакомились с паттерном проектирования, нацеленным исключительно на повышение производительности. Эффективность – важное соображение в языке C++, поэтому сообщество C++ разработало паттерны, призванные устранить наиболее распространенные причины неэффективности. Пожалуй, одной из самых типичных является многократное или расточительное выделение памяти. Рассмотренный выше паттерн – оптимизация локального буфера – действенное средство для сокращения количества таких выделений. Мы видели, что он применяется, чтобы сделать структуры данных компактными, а также для хранения небольших объектов, например вызываемых сущностей. Мы также обсудили недостатки этого паттерна.

В следующей главе «Охрана области видимости» мы перейдем к изучению паттернов посложнее, относящихся к более пространным вопросам проектирования. Рассмотренные выше идиомы часто применяются в реализации этих паттернов.

Вопросы

- Как измерить производительность небольшого фрагмента кода?
- Почему частое выделение небольших блоков памяти особенно вредит производительности?
- Что такое оптимизация локального буфера и как она работает?
- Почему выделение памяти для дополнительного буфера внутри объекта обходится по существу *бесплатно*?
- Что такое оптимизация короткой строки?
- Что такое оптимизация короткого вектора?
- Почему оптимизация локального буфера особенно эффективна для вызываемых объектов?
- Какие компромиссы необходимо рассмотреть при использовании оптимизации локального буфера?
- Когда объект не следует помещать в локальный буфер?

Для дальнейшего чтения

- Viktor Sehr and Björn Andrist. *C++ High Performance*: https://www.packtpub.com/application-development/c-high-performance.
- Wisnu Anggoro. *C++ Data Structures and Algorithms*: https://www.packtpub.com/application-development/c-data-structures-and-algorithms.
- Jeganathan Swaminathan. *High Performance Applications with C++* [видео]: https://www.packtpub.com/application-development/highperformance-applications-c-video.

Глава 11

Охрана области видимости

В этой главе описывается паттерн, который можно рассматривать как обобщение изученной ранее идиомы RAII. В своей ранней форме это старый и давно «устаканившийся» паттерн C++, однако он существенно выиграл от новых возможностей, добавленных в язык в стандартах C++11, C++14 и C++17. Мы будем свидетелями эволюции этого паттерна по мере того, как язык становился все более мощным. Паттерн ScopeGuard расположен на пересечении декларативного программирования (говори, что должно получиться, а не как это сделать) и программирования, защищенного от ошибок (особенно в части безопасности относительно исключений). Нам придется немного поговорить на эти темы, чтобы в полной мере понять и оценить паттерн ScopeGuard.

В этой главе рассматриваются следующие вопросы:

- как написать код, безопасный относительно ошибок и исключений;
- как RAII упрощает обработку ошибок;
- что такое компонуемость в применении к обработке ошибок;
- почему идиома RAII – недостаточно действенное средство обработки ошибок и как ее обобщить;
- как реализовать декларативную обработку ошибок в C++.

Технические требования

Примеры кода: **https://github.com/PacktPublishing/Hands-On-Design-Patterns-with-CPP/tree/master/Chapter11**.

Потребуется установить и сконфигурировать библиотеку Google Benchmark. Подробности приведены по адресу **https://github.com/google/benchmark** (инструкции по установке см. в главе 5).

В этой главе активно используются передовые средства C++, так что держите под рукой справочное руководство по C++ (**https://en.cppreference.com**, если не хотите копаться в самом стандарте).

Наконец, очень полную и скрупулезную реализацию паттерна ScopeGuard можно найти в библиотеке Folly по адресу **https://github.com/facebook/folly/**

blob/master/folly/ScopeGuard.h; она включает такие детали программирования библиотек на C++, которые выходят за рамки этой книги.

Обработка ошибок и идиома RAII

Начнем с обзора принципов обработки ошибок и, в частности, написания безопасного относительно исключений кода на C++. Идиома **захват ресурса есть инициализация** (Resource Acquisition Is Initialization – **RAII**) – один из основных методов обработки ошибок в C++. Мы уже посвятили ей целую главу, и теперь эти знания понадобятся, чтобы понять то, что мы собираемся сделать. Сначала разберемся, какая проблема перед нами стоит.

Безопасность относительно ошибок и исключений

В оставшейся части этой главы мы будем рассматривать следующую проблему. Пусть требуется реализовать базу данных, состоящую из записей. Записи хранятся на диске, но имеется также размещенный в памяти индекс для быстрого доступа к записям. API базы данных предоставляет метод для вставки записей в базу:

```
class Record { ... };
class Database {
    public:
    void insert(const Record& r);
    ...
};
```

Если вставка завершается успешно, то индекс и хранилище на диске обновляются и остаются согласованными между собой. В противном случае возбуждается исключение.

Хотя клиентам базы данных кажется, что вставка – это одна транзакция, в действительности эта операция состоит из нескольких шагов: нужно вставить запись в индекс и записать ее на диск. Поэтому реализация содержит два класса, каждый из которых отвечает за один вид хранилища:

```
class Database {
    class Storage { ... };   // хранилище на диске
    Storage S;
    class Index { ... };     // индекс в памяти
    Index I;
    public:
    void insert(const Record& r);
    ...
};
```

Реализация функции `insert()` должна вставить запись и в хранилище, и в индекс, никаких вариантов тут быть не может:

```
void Database::insert(const Record& r) {
    S.insert(r);
```

```
    I.insert(r);
}
```

К несчастью, любая из этих операций может завершиться неудачно. Сначала посмотрим, что произойдет, если не получается вставить запись в хранилище. Предположим, что обо всех ошибках программа сигнализирует с помощью исключений. Если вставка в хранилище не проходит, то хранилище остается в прежнем состоянии, программа даже не пытается вставлять в индекс, и исключение распространяется наружу из функции `Database::insert()`. Это именно то, чего мы хотим – вставка не прошла, база данных не изменилась, и возбуждено исключение.

А что будет, если вставка в хранилище завершилась успешно, а вставка в индекс – нет? Теперь ситуация выглядит не так радужно – данные на диске изменены, затем вставка в индекс не прошла, вызывающая сторона получила от `Database::insert()` сигнал об ошибке вставки, но дело-то в том, что вставка не то чтобы совсем не прошла. Но и полностью успешной она тоже не была. База данных осталась в несогласованном состоянии, на диске имеется запись, недоступная по индексу. Это неправильная обработка ошибки, небезопасный относительно исключений код – такого быть не должно.

Импульсивная попытка поменять местами шаги операции не поможет:

```
void Database::insert(const Record& r) {
    I.insert(r);
    S.insert(r);
}
```

Конечно, теперь при неудачной попытке вставить в индекс все будет хорошо. Но точно такая же проблема возникает, если вставка в хранилище возбуждает исключение – теперь у нас имеется запись в индексе, указывающая в никуда, поскольку на диск ничего записано не было.

Очевидно, что мы не можем просто игнорировать исключения, возбужденные объектом `Index` или `Storage`, мы должны как-то обработать их, чтобы не нарушить согласованность базы данных. Как обрабатывать исключения, мы знаем, для того и существует блок try-catch:

```
void Database::insert(const Record& r) {
    S.insert(r);
    try {
        I.insert(r);
    } catch (...) {
        S.undo();
        throw;        // возбудить повторно
    }
}
```

И на этот раз, если не получается записать в хранилище, ничего специально делать не надо. Если же не получается вставить в индекс, то нужно отменить последнюю операцию с хранилищем (предположим, что в API есть средства

для этого). Теперь база данных снова согласована, как будто вставка никогда и не происходила. Но пусть даже исключение, возбужденное индексом, перехвачено, мы все равно должны сигнализировать об ошибке вызывающей программе, поэтому возбуждаем исключение повторно. Пока все хорошо.

Ситуация не слишком меняется, если вместо исключений использовать коды ошибок. Рассмотрим вариант, когда все функции `insert()` возвращают `true` в случае успеха и `false` в случае неудачи:

```
bool Database::insert(const Record& r) {
    if (!S.insert(r)) return false;
    if (!I.insert(r)) {
        S.undo();
        return false;
    }
    return true;
}
```

Мы должны проверять значение, возвращенное каждой функцией, отменить последнее действие, если вторая функция завершилась неудачно, и возвратить `true`, только если оба действия прошли успешно.

Пока все нормально. Мы смогли довести до ума простейшую задачу с двумя шагами, сделав код безопасным относительно ошибок. Настало время повысить сложность. Предположим, что в конце транзакции хранилище нужно почистить, например вставленная запись не переходит в окончательное состояние, пока не будет вызван метод `Storage::finalize()` (быть может, так делается для того, чтобы мог работать метод `Storage::undo()`, а после «финализации» вставки ее уже нельзя отменить). Отметим разницу между функциями `undo()` и `finalize()`; первая должна вызываться, только если мы хотим откатить транзакцию, а вторая – если вставка в хранилище завершилась успешно, вне зависимости от того, что произойдет потом.

Нашим требованиям отвечает такой поток управления:

```
void Database::insert(const Record& r) {
    S.insert(r);
    try {
        I.insert(r);
    } catch (...) {
        S.undo();
        S.finalize();
        throw;
    }
    S.finalize();
}
```

И что-то подобное можно сделать в случае возврата кодов ошибок (далее в этой главе мы во всех примерах будем работать с исключениями, но перейти к кодам ошибок нетрудно).

Код уже принимает уродливые формы, особенно в той части, которая призвана обеспечить вызов кода очистки (в нашем случае `S.finalize()`) на каждом

пути выполнения. И будет только хуже по мере усложнения последовательности действий, которые должны быть все отменены, если какой-то шаг операции завершился неудачно. Вот как выглядит поток управления для трех действий, у каждого из которых есть свой откат и своя очистка:

```
if (action1() == SUCCESS) {
    if (action2() == SUCCESS) {
        if (action3() == FAIL) {
            rollback2();
            rollback1();
        }
        cleanup2();
    } else {
          rollback1();
    }
    cleanup1();
}
```

Очевидная проблема – необходимость явной проверки успешности завершения, будь то блок try-catch или проверка кода ошибки. А более серьезная заключается в том, что этот способ обработки ошибок не допускает композиции. Решение для N + 1 действий не сводится к коду для N действий с небольшими добавлениями – вместо этого приходится глубоко залезать в код и добавлять нужные куски внутрь. Но ведь мы уже видели идиому C++ для решения именно этой проблемы.

Захват ресурса есть инициализация

Идиома RAII связывает ресурсы с объектами. В момент захвата ресурса конструируется объект, а когда этот объект уничтожается, ресурс освобождается. Нас сейчас интересует только вторая половина – уничтожение. Ценность идиомы RAII проистекает из того факта, что деструкторы всех локальных объектов должны быть вызваны до того, как управление достигнет конца области видимости – независимо от того, как это произойдет (`return`, `throw`, `break` и т. д.). Мы воюем с очисткой, так поручим ее RAII-объекту:

```
class StorageFinalizer {
    public:
    StorageFinalizer(Storage& S) : S_(S) {}
    ~StorageFinalizer() { S_.finalize(); }
    private:
    Storage& S_;
};
void Database::insert(const Record& r) {
    S.insert(r);
    StorageFinalizer SF(S);
    try {
        I.insert(r);
    } catch (...) {
        S.undo();
```

```
        throw;
    }
}
```

Объект `StorageFinalizer` связывается с объектом `Storage` в момент конструирования и вызывает `finalize()`, когда уничтожается. Поскольку никаким способом нельзя покинуть область видимости, в которой определен объект `StorageFinalizer`, не вызвав его деструктор, нам не нужно беспокоиться о потоке управления, по крайней мере для очистки, – она произойдет в любом случае. Заметим, что `StorageFinalizer` конструируется после успешной вставки в хранилище, как и должно быть; ведь если первая вставка завершилась неудачно, то и финализировать нечего.

Этот код работает, но выглядит каким-то половинчатым. В конце функции должно быть выполнено два действия, но первое (очистка, или `finalize()`) автоматизировано, а второе (откат, или `undo()`) – нет. К тому же эта техника попрежнему не допускает композиции, вот как выглядит поток управления для трех действий:

```
class Cleanup1() {
    ~Cleanup1() { cleanup1(); }
    ...
};
class Cleanup2() {
    ~Cleanup2() { cleanup2(); }
    ...
};

action1();
Cleanup1 c1;
try {
    action2();
    Cleanup2 c2;
    try {
        action3();
    } catch (...) {
        rollback2();
        throw;
    }
} catch (...) {
    rollback1();
}
```

Снова, чтобы добавить еще одно действие, мы должны вставить новый блок try-catch глубоко внутри кода. С другой стороны, часть, связанная с очисткой, компонуется идеально. Вот как выглядел бы предыдущий код, если бы не надо было делать откат:

```
action1();
Cleanup1 c1;
action2();
Cleanup2 c2;
```

Если понадобится еще одно действие, мы просто добавим две строки в конец функции, и очистка произойдет в правильном порядке. Если бы мы могли сделать то же самое для отката, то все было бы отлично.

Мы не можем просто переместить вызов undo() в деструктор другого объекта; деструкторы вызываются всегда, а откат должен происходить только в случае ошибки. Но мы можем сделать так, чтобы деструктор производил откат условно:

```
class StorageGuard {
    public:
    StorageGuard(Storage& S) : S_(S), commit_(false) {}
    ~StorageGuard() { if (!commit_) S_.undo(); }
    void commit() noexcept { commit_ = true; }
    private:
    Storage& S_;
    bool commit_;
};
void Database::insert(const Record& r) {
    S.insert(r);
    StorageFinalizer SF(S);
    StorageGuard SG(S);
    I.insert(r);
    SG.commit();
}
```

Рассмотрим этот код внимательно. Если вставка в хранилище не проходит, то возбуждается исключение, и база данных остается неизменной. Если проходит, то конструируются два RAII-объекта. Первый будет безусловно вызывать S.finalize() в конце области видимости. Второй будет вызывать S.undo(), если мы не зафиксировали первое изменение, вызвав метод commit() объекта StorageGuard. Это произойдет, если только вставка в индекс не завершится неудачно, а в таком случае будет возбуждено исключение, оставшийся в области видимости код будет пропущен, и управление передано сразу в конец области видимости (туда, где стоит закрывающая скобка }), где и будут вызваны деструкторы всех локальных объектов. Поскольку при таком развитии событий commit() не вызывается, то StorageGuard все еще активен и, стало быть, откатит вставку.

Деструкторы локальных объектов вызываются в порядке, противоположном порядку их конструирования. Это важно; откатить вставку можно только до финализации действия, поэтому откат должен предшествовать очистке. Это значит, что RAII-объекты конструируются в правильном порядке – сначала очистка (которая должна быть выполнена последней), а затем охранник отката (он должен быть выполнен первым, если возникнет необходимость).

Теперь код выглядит очень симпатично, в нем вообще нет блоков try-catch. В каком-то смысле он даже выглядит не как обычный код на C++. Такой стиль программирования называется **декларативным**; в этой парадигме логика программы выражается без явной спецификации потока управления (проти-

воположный стиль, более распространенный в C++, называется **императивным** – программа описывает, какие шаги выполнить и в каком порядке, но необязательно указывает причины). Существуют языки декларативного программирования (очевидный пример – SQL), но C++ не из них. Тем не менее C++ очень хорош для реализации конструкций, которые позволяют надстроить над ним языки более высокого уровня. Вот и мы реализовали язык декларативной обработки ошибок. Теперь наша программа говорит, что после вставки записи в хранилище осталось выполнить две операции: очистку и откат. Откат отменяется, если функция в целом завершилась успешно. Код выглядит линейным, без явного потока управления, иными словами – декларативным.

Но, несмотря на свою симпатичность, этот код далек от совершенства. Очевидная проблема – необходимость писать класс охранника или финализатора для каждого действия программы. Менее очевидная – то, что правильно написать такие классы нелегко, и пока что гордиться своей работой нам рано. Сделайте паузу и подумайте, чего не хватает, прежде чем переходить к исправленной версии, показанной ниже.

```
class StorageGuard {
    public:
    StorageGuard(Storage& S) : S_(S), commit_(false) {}
    ~StorageGuard() { if (!commit_) S_.undo(); }
    void commit() noexcept { commit_ = true; }
    private:
    Storage& S_;
    bool commit_;
    // Важно: при копировании этого охранника произойдет катастрофа!
    StorageGuard(const StorageGuard&) = delete;
    StorageGuard& operator=(const StorageGuard&) = delete;
};
void Database::insert(const Record& r) {
    S.insert(r);
    StorageFinalizer SF(S);
    StorageGuard SG(S);
    I.insert(r);
    SG.commit();
}
```

Что нам нужно, так это общий каркас, который позволит планировать выполнение произвольного действия в конце области видимости, условно или безусловно. В следующем разделе представлен паттерн ScopeGuard, обеспечивающий как раз такой каркас.

Паттерн ScopeGuard

В этом разделе мы научимся писать RAII-классы для действий при выходе, похожие на то, что разработали в предыдущем разделе, но без трафаретного кода. Это можно сделать на C++03, но гораздо лучше – с помощью средств, добавленных в C++14 и особенно в C++17.

Основы ScopeGuard

Начнем с более трудной задачи – как реализовать обобщенный класс отката, т. е. обобщенный вариант класса StorageGuard из предыдущего раздела. Единственная разница между ним и классом очистки заключается в том, что очистка должна выполняться всегда, а откат отменяется после фиксации действия. Имея версию с условным откатом, мы всегда можем убрать из нее проверку условия и получить версию для очистки, так что на время забудем о ней.

В нашем примере откат – это обращение к методу S.undo(). Но для простоты начнем с отката, который вызывает свободную функцию, а не функцию-член:

```
void undo(Storage& S) { S.undo(); }
```

Когда мы доведем реализацию до конца, программа будет выглядеть как-то так:

```
{
    S.insert(r);
    ScopeGuard SG(undo, S); // пример желаемого синтаксиса (приблизительный)
    ...
    SG.commit();            // деактивировать охрану области видимости
}
```

Этот код говорит (декларативно!), что если вставка завершилась успешно, то нужно запланировать откат по выходе из области видимости. В процессе отката вызывается функция undo() с аргументом S, которая, в свою очередь, отменит вставку. Если мы дойдем до конца функции, то деактивируем охранника и подавим вызов отката, в результате чего вставка зафиксируется и станет постоянной.

Гораздо более общее и допускающее повторное использование решение было предложено Андреем Александреску в статье, опубликованной в журнале *Dr. Dobbs* в 2000 году (http://www.drdobbs.com/cpp/generic-change-the-way-youwrite-excepti/184403758?). Рассмотрим и проанализируем реализацию.

```
class ScopeGuardImplBase {
    public:
    ScopeGuardImplBase() : commit_(false) {}
    void commit() const throw() { commit_ = true; }

    protected:
    ScopeGuardImplBase(const ScopeGuardImplBase& other)
        : commit_(other.commit_) { other.commit(); }
    ~ScopeGuardImplBase() {}
    mutable bool commit_;

    private:
    ScopeGuardImplBase& operator=(const ScopeGuardImplBase&);
};

template <typename Func, typename Arg>
class ScopeGuardImpl : public ScopeGuardImplBase {
    public:
```

```
    ScopeGuardImpl(const Func& func, Arg& arg) : func_(func), arg_(arg) {}
    ~ScopeGuardImpl() { if (!commit_) func_(arg_); }
    private:
    const Func& func_;
    Arg& arg_;
};

template <typename Func, typename Arg>
ScopeGuardImpl<Func, Arg> MakeGuard(const Func& func, Arg& arg) {
    return ScopeGuardImpl<Func, Arg>(func, arg);
}
```

Вначале определен базовый класс для всех охранников области видимости, `ScopeGuardImplBase`. В базовом классе имеется флаг фиксации `commit_` и код для работы с ним. Конструктор создает охранника в активированном состоянии, так чтобы отложенное действие произошло в деструкторе. Вызов `commit()` предотвратит такое развитие события – деструктор не будет делать ничего. Наконец, существует копирующий конструктор, который создает нового охранника в таком же состоянии, как исходный, но затем деактивирует исходного охранника. Это делается для того, чтобы откат не был вызван дважды, из деструкторов обоих объектов. Этот объект допускает копирование, но не допускает присваивания. Здесь мы пользовались средствами из стандарта C++03 для всего, включая запрет оператора присваивания. По существу, эта реализация основана на C++03, а немногие заимствования из C++11 – не более чем вишенка на торте (в следующем разделе это изменится).

В реализации `ScopeGuardImplBase` есть несколько деталей, которые могут показаться странными и требующими разъяснений. Во-первых, деструктор не виртуальный, но это не ошибка и не опечатка, так и задумано, а почему, мы увидим ниже. Во-вторых, флаг `commit_` объявлен как `mutable`. Это, конечно, сделано для того, чтобы его можно было изменить методом `commit()`, объявленным с квалификатором `const`. Но зачем объявлять `commit()` как `const`? Первая причина заключается в том, чтобы можно было вызвать его из копирующего конструктора исходного объекта с целью передать ответственность за откат от другого объекта этому. В этом смысле копирующий конструктор выполняет перемещение и будет официально объявлен таковым позже. Вторая причина добавления `const` станет ясна позднее (кстати, она связана с тем, что деструктор не виртуальный).

Теперь обратимся к производному классу, `ScopeGuardImpl`. Это шаблон с двумя параметрами-типами: первый – тип функции или любого другого вызываемого объекта, который мы будем вызывать для выполнения отката, второй – тип его аргумента. Пока что у нашей функции отката может быть всего один аргумент. Она будет вызвана в деструкторе объекта `ScopeGuard`, если только охранник не был деактивирован обращением к `commit()`.

Наконец, имеется шаблон фабричной функции, `MakeGuard`. В C++ это очень распространенная идиома: если требуется создать экземпляр шаблона класса по аргументам конструктора, воспользуйтесь шаблонной функцией, которая

может вывести типы параметров и возвращаемого значения из аргументов (в C++17 это могут делать и шаблоны классов, как мы увидим ниже).

Как все это используется для создания объекта-охранника, который вызовет за нас функцию undo(S)? Вот так:

```
void Database::insert(const Record& r) {
    S.insert(r);
    const ScopeGuardImplBase& SG = MakeGuard(undo, S);
    I.insert(r);
    SG.commit();
}
```

Функция MakeGuard выводит типы функции undo() и аргумента S и возвращает объект ScopeGuard соответствующего типа. Возврат осуществляется по значению, так что производится копирование (компилятор может устранить копирование в процессе оптимизации, но не обязан это делать). Возвращенный объект – безымянная временная переменная, которая связывает его со ссылкой на базовый класс SG (приведение производного класса к базовому осуществляется неявно для указателей и ссылок). Всем известно, что время жизни временной переменной простирается до точки с запятой в конце предложения, в котором переменная была создана. Но тогда на что указывает ссылка SG после конца этого предложения? Ее необходимо связать с чем-то, поскольку несвязанных ссылок быть не может, это не нулевые указатели. Так вот, то, что *всем известно*, – неправильно, а точнее почти правильно – обычно временные переменные действительно существуют лишь до конца предложения. Однако связывание временной переменной с константной ссылкой продлевает ее жизнь, так что она живет столько же, сколько сама ссылка. Иными словами, безымянный временный объект ScopeGuard, созданный функцией MakeGuard, не будет уничтожен, пока ссылка SG не покинет область видимости. Константность здесь важна, но не волнуйтесь – забыть о ней не получится, язык не позволяет связывать неконстантную ссылку со временной переменной, так что компилятор поставит вас на место. Это объясняет странность метода commit(): он должен быть объявлен как const, потому что мы собираемся вызывать его от имени константной ссылки (и, следовательно, флаг commit_ должен быть объявлен как mutable).

Но как насчет деструктора? В конце области видимости будет вызван деструктор класса ScopeGuardImplBase, поскольку это тип ссылки, покидающей область видимости. Деструктор базового класса не делает ничего; то, что нам нужно, делает деструктор производного класса. Полиморфный класс с виртуальным деструктором сослужил бы нам службу, но мы этим путем не пошли. Вместо этого мы воспользовались еще одним специальным правилом из стандарта C++, касающимся константных ссылок и временных переменных, – мало того что жизнь временной переменной продлевается, так еще и в конце области вызывается деструктор производного класса, т. е. фактически сконструированного класса. Отметим, что это правило относится только к деструкторам, вызывать методы производного класса от имени ссылки SG на базовый класс

по-прежнему нельзя. Кроме того, расширение времени жизни касается только временной переменной, связанной непосредственно с константной ссылкой. Это не будет работать, например, если инициализировать еще одну константную ссылку первой ссылкой. Вот почему мы должны были вернуть объект ScopeGuard из функции MakeGuard по значению; если бы мы попробовали вернуть его по ссылке, то временная переменная оказалась бы связана с этой ссылкой, которая в конце предложения исчезла бы. Вторая ссылка SG, инициализированная первой, не продлила бы время жизни объекта.

Показанная только что реализация функции очень близка к намеченной цели, только чуть более многословна, чем хотелось бы (и еще в ней упоминается класс ScopeGuardImplBase вместо обещанного ScopeGuard). Не пугайтесь, последний шаг – всего лишь синтаксический сахар:

```
typedef const ScopeGuardImplBase& ScopeGuard;
```

Теперь мы можем написать:

```
void Database::insert(const Record& r) {
    S.insert(r);
    ScopeGuard SG = MakeGuard(undo, S);
    I.insert(r);
    SG.commit();
}
```

Это все, чего можно достичь теми языковыми средствами, которыми мы до сих пор пользовались. В идеале хотелось бы прийти к такому синтаксису (и мы не так уж далеки от него):

```
ScopeGuard SG(undo, S);
```

Наш класс ScopeGuard можно немного подрихтовать, воспользовавшись возможностями C++11. Во-первых, можно запретить оператор присваивания, как подобает. Во-вторых, можно уже не притворяться, что наш копирующий конструктор чем-то отличается от перемещающего. И наконец, чтобы объявить, что функция не возбуждает исключений, мы воспользуемся квалификатором noexcept вместо throw():

```
class ScopeGuardImplBase {
    public:
    ScopeGuardImplBase() : commit_(false) {}
    void commit() const noexcept { commit_ = true; }

    protected:
    ScopeGuardImplBase(ScopeGuardImplBase&& other)
        : commit_(other.commit_) { other.commit(); }
    ~ScopeGuardImplBase() {}
    mutable bool commit_;

    private:
    ScopeGuardImplBase& operator=(const ScopeGuardImplBase&) = delete;
};
typedef const ScopeGuardImplBase& ScopeGuard;
```

```
template <typename Func, typename Arg>
class ScopeGuardImpl : public ScopeGuardImplBase {
    public:
    ScopeGuardImpl(const Func& func, Arg& arg)
        : func_(func), arg_(arg) {}
    ~ScopeGuardImpl() { if (!commit_) func_(arg_); }
    ScopeGuardImpl(ScopeGuardImpl&& other)
        : ScopeGuardImplBase(std::move(other)),
          func_(other.func_),
          arg_(other.arg_) {}
    private:
    const Func& func_;
    Arg& arg_;
};

template <typename Func, typename Arg>
ScopeGuardImpl<Func, Arg> MakeGuard(const Func& func, Arg& arg) {
    return ScopeGuardImpl<Func, Arg>(func, arg);
}
```

Перейдя к C++14, мы можем внести еще одно упрощение – вывести тип значения, возвращаемого функцией `MakeGuard`:

```
template <typename Func, typename Arg>
    auto MakeGuard(const Func& func, Arg& arg) {
    return ScopeGuardImpl<Func, Arg>(func, arg);
}
```

Но все еще остается одна уступка – мы хотели получить функцию `S.undo()`, а не `undo(S)`. Это легко сделать с помощью варианта `ScopeGuard`, являющегося функцией-членом. Единственная причина, по которой мы не поступили так с самого начала, – желание упростить пример, поскольку синтаксис указателя на функцию-член – не самый очевидный аспект C++:

```
template <typename MemFunc, typename Obj>
class ScopeGuardImpl : public ScopeGuardImplBase {
    public:
    ScopeGuardImpl(const MemFunc& memfunc, Obj& obj)
        : memfunc_(memfunc), obj_(obj) {}
    ~ScopeGuardImpl() { if (!commit_) (obj_.*memfunc_)(); }
    ScopeGuardImpl(ScopeGuardImpl&& other)
        : ScopeGuardImplBase(std::move(other)),
          memfunc_(other.memfunc_),
          obj_(other.obj_) {}
    private:
    const MemFunc& memfunc_;
    Obj& obj_;
};

template <typename MemFunc, typename Obj>
auto MakeGuard(const MemFunc& memfunc, Obj& obj) { // C++14
    return ScopeGuardImpl<MemFunc, Obj>(memfunc, obj);
}
```

Конечно, если обе версии шаблона ScopeGuard используются в одной программе, то один из них придется переименовать. Кроме того, наш охранник-функция может вызывать только функции с одним аргументом, а охранник, являющийся функцией-членом, – только функции-члены без аргументов. В C++03 эта проблема решается утомительным, но надежным способом – необходимо создать версии реализации `ScopeGuardImpl0`, `ScopeGuardImpl1`, `ScopeGuardImpl2` и т. д. для функций с нулем, одним, двумя и т. д. аргументами. Затем создаются версии `ScopeObjGuardImpl0`, `ScopeObjGuardImpl1` и т. д. для функций-членов с разным числом аргументов. Если мы создадим недостаточно версий, то компилятор сообщит об этом. Для всех этих вариантов производного класса базовый класс остается одним и тем же, как псевдоним типа `typedef ScopeGuard`.

В C++11 имеются шаблоны с переменным числом аргументов, призванные решить именно эту проблему, но здесь мы данную реализацию не приводим. А причина в том, что можно сделать гораздо лучше, и сейчас мы увидим, как.

ScopeGuard в общем виде

Теперь мы обеими ногами стоим на территории C++11, ничто из показанного ниже не имеет практических аналогов в C++03.

До сих пор наш `ScopeGuard` разрешал использовать произвольные функции для отката любого действия. Как и написанные вручную объекты-охранники, охранники области действия допускают композицию и гарантируют безопасность относительно исключений. Но у нашей реализации имеются ограничения в плане того, что именно можно вызывать для реализации отката: это должна быть свободная функция или функция-член. И хотя это уже немало, может, например, возникнуть желание вызвать две функции для выполнения одного отката. Конечно, в этом случае никто не мешает написать функцию-обертку, но это вернуло бы нас на путь, ведущий к объектам отката, созданным вручную для какой-то одной цели.

Честно говоря, в нашей реализации есть еще одна проблема. Мы решили запоминать аргумент функции по ссылке:

```
ScopeGuardImpl(const Func& func, Arg& arg);
```

Это работает, но только если аргумент не является константой или временной переменной, иначе код не откомпилируется.

C++11 предлагает еще один способ создания произвольных вызываемых объектов – лямбда-выражения. На самом деле это классы, но ведут они себя как функции, т. е. вызываются с круглыми скобками. Они могут принимать аргументы, но при этом еще запоминают, или, как говорят, захватывают переменные из объемлющей области видимости, и часто это позволяет обойтись без передачи аргументов самой функции. Мы можем также написать произвольный код и оформить его как лямбда-выражение. Выглядит идеальным решением для охраны области видимости: мы можем написать нечто такое, что будет говорить «*выполни этот код в конце области видимости*».

Посмотрим, как выглядит лямбда-выражение ScopeGuard:

```
class ScopeGuardBase {
    public:
    ScopeGuardBase() : commit_(false) {}
    void commit() const noexcept { commit_ = true; }

    protected:
    ScopeGuardBase(ScopeGuardBase&& other)
        : commit_(other.commit_) { other.commit(); }
    ~ScopeGuardBase() {}
    mutable bool commit_;

    private:
    ScopeGuardBase& operator=(const ScopeGuardBase&) = delete;
};

template <typename Func>
class ScopeGuard : public ScopeGuardBase {
    public:
    ScopeGuard(Func&& func) : func_(func) {}
    ScopeGuard(const Func& func) : func_(func) {}
    ~ScopeGuard() { if (!commit_) func_(); }
    ScopeGuard(ScopeGuard&& other)
        : ScopeGuardBase(std::move(other)),
          func_(other.func_) {}
    private:
    Func func_;
};

template <typename Func>
ScopeGuard<Func> MakeGuard(Func&& func) {
    return ScopeGuard<Func>(std::forward<Func>(func));
}
```

Базовый класс по существу такой же, как раньше, только теперь мы не собираемся использовать трюк с константной ссылкой, поэтому суффикс `Impl` исчез; то, что вы видите, – не подмога реализации, а сам базовый класс охранника.

С другой стороны, производный класс сильно отличается. Прежде всего существует всего один класс для всех видов отката, и параметра-типа в нем больше нет. Вместо этого имеется функциональный объект, в качестве которого мы передадим лямбда-выражение, и оно будет содержать все необходимые аргументы. Деструктор такой же, как раньше (только вызываемому объекту `func_` больше не передается аргумент). И перемещающий конструктор не изменился. Но основной конструктор объекта стал совсем другим; вызываемый объект хранится по значению и инициализируется константной ссылкой или ссылкой на r-значение, причем подходящий перегруженный вариант автоматически выбирается компилятором.

Функция `MakeGuard` почти не изменилась, и две функции нам больше не нужны; мы можем использовать идеальную передачу (`std::forward`), чтобы передать аргумент любого типа одному из конструкторов `ScopeGuard`.

Вот как используется шаблон ScopeGuard:

```
void Database::insert(const Record& r) {
    S.insert(r);
    auto SG = MakeGuard([&] { S.undo(); });
    I.insert(r);
    SG.commit();
}
```

Изобилующая знаками препинания конструкция, передаваемая MakeGuard в качестве аргумента, – это лямбда-выражение. Оно создает вызываемый объект, а вызов этого объекта приводит к выполнению кода в теле лямбда-выражения, в нашем случае S.undo(). В самом объекте лямбда-выражения переменная S не объявлена, поэтому она захватывается из объемлющей области видимости. Все переменные захватываются по ссылке ([&]). Наконец, объект вызывается без аргументов, скобки можно опустить, хотя запись MakeGuard([&]() { S.undo(); }); тоже допустима. Функция ничего не возвращает, поэтому тип возвращаемого значения – void, но объявлять это явно необязательно. Заметим, что до сих пор мы обходились лямбдами из C++11 и не пользовались преимуществами более мощных лямбда-выражений из C++14. Это вообще характерно для паттерна ScopeGuard, хотя на практике, наверное, стоило бы использовать C++14 хотя бы для автоматического выведения возвращаемых типов.

До сих пор мы сознательно откладывали в сторону вопрос о регулярной очистке, сосредоточившись на обработке ошибок и откате. Теперь, располагая достойным и работоспособным классом ScopeGuard, можно легко подчистить хвосты:

```
void Database::insert(const Record& r) {
    S.insert(r);
    auto SF = MakeGuard([&] { S.finalize(); });
    auto SG = MakeGuard([&] { S.undo(); });
    I.insert(r);
    SG.commit();
}
```

Как видите, для поддержки очистки не пришлось добавлять в наш каркас ничего специального. Мы просто создали еще один объект ScopeGuard, который никогда не деактивируем. Отметим также, что в C++17 функция MakeGuard больше не нужна, потому что компилятор умеет выводить аргументы из конструктора:

```
void Database::insert(const Record& r) {
    S.insert(r);
    auto SF = ScopeGuard([&] { S.finalize(); }); // только в C++17
    auto SG = ScopeGuard([&] { S.undo(); });
    I.insert(r);
    SG.commit();
}
```

И раз уж мы заговорили о том, как облагородить использование ScopeGuard, стоит упомянуть некоторые полезные макросы. Легко написать макрос для

охранника очистки, который должен выполняться всегда. Хотелось бы, чтобы окончательная синтаксическая конструкция выглядела как-то так (если и это недостаточно декларативно, то я уж и не знаю):

```
ON_SCOPE_EXIT { S.finalize(); };
```

И действительно можно получить именно такой синтаксис. Прежде всего нам нужно сгенерировать имя охранника, который раньше назывался SF, и имя должно быть уникальным. С переднего края современного C++ мы возвращаемся на десятилетия назад, к классическому C и его препроцессорным штучкам, которые позволяют нам сгенерировать уникальное имя для анонимной переменной:

```
#define CONCAT2(x, y) x##y
#define CONCAT(x, y) CONCAT2(x, y)
#ifdef __COUNTER__
#define ANON_VAR(x) CONCAT(x, __COUNTER__)
#else
#define ANON_VAR(x) CONCAT(x, __LINE__)
#endif
```

Макрос CONCAT показывает, как можно конкатенировать две лексемы в препроцессоре (и да, нужны оба, так уж работает препроцессор). Первой лексемой будет заданный пользователем префикс, второй – нечто уникальное. Многие компиляторы поддерживают препроцессорную переменную __COUNTER__, которая увеличивается на 1 при каждом использовании, так что никогда не принимает одинаковые значения. Однако в стандарте она не описана. Если переменная __COUNTER__ недоступна, то возьмем в качестве уникального идентификатора номер строки __LINE__. Конечно, он уникален, только если мы не поместим двух охранников в одной строке, так что не делайте этого.

Теперь, когда мы умеем генерировать имя анонимной переменной, можно реализовать макрос ON_SCOPE_EXIT. Это было бы тривиально, если бы мы согласились передавать код в виде аргумента макроса, но тогда мы не получим желаемого синтаксиса – поскольку аргумент должен быть заключен в круглые скобки, то лучшее, на что можно рассчитывать, – это ON_SCOPE_EXIT(S.finalize();). К тому же запятые в коде сбивают с толку препроцессор, который считает их разделителями аргументов макроса. Внимательно приглядевшись к искомому синтаксису, ON_SCOPE_EXIT { S.finalize(); };, вы поймете, что у этого макроса вообще нет аргументов, а тело лямбда-выражения записано прямо после имени макроса. Следовательно, расширение макроса заканчивается на чем-то, после чего может следовать открывающая фигурная скобка. Вот как это делается:

```
struct ScopeGuardOnExit {};
template <typename Func>
ScopeGuard<Func> operator+(ScopeGuardOnExit, Func&& func) {
    return ScopeGuard<Func>(std::forward<Func>(func));
}
#define ON_SCOPE_EXIT \
auto ANON_VAR(SCOPE_EXIT_STATE) = ScopeGuardOnExit() + [&]()
```

В расширении макроса объявлена анонимная переменная, имя которой начинается префиксом SCOPE_EXIT_STATE, за которым следует уникальное число, а заканчивается расширение неполным лямбда-выражением [&](), которое дополняется кодом в фигурных скобках. Чтобы не возникало нужды в закрывающей скобке, завершающей прежний вызов функции MakeGuard, которую макрос не может сгенерировать (расширение макроса заканчивается перед телом лямбда-выражения, поэтому после него уже никакой код сгенерировать нельзя), мы должны заменить функцию MakeGuard (или конструктор ScopeGuard в C++17) оператором. Выбор оператора не играет роли; мы остановились на +, но подошел бы любой бинарный оператор. Первым аргументом оператора является временный объект уникального типа, он ограничивает разрешение перегрузки только функцией operator+(), определенной ранее (сам объект не используется вовсе, нам нужен только его тип). Сама функция operator+() – ровным счетом то, чем раньше была MakeGuard, она выводит тип лямбда-выражения и создает соответствующий объект ScopeGuard. Единственный недостаток этой техники – необходимость точки с запятой в конце предложения ON_SCOPE_EXIT, а если вы забудете ее поставить, то компилятор напомнит самым что ни на есть туманным и загадочным сообщением.

Теперь наш код можно еще немного облагородить:

```
void Database::insert(const Record& r) {
    S.insert(r);
    ON_SCOPE_EXIT { S.finalize(); };
    auto SG = ScopeGuard([&] { S.undo(); });
    I.insert(r);
    SG.commit();
}
```

Соблазнительно было бы применить такую же технику ко второму охраннику, но, к сожалению, это не так просто – нам необходимо знать имя переменной, чтобы вызвать для нее метод commit(). Мы можем определить похожий макрос, в котором используется не анонимная переменная, а переменная с именем, заданным пользователем:

```
#define ON_SCOPE_EXIT_ROLLBACK(NAME) \
auto NAME = ScopeGuardOnExit() + [&]()
```

И воспользуемся им, чтобы завершить преобразование нашего кода:

```
void Database::insert(const Record& r) {
    S.insert(r);
    ON_SCOPE_EXIT { S.finalize(); };
    ON_SCOPE_EXIT_ROLLBACK(SG){ S.undo(); };
    I.insert(r);
    SG.commit();
}
```

Теперь самое время вернуться к вопросу о компонуемости. Для трех действий, каждое со своими откатом и очисткой, код выглядел бы так:

```
action1();
ON_SCOPE_EXIT { cleanup1; };
ON_SCOPE_EXIT_ROLLBACK(g2){ rollback1(); };
action2();
ON_SCOPE_EXIT { cleanup2; };
ON_SCOPE_EXIT_ROLLBACK(g4){ rollback2(); };
action3();
g2.commit();
g4.commit();
```

Легко видеть, что эта схема тривиально обобщается на любое число действий. Однако внимательный читатель мог бы подумать, что мы проглядели ошибку в коде – разве охранники откатов не должны деактивироваться в порядке, обратном конструированию? Но это не ошибка, впрочем, и противоположный порядок обращений к `commit()` был бы правильным. Дело в том, что `commit()` не может возбуждать исключений, поскольку объявлена с квалификатором `noexcept`, и в действительности она написана так, что исключения не возбуждаются. Это критически важно для работы паттерна ScopeGuard; если бы `commit()` могла возбуждать исключения, то нельзя было бы дать гарантию, что все охранники откатов должным образом деактивированы. Тогда в конце области видимости некоторые действия откатывались бы, а некоторые – нет, что оставило бы систему в несогласованном состоянии.

Хотя паттерн ScopeGuard проектировался в первую очередь для того, чтобы упростить написание кода, безопасного относительно исключений, его взаимодействие с исключениями далеко не тривиально, и мы должны посвятить этому вопросу еще какое-то время.

ScopeGuard и исключения

Паттерн ScopeGuard предназначен для корректного и автоматического выполнения различных операций отката и очистки при выходе из области видимости вне зависимости от того, по какой причине был произведен выход: нормальное завершение через закрывающую скобку, досрочный возврат или исключение. Это намного упрощает написание кода, безопасного относительно ошибок вообще и особенно безопасного относительно исключений. Коль скоро мы помещаем в очередь надлежащих охранников после каждого действия, правильная очистка и обработка ошибок будут происходить автоматически. Но, конечно, только в предположении, что сам ScopeGuard правильно работает при наличии исключений. Далее мы узнаем, как это обеспечить и как воспользоваться этим паттерном, чтобы сделать весь остальной код безопасным относительно исключений.

Что не должно возбуждать исключения

Мы уже видели, что функция `commit()`, которая фиксирует действие и деактивирует охранника отката, никогда не должна возбуждать исключений. По

счастью, это легко гарантировать, потому что эта функция всего лишь поднимает флажок. Но что, если в функции отката возникнет ошибка и она возбудит исключение?

```
void Database::insert(const Record& r) {
    S.insert(r);
    auto SF = MakeGuard([&] { S.finalize(); });
    auto SG = MakeGuard([&] { S.undo(); }); // что, если undo() может возбуждать
                                            // исключения?
    I.insert(r);                            // допустим, что здесь произошла ошибка
    SG.commit();                            // тогда фиксации не будет
}                                           // управление попадает сюда, и
                                            // undo() возбуждает исключение
```

Краткий ответ – *ничего хорошего*. Вообще, налицо парадоксальная ситуация – мы не можем позволить действию, в нашем случае вставке в хранилище, остаться, но не можем и отменить его, потому что эта операция тоже завершается ошибкой. В C++ два исключения не могут распространяться одновременно. Именно по этой причине деструкторам не разрешено возбуждать исключения – деструктор может быть вызван, когда исключение уже возбуждено, и если он тоже возбуждает исключение, то получается, что два исключения распространяются одновременно. В таком случае программа немедленно завершается. Это не столько недостаток языка, сколько отражение неразрешимости ситуации: оставить вещи, как есть, нельзя, но и попытка что-то изменить приводит к ошибке. Хороших вариантов не просматривается.

Вообще говоря, в C++ есть три способа выйти из этой ситуации. Самый лучший – не загонять себя в эту ловушку; если откат не может возбудить исключения, то ничего такого и не случится. Поэтому хорошо написанная безопасная относительно исключений программа сделает все возможное, чтобы откат и очистка не возбуждали исключений. Например, главное действие может произвести новые данные и держать их наготове, тогда чтобы сделать данные доступными вызывающей стороне, достаточно просто обменять указатели – очевидно, что при этом никаких исключений быть не может. Откат сводится к обратному обмену указателей и, быть может, удалению чего-то (а как мы сказали, деструкторы не могут возбуждать исключений, иначе поведение программы не определено).

Второй способ – подавить исключение при откате. Мы попытались отменить операцию, не вышло, сделать мы ничего не можем, ну, пусть остается как есть. Опасность здесь в том, что программа может остаться в неопределенном состоянии, и, начиная с этого момента, любая операция может оказаться некорректной. Впрочем, это худший сценарий. На практике последствия могут быть не столь трагичны. Например, в случае нашей базы данных мы знаем, что если откат не прошел, то на диске имеется участок, занятый записью, но недоступный по индексу. Вызывающая сторона может быть проинформирована о том, что вставка не удалась, но сколько-то места на диске мы потеряли. Иногда это все же лучше, чем аварийно завершить программу. Если такой

выход нас устраивает, то мы можем перехватить исключения, возбуждаемые действием ScopeGuard:

```
template <typename Func>
class ScopeGuard : public ScopeGuardBase {
    public:
    ...
    ~ScopeGuard() {
        if (!commit_) try { func_(); } catch (...) {}
    }
    ...
};
```

Блок `catch` пуст; мы перехватываем всё, но ничего не делаем. Иногда такую реализацию называют *экранированным ScopeGuard*.

И последний способ – дать программе аварийно завершиться. Это не потребует от нас никаких усилий, если мы просто не станем препятствовать одновременному возникновению двух исключений, но можно еще вывести сообщение или каким-то иным способом уведомить пользователя о том, что сейчас произойдет и почему. Чтобы вставить наше предсмертное послание до завершения программы, нужно написать код, очень похожий на предыдущий:

```
template <typename Func>
class ScopeGuard : public ScopeGuardBase {
    public:
    ...
    ~ScopeGuard() {
        if (!commit_) try { func_(); } catch (...) {
            std::cout << "Rollback failed" << std::endl;
            throw;        // повторно возбудить исключение
        }
    }
    ...
};
```

Ключевое отличие – предложение `throw;` без аргументов. При этом перехваченное исключение повторно возбуждается и может распространяться дальше.

Различие между двумя показанными фрагментами кода высвечивает тонкую деталь, на которую мы раньше не обращали внимания, но которая окажется важной в дальнейшем. Когда говорят, что в C++ деструктор не может возбуждать исключений, это не совсем точно. На самом деле исключение не должно выходить за пределы деструктора. Деструктор может возбуждать что угодно, при условии что сам же и перехватывает это:

```
class LivingDangerously {
    public:
    ~LivingDangerously() {
    try {
        if (cleanup() != SUCCESS) throw 0;
            more_cleanup();
```

```
        } catch (...) {
            std::cout << "Очистка не прошла, но работа продолжается"
                      << std::endl;
            // Повторно не возбуждается - это очень важно!
        }
    }
};
```

До сих пор мы рассматривали исключения в основном как досадную помеху; программа должна оставаться в согласованном состоянии, если кто-то где-то что-то возбуждает, но никаких других применений исключениям мы не видели и просто пропускали их дальше. С другой стороны, наш код мог бы работать с любым видом обработки ошибок, будь то исключения или коды ошибок. Если бы мы знали наверняка, что об ошибке всегда сигнализирует исключение и что любой возврат из функции, кроме исключительного, знаменует успех, то могли бы воспользоваться этим фактом, чтобы автоматизировать обнаружение успеха или неудачи и таким образом выполнить фиксацию или откат – то, что необходимо в конкретной ситуации.

ScopeGuard, управляемый исключениями

Далее мы будем предполагать, что если функция возвращает управление без исключений, значит, операция прошла успешно. Если же функция возбуждает исключение, то, очевидно, что-то пошло не так. Наша цель – избавиться от явного обращения к `commit()` и вместо этого обнаруживать, был ли деструктор ScopeGuard вызван в результате исключения или потому, что функция вернула управление нормально.

Реализация этого плана состоит из двух частей. Первая – описать, когда мы хотим предпринять действие. Охранник очистки должен быть выполнен вне зависимости от того, как мы вышли из области действия. Охранник отката выполняется только в случае ошибки. Для полноты картины мы можем еще завести охранника, который выполняется, только если функция завершилась успешно. Вторая часть – определить, что произошло на самом деле.

Начнем со второй части. Нашему ScopeGuard теперь нужно передать два дополнительных параметра, которые говорят, нужно ли его выполнять в случае успеха и в случае ошибки (оба могут быть активны одновременно). Модифицировать придется только деструктор `ScopeGuard`:

```
template <typename Func, bool on_success, bool on_failure>
class ScopeGuard {
  public:
    ...
    ~ScopeGuard() {
        if ((on_success && is_success()) ||
            (on_failure && is_failure())) func_();
    }
    ...
};
```

Нам еще нужно решить, как реализовать псевдофункции is_success() и is_failure(). Напомним, что ошибка означает, что было возбуждено исключение. В C++ имеется для этой цели функция, std::uncaught_exception(). Она возвращает true, если в данный момент распространяется исключение, и false в противном случае. Вооружившись этим знанием, мы можем реализовать нашего охранника:

```
template <typename Func, bool on_success, bool on_failure>
class ScopeGuard {
    public:
    ...
    ~ScopeGuard() {
        if ((on_success && !std::uncaught_exception()) ||
            (on_failure && std::uncaught_exception())) func_();
    }
    ...
};
```

Теперь вернемся к первой части: ScopeGuard будет выполнять отложенное действие, если для этого сложились подходящие условия, но как сказать, какие условия подходящие? С помощью описанного выше подхода на основе макросов мы можем определить три варианта охранника: ON_SCOPE_EXIT выполняется всегда, ON_SCOPE_SUCCESS – только если не было исключений, а ON_SCOPE_FAILURE – если было возбуждено исключение. Последний заменяет наш прежний макрос ON_SCOPE_EXIT_ROLLBACK, только теперь в нем тоже можно использовать имя анонимной переменной, поскольку явных обращений к commit() больше нет. Все три макроса определены похоже, только нужно три разных уникальных типа вместо одного ScopeGuardOnExit, чтобы можно было решить, какой operator+() вызывать:

```
struct ScopeGuardOnExit {};
template <typename Func>
auto operator+(ScopeGuardOnExit, Func&& func) {
    return ScopeGuard<Func, true, true>(std::forward<Func>(func));
}
#define ON_SCOPE_EXIT \
    auto ANON_VAR(SCOPE_EXIT_STATE) = ScopeGuardOnExit() + [&]()

struct ScopeGuardOnSuccess {};
template <typename Func>
auto operator+(ScopeGuardOnSuccess, Func&& func) {
    return ScopeGuard<Func, true, false>(std::forward<Func>(func));
}
#define ON_SCOPE_SUCCESS \
    auto ANON_VAR(SCOPE_EXIT_STATE) = ScopeGuardOnSuccess() + [&]()

struct ScopeGuardOnFailure {};
template <typename Func>
auto operator+(ScopeGuardOnFailure, Func&& func) {
    return ScopeGuard<Func, false, true>(std::forward<Func>(func));
}
```

```
#define ON_SCOPE_FAILURE \
    auto ANON_VAR(SCOPE_EXIT_STATE) = ScopeGuardOnFailure() + [&]()
```

Каждый перегруженный вариант `operator+()` конструирует объект `ScopeGuard` с разными булевыми аргументами, которые определяют, когда этот объект выполняется, а когда – нет. Каждый макрос направляет лямбда-выражение к желаемому перегруженному варианту, задавая один из трех типов первого аргумента `operator+()`, специально определенных для этой цели: `ScopeGuardOnExit`, `ScopeGuardOnSuccess`, `ScopeGuardOnFailure`.

Эта реализация позволяет передавать простые и даже довольно хитрые проверки и, на первый взгляд, работает. К сожалению, в ней есть фатальный дефект – она неправильно распознает успех и неудачу. Точнее, она хорошо работает, если функция `Database::insert()` была вызвана из нормального потока управления, где может завершиться как успешно, так и неудачно. Проблема в том, что мы можем вызывать `Database::insert()` из деструктора какого-то другого объекта, и этот объект может использоваться в области видимости, где возбуждено исключение:

```
class ComplexOperation {
    Database db_;
    public:
    ...
    ~ComplexOperation() {
        try {
            db_.insert(some_record);
        } catch (...) {} // экранировать все исключения, возбуждаемые insert()
    }
};

{
    ComplexOperation OP;
    throw 1;
}                       // здесь вызывается OP.~ComplexOperation()
```

Теперь `db_.insert()` работает при наличии неперехваченного исключения, поэтому `std::uncaught_exception()` возвращает `true`. Беда в том, что это не то исключение, которое нас интересует. Оно не означает, что в `insert()` произошла ошибка, но рассматривается так, будто именно это и случилось и вставку в базу данных нужно откатить.

На самом деле нам нужно знать, сколько исключений распространяется в настоящий момент. Это может показаться странным, поскольку C++ не позволяет нескольким исключениям распространяться одновременно. Однако мы уже видели, что это утверждение не совсем верно; второе исключение может распространяться, пока не выходит за пределы деструктора. Может распространяться и три, и более исключений, если вызовы деструкторов вложены, нужно только перехватывать их вовремя. Чтобы правильно решить эту проблему, нужно знать, сколько исключений распространялось в момент вызова функции `Database::insert()`. Тогда мы сможем сравнить это число с числом ис-

ключений, распространяющихся в конце функции, вне зависимости от того, как мы туда попали. Если оба числа совпадают, то `insert()` не возбудила исключений, а что было до этого – не наша забота. Если же появилось новое исключение, значит, `insert()` завершилась неудачно, и обработку при выходе нужно соответственно модифицировать.

C++17 позволяет реализовать такое обнаружение; помимо старой функции `std::uncaught_exception()`, которая объявлена нерекомендуемой (и будет удалена в C++20), у нас теперь есть новая функция, `std::uncaught_exceptions()`, которая возвращает количество исключений, распространяющихся в данный момент. Теперь можно реализовать класс `UncaughtExceptionDetector`, обнаруживающий появление новых исключений:

```
class UncaughtExceptionDetector {
    public:
    UncaughtExceptionDetector()
        : count_(std::uncaught_exceptions()) {}
    operator bool() const noexcept {
        return std::uncaught_exceptions() > count_;
    }
    private:
    const int count_;
};
```

Имея такой детектор, мы наконец можем реализовать автоматический `ScopeGuard`:

```
template <typename Func, bool on_success, bool on_failure>
class ScopeGuard {
    public:
    ...
    ~ScopeGuard() {
        if ((on_success && !detector_) ||
            (on_failure && detector_)) func_();
    }
    ...
    private:
    UncaughtExceptionDetector detector_;
    ...
};
```

Необходимость использовать C++17 может оказаться препятствием (хочется надеяться, временным) на пути применения этой техники. Но в большинстве современных компиляторов имеется способ получить счетчик неперехваченных исключений, пусть даже несовместимый со стандартом и непереносимый. Вот как это делается в GCC и Clang (имена, начинающиеся с `__`, относятся к внутренним типа и функциям GCC):

```
namespace __cxxabiv1 {
    struct __cxa_eh_globals;
    extern "C" __cxa_eh_globals* __cxa_get_globals() noexcept;
```

```
}
class UncaughtExceptionDetector {
    public:
    UncaughtExceptionDetector()
        : count_(uncaught_exceptions()) {}
    operator bool() const noexcept {
        return uncaught_exceptions() > count_;
    }
    private:
    const int count_;
    int uncaught_exceptions() const noexcept {
        return *(reinterpret_cast<int*>(
                  static_cast<char*>(
                  static_cast<void*>(
                  __cxxabiv1::__cxa_get_globals())) + sizeof(void*)));
    }
};
```

Не важно, используем ли мы ScopeGuard, управляемый исключениями, или явно поименованный ScopeGuard (быть может, чтобы обрабатывать не только исключения, но и ошибки), поставленных целей мы достигли – теперь мы можем задавать отложенные действия, которые следует выполнить в конце функции или другой области видимости.

В конце этой главы мы покажем еще одну реализацию ScopeGuard, которую можно найти в нескольких источниках в сети. Она заслуживает внимания, но читатель должен знать и о ее недостатках.

ScopeGuard со стертым типом

Если поискать в сети пример ScopeGuard, то можно наткнуться на реализацию, в которой вместо шаблона класса используется `std::function`. Сама по себе реализация очень проста:

```
class ScopeGuard {
    public:
    template <typename Func> ScopeGuard(Func&& func)
        : commit_(false), func_(func) {}
    template <typename Func> ScopeGuard(const Func& func)
        : commit_(false), func_(func) {}
    ~ScopeGuard() { if (!commit_) func_(); }
    void commit() const noexcept { commit_ = true; }
    ScopeGuard(ScopeGuard&& other)
        : commit_(other.commit_),
          func_(other.func_)
        { other.commit(); }
    private:
    mutable bool commit_;
    std::function<void()> func_;
    ScopeGuard& operator=(const ScopeGuard&) = delete;
};
```

Заметим, что этот ScopeGuard – не класс, а шаблон класса. В нем имеются шаблонные конструкторы, которые могут принимать такое же лямбда-выражение или иной вызываемый объект, как любой другой охранник. Но переменная, используемая для хранения выражения, имеет один и тот же тип вне зависимости от типа вызываемого объекта. Это тип `std::function<void()>`, обертка для любой функции, которая не принимает аргументов и ничего не возвращает. Как же можно сохранить значение произвольного типа в объекте некоторого фиксированного типа? Тут в дело вступает магия стирания типа, которой мы посвятили целую главу. Этот нешаблонный ScopeGuard упрощает использующий его код, поскольку не нужно выводить никаких типов:

```
void Database::insert(const Record& r) {
    S.insert(r);
    ScopeGuard SF([&] { S.finalize(); });
    ScopeGuard SG([&] { S.undo(); });
    I.insert(r);
    SG.commit();
}
```

Однако у такого подхода есть серьезный недостаток – для работы объекта со стертым типом нужны нетривиальные вычисления. Как минимум, он предполагает вызов виртуальной функции, а зачастую также выделение и освобождение памяти.

Для сравнения издержек шаблонного ScopeGuard и ScopeGuard со стертым типом мы воспользуемся библиотекой Google Benchmark. Результат зависит от охраняемой операции. Самый крайний случай – это пустой охранник, который не делает ничего (мы пользуемся макросом `REPEAT`, чтобы повторить трафаретный код 32 раза и тем самым уменьшить накладные расходы цикла):

```
void BM_type_erased_noop(benchmark::State& state) {
    for (auto _ : state) {
        REPEAT({ScopeGuardTypeErased::ScopeGuard SG([&] { noop(); });})
    }
    state.SetItemsProcessed(32*state.iterations());
}

void BM_template_noop(benchmark::State& state) {
    for (auto _ : state) {
        REPEAT({auto SG = ScopeGuardTemplate::MakeGuard([&] { noop(); });})
    }
    state.SetItemsProcessed(32*state.iterations());
}
```

Результат поражает:

```
-------------------------------------------------------------
Benchmark                   Time           CPU Iterations
-------------------------------------------------------------
BM_type_erased_noop        55 ns         55 ns    12658705   559.464M items/s
BM_template_noop            0 ns          0 ns 1000000000   61.6523P items/s
```

Версия со стертым типом занимает некоторое время, чуть меньше 2 нс на вызов, но шаблонная выглядит бесконечно быстрее. Все дело в том, что в процессе оптимизации компилятор почти полностью убрал код; он понял, что в конце области действия делать нечего, и удалил весь механизм ScopeGuard как неиспользуемый код.

Возьмем более реалистичный пример – используем ScopeGuard для удаления ранее выделенной памяти (в настоящей программе это лучше делать с помощью `std::unique_ptr`, но в качестве эталонного теста задача полезна):

```
void BM_free(benchmark::State& state) {
    void* p = NULL;
    for (auto _ : state) {
        benchmark::DoNotOptimize(p = malloc(8));
        free(p);
    }
    state.SetItemsProcessed(state.iterations());
}

void BM_type_erased_free(benchmark::State& state) {
    void* p = NULL;
    for (auto _ : state) {
        benchmark::DoNotOptimize(p = malloc(8));
        ScopeGuardTypeErased::ScopeGuard SG([&] { free(p); });
    }
    state.SetItemsProcessed(state.iterations());
}

void BM_template_free(benchmark::State& state) {
    void* p = NULL;
    for (auto _ : state) {
        benchmark::DoNotOptimize(p = malloc(8));
        auto SG = ScopeGuardTemplate::MakeGuard([&] { free(p); });
    }
    state.SetItemsProcessed(state.iterations());
}
```

Чтобы установить эталон, т. е. понять, что в данном случае *быстро*, мы также выполним последовательность выделения и освобождения без охранников:

```
-------------------------------------------------------------
Benchmark                   Time           CPU Iterations
-------------------------------------------------------------
BM_free                  18 ns         18 ns   39010078   53.1427M items/s
BM_type_erased_free      27 ns         27 ns   25172157   34.7562M items/s
BM_template_free         18 ns         18 ns   40311482   52.6061M items/s
```

Результат, конечно, не такой поразительный, но куда более характерный для реальной программы. У шаблонного ScopeGuard нет существенных накладных расходов, тогда как охранник со стертым типом увеличивает время выполнения в полтора раза.

Резюме

В этой главе мы подробно рассмотрели один из лучших паттернов C++ для написания кода, безопасного относительно ошибок и исключений. Шаблон ScopeGuard позволяет запланировать произвольное действие, т. е. фрагмент кода на C++, для выполнения по выходе из области видимости. Областью видимости может быть функция, тело цикла или просто блок, вставленный в программу для управления временем жизни локальных переменных. Запланированные действия могут выполняться условно, только при успешном выходе из области действия. Паттерн ScopeGuard одинаково хорошо работает вне зависимости от того, как индицируется исход выполнения: исключением или кодом ошибки, хотя в первом случае распознать ошибку можно автоматически (при использовании кодов ошибок программист должен явно указывать, какие возвращаемые значения означают успех, а какие неудачу). Мы проследили за эволюцией паттерна ScopeGuard по мере добавления в язык новых средств. В своей оптимальной форме ScopeGuard дает простой декларативный способ задания постусловий и отложенных действий, таких как очистка или откат, причем так, что обеспечивается тривиальная композиция любого количества действий, подлежащих фиксации или отмене.

В следующей главе будет описан еще один паттерн, специфичный для C++, Фабрика друзей. Это вариант паттерна Фабрика, только вместо объектов во время выполнения производятся функции во время компиляции.

Вопросы

- Что такое программа, безопасная относительно ошибок или исключений?
- Как можно сделать безопасной относительно ошибок процедуру, выполняющую несколько взаимосвязанных действий?
- Как идиома RAII помогает писать программы, безопасные относительно ошибок?
- Как паттерн ScopeGuard обобщает идиому RAII?
- Как программа может автоматически определить, когда функция завершилась успешно, а когда – неудачно?
- Каковы достоинства и недостатки паттерна ScopeGuard со стертым типом?

Глава 12

Фабрика друзей

В этой главе мы поговорим о том, как заводить друзей. Имеются в виду друзья в языке C++, а не друзья языка C++ (последних вы можете найти в местной группе пользователей C++). В C++ друзьями класса называются функции или другие классы, которым предоставлен специальный доступ к классу. В этом смысле они не так уж сильно отличаются от ваших собственных друзей. Но C++ может производить друзей по запросу, когда в них возникает необходимость!

В этой главе рассматриваются следующие вопросы:

- Как устроены друзья в C++ и что они делают?
- Когда следует использовать дружественные функции, а когда дружественные функции-члены класса?
- Как сочетаются друзья и шаблоны?
- Как генерировать дружественные функции по шаблону?

Технические требования

Примеры кода: **https://github.com/PacktPublishing/Hands-On-Design-Patterns-with-CPP/tree/master/Chapter12**.

Друзья в C++

Начнем с обзора того, как в C++ предоставляется дружественный доступ к классам, что при этом происходит и для чего вообще следует использовать дружбу (*мой код не компилируется, если я не сделаю всех друзьями* – не основательная причина, а признак плохо спроектированного интерфейса, поэтому перепроектируйте свои классы).

Как предоставить дружественный доступ в C++

Friend — это концепция C++, которая применяется к классам и влияет на доступ к членам класса (access — это то, что public и private контроль). Обычно общедоступные функции-члены и данные-члены доступны для всех, а частные — только для других функций-членов самого класса. Следующий код не компилируется, поскольку член данных `C::x_` является закрытым:

```
class C {
    int x_;
    public:
    C(int x) : x_(x) {}
};
C increase(C c, int dx) { return C(c.x_ + dx); }    // не компилируется
```

Самый простой способ решить эту конкретную проблему – сделать `increase()` функцией-членом, но пока оставим эту версию. Другой вариант – ослабить контроль доступа и сделать член `C::x_` открытым. Это неудачная мысль, потому что тогда `x_` становится виден не только функции `increase()`, но и вообще любому коду, который пожелает напрямую модифицировать объект типа `C`. На самом деле мы хотим сделать `x_` открытым или, по крайней мере, доступным только функции `increase()` и никому более. Для этого служит объявление другом:

```
class C {
    int x_;
    public:
    C(int x) : x_(x) {}
    friend C increase(C c, int dx);
};
C increase(C c, int dx) { return C(c.x_ + dx); }    // теперь компилируется
```

Объявление другом просто предоставляет указанной функции те же права доступа, какими обладают функции-члены класса. Существует также форма, предоставляющая дружественный доступ не функции, а классу, тогда друзьями становятся все функции-члены этого класса.

Друзья и функции-члены

Мы все же должны вернуться к вопросу, почему просто не сделать `increase()` функцией-членом класса `C`. В примере из предыдущего раздела именно так и следовало бы поступить – очевидно, что `increase()` должна быть частью открытого интерфейса класса `C`, поскольку это одна из операций, поддерживаемых `C`. Для работы ей нужен специальный доступ, поэтому она должна быть функцией-членом. Но в некоторых случаях у функций-членов имеются ограничения, а иногда их вообще нельзя использовать.

Рассмотрим оператор сложения для того же класса `C` – то, что необходимо для компиляции выражения `c1 + c2` в случае, когда обе переменные имеют тип `C`. Его можно объявить как функцию-член `operator+()`:

```
class C {
    int x_;
    public:
    C(int x) : x_(x) {}
    C operator+(const C& rhs) const { return C(x_ + rhs.x_); }
};
...
C x(1), y(2);
C z = x + y;
```

Этот код компилируется и делает ровно то, что мы хотим; никаких очевидных изъянов в нем вроде бы нет. Да не вроде бы, а просто нет – пока. Но мы можем складывать не только объекты типа C:

```
C x(1);
C z = x + 2;
```

Этот код тоже компилируется и вскрывает одну тонкую деталь в объявлении класса C – мы не сделали конструктор C(int) явным, т. е. не включили в его объявление квалификатор explicit. И теперь этот конструктор допускает неявное преобразование из int в C, вследствие чего выражение x + 2 и компилируется – сначала 2 преобразуется во временный объект C(2) с помощью предоставленного нами конструктора, а затем вызывается вторая функция-член, x.operator+(const C&), – в правой части находится только что созданный временный объект. Сам временный объект удаляется сразу после вычисления выражения. Неявное преобразование из целого типа довольно широкое, и, возможно, его наличие является недосмотром. Но предположим, что нет и что мы действительно хотели, чтобы выражение x + 2 компилировалось. Что тогда не нравится? Опять-таки ничего – пока. Сомнительная сторона нашего дизайна показана дальше:

```
C x(1);
C z = 2 + x;    // НЕ компилируется
```

Раз x + 2 компилируется, то разумно было бы ожидать, что и 2 + x тоже компилируется и дает такой же результат (существуют разделы математики, в которых сложение не коммутативно, но давайте будем придерживаться правил обычной арифметики). А не компилируется оно, потому что компилятор не может в этом контексте добраться до функции operator+() в классе C, и никакого другого оператора сложения для таких аргументов тоже не существует. Выражение x + y, используемое совместно с операторами, являющимися функциями-членами, – это просто синтаксический сахар для эквивалентного, но более длинного обращения x.operator+(y). То же самое справедливо для любого другого бинарного оператора, например умножения или сложения.

Суть дела в том, что оператор, являющийся функцией-членом, вызывается от имени первого аргумента выражения (т. е. технически x + y и y + x – не одно и то же; функция-член вызывается от имени разных объектов, но реализация устроена так, что оба обращения дают одинаковый результат). В нашем случае функцию-член следовало бы вызвать от имени числа 2, но это целое число, и никаких функций-членов у него нет. Но как тогда откомпилировалось выражение x + 2? Очень просто: x + уже подразумевает вызов x.operator+(), а аргументом будет то, что находится после знака +. В нашем случае это 2. Следовательно, компилируется выражение x.operator+(2) или нет, но поиск функции operator+ для этого вызова уже закончен. В данном случае в классе C имеется неявное преобразование из int, поэтому вызов и компилируется. Но почему компилятор не пробует применить преобразование к первому аргументу? Отвечаем,

он не делает этого никогда, потому что не знает, во что преобразовывать; может существовать сколько угодно типов, в которых определена функция-член operator+(), и некоторые из них, возможно, принимают аргумент типа C или какого-то типа, в который можно преобразовать C. Компилятор даже не пытается исследовать потенциально неограниченное множество возможных преобразований.

Если мы хотим использовать плюс в выражениях, где первое слагаемое может иметь встроенный или еще какой-то тип, в котором нет или не может быть функции-члена operator+(), то следует использовать свободную функцию. Ну что же, нет проблем, мы знаем, как их писать:

```
C operator+(const C& lhs, const C& rhs) {
    return C(lhs.x_ + rhs.x_);
}
```

Но теперь мы утратили доступ к закрытому члену C::x_, поэтому наша функция operator+() не откомпилируется. Решение этой проблемы мы видели в предыдущем разделе – нужно сделать ее другом:

```
class C {
    int x_;
    public:
    C(int x) : x_(x) {}
    friend C operator+(const C& lhs, const C& rhs);
};
C operator+(const C& lhs, const C& rhs) {
return C(lhs.x_ + rhs.x_);
}
C x(1), y(2);
C z1 = x + y;
C z2 = x + 2;
C z3 = 1 + y;
```

Теперь все компилируется и работает как надо. Свободная функция operator+() – обычная функция с двумя аргументами типа const C&. Для нее действуют такие же правила, как для любой другой подобной функции.

Можно не набирать объявление operator+(), если определить его тело *по месту* (сразу после объявления, внутри класса):

```
class C {
    int x_;
    public:
    C(int x) : x_(x) {}
    friend C operator+(const C& lhs, const C& rhs) {
        return C(lhs.x_ + rhs.x_);
    }
};
```

В этом примере делается все то же самое, что в предыдущем, так что это просто вопрос стиля – если поместить тело функции внутрь класса, то станет длиннее объявление самого класса, а если определить функцию вне класса, то

приходится больше стучать по клавишам (и возможны расхождения между объявлением и определением функции, если код когда-нибудь изменится).

Как бы то ни было, дружественная функция в действительности является частью открытого интерфейса класса, но по техническим причинам мы предпочитаем не делать ее функцией-членом. Есть даже случай, когда свободная функция – единственный вариант. Рассмотрим операторы ввода-вывода в C++, например `operator<<()`, который служит для вывода объектов в поток (например, `std::cout`). Мы хотим иметь возможность следующим образом напечатать объект типа `C`:

```
C c1(5);
std::cout << c1;
```

Для нашего типа `C` не существует стандартного оператора `operator<<()`, поэтому придется объявить свой собственный. Оператор вывода – это бинарный оператор, как и плюс (параметры задаются с обеих сторон), поэтому будь он функцией-членом, определить его нужно было бы в классе объекта в левой части. Но взгляните на пример выше – в левой части выражения `std::cout << c1` находится не наш объект `c1`, а стандартный поток вывода, `std::cout`. Именно в этот объект следовало бы добавить функцию-член, но сделать этого мы не можем, потому что `std::cout` объявлен где-то в заголовочных файлах стандартной библиотеки C++ и расширить его интерфейс невозможно, по крайней мере не напрямую. Функции-члены класса `C` объявлены нами, но это ничем не помогает – рассматриваются только функции-члены объекта в левой части.

Единственная альтернатива – свободная функция. Ее первый аргумент должен иметь тип `std::ostream&`:

```
std::ostream& operator<<(std::ostream& out, const C& c) {
    out << c.x_;
    return out;
}
```

Эта функция должна быть объявлена другом, потому что ей необходим доступ к закрытым данным класса `C`. Можно также определить ее *по месту*:

```
class C {
    int x_;
    public:
    C(int x) : x_(x) {}
    friend std::ostream& operator<<(std::ostream& out, const C& c) {
        out << c.x_;
        return out;
    }
};
```

По соглашению, должен возвращаться тот же самый объект потока, чтобы операторы вывода можно было сцеплять:

```
C c1(5), c2(7);
std::cout << c1 << c2;
```

Последнее предложение интерпретируется как (std::cout << c1) << c2, т. е. operator<<(operator<< (std::cout, c1), c2). Внешний operator<<() вызывается от имени значения, возвращенного внутренним operator<<(), каковым является все тот же std::cout. Повторим, что этот оператор вывода – часть открытого интерфейса класса C, он обеспечивает возможность печати объектов типа C. Однако он обязан быть свободной функцией.

Пока что мы рассматривали только обычные классы, а не шаблон, а свободными дружественными функциями тоже были обычные, а не шаблонные функции. А теперь посмотрим, что нужно изменить и нужно ли вообще что-то менять, если вместо класса фигурирует шаблон.

Друзья и шаблоны

В C++ и классы, и функции могут быть шаблонами, причем возможны различные сочетания: шаблон класса может объявить другом нешаблонную функцию, если ее параметры не зависят от параметров шаблона; этот случай не особенно интересен и, безусловно, не решает ни одной из рассматриваемых нами задач. Если дружественная функция должна работать с типами-параметрами шаблона, то обзаведение правильными друзьями становится более хитрым делом.

Друзья шаблонов классов

Для начала превратим наш класс C в шаблон:

```
template <typename T> class C {
    T x_;
    public:
    C(T x) : x_(x) {}
};
```

Мы по-прежнему хотим складывать объекты типа C и печатать их. Мы уже обсудили, почему первую задачу лучше решать с помощью свободной функции, а вторую – только таким способом и никак иначе. Эти причины остаются в силе и для шаблонов.

Никаких проблем – мы можем объявить шаблонные функции, сопровождающие шаблон класса, которые будут делать то же самое, что нешаблонные функции в предыдущем разделе. Начнем с operator+():

```
template <typename T>
    C<T> operator+(const C<T>& lhs, const C<T>& rhs) {
    return C<T>(lhs.x_ + rhs.x_);
}
```

Это та же функция, что и раньше, только мы преобразовали ее в шаблон, который принимает любую конкретизацию шаблона класса C. Заметим, что мы параметризовали этот шаблон типом T, т. е. параметром шаблона C. Можно было бы, конечно, объявить ее и так:

```
template <typename C>
    C operator+(const C& lhs, const C& rhs) {   // НИКОГДА так не поступайте!
    return C<T>(lhs.x_ + rhs.x_);
}
```

Однако при этом мы вводим – в глобальную область видимости, никак не меньше – `operator+()`, который заявляет о готовности принять два аргумента любого типа. Разумеется, на самом деле он будет обрабатывать только типы, в которых имеется член данных `x_`. И что делать, если имеется также шаблон класса `D`, который тоже допускает сложение, но вместо `x_` содержит член `y_`?

Предыдущая версия шаблона хотя бы ограничивала тип возможными конкретизациями шаблона класса `C`. Конечно, она страдает от той же проблемы, что наша первая попытка написать свободную функцию, – у нее нет доступа к закрытому члену данных `C<T>::x_`. Не страшно, ведь это глава про друзей. Только чьих друзей? Для всего шаблона класса `C` появится объявление дружественной функции, одно для всех типов `T`, и она должна будет работать для любой конкретизации шаблонной функции `operator+()`. Похоже, мы должны предоставить дружественный доступ всей шаблонной функции:

```
template <typename T> class C {
    int x_;
    public:
    C(int x) : x_(x) {}
    template <typename U>
    friend C<U> operator+(const C<U>& lhs, const C<U>& rhs);
};
template <typename T>
C<T> operator+(const C<T>& lhs, const C<T>& rhs) {
    return C<T>(lhs.x_ + rhs.x_);
}
```

Обратите внимание на правильный синтаксис – ключевое слово `friend` должно располагаться после слова `template` и параметров шаблона, но перед возвращаемым типом функции. Также заметьте, что нам пришлось переименовать параметр шаблона во вложенном объявлении друга – идентификатор `T` уже занят параметром шаблона класса. Мы могли бы переименовать и параметр шаблона `T` в определении функции, но это необязательно – как и в объявлениях и определениях функций, параметр – это просто имя, которое имеет смысл только внутри объявления. В двух объявлениях одной и той же функции для одного и того же параметра можно иметь разные имена. Можно вместо этого перенести тело функции внутрь класса:

```
template <typename T> class C {
    int x_;
    public:
    C(int x) : x_(x) {}
    template <typename U>
    friend C<U> operator+(const C<U>& lhs, const C<U>& rhs) {
```

```
        return C<U>(lhs.x_ + rhs.x_);
    }
};
```

Читатель может возразить, что мы распахнули довольно широкую дыру в инкапсуляции шаблона класса `C` – предоставив дружественный доступ любой конкретизации `C<T>` всей шаблонной функции, мы, например, сделали конкретизацию `operator+(const C&<double>, const C&<double>)` другом `C<int>`. Ясно, что это необязательно, хотя сразу не понятно, в чем тут вред (пример, демонстрирующий фактическое причинение вреда, по необходимости был бы довольно запутанным). Но мы упускаем из виду гораздо более серьезную проблему этого дизайна, которая становится очевидной, как только мы начнем использовать класс для сложения. Он работает:

```
C<int> x(1), y(2);
C<int> z = x + y;     // Пока все хорошо...
```

Однако лишь до определенного момента:

```
C<int> x(1), y(2);
C<int> z1 = x + 2;    // Не компилируется!
C<int> z2 = 1 + 2;    // И это тоже!
```

Но разве это не та самая причина, по которой мы стали использовать свободную функцию? Что случилось с неявными преобразованиями? Все же работало! Дьявол кроется в деталях – да, работало, но для нешаблонной функции `operator+()`. Для шаблонных функций действуют совсем другие правила преобразования. Точные технические детали можно выудить из стандарта, при должном усердии, а мы приведем выжимку: рассматривая свободные нешаблонные функции, компилятор ищет все функции с данным именем (в нашем случае `operator+`), затем проверяет, принимают ли они нужное число параметров, потом для каждой функции и каждого параметра проверяет, существует ли преобразование из типа переданного аргумента в указанный тип параметра (правила, описывающие, какие точно преобразования рассматриваются, тоже довольно сложны, но отметим, что определенные пользователем неявные преобразования и встроенные преобразования, такие как не `const` в `const`, допустимы). Если в результате этого процесса остается только одна функция, то она и вызывается (иначе компилятор выбирает наилучший перегруженный вариант или ругается на то, что несколько кандидатов одинаково хороши и потому вызов неоднозначный).

Для шаблонных функций этот процесс дал бы практически неограниченное количество кандидатов – каждую шаблонную функцию с именем `operator+()` пришлось бы конкретизировать всеми известными типами, только чтобы проверить, достаточно ли имеется преобразований типов, чтобы она заработала. Вместо этого применяется гораздо более простой процесс – в дополнение ко всем нешаблонным функциям, описанным в предыдущем абзаце (в нашем

случае таковых нет), компилятор рассматривает также конкретизации шаблонных функций с данным именем (все ту же operator+), для которых типы всех параметров совпадают с типами аргументов функции, переданных при вызове (допускаются также тривиальные преобразования, например добавление const).

В нашем случае аргументы в выражении x + 2 имеют типы C<int> и int соответственно. Компилятор ищет конкретизацию шаблона функции operator+, которая принимает два аргумента такого типа, не рассматривая пользовательские преобразования. Такой функции, конечно же, нет, и разрешить обращение к operator+() невозможно.

Корень проблемы в том, что мы хотим, чтобы компилятор автоматически использовал определенные пользователем преобразования, но при попытке конкретизировать шаблон функции этого не происходит. Можно было бы объявить нешаблонную функцию operator+(const C<int>&, const C<int>&), но если C – шаблон класса, то нам пришлось объявлять такую функцию для каждого типа T, которым может быть конкретизирован класс C.

Фабрика друзей шаблона

На самом деле нам требуется автоматически генерировать нешаблонную функцию для каждого типа T, используемого для конкретизации шаблона класса C. Конечно, сгенерировать все такие функции заранее невозможно, т. к. теоретически множество типов T, которые можно использовать совместно с шаблоном класса C, не ограничено. По счастью, нам и не нужно генерировать operator+() для каждого такого типа, а лишь для типов, которые фактически встречаются с этим шаблоном в программе.

Генерация друзей по запросу

Паттерн, о котором мы собираемся рассказать, очень старый, он был придуман Джоном Бартоном (John Barton) и Ли Нэкманом (Lee Nackman) в 1994 году совсем для другой цели – они использовали его, чтобы обойти некоторые ограничения тогдашних компиляторов. Авторы предложили название *ограниченное расширение шаблона* (Restricted Template Expansion), которое так и не обрело популярности. Спустя много лет Дэн Сакс (Dan Sacks) предложил название *Фабрика друзей* (Friends Factory), но иногда этот паттерн называют просто *трюк Бартона–Нэкмана*.

Выглядит паттерн совсем просто и очень похож на код, который мы писали в этой главе:

```
template <typename T> class C {
    int x_;
    public:
    C(int x) : x_(x) {}
    friend C operator+(const C& lhs, const C& rhs) {
```

```
        return C(lhs.x_ + rhs.x_);
    }
};
```

Мы пользуемся довольно специфичной особенностью C++, поэтому код надо писать предельно точно. Нешаблонная дружественная функция определена внутри шаблона класса. Эта функция обязательно должна быть внутри, ее нельзя объявить другом, а определить позже, разве что в виде явной конкретизации шаблона – мы могли бы объявить дружественную функцию внутри класса, а затем определить `operator+<const C<int>>&, const C<int>&)`, и это работало бы для `C<int>`, но не для `C<double>` (но поскольку мы не знаем, какими типами программа будет конкретизировать шаблон впоследствии, это не слишком полезно). Функция может иметь параметры типа `T` (параметр шаблона), типа `C<T>` (внутри класса его можно обозначать просто `C`) и любого другого типа, который либо фиксирован, либо зависит только от параметров шаблона, но этот тип сам не может быть шаблоном. Любая конкретизация шаблона класса `C` с любой комбинацией параметров-типов генерирует ровно одну нешаблонную свободную функцию с указанным именем. Заметим, что сгенерированные функции нешаблонные, это обычные функции, к которым применимы все стандартные правила разрешения перегрузки. Если теперь вернуться к нешаблонной `operator+()`, то все преобразования работают, как мы и хотели:

```
C<int> x(1), y(2);
C<int> z1 = x + y;          // работает
C<int> z2 = x + 2;          // и это работает
C<int> z3 = 1 + 2;          // и это тоже
```

Вот и весь паттерн. Но обратим внимание на некоторые детали. Во-первых, ключевое слово `friend` нельзя опустить. Класс может сгенерировать свободную функцию, только если она объявлена другом. Даже если функции не нужен доступ ни к каким закрытым данным, она должна быть объявлена другом, если мы хотим автоматически генерировать нешаблонные функции по конкретизации шаблона класса (точно так же могут быть сгенерированы статические свободные функции, но бинарные операторы не могут быть статическими функциями – стандарт явно запрещает это). Во-вторых, сгенерированная функция помещается в область видимости, объемлющую класс. Например, определим оператор вывода для нашего шаблона класса `C`, предварительно поместив весь класс в пространство имен:

```
namespace NS {
template <typename T> class C {
    int x_;
    public:
    C(int x) : x_(x) {}
    friend C operator+(const C& lhs, const C& rhs) {
        return C(lhs.x_ + rhs.x_);
    }
    friend std::ostream& operator<<(std::ostream& out, const C& c) {
```

```
        out << c.x_;
        return out;
    }
};
}
```

Теперь можно складывать и печатать объекты типа C:

```
NS::C<int> x(1), y(2);
std::cout << (x + y) << std::endl;
```

Заметим, что хотя шаблон класса C теперь находится в пространстве имен NS и должен соответствующим образом использоваться (NS::C<int>), нам не нужно делать ничего особенного, чтобы вызвать operator+() или operator<<(). Это не означает, что обе функции сгенерированы в глобальной области видимости. Нет, они по-прежнему находятся в пространстве имен NS, а то, что мы видим, – это результат поиска, зависящего от аргументов, – при поиске функции с именем operator+() компилятор рассматривает кандидатов как в текущей области видимости (т. е. глобальной, где их нет), так и в той области видимости, где определены аргументы функции. В нашем случае по крайней мере один из аргументов operator+() имеет тип NS::C<int>, что автоматически включает в число кандидатов функции, объявленные в пространстве имен NS. Фабрика друзей генерирует функции в области видимости, объемлющей шаблон класса, каковой, конечно, является пространство имен NS. Таким образом, поиск находит определение, и обе операции + и << разрешаются именно так, как нам и нужно. Будьте уверены, что это не случайность, а сознательное проектное решение: правила поиска, зависящего от аргументов, тщательно определены, так чтобы давать желаемый и ожидаемый результат.

Отметим также, что хотя дружественные функции генерируются в области видимости, объемлющей класс, в нашем случае – в пространстве имен NS, найти их можно только поиском, зависящим от аргументов (и снова это специальное положение стандарта). Попытка напрямую указать имя функции, не прибегая к поиску, зависящему от аргументов, обречена на неудачу:

```
auto p = &NS::C<int>::operator+;    // не компилируется
```

Фабрика друзей и Рекурсивный шаблон

Фабрика друзей – это паттерн, который синтезирует свободные нешаблонные функции для каждой конкретизации шаблона класса – всякий раз, как конкретизируется новый тип, генерируется новая функция. Параметры этой функции могут иметь любые типы, которые можно объявить в данной конкретизации шаблона класса. Обычно это сам класс, но может быть любой тип, о котором шаблон знает.

Это позволяет использовать Фабрику друзей совместно с паттерном **Рекурсивный шаблон** (**CRTP**), который мы изучали в главе 7. Напомним, что основ-

ная идея CRTP заключается в том, что класс наследует конкретизации шаблона базового класса, параметризованной типом производного класса. Посмотрим, что мы здесь имеем – шаблон базового класса, который автоматически конкретизируется любым производным от него классом и знает, какого он типа. Идеальное место для размещения Фабрики друзей. Конечно, операторы, сгенерированные базовым классом, должны знать не только тип объекта, с которым работают, но и что с ним делать (например, как его напечатать). Иногда необходимые данные фактически находятся в базовом классе, и тогда базовый класс может предоставить полную реализацию. Но редко бывает, что производный класс так мало добавляет к базовому. Чаще комбинация CRTP с Фабрикой друзей используется для реализации операторов стандартным способом посредством какой-то другой функциональности. Например, `operator!=()` можно реализовать посредством `operator==()`:

```
template <typename D> class B {
    public:
    friend bool operator!=(const D& lhs, const D& rhs) {
        return !(lhs == rhs);
    }
};
template <typename T> class C : public B<C<T>> {
    T x_;
    public:
    C(T x) : x_(x) {}
    friend bool operator==(const C& lhs, const C& rhs) {
        return lhs.x_ == rhs.x_;
    }
};
```

Здесь в производном классе `C` используется паттерн Фабрика друзей для генерации нешаблонной функции бинарного оператора `operator==()` непосредственно по конкретизации шаблона класса. Производный класс также наследует базовому классу `B`, который активирует конкретизацию этого шаблона, что, в свою очередь, генерирует нешаблонную функцию `operator!=()` для каждого типа, для которого сгенерирована `operator==()`.

Второе применение CRTP – преобразование функций-членов в свободные функции. Например, бинарный оператор `operator+()` иногда реализуется в терминах функции `operator+=()`, которая всегда является функцией-членом (она вызывается от имени первого операнда). Чтобы реализовать бинарный `operator+()`, кто-то должен позаботиться о преобразованиях в тип этого объекта, а затем можно вызывать `operator+=()`. Эти преобразования предоставляют бинарные операторы, сгенерированные базовым классом CRTP при использовании фабрики друзей. Аналогично оператор вывода можно сгенерировать, если принять соглашение, что во всех наших классах имеется функция-член `print()`:

```
template <typename D> class B {
    public:
    friend D operator+(const D& lhs, const D& rhs) {
        D res(lhs);
        res += rhs; // Convert += to +
        return res;
    }
    friend std::ostream& operator<<(std::ostream& out, const D& d) {
        d.print(out);
        return out;
    }
};
template <typename T> class C : public B<C<T>> {
    T x_;
    public:
    C(T x) : x_(x) {}
    C operator+=(const C& incr) {
        x_ += incr.x_;
        return *this;
    }
    void print(std::ostream& out) const {
        out << x_;
    }
};
```

Таким образом, CRTP можно использовать для добавления трафаретных интерфейсов, когда реализация делегируется производным классам. В конце концов, это паттерн статического (на этапе выполнения) делегирования.

Резюме

В этой главе мы узнали о специфическом для C++ паттерне, который первоначально был предложен для обхода ошибок в ранних компиляторах C++, но спустя много лет нашел новые применения. Фабрика друзей применяется для генерирования нешаблонных функций по конкретизациям шаблонов классов. Будучи нешаблонными, сгенерированные друзья подчиняются гораздо более гибким, по сравнению с шаблонными функциями, правилам преобразования типов аргументов. Мы также узнали о том, как поиск, зависящий от аргументов, преобразования типов и Фабрика друзей, работая совместно, дают естественно выглядящий результат, хотя процесс, благодаря которому это достигается, далек от интуитивно очевидного.

В следующей главе описывается совершенно другой вид Фабрики – паттерна C++, который основан на классическом паттерне Фабрика и устраняет некоторую асимметрию в языке – все функции-члены, даже деструкторы, могут быть виртуальными, а конструкторы – нет.

Вопросы

- Каков эффект объявления функции *другом*?
- В чем разница между предоставлением дружественного доступа функции и шаблону функции?
- Почему бинарные операторы обычно реализуются как свободные функции?
- Почему оператор вывода в поток всегда реализуется в виде свободной функции?
- В чем основная разница между преобразованиями аргументов шаблонных и нешаблонных функций?
- Как сделать так, чтобы при конкретизации шаблона всегда генерировалась также уникальная нешаблонная свободная функция?

Глава 13

Виртуальные конструкторы и фабрики

В C++ любая функция-член класса, в т. ч. его деструктор, может быть виртуальной, любая, кроме одной – конструктора. Если функция-член не виртуальная, то точный тип объекта, от имени которого она вызывается, известен на этапе компиляции. Поэтому тип конструируемого объекта всегда известен во время компиляции, в той точке, где конструктор вызывается. Тем не менее нам часто необходимо конструировать объекты, тип которых становится известен только на этапе выполнения. В данной главе описаны паттерны и идиомы, решающие эту проблему, в т. ч. паттерн Фабрика.

В этой главе рассматриваются следующие вопросы:

- почему нельзя сделать конструктор виртуальным;
- как использовать паттерн Фабрика, чтобы отложить выбор типа конструируемого объекта до этапа выполнения;
- использование идиом C++ для полиморфного конструирования и копирования объектов.

Технические требования

Примеры кода: **https://github.com/PacktPublishing/Hands-On-Design-Patterns-with-CPP/tree/master/Chapter13**.

Почему конструкторы не могут быть виртуальными

Мы уже понимаем, как работает полиморфизм. Когда виртуальная функция вызывается через указатель или ссылку на базовый класс, этот указатель или ссылка используется для доступа к v-указателю в классе (указателю на таблицу виртуальных функций, или v-таблицу). V-указатель позволяет определить истинный тип объекта, т. е. тип, который был указан в момент его создания. Это может быть как сам базовый класс, так и один из производных от него классов. Затем вызывается функция-член этого класса. Так почему то же самое нельзя проделать с конструкторами? Разберемся.

Когда объект получает свой тип?

Довольно легко понять, почему описанная выше процедура не может работать для создания *виртуальных конструкторов*. Прежде всего из описания понятно, что частью процесса является *определение типа объекта, указанного в момент его создания*. Это возможно только после того, как объект сконструирован – до этого момента еще не существует объекта данного типа, а лишь неинициализированная область памяти. Можно взглянуть и по-другому – до того как будет произведена диспетчеризация виртуальной функции к нужному типу, необходимо справиться с v-указателем. Но кто помещает правильное значение в v-указатель? Учитывая, что v-указатель однозначно определяет тип объекта, он может быть инициализирован только в процессе конструирования. Следовательно, он не был инициализирован до конструирования. Но если он не был инициализирован, то его нельзя использовать для диспетчеризации вызовов виртуальных функций. Так что, как ни крути, конструкторы виртуальными быть не могут.

Для производных классов в иерархии процедура установления типа еще сложнее. Мы можем попытаться понаблюдать за типом объекта в процессе его конструирования. Проще всего это сделать с помощью оператора `typeid`, который возвращает информацию о типе объекта, включая и имя типа:

```
class A {
    public:
    A() { std::cout << "A::A(): " << typeid(*this).name() << std::endl; }
    virtual ~A() {
        std::cout << "A::~A(): " << typeid(*this).name() << std::endl;
    }
};
class B : public A {
    public:
    B() { std::cout << "B::B(): " << typeid(*this).name() << std::endl; }
    ~B() { std::cout << "B::~B(): " << typeid(*this).name() << std::endl; }
};
class C : public B {
    public:
    C() { std::cout << "C::C(): " << typeid(*this).name() << std::endl; }
    ~C() { std::cout << "C::~C(): " << typeid(*this).name() << std::endl; }
};

int main() {
    C c;
}
```

Выполнение этой программы дает следующий результат:

```
A::A(): 1A
B::B(): 1B
C::C(): 1C
C::~C(): 1C
B::~B(): 1B
A::~A(): 1A
```

Обращение к функции `std::typeinfo::name()` возвращает так называемое искаженное имя типа – внутреннее имя, по которому компилятор идентифицирует типы, а не понятное человеку имя вида `class A`. Сколько объектов было конструировано в этом примере? Согласно исходному коду, всего один – объект с типа `C`. Но исполняющая среда говорит, что три, по одному каждого типа. Оба ответа правильны – когда конструируется объект типа `C`, сначала необходимо сконструировать базовый класс `A`, и его конструктор вызывается первым. Затем конструируется промежуточный базовый класс `B`, и только потом `C`. Деструкторы вызываются в обратном порядке. Тип объекта внутри конструктора или деструктора, сообщаемый оператором `typeid`, совпадает с типом объекта, для которого выполняется конструктор или деструктор.

Создается впечатление, что тип, определяемый виртуальным указателем, изменяется в процессе конструирования! Разумеется, в предположении, что оператор `typeid` возвращает динамический тип, определяемый виртуальным указателем, а не статический тип, который можно узнать во время компиляции. Стандарт говорит, что так оно и есть. Означает ли это, что если вызвать один и тот же виртуальный метод из каждого конструктора, то на самом деле будут вызываться три разных перегруженных варианта этого метода? Легко проверить:

```
class A {
    public:
    A() { whoami(); }
    virtual ~A() { whoami(); }
    virtual void whoami() const { std::cout << "A::whoami" << std::endl; }
};
class B : public A {
    public:
    B() { whoami(); }
    ~B() { whoami(); }
    void whoami() const { std::cout << "B::whoami" << std::endl; }
};
class C : public B {
    public:
    C() { whoami(); }
    ~C() { whoami(); }
    void whoami() const { std::cout << "C::whoami" << std::endl; }
};

int main() {
    C c;
    c.whoami();
}
```

Если создать объект типа `C`, то вызов `whoami()` после создания подтвердит – динамический тип объекта равен `C`. Это было справедливо с самого начала процесса конструирования; мы просили компилятор сконструировать один объект `C`, но динамический тип объекта менялся по ходу конструирования:

```
A::whoami
B::whoami
C::whoami
C::whoami
C::whoami
B::whoami
A::whoami
```

Видно, что значение виртуального указателя меняется по ходу конструирования объекта. Вначале он идентифицировал тип объекта как `A`, хотя конечным типом является `C`. Не связано ли это с тем, что объект создается в стеке? Может быть, при создании объекта в куче все будет по-другому? Проверим:

```
C* c = new C;
c->whoami();
delete c;
```

Выполнение модифицированной программы дает точно такой же результат, как исходной.

Еще одна причина, по которой конструктор не может быть виртуальным и вообще почему тип конструируемого объекта должен быть известен во время компиляции в точке конструирования, заключается в том, что компилятору нужно знать, сколько памяти выделить под объект. Объем памяти определяется размером типа, который сообщает оператор `sizeof`. Результатом `sizeof(C)` является константа времени компиляции, поэтому объем памяти, выделенной для нового объекта, всегда известен на этапе компиляции. Это верно вне зависимости от того, создается объект в стеке или в куче.

Итог таков: если программа создает объект типа `T`, то где-то в коде имеется явный вызов конструктора `T::T`. Затем тип `T` можно скрыть от программы, например если мы будем обращаться к объекту через указатель на базовый класс или сотрем его тип (см. главу 6). Но хотя бы одно явное упоминание типа `T` должно быть – в точке конструирования.

С одной стороны, у нас теперь есть весьма разумное объяснение того, почему конструирование объектов не может быть полиморфным. С другой стороны, это никак не приближает нас к решению проблемы проектирования: как сконструировать объект, тип которого неизвестен во время компиляции. Рассмотрим проектирование игры – игрок может набрать или призвать сколько угодно искателей приключений в свою партию и начать строить поселения и города. Было бы разумно иметь отдельный класс для каждого типа авантюриста и каждого типа строения, но ведь мы должны будем конструировать объекты этих типов, когда искатель приключений присоединяется к партии или когда здание возводится, а пока игрок не выберет тип, игра не знает, какой объект конструировать.

Как часто бывает при разработке программного обеспечения, для решения нужно ввести еще один уровень косвенности.

Паттерн Фабрика

Проблема, с которой мы столкнулись, – как во время выполнения выбрать тип создаваемого объекта – конечно, очень часто встречается при проектировании. Паттерны проектирования – это решения как раз таких проблем, и для этой тоже есть паттерн, называется он Фабрика. Паттерн Фабрика относится к категории порождающих и предлагает решения нескольких родственных проблем – как делегировать решение о том, какой объект создавать, производному классу, как создавать объекты с помощью отдельного фабричного метода и т. д. Мы рассмотрим эти вариации паттерна Фабрика поочередно и начнем с простого фабричного метода.

Основа паттерна Фабричный метод

В своей простейшей форме Фабричный метод конструирует объект типа, указанного во время выполнения:

```
class Base { ... };
class Derived : public Base { ... };
Base* p = ClassFactory( some_type_identifier, ... arguments );
```

Как во время выполнения понять, какой объект создавать? Нам нужен какой-то идентификатор для каждого типа, который умеет создавать Фабрика. В простейшем случае список типов известен на этапе компиляции. Рассмотрим проектирование игры, в которой игрок выбирает тип строящегося здания из меню. В программе имеется список зданий, которые можно построить, и каждое представлено объектом, имеющим идентификатор из этого списка:

```
enum Buildings {
    FARM, FORGE, MILL, GUARDHOUSE, KEEP, CASTLE
};
class Building {
    virtual ~Building() {}
};
class Farm : public Building { ... };
class Forge : public Building { ... };
```

Когда игрок выбирает тип здания, игровая программа выбирает соответствующий идентификатор, после чего можем сконструировать объект здания с помощью фабричного метода:

```
Building* new_building = MakeBuilding(Buildings building_type);
```

Заметим, что Фабрика принимает идентификатор типа в качестве аргумента и возвращает указатель на базовый класс. Возвращенный объект должен иметь тип, соответствующий идентификатору. Как реализуется Фабрика? Напомним вывод из предыдущего раздела: где-то в программе каждый объект должен быть сконструирован с указанием своего истинного типа. Паттерн Фабрика не

отменяет это требование, он просто скрывает место, в котором производится конструирование:

```
Building* MakeBuilding(Buildings building_type) {
    switch (building_type) {
        case FARM: return new Farm;
        case FORGE: return new Forge;
        ...
    }
}
```

Соответствие между идентификатором типа и типом объекта зашито в предложении `switch` внутри Фабрики. Возвращаемый тип должен быть одинаковым для всех объектов, конструируемых Фабрикой, поскольку существует всего один фабричный метод, и его тип объявлен на этапе компиляции. В простейшем случае это указатель на базовый класс, хотя если вы следуете современным идиомам владения памятью, описанным в главе 3, то понимаете, что Фабрика может возвращать уникальный указатель, `std::unique_ptr<Base>`, если владелец объекта понятен, а в редких случаях, когда требуется совместное владение, можно с помощью функции `std::make_shared()` сконструировать и вернуть разделяемый указатель типа `std::shared_ptr<Base>`.

Это основная форма фабричного метода. Но есть много вариаций, ориентированных на конкретные задачи. Некоторые из них мы рассмотрим ниже.

Фабричные методы с аргументами

В нашем простом примере конструктор не принимал никаких аргументов. Передача аргументов конструктору составляет проблему, если конструкторы разных типов принимают разные аргументы, ведь в объявлении функции `MakeBuilding()` должны быть указаны конкретные параметры. Первое, что приходит на ум, – воспользоваться функциями с переменным числом аргументов. Действительно, раз аргументы могут быть разного типа и в разном количестве, то функция с переменными аргументами – именно то, что нужно. Но это сложное решение, которое становится куда менее привлекательным, стоит приняться за его реализацию. Более простой вариант – создать иерархию объектов параметров, соответствующую иерархии самих объектов. Предположим, что в нашей игре игрок может выбирать модификации возводимого здания. Опции, выбранные игроком в пользовательском интерфейсе, будут сохраняться в объекте, специфичном для типа здания:

```
struct BuildingSpec {
    virtual Buildings building_type() const = 0;
};
struct FarmSpec : public BuildingSpec {
    Buildings building_type() const { return FARM; }
    bool with_pasture;
    int number_of_stalls;
};
```

```
struct ForgeSpec : public BuildingSpec {
    Buildings building_type() const { return FORGE; }
    bool magic_forge;
    int number_of_apprentices;
};
```

Обратите внимание, что мы включили в объект параметров идентификатор типа, – нет никаких причин вызывать фабричный метод с двумя аргументами, типы которых всегда должны точно соответствовать друг другу, это только откроет возможность для ошибок. При таком подходе гарантируется, что при каждом вызове фабричного метода идентификатор типа будет соответствовать аргументам.

```
Building* MakeBuilding(const BuildingSpec* building_spec) {
    switch (building_spec->building_type()) {
        case FARM: return new Farm(static_cast<const FarmSpec*>(building_spec));
        case FORGE: return new Forge(static_cast<const ForgeSpec*>(building_spec));
        ...
    }
}
```

Эту технику можно сочетать с некоторыми другими вариациями Фабрики, показанными в следующих разделах, когда требуется передавать аргументы конструкторам.

Динамический реестр типов

До сих пор мы предполагали, что весь список типов известен на этапе компиляции и может быть представлен в виде таблицы соответствия между типами и их идентификаторами (которая в нашем примере была реализована с помощью предложения `switch`). Это требование неизбежно в контексте программы в целом: т. к. вызов каждого конструктора где-то присутствует явно, полный список допускающих конструирование типов известен во время компиляции. Однако наше решение налагает дополнительное ограничение – весь список типов зашит в коде фабричного метода. Невозможно создать новые производные классы, не включив их в Фабрику. Иногда это ограничение не так плохо, как кажется; например, список здания в игре изменяется не слишком часто, а когда все-таки изменяется, полный перечень необходимо вручную обновить, чтобы строилось правильное меню, дабы в нужных местах появлялись картинки и воспроизводились звуки и т. д.

Тем не менее одно из преимуществ иерархического проектирования состоит в том, что производные классы можно добавлять позже, не изменяя ранее написанный код. Новая виртуальная функция просто встраивается в уже существующий поток управления и предоставляет необходимое поведение. Эту идею можно реализовать и в фабричном конструкторе.

Прежде всего каждый производный класс должен отвечать за конструирование себя. Это необходимо, потому что, как мы выяснили, явный вызов кон-

структора где-то должен присутствовать. Если его нет в общем коде, значит, он должен находиться в коде, добавляемом при создании нового производного класса:

```
class Farm : public Building {
    public:
    static Building* MakeBuilding() { return new Farm; }
};
class Forge : public Building {
    public:
    static Building* MakeBuilding() { return new Forge; }
};
```

Во-вторых, список типов должен быть расширяемым на этапе выполнения, а не фиксированным на этапе компиляции. Можно по-прежнему использовать enum, но тогда его нужно будет обновлять при каждом добавлении нового производного класса. Альтернативно можно было бы назначать каждому производному классу целочисленный идентификатор во время выполнения, позаботившись о том, чтобы идентификаторы были уникальными. В любом случае нам необходимо отображать эти идентификаторы на фабричные функции, и это нельзя сделать с помощью предложения switch или еще чего-то, фиксированного на этапе компиляции. Отображение должно быть расширяемым. Можно было бы использовать в качестве реестра всех типов зданий таблицу указателей на функции и хранить ее, например, в векторе:

```
class Building;
typedef Building* (*BuildingFactory)();

int building_type_count = 0;

std::vector<std::pair<int, BuildingFactory>> building_registry;
void RegisterBuilding(BuildingFactory factory) {
    building_registry.push_back(std::make_pair(building_type_count++,
                                factory));
}
```

Эта таблица – глобальный объект в программе, одиночка (синглтон). Ее можно было бы сделать просто глобальным объектом, как в примере выше, или воспользоваться одним из вариантов паттерна Одиночка, который будет описан в главе 15. Чтобы гарантировать уникальность идентификаторов типов зданий, мы храним глобальный счетчик типов и увеличиваем его на единицу при добавлении нового типа в реестр.

Теперь нужно только добавлять новые типы в реестр. Эта операция состоит из двух шагов. Сначала нужно добавить метод регистрации в каждый класс здания:

```
class Farm : public Building {
    public:
    static void Register() {
        RegisterBuilding(Farm::MakeBuilding);
    }
```

```
};
class Forge : public Building {
    public:
    static void Register() {
        RegisterBuilding(Forge::MakeBuilding);
    }
};
```

Затем нужно сделать так, чтобы все методы `Register()` вызывались до начала игры. Для этого есть несколько способов. Можно было бы зарегистрировать все типы в процессе статической инициализации, до вызова `main()`, но такой подход чреват ошибками, потому что порядок статической инициализации в различных единицах компиляции не определен. С другой стороны, в игре должно быть место, где инициализируются и рисуются элементы пользовательского интерфейса для каждого здания, которое может создать игрок, и именно там естественно было бы зарегистрировать и фабричные методы:

```
Farm::Register();   // Farm получает ID 0
Forge::Register();  // Forge получает ID 1
```

Заметим, что если тот же реестр-одиночка используется для других целей, например для графических образов зданий, то он должен быть инкапсулирован в отдельном классе.

Зарегистрировав все типы и их фабричные конструкторы, мы можем реализовать главный фабричный метод, который будет просто делегировать вызов фабричному конструктору нужного типа:

```
class Building {
    public:
    static Building* MakeBuilding(int building_type) {
        BuildingFactory factory = building_registry[building_type].second;
        return factory();
    }
};
Building* b0 = Building::MakeBuilding(0);    // это Farm
Building* b1 = Building::MakeBuilding(1);    // а это Forge
```

Теперь обращение к функции `Building::MakeBuilding(i)` будет конструировать объект того типа, который зарегистрирован под номером `i`. В нашем решении соответствие между значениями идентификаторов и типами неизвестно до начала выполнения – мы не можем сказать, какое здание имеет идентификатор 5, пока не запустим программу. Если это нежелательно, то можно назначить каждому типу статический идентификатор, но тогда нужно будет гарантировать их уникальность. Или можно было бы устроить так, чтобы обращения к регистрации всегда происходили в одном и том же порядке.

Заметим, что эта реализация очень похожа на код, генерируемый компилятором для настоящих виртуальных функций – вызовы виртуальных функций производятся через указатели на функции, которые хранятся в таблицах, каждая из которых доступна по уникальному идентификатору (виртуальному

указателю). Основное различие заключается в том, что уникальный идентификатор не может находиться в классе, поскольку мы должны использовать его до того, как объект сконструирован. Тем не менее это настолько близко к *виртуальному конструктору*, насколько вообще возможно.

В хорошо написанной программе на C++ должно быть понятно, кто владеет каждым объектом. Чаще всего встречается случай, когда имеется один очевидный владелец, что можно выразить и одновременно гарантировать с помощью владеющего указателя, например `std::unique_ptr`. Фабрика должна возвращать владеющий указатель, а клиент должен сохранить результат в другом владеющем указателе:

```
class Building {
    public:
    static std::unique_ptr<Building> MakeBuilding(int building_type) {
        BuildingFactory factory = building_registry[building_type].second;
        return std::unique_ptr<Building>(factory());
    }
};
std::unique_ptr<Building> b0 = Building::MakeBuilding(0);
std::unique_ptr<Building> b1 = Building::MakeBuilding(1);
```

Во всех рассмотренных выше фабричных конструкторах решение о том, какой объект конструировать, принималось на основе данных, внешних по отношению к программе, а для конструирования вызывался один и тот же фабричный метод (возможно, делегирующий работу производным классам). Теперь мы рассмотрим другой вариант Фабрики, применяемый в несколько иной ситуации.

Полиморфная фабрика

Рассмотрим слегка отличающуюся задачу – пусть в нашей игре каждое здание производит сущности некоторого вида, и тип сущности однозначно связан с типом здания. Замок (Castle) выпускает рыцарей, Башня магов (Wizard Tower) обучает волшебников, а Паучий холм (Spider Mound) порождает гигантских пауков. Теперь наш общий код не только конструирует здание выбранного во время выполнения типа, но и создает новые сущности, типы которых тоже неизвестны на этапе компиляции. Фабрика зданий у нас уже есть. Можно было бы реализовать аналогичную Фабрику сущностей и ассоциировать с каждым зданием уникальный идентификатор сущности. Но такой дизайн раскрывает связь между зданиями и сущностями всей программе, а это совсем не обязательно – каждое здание знает, как создать *правильную* сущность, и нет никаких причин сообщать об этом другим частям программы.

Для этой задачи проектирования нужна несколько иная Фабрика – фабричный метод определяет, что создается некоторая сущность, но какая именно, решает само здание. Это паттерн Шаблонный метод в сочетании с паттерном Фабрика – в целом дизайн основан на Фабрике, но тип сущности настраивается производными классами:

```
class Unit {};
class Knight : public Unit { ... };
class Mage : public Unit { ... };
class Spider : public Unit { ... };

class Building {
    public:
    virtual Unit* MakeUnit() const = 0;
};
class Castle : public Building {
    public:
    Knight* MakeUnit() const { return new Knight; }
};
class Tower : public Building {
    public:
    Mage* MakeUnit() const { return new Mage; }
};
class Mound : public Building {
    public:
    Spider* MakeUnit() const { return new Spider; }
};

Building* building = MakeBuilding(identifier);
Unit* unit = building->MakeUnit();
```

Фабричные методы для самих Фабрик в этом примере не показаны – Фабрика сущностей может сосуществовать с любой рассмотренной выше реализацией Фабрики зданий. Общий код, показанный в конце листинга, пишется один раз и не изменяется при добавлении новых производных классов для зданий и сущностей.

Заметим, что функции `MakeUnit()` возвращают разные типы. Однако же все они переопределяют одну и ту же виртуальную функцию `Building::MakeUnit()`. Это называется *ковариантными возвращаемыми типами* – тип, возвращаемый переопределенным методом, может быть производным от типа, который возвращает переопределенный им метод. В нашем случае возвращаемые типы совпадают с типами классов, но, вообще говоря, это не обязательно. В качестве ковариантных типов можно использовать любые базовые и производные классы, даже из разных иерархий. Однако ковариантность разрешена только для таких типов, и, за исключением этого случая, тип, возвращаемый переопределенной функцией, должен совпадать с типом виртуальной функции, определенным в базовом классе.

Строгие правила для ковариантных возвращаемых типов представляют некоторую проблему, когда мы пытаемся написать Фабрику, возвращающую что-то, кроме простого указателя. Например, предположим, что мы хотим вернуть `std::unique_ptr` вместо простого указателя. Но, в отличие от `Unit*` и `Knight*`, типы `std::unique_ptr<Unit>` и `std::unique_ptr<Knight>` не являются ковариантными и потому не могут использоваться как типы, возвращаемые виртуальной функцией и функцией, переопределяющей ее.

В следующем разделе мы рассмотрим решения этой и еще нескольких специфичных для C++ проблем, относящихся к фабричным методам.

Похожие на Фабрику паттерны в C++

В C++ используется много вариаций на тему основного паттерна Фабрика для удовлетворения специфических проектных потребностей и ограничений. В этом разделе мы рассмотрим некоторые из них. Это ни в коем случае не исчерпывающий перечень фабрикоподобных паттернов в C++, но понимание описанных вариантов должно подготовить читателя к сочетанию приемов, описанных в этой книге, на случай столкновения с разнообразными вызовами, относящимися к Фабрикам объектов.

Полиморфное копирование

До сих пор мы рассматривали фабричные альтернативы конструктору объектов – без аргументов или с аргументами. Но похожий паттерн применим и к копирующему конструктору, когда имеется объект и надо получить его копию.

Эта задача похожа на предыдущую во многих отношениях – у нас имеется объект, доступный по указателю на базовый класс, и мы хотим вызвать его копирующий конструктор. По причинам, обсуждавшимся выше, и не в последнюю очередь потому, что компилятору нужно знать, сколько памяти выделить, вызвать конструктор можно только для статически определенного типа. Однако поток управления, приведший к вызову конструктора, может стать известен только на этапе выполнения, поэтому нам снова необходим паттерн Фабрика.

Фабричный метод, который мы будем использовать для реализации полиморфного копирования, в чем-то напоминает пример Фабрики сущностей из предыдущего раздела – фактическое конструирование должно выполняться в каждом производном классе, а производный класс знает, объект какого типа конструировать. Базовый класс реализует поток управления, который понуждает сконструировать копию чего-то, а производный класс подставляет собственно конструирование:

```
class Base {
    public:
    virtual Base* clone() const = 0;
};
class Derived : public Base {
    public:
    Derived* clone() const { return new Derived(*this); }
};

Base* b = ... получить откуда-то объект ...
Base* b1 = b->clone();
```

Мы снова используем ковариантные возвращаемые типы и потому вынуждены ограничиться простыми указателями.

Предположим, что вместо этого хочется возвращать уникальные указатели. Поскольку ковариантными считаются только простые указатели на базовый и производный классы, мы всегда должны возвращать уникальный указатель на базовый класс:

```
class Base {
public:
    virtual std::unique_ptr<Base> clone() const = 0;
};
class Derived : public Base {
    public:
    std::unique_ptr<Base> clone() const {          // не unique_ptr<Derived>
        return std::unique_ptr<Base>(new Derived(*this));
    }
};

std::unique_ptr<Base> b(...);
std::unique_ptr<Base> b1 = b->clone();
```

Во многих случаях это ограничение несущественно. Но иногда оно приводит к лишним преобразованиям и приведениям. На случай, когда возврат интеллектуального указателя на точный тип важен, имеется другая вариация этого паттерна, которую мы рассмотрим ниже.

CRTP-фабрика и возвращаемые типы

Единственный способ вернуть `std::unique_ptr<Derived>` из фабричного копирующего конструктора производного класса – сделать так, чтобы виртуальный метод `clone()` базового класса возвращал такой же тип. Но это невозможно, по крайней мере если производных классов больше одного, – для каждого производного класса нужно, чтобы тип, возвращаемый методом `Base::clone()`, совпадал с этим классом. Но метод `Base::clone()` только один! А так ли это? По счастью, в C++ есть простой способ получить много из одного – шаблоны. Если превратить базовый класс в шаблон, то мы могли бы сделать так, что базовый класс каждого производного класса будет возвращать правильный тип. Но для этого нужно, чтобы базовый класс каким-то образом мог узнать тип производного от него класса. Однако, разумеется, и для этого есть паттерн – в C++ он называется Рекурсивным шаблоном (CRTP), и мы рассматривали его в главе 8. Теперь можно объединить паттерны CRTP и Фабрика:

```
template <typename Derived> class Base {
    public:
    virtual std::unique_ptr<Derived> clone() const = 0;
};
class Derived : public Base<Derived> { // благодаря CRTP Base знает о Derived
    public:
    std::unique_ptr<Derived> clone() const {
        return std::unique_ptr<Derived>(new Derived(*this));
    }
};
```

```
std::unique_ptr<Derived> b0(new Derived);
std::unique_ptr<Derived> b1 = b0->clone();
```

Возвращаемый тип auto позволяет писать подобный код гораздо лаконичнее. В этой книге мы нечасто используем его, чтобы было видно, что именно возвращает каждая функция. Заметим, что поскольку класс Base теперь знает тип производного класса, даже нет нужды делать метод clone() виртуальным:

```
template <typename Derived> class Base {
    public:
    std::unique_ptr<Derived> clone() const {
        return std::unique_ptr<Derived>(
            new Derived(*static_cast<const Derived*>(this)));
    }
};
class Derived : public Base<Derived> { // благодаря CRTP Base знает о Derived
    ...
};
```

У этого способа есть существенные недостатки, по крайней мере в том виде, в каком он сейчас реализован. Во-первых, нам пришлось сделать базовый класс шаблоном, а это означает, что в нашем общем коде больше нет общего типа указателя (или мы должны будем использовать шаблоны еще шире). Во-вторых, этот метод работает, только если от Derived не произведено других классов, потому что тип базового класса не прослеживает второй уровень наследования – только тот, которым был конкретизирован шаблон Base. В общем, если не считать нескольких специальных случаев, когда очень важно возвращать точный тип, а не базовый, этот подход не рекомендуется.

С другой стороны, эта реализация обладает кое-какими привлекательными чертами, которые мы хотели бы сохранить. Точнее, мы избавились от нескольких копий функции clone(), по одной в каждом производном классе, и заставили шаблон генерировать их автоматически. В следующем разделе мы покажем, как сохранить эту полезную особенность реализации на основе CRTP, даже если придется отказаться от обобщения понятия *ковариантных возвращаемых типов* на интеллектуальные указатели посредством трюков с шаблонами.

CRTP-фабрика с меньшим объемом копирования и вставки

Теперь мы сосредоточимся на том, как с помощью CRTP избежать включения функции clone() в каждый производный класс. Это делается не просто для того, чтобы меньше стучать по клавишам. Чем больше написано кода, особенно похожего, который копируется из одного места и вставляется в другое с небольшими модификациями, тем больше шансов допустить ошибку. Мы уже видели, как использовать CRTP для автоматического генерирования варианта функции clone() в каждом производном классе. Мы просто не хотим отказываться от обычного (нешаблонного) базового класса ради этого. Но и не придется, если делегировать клонирование специальному базовому классу, который только это и будет делать:

```
class Base {
    public:
    virtual Base* clone() const = 0;
};
template <typename Derived> class Cloner : public Base {
    public:
    Base* clone() const {
        return new Derived(*static_cast<const Derived*>(this));
    }
};
class Derived : public Cloner<Derived> {
    ...
};

Base* b0(new Derived);
Base* b1 = b0->clone();
```

Здесь мы для простоты вернулись к возврату простых указателей, хотя можно было бы возвращать и `std::unique_ptr<Base>`. Чего нельзя сделать, так это вернуть `Derived*`, поскольку в тот момент, когда компилятор разбирает шаблон `Cloner`, еще неизвестно, что `Derived` всегда наследует `Base`.

При таком подходе мы можем произвести сколько угодно классов от `Base`, косвенно через `Cloner`, и больше ни разу не должны будем писать функцию `clone()`. Остается ограничение – если произвести еще один класс от `Derived`, то он будет копироваться неправильно. Есть много проектных решений, в которых это не проблема – разумный эгоизм подсказывает избегать глубоких иерархий и создавать классы только двух видов: абстрактные базовые классы, экземпляры которых никогда не создаются, и конкретные классы, производные от этих базовых, но ни в коем случае не от других конкретных классов. Но если разумного эгоизма недостаточно, то можно применить другой вариант CRTP-фабрики, чтобы сократить объем копирования кода даже в глубоких иерархиях:

```
class Base {
    public:
    virtual ~Base() {}
    Base* clone() const;
};

class ClonerBase {
    public:
    virtual Base* clone() const = 0;
};

Base* Base::clone() const {
    dynamic_cast<const ClonerBase*>(this)->clone(); // перекрестное приведение
};

template <typename Derived> class Cloner : public ClonerBase {
    public:
    Base* clone() const {
```

```
        return new Derived(*static_cast<const Derived*>(this));
    }
};

class Derived : public Base,
                public Cloner<Derived> { // множественное наследование
    ...
};

Base* b0(new Derived);
Base* b1 = b0->clone();
```

Это показательный пример того, что, сказав А, надо говорить и Б. Раз уж мы решили баловаться сложностью и использовать глубокие иерархии, то придется насладиться всеми прелестями множественного наследования и сложного динамического приведения. Любой производный класс, не важно, наследует он `Base` напрямую или нет, является производным также от `Cloner<Derived>`, который, в свою очередь, наследует `ClonerBase`. Таким образом, каждый класс `Derived` получает соответствующий базовый класс с модифицированным виртуальным методом `clone()` – все благодаря любезности CRTP. Все эти методы `clone()` переопределяют один и тот же метод базового класса `ClonerBase`, поэтому мы должны перейти от `Base` к `ClonerBase`, хотя они даже не принадлежат одной иерархии. Странный `dynamic_cast`, который решает эту задачу, называется *перекрестным приведением* – он осуществляет приведение от одного базового класса объекта к другому базовому классу того же объекта. Если во время выполнения выяснится, что фактический производный объект имеет тип, не являющийся комбинацией обоих базовых классов, то `dynamic_cast` завершится неудачно и вернет `NULL` (или возбудит исключение, если мы приводим не указатели, а ссылки). Надежная программа должна проверять успешность перекрестного приведения и обрабатывать возможные ошибки.

Резюме

В этой главе мы узнали, почему конструкторы не могут быть виртуальными и что делать, если виртуальный конструктор все-таки нужен. Мы научились конструировать и копировать объекты, тип которых становится известен только во время выполнения, – с помощью паттерна Фабрика и его вариаций. Мы также изучили несколько реализаций фабричного конструктора, различающихся организацией кода и тем, какое поведение делегируется различным компонентам системы, и сравнили их достоинства и недостатки. Кроме того, мы видели, как несколько паттернов проектирования взаимодействуют между собой.

Хотя в C++ при вызове конструктора всегда должен быть известен истинный тип конструируемого объекта, это не означает, что код приложения должен указать полный тип. Паттерн Фабрика позволяет писать код, где тип задается косвенно, посредством идентификатора, который ассоциирован с типом, объ-

явленным где-то еще (*создай объект такого вида*), или ассоциирован с типом другого объекта (*создай сущность, связанную с типом этого здания*), или даже совпадает с типом указанного объекта (*дай мне копию вот этого, чем бы оно ни было*).

В следующей главе мы будем изучать паттерн Шаблонный метод – один из классических объектно-ориентированных паттернов, который в C++ имеет дополнительные последствия для способа проектирования иерархий классов.

Вопросы

- Почему в C++ не разрешены виртуальные конструкторы?
- Что такое паттерн Фабрика?
- Как паттерн Фабрика используется для создания эффекта виртуального конструктора?
- Как добиться эффекта виртуального копирующего конструктора?
- Как паттерны Шаблонный метод и Фабрика используются совместно?

Глава 14

Паттерн Шаблонный метод и идиома невиртуального интерфейса

Шаблонный метод – один из классических паттернов проектирования *Банды четырех*, или, более формально, один из 23 паттернов, описанных в книге Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides «Design Patterns – Elements of Reusable Object-Oriented Software»[1]. Это поведенческий паттерн, т. е. он описывает способ взаимодействия различных объектов. Будучи объектно-ориентированным языком, C++, конечно, в полной мере поддерживает паттерн Шаблонный метод. Но есть некоторые детали реализации, специфичные или даже уникальные для C++, которые мы осветим в данной главе.

В этой главе рассматриваются следующие вопросы:

- что такое паттерн Шаблонный метод и какие задачи он решает;
- что такое невиртуальный интерфейс;
- следует ли делать виртуальные функции открытыми, закрытыми или защищенными по умолчанию;
- всегда ли надо делать деструкторы виртуальными и открытыми в полиморфных классах.

Технические требования

Для чтения этой главы не требуется никаких инструментов или технических знаний.

---

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. М.: ДМК-Пресс; Питер, 2019.

Паттерн Шаблонный метод

Паттерн Шаблонный метод – это общий способ реализовать алгоритм, общая структура которого определена заранее, но некоторые детали реализации необходимо настраивать. Если вам приходит в голову примерно такое решение – сначала сделать *X*, потом *Y*, затем *Z*, но что конкретно делает *Y*, зависит от обрабатываемых данных, – то это как раз и есть Шаблонный метод. Будучи паттерном, который допускает динамическое изменение поведения программы, Шаблонный метод чем-то напоминает паттерн Стратегия. Ключевое различие заключается в том, что Стратегия во время выполнения изменяет алгоритм целиком, а Шаблонный метод позволяет модифицировать отдельные части алгоритма. В этом разделе мы будем рассматривать именно Шаблонный метод, а Стратегию отложим до главы 16.

Шаблонный метод в C++

Паттерн Шаблонный метод легко реализуется на любом объектно-ориентированном языке. В C++ для реализации используются наследование и виртуальные функции. Заметим, что этот паттерн не имеет ничего общего с шаблонами C++ как средством обобщенного программирования. Здесь под *шаблоном* понимается эскиз алгоритма:

```
class Base {
    public:
    bool TheAlgorithm() {
        if (!Step1()) return false; // шаг 1 завершился неудачно
        Step2();
        return true;
    }
};
```

Шаблон определяет общую структуру алгоритма – любая реализация должна сначала выполнить шаг 1, который может завершиться неудачно. Если такое происходит, то весь алгоритм завершается неудачно, и больше ничего делать не надо. Если шаг 1 завершился успешно, то следует выполнить шаг 2. По определению, шаг 2 не может завершиться неудачно, и после его завершения считается, что весь алгоритм завершился успешно.

Заметим, что метод `TheAlgorithm()` открытый, но не виртуальный – любой класс, производный от `Base`, наследует его как часть своего интерфейса, но не может переопределить. А переопределить производные классы могут реализации шага 1 и шага 2, соблюдая ограничения шаблона алгоритма – шаг 1 может завершиться неудачно и должен известить об этом, вернув `false`, а шаг 2 всегда завершается успешно:

```
class Base {
    public:
    ...
    virtual bool Step1() { return true };
```

```
    virtual void Step2() = 0;
};
class Derived1 : public Base {
    public:
    void Step2() { ... поработать ... }
};
class Derived2 : public Base {
    public:
    bool Step1() { ... проверить предусловия ... }
    void Step2() { ... поработать ... }
};
```

В этом примере переопределение шага 1, в котором может возникнуть ошибка, необязательно, а реализация по умолчанию тривиальна – она ничего не делает и всегда завершается успешно. Шаг 2 обязательно должен быть переопределен в любом производном классе – никакой реализации по умолчанию нет, и он объявлен как чисто виртуальная функция.

Как видим, общий поток управления – каркас – остается инвариантным, но имеются *места для подстановки* настраиваемых шагов, возможно, в виде реализаций по умолчанию, предлагаемых самим каркасом. Такой поток называется *инверсией управления*. В традиционном потоке управления конкретная реализация определяет поток вычислений и последовательность операций, она сама обращается к библиотечным функциям и другим низкоуровневым средствам для реализации общего алгоритма. Напротив, в Шаблонном методе каркас вызывает конкретные реализации, предоставленные пользовательским кодом.

Применения Шаблонного метода

Есть много причин использовать Шаблонный метод. Вообще говоря, он применяется для того, чтобы провести границу между тем, что можно и нельзя изменять в подклассах. В противоположность общему полиморфному переопределению, когда заменить можно виртуальную функцию целиком, здесь базовый класс решает, что можно переопределить, а что нельзя. Еще одно типичное применение Шаблонного метода – предотвращение дублирования кода, и в таком контексте прийти к этому паттерну можно следующим образом. Предположим, что мы начали с обычного полиморфизма – виртуальной функции, допускающей переопределение. Рассмотрим, к примеру, следующий модельный дизайн игры, в которой персонажи по очереди наносят удары:

```
class Character {
    public:
    virtual void CombatTurn() = 0;
    protected:
    int health_;
};
class Swordsman : public Character {
    bool wielded_sword_;
```

```
    public:
    void CombatTurn() {
        if (health_ < 5) { // критическое повреждение
            Flee();
            return;
        }
        if (!wielded_sword_) {
            Wield();
            return; // вскидывание меча занимает весь ход
        }
        Attack();
    }
};
class Wizard : public Character {
    int mana_;
    bool scroll_ready_;
    public:
    void CombatTurn() {
        if (health_ < 2 ||
            mana_ == 0) { // критическое повреждение или кончилась волшебная сила
            Flee();
            return;
        }
        if (!scroll_ready_) {
            ReadScroll();
            return;       // чтение свитка занимает весь ход
        }
        CastSpell();
    }
};
```

В этом коде много повторов – каждый персонаж может быть вынужден покинуть поле битвы в свой ход, затем он должен подготовиться к битве и только потом, если готов и достаточно силен, может воспользоваться своими боевыми навыками. Если такая схема повторяется снова и снова, то самое время подумать о Шаблонном методе. Этот паттерн полезен, когда общая последовательность проведения поединка фиксирована, но то, как персонажи переходят к следующему шагу и что делают на этом шаге, зависит от персонажа:

```
class Character {
    public:
    void CombatTurn() {
        if (MustFlee()) {
            Flee();
            return;
        }
        if (!Ready()) {
            GetReady();
            return;                 // Подготовка занимает целый ход
        }
        CombatAction();
```

```
    }
    virtual bool MustFlee() const = 0;
    virtual bool Ready() const = 0;
    virtual void GetReady() = 0;
    virtual void CombatAction() = 0;
    protected:
    int health_;
};
class Swordsman : public Character {
    bool wielded_sword_;
    public:
        bool MustFlee() const { return health_ < 5; } // Критическое повреждение
        bool Ready() const { return wielded_sword_; }
        void GetReady() { Wield(); }
        void CombatAction() { Attack(); }
    };
class Wizard : public Character {
    int mana_;
    bool scroll_ready_;
    public:
    bool MustFlee() const { return health_ < 2 ||      // Критическое повреждение
        mana_ == 0; }                                  // Кончилась волшебная сила
    bool Ready() const { return scroll_ready_; }
    void GetReady() { ReadScroll(); }
    void CombatAction() { CastSpell(); }
};
```

Теперь повторов стало куда меньше. Но приятным видом преимущества Шаблонного метода не исчерпываются. Предположим, что в следующей версии игры мы добавили целебные настои, и в начале своего хода персонаж может выпить настой. Теперь представьте, что в каждый производный класс нужно добавить код вида `if (health_ < ... зависящее от класса значение ... && potion_count_ > 0) ...`. Если в дизайне уже используется Шаблонный метод, то логику приема снадобья нужно закодировать только один раз, а разные классы будут по-своему реализовывать условия использования настоя, а также последствия его приема. Но не спешите с реализацией этого решения, не дочитав главу до конца, поскольку в C++ это не лучшее, что можно сделать.

Предусловия, постусловия и действия

Еще одно распространенное применение Шаблонного метода – обработка пред- и постусловий или действий. В иерархии классов пред- и постусловия обычно проверяют, что ни в какой точке выполнения реализация не нарушает инварианты абстракции, предоставляемой интерфейсом. Для такой проверки естественно прибегнуть к паттерну Шаблонный метод:

```
class Base {
    public:
    void VerifiedAction() {
        assert(StateIsValid());
```

```
        ActionImpl();
        assert(StateIsValid());
    }
    virtual void ActionImpl() = 0;
};
class Derived : public Base {
    public:
    void ActionImpl() { ... }
};
```

Разумеется, характер инвариантов – еще одна точка настройки дизайна программы. Иногда главный код не меняется, но что происходит до и после него, зависит от конкретного приложения. В таком случае мы, вероятно, не проверяем инварианты, а выполняем начальное и конечное действия:

```
class FileWriter {
    public:
    void Write(const char* data) {
        Preamble(data);
        ... записать данные в файл ...
        Postscript(data);
    }
    virtual void Preamble(const char* data) {}
    virtual void Postscript(const char* data) {}
};
class LoggingFileWriter : public FileWriter {
    void Preamble(const char* data) {
        std::cout << "Данные " << data << " записываются в файл" << std::endl;
    }
};
```

Невиртуальный интерфейс

Динамическая настройка частей шаблонного алгоритма обычно реализуется с помощью виртуальных функций. Вообще говоря, паттерн Шаблонный метод не требует этого, но в C++ другой способ редко бывает необходим. Сейчас мы сосредоточимся на использовании виртуальных функций и улучшим то, чему уже научились.

Виртуальные функции и контроль доступа

Начнем с общего вопроса – должны ли виртуальные функции быть открытыми или закрытыми? В учебниках по объектно-ориентированному проектированию используются открытые виртуальные функции, поэтому мы тоже часто так поступаем, не особенно задумываясь. Но в паттерне Шаблонный метод эту практику стоит переосмыслить, поскольку открытая функция – часть интерфейса класса. В нашем случае интерфейс класса включает весь алгоритм и каркас, который мы поместили в базовый класс. Эта функция должна быть открытой, но невиртуальной. Модифицируемые реализации некоторых частей

алгоритма не рассчитаны на прямой вызов клиентами иерархии классов. Они используются только в одном месте – в невиртуальной открытой функции, где переопределяют реализации по умолчанию, присутствующие в шаблоне алгоритма.

Эта идея может показаться тривиальной, но для многих программистов оказывается откровением. Я не раз задавал вопрос: «Могут *ли* виртуальные функции в C++ не быть открытыми?» На самом деле язык не налагает никаких ограничений на уровень доступа к виртуальным функциям; они могут быть закрытыми, защищенными или открытыми, как и любые другие функции-члены класса. Требуется время, чтобы уложить эту мысль в голове; возможно, пример поможет:

```
class Base {
    public:
    void method1() { method2(); }
    private:
    virtual void method2() { ... }
};
class Derived : public Base {
    private:
    virtual void method2() { ... }
};
```

Здесь метод `Derived::method2()` закрытый. Но разве базовый класс может вызывать закрытые методы производных от него классов? Ответ такой – ему и не нужно: метод `Base::method2()` вызывается только из `Base::method1()`, а это закрытая функция-член того же класса, но ведь никто не запрещает классу вызывать свои собственные закрытые функции-члены. Однако если фактический тип класса – `Derived`, то во время выполнения будет вызвана переопределенная в нем функция `method2()`. Эти два решения – *могу ли я вызвать функцию* `method2()`*?* и *какую именно* `method2()`*?* – принимаются в разное время: первое – на этапе компиляции модуля, содержащего класс `Base` (а класс `Derived` к этому моменту, возможно, еще и не написан), а второе – на этапе выполнения программы (и в этот момент слова `private` и `public` уже ничего не значат).

Есть и еще одна, более фундаментальная причина избегать открытых виртуальных функций. Открытый метод – часть интерфейса класса. Переопределение виртуальной функции – это настройка реализации. Открытая виртуальная функция по существу играет обе эти роли сразу. Одна и та же программная сущность решает две совершенно разные задачи, которые не должны быть связаны: объявление открытого интерфейса и предоставление альтернативной реализации. У этих задач различные ограничения – реализацию можно изменять любым способом, коль скоро инварианты иерархии соблюдаются. Но виртуальная функция не может изменить интерфейс (если не считать возврата ковариантных типов, однако это в действительности не есть изменение интерфейса). Единственное, что делает открытая виртуальная функция, – подтверждает, что да, открытый интерфейс по-прежнему выглядит, как объявлено

в базовом классе. Такое смешение двух очень разных ролей наводит на мысль о необходимости лучшего разделения обязанностей. Паттерн Шаблонный метод – ответ на эту проблему проектирования, и в C++ он принимает форму идиомы **невиртуального интерфейса** (Non-Virtual Interface – **NVI**).

Идиома NVI в C++

Противоречие между двумя ролями открытой виртуальной функции и необходимость раскрытия точек настройки, обусловленная такими функциями, приводят нас к идее сделать виртуальные функции, содержащие реализацию, закрытыми. Герб Саттер (Herb Sutter) в статье «Virtuality» (`http://www.gotw.ca/publications/mill18.htm`) высказывает предположение, что многие, если не все, виртуальные функции должны быть закрытыми.

В случае Шаблонного метода перемещение виртуальных функций из открытой секции класса в закрытую не влечет за собой никаких последствий (не считая кратковременного шока от встречи с закрытой виртуальной функцией у тех, кто не знал, что C++ их допускает):

```
class Base {
    public:
    bool TheAlgorithm() {
        if (!Step1()) return false; // шаг 1 завершился неудачно
        Step2();
        return true;
    }
    private:
    virtual bool Step1() { return true };
    virtual void Step2() = 0;
};
class Derived1 : public Base {
    private:
    void Step2() { ... поработать ... }
};
class Derived2 : public Base {
    private:
    bool Step1() { ... проверить предусловия ... }
    void Step2() { ... поработать ... }
};
```

В этом дизайне интерфейс и его реализация изящно разделены – клиентский интерфейс состоит и всегда состоял из одного вызова алгоритма в целом. Возможность изменять части реализации алгоритма не отражена в интерфейсе, но пользователю иерархии классов, который обращается к ней только через открытый интерфейс и не собирается расширять иерархию (писать дополнительные производные классы), об этом и знать не нужно.

Идиома NVI дает полный контроль над интерфейсом базового класса. Производный класс может только настраивать реализацию этого интерфейса. Базовый класс определяет и проверяет инварианты, фиксирует общую структуру реализации и специфицирует, какие части можно, какие должно, а какие

нельзя модифицировать. NVI также явно разделяет интерфейс и реализацию. Программисты, реализующие производные классы, могут не думать о том, как бы по оплошности не раскрыть части реализации вызывающей стороне, – закрытые методы, относящиеся только к реализации, не может вызывать никто, кроме базового класса.

До сих пор мы делали все виртуальные функции, настраивающие реализацию, закрытыми. Но не в этом состоит суть NVI – задача этой идиомы, как и более общего паттерна Шаблонный метод, – сделать открытый интерфейс невиртуальным. Отсюда, в частности, следует, что зависящие от реализации переопределенные функции не должны быть открытыми, поскольку не являются частью интерфейса. Но это не означает, что они обязательно должны быть закрытыми. Остается еще одна возможность – *защищенные*. Итак, должны ли виртуальные функции, обеспечивающие настройку алгоритма, быть закрытыми или защищенными? Шаблонный метод допускает то и другое – клиент иерархии не может напрямую вызывать ни закрытую, ни защищенную функцию, так что на каркас алгоритма это не влияет. Ответ зависит от того, возникает ли у производных классов необходимость вызывать реализации, предоставленные базовым классом. В качестве примера рассмотрим иерархию классов, которые можно сериализовать и передать удаленной машине через сокет:

```
class Base {
    public:
    void Send() {    // здесь используется Шаблонный метод
        ... открыть соединение ...
        SendData();
        ... закрыть соединение ...
    }
    protected:
    virtual void SendData() { ... отправить данные базового класса ... }
    private:
    ... данные ...
};
class Derived : public Base {
    protected:
    void SendData() {
        ... отправить данные производного класса ...
        Base::SendData();
    }
};
```

Здесь каркас предоставляет открытый невиртуальный метод `Base::Send()`, который отвечает за реализацию коммуникационного протокола и в нужное время отправляет по сети данные. Конечно, он может отправить только данные, о которых знает базовый класс. Именно поэтому метод `SendData` является точкой настройки и сделан виртуальным. Естественно, производный класс должен сам отправлять свои данные, но кому-то же нужно отправить и данные базового класса, поэтому производный класс обращается к защищенной виртуальной функции, находящейся в базовом классе.

Замечание о деструкторах

Вся дискуссия по поводу NVI представляет собой уточнение простой рекомендации – делать виртуальные функции закрытыми (или защищенными) и описывать открытый интерфейс в терминах невиртуальных функций базового класса. Звучит прекрасно, пока не сталкивается с другой хорошо известной рекомендацией – если в классе имеется хотя бы одна виртуальная функция, его деструктор тоже должен быть виртуальным. Поскольку налицо конфликт, необходимы пояснения.

Зачем мы делаем деструкторы виртуальными? Затем, что если объект удаляется полиморфно, например объект производного класса удаляется через указатель на базовый класс, то деструктор должен быть виртуальным, иначе будет уничтожена только часть объекта, относящаяся к базовому классу (результатом обычно является *расслоение* класса, т. е. частичное удаление, хотя стандарт просто утверждает, что поведение не определено). Итак, если объекты удаляются через указатель на базовый класс, то деструктор должен быть виртуальным – и точка. Но никаких других причин нет. Если объекты всегда удаляются через указатель на истинный производный тип, то эта причина неприменима. А такая ситуация не является необычной; например, если объекты производного класса хранятся в контейнере, то при удалении будет указан их истинный тип.

Контейнер должен знать, сколько памяти выделить под объект, поэтому в нем нельзя хранить одновременно объекты базового и производного классов или удалять объекты как базовые (заметим, что контейнер указателей на базовый класс – совершенно другая конструкция, которая обычно создается специально для того, чтобы хранить и удалять объекты полиморфно).

Итак, если объект производного класса удаляется как таковой, то его деструктор не обязан быть виртуальным. Однако если кто-то все же вызовет деструктор базового класса, когда объект фактически принадлежит производному классу, то неприятностей не миновать. Чтобы предотвратить такое развитие событий, мы можем объявить невиртуальный деструктор базового класса защищенным, а не открытым. Конечно, если базовый класс не абстрактныйи в программе существуют объекты и производного, и базового классов, то оба деструктора должны быть открытыми, и тогда безопаснее сделать их виртуальными (можно во время выполнения проверять, что деструктор базового класса не вызывается для уничтожения объекта производного).

Следует также предостеречь читателя от использования Шаблонного метода или идиомы невиртуального интерфейса для деструкторов класса. Быть может, когда-нибудь возникнет соблазн написать такой код:

```
class Base {
    public:
    ~Base() {          // невиртуальный интерфейс!
        std::cout << "Производится удаление" << std::endl;
        clear();       // здесь применяется Шаблонный метод
        std::cout << "Удаление завершено" << std::endl;
```

```
    }
    protected:
    virtual void clear() { ... } // настраиваемая часть
};
class Derived : public Base {
    private:
    void clear() {
        ...
        Base::clear();
    }
};
```

Однако это работать не будет (если в базовом классе имеется чисто виртуальный метод `Base::clear()` вместо реализации по умолчанию, то работать все равно не будет, но весьма своеобразно). Причина в том, что внутри деструктора базового класса, `Base::~Base()`, настоящий, истинный, фактический тип объекта – уже не `Derived`. А `Base`. И это правильно – когда деструктор `Derived::~Derived()` заканчивает свою работу и управление передается деструктору базового класса, динамический тип объекта изменяется на `Base`.

Есть еще только один член класса, который ведет себя подобным образом, – конструктор; объект имеет тип `Base`, пока работает конструктор базового класса, а когда начинает работать конструктор производного класса, его тип меняется на `Derived`. Во всех остальных функциях-членах объект имеет тот тип, с которым был создан. Если объект был создан как `Derived`, то он будет иметь этот тип, даже если вызывался метод базового класса. А что произойдет, если в предыдущем примере метод `Base::clear()` чисто виртуальный? Он все равно вызывается! Результат зависит от компилятора; большинство компиляторов генерирует код, который аварийно завершает программу с диагностическим сообщением «была вызвана чисто виртуальная функция».

Недостатки невиртуального интерфейса

У идиомы NVI недостатков не много. Именно поэтому рекомендация всегда делать виртуальные функции закрытыми и использовать для их вызова NVI получила такое широкое признание. Однако есть некоторые соображения, о которых следует знать, принимая решение о том, подходит паттерн Шаблонный метод в конкретной ситуации или нет. Использование этого паттерна может привести к хрупким иерархиям. Кроме того, имеется ряд задач проектирования, которые можно решить как применением Шаблонного метода, так и (быть может, даже лучше) паттерна Стратегия, или, в терминах C++, Политика. В этом разделе мы рассмотрим оба соображения.

Компонуемость

Рассмотрим показанный выше дизайн класса `LoggingFileWriter`. Предположим теперь, что нам также нужен класс `CountingFileWriter`, который подсчитывает, сколько символов было записано в файл:

```
class CountingFileWriter : public FileWriter {
    size_t count_;
    void Preamble(const char* data) {
        count_ += strlen(data);
    }
};
```

Пока все просто. Но почему бы подсчитывающему писателю не заняться также протоколированием? Как мы стали бы реализовывать класс `CountingLoggingFileWriter`? Да никаких проблем, у нас же есть технология – сделать закрытые виртуальные функции защищенными и вызвать реализацию базового класса из производного:

```
class CountingLoggingFileWriter : public LoggingFileWriter {
    size_t count_;
    void Preamble(const char* data) {
        count_ += strlen(data);
        LoggingFileWriter::Preamble(data);
    }
};
```

А может, это `LoggingCountingFileWriter` должен наследовать `CountingFileWriter`? Заметим, что при любом решении часть кода дублируется – в нашем случае код подсчета присутствует как в `CountingLoggingFileWriter`, так и в `CountingFileWriter`. И дублирование станет только хуже, если увеличить количество вариаций. Паттерн Шаблонный метод просто не подходит, если требуются настройки, допускающие композицию. О том, что делать в такой ситуации, читайте главу 16.

Проблема хрупкого базового класса

Проблема хрупкого базового класса касается не только Шаблонного метода, но до некоторой степени присуща всем объектно-ориентированным языкам. Возникает она, когда изменения в базовом классе делают неработоспособным производный. Чтобы понять, как это может случиться конкретно при использовании невиртуального интерфейса, вернемся к классу записи в файл и добавим возможность записывать сразу много строк:

```
class FileWriter {
    public:
    void Write(const char* data) {
        Preamble(data);
        ... записать данные в файл ...
        Postscript(data);
    }
    void Write(std::vector<const char*> huge_data) {
        Preamble(huge_data);
        for (auto data: huge_data) { ... записать данные в файл ... }
        Postscript(huge_data);
    }
    private:
    virtual void Preamble(std::vector<const char*> huge_data) {}
```

```
    virtual void Postscript(std::vector<const char*> huge_data) {}
    virtual void Preamble(const char* data) {}
    virtual void Postscript(const char* data) {}
};
```

В класс подсчитывающего писателя вносятся соответствующие изменения:

```
class CountingFileWriter : public FileWriter {
    size_t count_;
    void Preamble(std::vector<const char*> huge_data) {
        for (auto data: huge_data) count_ += strlen(data);
    }
    void Preamble(const char* data) {
        count_ += strlen(data);
    }
};
```

Пока все хорошо. Но позже какой-то программист, из самых лучших побуждений, замечает, что в базовом классе присутствует некоторое дублирование кода, и решает подвергнуть его рефакторингу:

```
class FileWriter {
    public:
    void Write(const char* data) { ... здесь изменений нет ... }
    void Write(std::vector<const char*> huge_data) {
      Preamble(huge_data);
      for (auto data: huge_data) Write(data); // повторное использование кода!
      Postscript(huge_data);
    }
    private:
    ... здесь изменений нет ...
};
```

И производный класс перестает работать – при записи вектора строк вызываются подсчитывающие модификации обеих версий `Write`, и размер данных учитывается дважды.

Если наследование вообще используется, то у проблемы хрупкого базового класса нет общего решения, но есть очевидная рекомендация, которая позволяет избежать ее в Шаблонном методе, – при изменении базового класса и структуры алгоритмов или каркаса старайтесь не менять состав вызываемых точек настройки. Точнее, не опускайте те точки настройки, которые вызывались ранее, и не добавляйте новых вызовов к уже существующим (можно добавлять новые точки настройки при условии, что их реализация по умолчанию разумна). Если избежать таких изменений невозможно, то придется проанализировать каждый производный класс и выяснить, зависел ли он от переопределения реализации, которая теперь удалена или заменена, и если да, то какие последствия влечет изменение.

Резюме

В этой главе мы рассмотрели классический объектно-ориентированный паттерн проектирования Шаблонный метод в применении к программам на C++. Этот паттерн работает в C++, как и в любом другом объектно-ориентированном языке, но в C++ также имеется его особая разновидность – идиома невиртуального интерфейса. Преимущества этого паттерна позволяют сформулировать довольно широкую рекомендацию – делайте все виртуальные функции закрытыми или защищенными. Но не забывайте о специфике деструкторов в том, что касается полиморфизма.

В следующей главе мы рассмотрим несколько противоречивый паттерна Одиночка (или Синглтон). Мы узнаем как о добропорядочных применениях этого паттерна, так и о неправильном употреблении, снискавшем ему дурную репутацию.

Вопросы

- Что такое поведенческий паттерн проектирования?
- Что такое паттерн Шаблонный метод?
- Почему Шаблонный метод считается поведенческим паттерном?
- Что такое инверсия управления и каким образом она применима к Шаблонному методу?
- Что такое невиртуальный интерфейс?
- Почему в C++ рекомендуется делать все виртуальные функции закрытыми?
- Когда следует делать виртуальные функции защищенными?
- Почему Шаблонный метод нельзя использовать для деструкторов?
- Что такое проблема хрупкого базового класса и как избежать ее при использовании Шаблонного метода?

Для дальнейшего чтения

- https://www.packtpub.com/application-development/learn-example-c-programming-75-solved-problems-video.
- https://www.packtpub.com/application-development/c-data-structures-andalgorithms.

Глава 15

Одиночка – классический объектно-ориентированный паттерн

В этой главе мы рассмотрим один из самых простых классических объектно-ориентированных паттернов в C++ – Одиночка (Синглтон). И в то же время это один из паттернов, которые чаще всего используют неправильно. Благодаря кажущейся простоте он таит в себе необычную опасность – каждый стремится реализовать Одиночку самостоятельно, а в этих реализациях часто встречаются тонкие ошибки.

В этой главе рассматриваются следующие вопросы:

- что такое паттерн Одиночка;
- когда следует использовать паттерн Одиночка;
- как Одиночки реализуются на C++;
- каковы недостатки и компромиссы различных реализаций Одиночки.

Технические требования

Примеры кода: **https://github.com/PacktPublishing/Hands-On-Design-Patterns-with-CPP/tree/master/Chapter15**.

Потребуется установить и сконфигурировать библиотеку Google Benchmark. Подробности приведены по адресу **https://github.com/google/benchmark**.

Паттерн Одиночка – для чего он предназначен, а для чего – нет

Начнем с обзора паттерна Одиночка – что это такое, что он делает и когда его следует использовать. Любой паттерн – это общепринятое решение какой-нибудь часто возникающей проблемы проектирования. Так какую же проблему решает паттерн Одиночка?

Паттерн Одиночка используется, когда нужно ограничить количество экземпляров класса ровно одним. О нем стоит подумать, если, согласно проекту, во всей системе должен быть только один объект некоторого типа. Часто Одиночку критикуют за то, что это замаскированная глобальная переменная, а глобальные переменные – зло. При этом упускают из виду важный момент: паттерн – решение некоторой проблемы проектирования. Можно было бы возразить, что наличие глобального объекта в системе – признак плохого проектирования, но это аргумент против самой проблемы, а не ее конкретного решения. В предположении, что в проекте все-таки есть основания для заведения глобального объекта, паттерн Одиночка дает каноническое решение проблемы.

На время отложим в сторону вопрос о том, могут ли в хорошем проекте быть глобальные объекты, мы еще вернемся к нему позже. А начнем с демонстрации простого Одиночки.

Что такое Одиночка?

Предположим, что мы хотим иметь в программе класс для протоколирования всех сообщений: отладочных, информационных и об ошибках. Регистратор сообщений должен гарантировать, что если несколько потоков или процессов выводят сообщения одновременно, то они будут напечатаны по порядку, без искажений и чередования. Мы также хотим поддерживать счетчики сообщений об ошибках и предупреждений разных типов, а некоторые сообщения требуется сохранять для последующего включения в отчет, быть может, в виде сводки ошибок в конце работы программы. Существует только один набор счетчиков сообщений для всей программы, а для синхронизации потоков должна использоваться единственная блокировка.

Существуют разные способы решения этой задачи, но самый простой – класс, для которого можно создать только один экземпляр. Реализаций этой идеи несколько, но вот одна из тех, что попроще:

```
// В заголовке logger.h:
class Logger {
    public:
    static Logger& instance() {
        return instance_;
    }

    // API регистратора:
    void LogError(const char* msg) {
        std::lockguard<std::mutex> guard(lock_);
        std::cerr << "ОШИБКА: " << msg << std::endl;
        ++error_count_;
        ... посчитать и запротоколировать ошибку ...
    }
    ... остальная часть API ...

    private:
```

```
    Logger() : error_count_(0) {}
    ~Logger() {}
    Logger(const Logger&) = delete;
    Logger& operator=(const Logger&) = delete;

    private:
    static Logger instance_;
    std::mutex lock_;
    size_t error_count_;
};

// В файле реализации logger.C:
Logger Logger::instance_;
```

В программе существует только один экземпляр класса `Logger` – статический член данных `Logger::instance_` этого класса. Больше никаких экземпляров создать нельзя, потому что конструктор закрытый и все операции копирования удалены. К единственному экземпляру класса `Logger` можно обратиться из любого места программы, вызвав статическую функцию `Logger::instance()`, которая возвращает ссылку на сам объект-одиночку:

```
void func(int* p) {
    if (!p) Logger::instance().LogError("Unexpected NULL pointer");
}
```

Эта статическая функция – единственный способ получить доступ к объекту `Logger`, а, не имея объекта, мы не можем вызывать методы API класса `Logger`. Тем самым гарантируется, что все протоколирование производится через один объект `Logger`, что, в свою очередь, гарантирует, что весь ввод-вывод защищен одной и той же блокировкой, что используется единственный набор счетчиков и т. д. Мы хотели, чтобы протоколирование велось через один объект, и именно это получили от паттерна Одиночка.

При использовании этого Одиночки может возникнуть несколько тонких осложнений. Но прежде чем подробно разбирать их, зададимся вопросом, а нужно ли вообще решать эту задачу.

Когда использовать паттерн Одиночка

Лежащий на поверхности ответ на вопрос, когда использовать Одиночку, очень прост – когда в программе нужен только один объект некоторого типа. Но это, конечно, переливание из пустого в порожнее. Ладно, тогда поставим вопрос иначе: когда наличие только одного объекта некоторого типа во всей программе считается хорошим принципом проектирования, и считается ли вообще? На этот вопрос нет простого универсального ответа, и потому выдвигается аргумент, что Одиночка, как и любой глобальный объект, – признак плохого проекта.

Есть две категории проблем проектирования, которые можно решить с помощью паттерна Одиночка или вообще с помощью единственного глобального объекта. Первая – объекты, представляющие некоторый физический предмет

или ресурс, который существует в единственном экземпляре, по крайней мере в контексте программы. Последнее уточнение очень важно – в мире миллионы автомобилей, но, с точки зрения программы, управляющей работой двигателя, электроники и прочих автомобильных систем, существует всего один *автомобиль*, а именно тот, в котором эта программа работает (другие машины могут присутствовать в картине мира как препятствия, которых нужно избегать, но они фундаментально отличаются от *того самого автомобиля*). В программном комплексе, работающем на различных бортовых микропроцессорах и микрокомпьютерах, *объект автомобиля* должен относиться к одной сущности – ничего хорошего не вышло бы, если бы один компонент программы нажимал на тормоз, а другой в то же время пытался ускориться, поскольку проверяет состояние тормозов другого объекта.

Точно так же программа, которая моделирует Солнечную систему и пытается вычислить траекторию солнечного зонда или аппарата на орбите Марса, с полным основанием может предполагать, что в системе только одна звезда и только один объект типа *звезда*, ее представляющий. Педант мог бы возразить, что это ограничивает применимость программы лишь звездными системами с одной звездой, исключая тем самым двойные звезды. Но программиста можно извинить за то, что он отложил это обобщение на потом и создал технический долг, который будет оплачен, когда межзвездные полеты станут достаточно обыденными, чтобы побеспокоиться о таких вещах (быть может, он полагал, что когда такое время настанет, эта программа уже не будет эксплуатироваться).

Вторая категория – глобальные объекты, созданные таковыми из проектных соображений, хотя за ними не стоит никакая физическая сущность. Например, диспетчеры ресурсов часто реализуются как одиночки. Иногда это отражает природу самого ресурса, например памяти, – ее объем фиксирован, поэтому все диспетчеры, не важно, один или много, должны координировать свою работу в рамках общего лимита. Бывает и так, что физической причины не существует – например, в параллельной системе можно спроектировать один объект-диспетчер для всех процессоров и всей выполняемой системой работы – клиенты передают работу, а глобальный диспетчер распределяет ее по процессорам.

И снова программист может сказать, что ни к чему без нужды усложнять программу – одного диспетчера ресурсов достаточно для всех предвидимых обстоятельств, поэтому обобщение можно отложить до момента, когда возникнет реальная необходимость (если повезет, программа будет полностью переписана или заменена раньше, чем это случится).

К сожалению, черта, разделяющая абсурдно общее и абсурдно близорукое, никогда не бывает четкой и ясной. Общий принцип гибкого (agile) программирования, известный под названием **«тебе это не понадобится»** (You Aren't Gonna Need It – **YAGNI**), гласит, что реализацию, достаточную для удовлетворения нужд сегодняшнего дня, следует предпочесть более общей. Но будьте осто-

рожнее, применяя это правило к проектированию, особенно когда проектное решение ограничивает будущую расширяемость, а отменить его будет трудно. Программы имеют обыкновение жить и эксплуатироваться куда дольше, чем предвидел автор. Конечно, трудно было бы порицать программиста, сказавшего, что *несколько солнц мы обработаем через тысячи лет, когда достигнем хотя бы одного, а пока хватит и одного солнца*. Но похожую аргументацию по поводу клавиатур и мониторов можно было бы услышать и лет 30 назад – к компьютеру подключена всего одна клавиатура, и любая программа, в которой есть ввод с клавиатуры, должна читать именно с нее, поэтому клавиатура, естественно, должна быть одиночкой. На самом деле объект клавиатуры когда-то являлся классическим примером паттерна Одиночка; если бы эта книга была написана в конце прошлого века, то в примере, наверное, фигурировал бы класс `Keyboard`, а не `Logger`, потому что все так делали. А сегодня у нас USB-клавиатуры, клавиатуры док-станций для ноутбуков, проводные и беспроводные клавиатуры, несколько клавиатур типа plug-and-play, но также есть дистанционно управляемые серверы вообще без клавиатур и мониторов. Нелегко, наверное, было переписывать драйверы ввода-вывода, написанные в предположении, что операционная система должна поддерживать одну и только одну клавиатуру, и выдававшие фатальную ошибку на этапе загрузки, если клавиатура не была подключена. С другой стороны, мы пережили это, и предстоящий труд всегда надо сравнивать с тем, что был сэкономлен программистами, которые в течение нескольких десятилетий не желали рассматривать общий случай нескольких клавиатур.

В общем случае размышлять о том, стоит ли использовать паттерн Одиночка, – значит ставить телегу впереди лошади. Вместо этого нужно спросить себя, должен ли проект гарантировать единственность некоторого объекта. Паттерн Одиночка – всего лишь решение этой проблемы проектирования. Оно настолько распространено и принято, что часто воспринимается как эквивалент проблемы, и программисты говорят «Я решил использовать Одиночку», имея в виду «Я решил остановиться на проекте с глобальным объектом и реализовать его как Одиночку» (если в этом предложении что-то и может вызвать возражения, так только первая часть). Итак, когда же проект должен опираться на единственный объект? Или, как говорят чаще, хотя и не совсем правильно, когда следует использовать Одиночку?

Первый случай, когда стоит рассматривать Одиночку, – когда имеется физическая причина для единственного объекта. Иногда причина универсальная – например, существует только одна скорость света. А иногда физическая причина существует лишь в контексте конкретной задачи – любая программа, которая моделирует или управляет полетом космического корабля в нашей Солнечной системе, должна иметь дело только с одной звездой; любая программа управления автомобилем или самолетом управляет только одним транспортным средством. Теоретически всякое решение использовать Одиночку ограничивает способность кода к обобщению и возможность его повтор-

ного применения. В первую очередь следует задать вопрос «*Насколько велики шансы столкнуться с такой ситуацией?*», а во вторую – «*Насколько трудно будет перепроектировать систему по сравнению с усилиями, которые придется приложить для проектирования, реализации и сопровождения кода, который сейчас не используется?*». Вопрос о сопровождении, пожалуй, самый важный из всех – хотя можно представить себе программную систему, управляющую несколькими автомобилями (способом менее тривиальным, чем оповещение друг друга с целью избежать столкновений), мы пока очень далеки от какого-либо практического использования такой системы. Даже если мы не станем делать автомобиль одиночкой и согласимся поддержать несколько двигателей в программе управления двигателем, этот код в течение многих лет останется невостребованным и непротестированным. Велики шансы, что к моменту, когда он действительно понадобится, работать он не будет, и его все равно придется переписывать. Но, оставляя в стороне некоторые очевидные случаи, решение использовать Одиночку – это поступок, который приносит обобщаемость в жертву срочности.

Итак, это обсуждение осталось позади, и, хочется надеяться, мы убедили читателя в том, что для паттерна Одиночка есть законное место в этом мире. Теперь вернемся к предпочтительной реализации.

Типы Одиночек

Реализация Одиночки, показанная в предыдущем разделе, работает, по крайней мере она гарантирует единственность глобального объекта и возможность доступа к нему из любого места программы. Однако есть другие соображения, которые могут повлиять на выбор реализации. Следует также отметить, что за прошедшие годы было создано немало вариантов реализации. В свое время многие из них были вполне корректными и по-разному подходили к конкретным проблемам. Но сегодня большинство из них устарело, так что их даже не следует рассматривать.

Когда дело доходит до реализации Одиночки, у нас есть выбор из нескольких реализаций разного типа. Прежде всего реализации можно классифицировать по тому, как в программе осуществляется доступ к одиночке – хотя экземпляр объекта-одиночки или, по крайней мере, данных, которые должны быть глобальными и уникальными, безусловно, один, существуют разные способы предоставить доступ к этим уникальным данным. Самый простой – завести один глобальный объект:

```
// В заголовке:
extern Singleton SingletonInstance;

// В С-файле:
Singleton SingletonInstance;
```

Это действительно глобальный объект, но его «одинокая» природа не гарантируется компилятором, а обеспечивается только соглашением. Ничто не по-

мешает некорректной программе создать другой объект типа Singleton (можно проверить единственность объекта во время выполнения). При правильном использовании существует всего один экземпляр объекта, который используется для доступа к глобальным данным – в этом случае к самому объекту Singleton. Можно представить реализацию, которая позволяет создавать произвольное количество объектов-*одиночек*, но все они одинаковы и ссылаются на одни и те же данные. Клиентская программа могла бы выглядеть так:

```
void f() {
    Singleton Sf;
    S.do_operation();
}
void g() {
    Singleton Sg;    // ссылается на те же данные, что Sf выше
    S.do_operation();
}
```

Мы увидим возможную реализацию в следующем разделе, а пока важно отметить различие – при таком подходе имеется несколько объектов, являющихся *описателями одиночки*, но все они соответствуют одному реальному одиночке. Никаких ограничений на конструирование описателей нет, но все они взаимозаменяемы и ссылаются на одного и того же одиночку. Таким образом, единственность последнего гарантируется на этапе компиляции. Напротив, в первом подходе есть только один *описатель одиночки* (в нашем примере описателем был сам одиночка).

Наконец, есть и третий вариант – пользователь вообще не может сконструировать объект-одиночку, так что описателей нет вовсе:

```
// В заголовке:
extern Singleton& SingletonInstance();

// В C-файле:
Singleton instance;
Singleton& SingletonInstance() { return instance; }
```

В этом случае программа не конструирует и не использует объекты-описатели, чтобы получить доступ к одиночке, потому-то описателей и нет. Единственность также можно обеспечить на этапе компиляции.

Реализация, которая полагается на дисциплину программирования для обеспечения единственности одиночки, очевидно, не идеальна. Выбор же между остальными двумя вариантами – в основном дело вкуса, т. к. существенных различий между ними нет.

Есть и совершенно другая классификация реализаций одиночек – по времени жизни. Одиночка может быть инициализирован при первом использовании объекта или раньше, обычно на этапе запуска программы. Он может уничтожаться в какой-то точке в конце программы или не уничтожаться вовсе (утекать). Мы увидим примеры каждого вида реализаций и обсудим их преимущества и недостатки.

Статический Одиночка

Одна из самых простых реализаций паттерна – статический Одиночка. В этом случае объект имеет только статические данные-члены:

```
class Singleton {
    public:
    Singleton() {}
    int& get() { return value_; }
    private:
    static int value_;
};
int Singleton::value_ = 0;
```

Далее в этом разделе мы будем рассматривать одиночку с одним членом типа int и функциями-членами, предоставляющими доступ к нему. Это сделано только для определенности – в настоящей реализации может быть сколько угодно членов данных различных типов и сколь угодно сложный API, предоставляемый функциями-членами.

Функции-члены могут быть статическими или нет. Если функция-член не статическая, то реализацию можно отнести к категории *несколько описателей, один набор данных*, поскольку для доступа к одиночке программист просто конструирует объект Singleton столько раз, сколько нужно:

```
Singleton S;
int i = S.get();
++S.get();
```

Объект можно даже конструировать на лету, как временную переменную:

```
int i = Singleton().get();
++Singleton().get();
```

Фактически никакого конструирования здесь нет, т. е. для создания такого объекта не генерируется и не выполняется никакой код – действительно, конструировать-то нечего, раз все данные статические. Можно ожидать, что такой Одиночка будет очень быстрым, и эти ожидания подтверждаются эталонным тестом:

```
#define REPEAT(X) ... повторить X 32 раза
void BM_singleton(benchmark::State& state) {
    Singleton S;
    for (auto _ : state) {
        REPEAT(benchmark::DoNotOptimize(++S.get());)
    }
    state.SetItemsProcessed(32*state.iterations());
}
```

В этом тесте мы воспользовались макросом REPEAT, чтобы сгенерировать 32 копии измеряемого кода внутри цикла. Это делается, чтобы уменьшить накладные расходы на организацию цикла, поскольку каждая итерация очень короткая:

```
-------------------------------------------------------------
Benchmark                   Time           CPU Iterations
-------------------------------------------------------------
BM_singleton/threads:1      10 ns         10 ns   66071182   2.85622G items/s
```

Производительность другого использования Одиночки, когда каждый раз создается временный объект, тоже легко измерить:

```
void BM_singletons(benchmark::State& state) {
    for (auto _ : state) {
        REPEAT(benchmark::DoNotOptimize(++Singleton().get());)
    }
    state.SetItemsProcessed(32*state.iterations());
}
```

Мы ожидаем такой же скорости – и измерения это подтверждают:

```
-------------------------------------------------------------
Benchmark                   Time           CPU Iterations
-------------------------------------------------------------
BM_singletons/threads:1     11 ns         11 ns   66099954   2.82814G items/s
```

Что касается времени жизни Одиночки (не объекта-описателя, а самих данных), то он инициализируется вместе со всеми статическими данными программы в какой-то момент перед вызовом `main()`. А уничтожается где-то в конце программы, после выхода из `main()`.

Альтернативный способ – объявить статическими не только данные-члены, но и функции-члены:

```
class Singleton {
    public:
    static int& get() { return value_; }
    private:
    Singleton() = delete;
    static int value_;
};
int Singleton::value_ = 0;
```

При таком подходе нам вообще никогда не придется конструировать объект `Singleton`:

```
int i = Singleton::get();
++Singleton::get();
```

Эту реализацию можно отнести к категории *ноль объектов-описателей, один набор данных*. Однако во всех существенных аспектах она идентична предыдущей – синтаксис вызова функций различается, но делают они то же самое. Производительность также одинаковая.

Важно учитывать потокобезопасность реализации Одиночки, а также ее производительность в конкурентной программе. Статический Одиночка сам по себе, очевидно, потокобезопасен – он инициализируется и уничтожается исполняющей системой C++ вместе с остальными статическими данными,

и в этот момент никакие пользовательские потоки не работают. Ответственность за потокобезопасное использование Одиночки всегда возлагается на программиста – если Одиночка позволяет модифицировать свои данные, то доступ к нему следует либо защитить мьютексом, либо реализовать потокобезопасным способом. При обсуждении потокобезопасности реализации Одиночки нас интересует инициализация, уничтожение и предоставление доступа к объекту-одиночке – эти операции являются частью реализации паттерна, все остальное – специфика программы.

Потокобезопасность статического Одиночки устанавливается тривиально, но как насчет производительности? В этом случае для доступа к Одиночке программа всего лишь должна прочитать статические данные, поэтому никаких накладных расходов нет. Разумеется, накладные расходы возможны, если программе необходимо синхронизировать модификации этих данных, но тут дело обстоит так же, как с доступом к любым разделяемым данным в конкурентной программе.

У этой реализации есть два потенциальных недостатка. Во-первых, этого Одиночку нельзя расширить путем наследования. Но это наименее важная из двух проблем – объекты-одиночки в хорошем проекте должны встречаться редко, поэтому повторное использование кода особого значения не имеет.

Более важная проблема связана со временем жизни объекта – одиночка инициализируется как статический объект еще до начала работы программы. Порядок инициализации статических объектов, вообще говоря, не определен и оставлен на усмотрение реализации – стандарт гарантирует, что объекты, определенные в одном файле, инициализируются в порядке следования определений, но для объектов, находящихся в разных файлах, не дается никаких гарантий относительно порядка инициализации.

Это не проблема, если Одиночка будет использоваться программой во время ее выполнения, т. е. в чем-то, что вызывается из `main()`, – в конце концов, Одиночка, безусловно, будет сконструирован до начала работы программы и не будет уничтожен до ее завершения. Проблема, однако, возникает, когда другой статический объект использует этого Одиночку. Такие зависимости между статическими объектами – не редкость; например, если наш одиночка – диспетчер памяти, который существует в единственном экземпляре, то любой статический объект, выделяющий память, должен получить ее от диспетчера. Однако если только вся программа не находится в одном файле, невозможно гарантировать, что такой диспетчер памяти будет инициализирован до первого использования. Описанная далее реализация – попытка решить эту проблему.

Одиночка Мейерса

Эта реализация названа в честь своего изобретателя, Скотта Мейерса (Scott Meyers). Если главная проблема статического Одиночки – то, что он может быть инициализирован до первого использования, то решение заключается в том, чтобы инициализировать Одиночку в тот момент, когда он будет впервые востребован:

```
class Singleton {
    public:
    static Singleton& instance() {
        static Singleton inst;
        return inst;
    }
    int& get() { return value_; }
    private:
    Singleton() : value_(0) {
        std::cout << "Singleton::Singleton()" << std::endl;
    }
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;
    ~Singleton() {
        std::cout << "Singleton::~Singleton()" << std::endl;
    }
    private:
    int value_;
};
```

Одиночка Мейерса обладает закрытым конструктором, поэтому программа не может сконструировать его непосредственно (мы включили в конструктор печать, просто чтобы видеть, когда Одиночка инициализируется). Программа также не может создавать копии объекта-одиночки. Поскольку Одиночку Мейерса нельзя сконструировать напрямую, эта реализация также относится к категории *нулевого числа объектов-описателей*. Единственный способ получить доступ к такому одиночке – воспользоваться статической функцией-членом `Singleton::instance()`:

```
int i = Singleton::instance().get();
++Singleton::instance().get();
```

Функция `Singleton::instance()` возвращает ссылку на объект-одиночку, но какой именно, и когда он был создан? Из показанного выше кода ясно, что возвращаемое значение – ссылка на локальный объект, определенный в теле самой функции `instance()`. Обычно возврат ссылок на локальные объекты является серьезной ошибкой программирования – такие объекты перестают существовать после выхода из функции. Но в Одиночке Мейерса используется не обычный локальный объект, а локальный статический объект. Как и в случае статических объектов с файловой областью видимости, в программе существует только один экземпляр статического объекта. Но, в отличие от статических объектов с файловой областью видимости, статические объекты с функциональной областью видимости инициализируются в момент первого обращения; в нашем случае – при первом вызове функции. На псевдокоде поведение статического объекта с функциональной областью видимости можно описать так:

```
static bool initialized = false;  // скрытая переменная, генерируемая компилятором
// Память для статического объекта, первоначально неинициализированная
```

```
char memory[sizeof(Singleton)];
class Singleton {
    public:
    static Singleton& instance() {
        if (!initialized) {         // происходит только один раз
            initialized = true;
            new (memory) Singleton; // new с размещением, вызывается конструктор Singleton
        }
        // теперь память содержит объект класса Singleton
        return *(Singleton*)(memory);
    }
    ...
};
```

Такая инициализация Одиночки может произойти после запуска программы, может быть даже спустя длительное время после запуска, если Одиночка долго не используется. С другой стороны, если какой-то другой статический объект (необязательно одиночка) попытается воспользоваться нашим Одиночкой и запросит ссылку на него, то инициализация гарантированно произойдет до того, как объект можно будет использовать. Это пример ленивой реализации – отложенной до момента, когда возникнет необходимость (если при конкретном прогоне программы к Одиночке не будет ни одного обращения, то он никогда и не инициализируется).

Возможное сомнение по поводу Одиночки Мейерса – производительность. Хотя инициализация производится только один раз, при каждом вызове `Singleton::instance()` необходимо проверять, инициализирован ли уже объект. Стоимость этой проверки можно измерить, сравнив время, необходимое для доступа к экземпляру в некоторых операциях, со временем, которое требуется для вызова тех же операций, но от имени уже сохраненной ссылки на экземпляр:

```
void BM_singletons(benchmark::State& state) {
    for (auto _ : state) {
        REPEAT(benchmark::DoNotOptimize(++Singleton::instance().get());)
    }
    state.SetItemsProcessed(32*state.iterations());
}

void BM_singleton(benchmark::State& state) {
    Singleton& S = Singleton::instance();
    for (auto _ : state) {
        REPEAT(benchmark::DoNotOptimize(++S.get());)
    }
    state.SetItemsProcessed(32*state.iterations());
}
```

В первом тесте `Singleton::instance()` вызывается каждый раз, а во втором те же функции-члены вызываются от имени одиночки, но доступ к экземпляру производится только один раз. Разница во времени показывает стоимость проверки на необходимость инициализации (стоимость самой инициализа-

ции несущественна, потому что тест выполняется много раз, а инициализация – всего один):

```
Running ./singleton_meyers
Run on (4 X 3400 MHz CPU s)
Singleton::Singleton()
-----------------------------------------------------
Benchmark              Time           CPU Iterations
-----------------------------------------------------
BM_singleton          11 ns         11 ns   64858630   2.62924G items/s
BM_singletons         70 ns         70 ns    9945667    437.73M items/s
Singleton::~Singleton()
```

Как видим, стоимость реализации статической переменной с функциональной областью видимости весьма заметна и значительно выше стоимости простой операции для объекта-одиночки (в нашем случае инкремента целого числа). Поэтому если объект-одиночку предполагается использовать интенсивно, то может быть выгоднее сохранить ссылку на него, а не запрашивать ее каждый раз. Кроме того, благодаря отладочной печати мы видим, что Одиночка действительно инициализируется при первом использовании – если программа (точнее, функция `main()`, предоставленная библиотекой Google Benchmark) печатает сообщения *Running...* и *Run on...*, значит, Одиночка инициализирован. Если бы в Одиночке использовался статический объект с файловой областью видимости, то конструктор был бы вызван до того, как у программы появился бы шанс что-то напечатать.

Не путайте следующую реализацию с Одиночкой Мейерса:

```
class Singleton {
    public:
    static Singleton& instance() {
        return instance_;
    }
    int& get() { return value_; }
    private:
    Singleton() : value_(0) {
        std::cout << "Singleton::Singleton()" << std::endl;
    }
    ~Singleton() {
        std::cout << "Singleton::~Singleton()" << std::endl;
    }
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;
    private:
    static Singleton instance_;
    int value_;
};
Singleton Singleton::instance_;
```

На первый взгляд, они похожи, однако эта реализация отличается в самом важном аспекте – моменте инициализации. Область видимости статического экземпляра здесь – не функция, он инициализируется вместе с остальными

статическими объектами вне зависимости от того, используется или нет (энергичная инициализация в противоположность ленивой). Доступ к экземпляру Одиночки выглядит точно так же, как в Одиночке Мейерса, но на этом сходство и заканчивается. На самом деле это просто еще один вариант статического Одиночки, только вместо объявления всех данных-членов статическими мы создаем статический экземпляр класса.

Можно ожидать, что производительность будет примерно такой, как для статического Одиночки или для Одиночки Мейерса, если бы мы оптимизировали код, так чтобы избежать многократных проверок на инициализацию.

```
Singleton::Singleton()
2018-09-01 15:17:01
Running ./singleton_eager
Run on (4 X 3400 MHz CPU s)
------------------------------------------------------------------
Benchmark                        Time           CPU Iterations
------------------------------------------------------------------
BM_singleton/threads:1          11 ns         11 ns   65362522   2.80485G items/s
Singleton::~Singleton()
```

Мы снова хотим обратить внимание читателя на момент конструирования – на этот раз конструктор статического Одиночки вызывается до того, как программа начинает печатать собственные сообщения.

Интересный вариант этой реализации – комбинация Одиночки Мейерса с *идиомой pimpl*, согласно которой заголовочный файл содержит только объявление интерфейса, а вся реализация, включая данные-члены, перемещена в другой класс и скрыта в C-файле, тогда как в заголовке остается только указатель на объект реализации (отсюда и название – *pointer to impl*, или просто *pimpl*). Эта идиома часто применяется для уменьшения количества зависимостей на этапе компиляции – если реализация объекта изменяется, но открытый API остается тем же самым, то заголовочный файл останется прежним, и все зависящие от него файлы не придется перекомпилировать. В случае Одиночки комбинация этих двух паттернов выглядит следующим образом:

```
// В заголовочном файле:
struct SingletonImpl;           // опережающее объявление
class Singleton {
    public:                     // открытый API
    int& get();
    private:
    static SingletonImpl& impl();
};

// В C-файле:
struct SingletonImpl {          // клиенту все равно, изменилось это или нет
    SingletonImpl() : value_(0) {}
    int value_;
};

int& Singleton::get() { return impl().value_; }

SingletonImpl& Singleton::impl() {
```

```
    static SingletonImpl inst;
    return inst;
}
```

В этой реализации программа может создать сколько угодно объектов Singleton, но все они работают с одним и тем же объектом, созданным методом impl() (в нашем случае этот метод возвращает ссылку, а не указатель на реализацию, и тем не менее мы используем название **pimpl**, потому что, по сути дела, это та же самая идиома). Заметим, что мы не стали как-то защищать класс реализации – поскольку он находится в одном C-файле и используется не напрямую, а только с помощью методов класса Singleton, то вполне возможно положиться на дисциплинированность программиста.

Достоинство этой реализации – улучшенное разделение между интерфейсом и реализацией, для чего, собственно, идиома *pimpl* всегда и используется. А недостаток – дополнительный уровень косвенности и связанные с ним накладные расходы. Отметим также, что теперь программа уже не может избежать проверки на ленивую инициализацию, поскольку она скрыта внутри реализации методов Singleton. Класс Singleton можно оптимизировать, чтобы не делать повторных проверок, но для этого нужно хранить ссылку на реализацию в каждом объекте:

```
// В заголовочном файле:
struct SingletonImpl;
class Singleton {
    public:
    Singleton();
    int& get();
    private:
    static SingletonImpl& impl();
    SingletonImpl& impl_;                    // кешированная ссылка
};

// В C-файле:
struct SingletonImpl {
    SingletonImpl() : value_(0) {}
    int value_;
};

Singleton::Singleton() : impl_(impl()) {}

int& Singleton::get() { return impl_.value_; }
    SingletonImpl& Singleton::impl() { // теперь вызывается один раз на объект
    static SingletonImpl inst;
    return inst;
}
```

Теперь экземпляр Одиночки создается при первом конструировании объекта Singleton, а не при первом вызове его функции-члена. Кроме того, у каждого объекта Singleton уже есть ссылочный член данных, поэтому мы потребляем чуть больше памяти в качестве платы за повышение производительности:

```
-------------------------------------------------------------
Benchmark                   Time           CPU Iterations
-------------------------------------------------------------
BM_singleton_pimpl          70 ns         70 ns    9918408   438.475M items/s
BM_singleton_pimpl_opt      11 ns         11 ns   64369401   2.80543G items/s
```

Видно, что оптимизированная реализация не уступает в производительности любой из рассмотренных ранее, тогда как *pimpl*-реализация в лоб значительно медленнее.

В современных программах есть один важный фактор – потокобезопасность. В случае Одиночки Мейерса этот вопрос нетривиален и сводится к такому: является ли потокобезопасной инициализация локальной статической переменной? Особый интерес представляет следующий код:

```
static Singleton&amp;amp; instance() {
    static Singleton inst;
    return inst;
}
```

Реальный код, скрывающийся за этой конструкцией на C++, довольно сложен – имеется условная проверка того, сконструирована ли уже переменная, и флаг, устанавливаемый после первого выполнения кода. Что будет, если несколько потоков одновременно вызовут функцию `instance()`? Существует ли гарантия, что будет создан единственный экземпляр статического объекта для всех потоков? В стандарте C++11 и более поздних ответ – твердое «да». Но до выхода C++11 стандарт не давал никаких гарантий относительно потокобезопасности. Поэтому появилось множество различных реализаций, которые все еще можно найти в сети и в печатных источниках. В общем и целом они устроены, как показано ниже, и отличаются только вариациями на тему блокировок:

```
static bool initialized = false;
static Singleton& instance() {
    if (!initialized) { ... инициализировать экземпляр под защитой блокировки ... }
    return ... ссылку на экземпляр Одиночки ...
}
```

Сейчас такие реализации считаются устаревшими и представляют только исторический интерес. Мы не будем тратить время на объяснение того, как они работают, и работают ли правильно (о многих этого не скажешь). Не осталось никаких причин делать что-то, помимо объявления локальной статической переменной и возврата ссылки на нее.

Как мы уже объяснили, Одиночка Мейерса решает проблему порядка инициализации за счет того, что производит инициализацию при первом использовании объекта. Даже если имеется несколько Одиночек (конечно, разного типа), которые ссылаются друг на друга, ни один объект не будет инициализирован раньше, чем в нем возникнет необходимость. Проблема порядка инициализации действительно решена. Но, как мы увидим далее, это не единственная проблема.

Утекающие Одиночки

Мы только что видели, как Одиночка Мейерса решает проблему порядка инициализации статических объектов. Но есть и совершенно другая проблема – порядка уничтожения. Порядок уничтожения точно определен в стандарте: статические переменные – как с функциональной, так и с файловой областью видимости – уничтожаются после завершения программы (после возврата из `main()`). Уничтожение производится в порядке, противоположном порядку конструирования, т. е. объект, сконструированный последним, уничтожается первым. Почему это так важно?

Прежде всего мы можем с уверенностью сказать, что любые ссылки на объект-одиночку из самой программы (а не из других статических объектов) абсолютно безопасны, поскольку выход из `main()` происходит до того, как он уничтожается. Поэтому предметом беспокойства могут быть только другие статические объекты, а точнее, их деструкторы. На первый взгляд, нет никакой проблемы: объекты уничтожаются в порядке, обратном порядку конструирования, поэтому любой объект, который был сконструирован позже и может зависеть от ранее созданных объектов, будет уничтожен раньше. Но это не единственный вид зависимости. Рассмотрим конкретный пример – в нашей программе имеется два объекта-одиночки. Первый – диспетчер памяти `MemoryManager`, который выделяет всю память, затребованную программой. Когда этот объект уничтожается, он возвращает всю выделенную память операционной системе. Второй объект – `ErrorLogger`, он используется в программе для протоколирования ошибок. Этому объекту нужна память для хранения информации об ошибках, но он освобождает эту память в момент уничтожения. Оба объекта – Одиночки с ленивой инициализацией, поэтому конструируются в момент первого использования.

На первый взгляд, все в порядке – безусловно, диспетчер памяти будет сконструирован первым, хотя бы потому, что он нужен регистратору ошибок, правда? Поэтому `ErrorLogger` будет уничтожен раньше и вернет всю память диспетчеру. Однако на самом деле взаимодействие сложнее. Представьте, что не каждая операция объекта `ErrorLogger` нуждается в памяти – некоторые сообщения просто печатаются без запоминания. Тогда могла бы иметь место такая последовательность операций:

```
ErrorLogger::instance().MessageNoLog("Выполнение началось");    // 1
ErrorLogger::instance().Error("Обнаружена проблема");           // 2
```

В строке 1 объект-одиночка `ErrorLogger` инициализируется, но памяти пока не требует, поэтому диспетчер памяти не инициализирован. В строке 2 `ErrorLogger` уже сконструирован, поэтому используется имеющийся экземпляр Одиночки. Методу `ErrorLogger::Error()` необходима память, поэтому он обращается к `MemoryManager`. Поскольку это первое обращение к объекту-одиночке, конструируется экземпляр `MemoryManager` и выделяется память. `ErrorLogger` сохраняет указатель на выделенный блок памяти. Теперь отправимся прямиком

в конец программы – MemoryManager был сконструирован последним, поэтому уничтожается первым и освобождает всю выделенную память. Но экземпляр ErrorLogger еще жив и хранит указатель на область памяти, полученную от диспетчера! ErrorLogger был сконструирован раньше диспетчера, и теперь настала его очередь подвергнуться уничтожению. Деструктор объекта вызывает API MemoryManager, чтобы вернуть память. Но объект MemoryManager уже уничтожен, т. е. мы вызываем функцию-член несуществующего объекта. Более того, вся память уже возвращена операционной системе, поэтому в ErrorLogger хранится висячий указатель. Оба эти действия приводят к неопределенному поведению.

Эта тонкая ошибка проявляется странным образом – если программа «грохается» в самом конце после печати последнего сообщения, то, возможно, в ней имеется ошибка, похожая на описанную выше. К сожалению, у этой проблемы нет общего решения. Изменить порядок вызова статических деструкторов невозможно, поэтому единственный способ избежать проблем, вызванных этим фиксированным порядком, – вообще не использовать статические экземпляры. Можно заменить статические объекты статическими указателями:

```
static Singleton&amp;amp; instance() {
    static Singleton* inst = new Singleton;
    return *inst;
}
```

Отличие от предыдущей реализации тонкое, но очень важное – статический указатель тоже инициализируется всего один раз, при первом вызове функции. В этот момент вызывается конструктор класса Singleton. Однако уничтожение указателя – пустая операция. В частности, при этом не вызывается оператор delete для объекта, мы должны сделать это сами. А это уже открывает возможность управлять порядком уничтожения. К сожалению, на практике воспользоваться этой возможностью крайне сложно. Иногда можно применить подсчет ссылок, но для этого нужно как-то подсчитывать все обращения к объекту-одиночке. В примере с регистратором ошибок и диспетчером памяти мы могли бы вести счетчик ссылок, при условии что регистратор имеет хотя бы один указатель на память, полученную от диспетчера. Такие реализации всегда специфичны для конкретной задачи – не существует общего способа сделать это правильно. Но даже частные реализации зачастую сложны, их трудно протестировать или доказать их правильность.

Одна из возможных альтернатив, которую следует рассмотреть, если порядок удаления становится проблемой, – вообще ничего не удалять. Тогда объекты-одиночки утекают, поскольку программа их никогда не уничтожает. Для некоторых типов ресурсов это неприемлемо, но обычно такое решение годится, особенно если эти объекты управляют только памятью: программа-то почти завершилась, и память все равно будет возвращена операционной системе, так что освобождение памяти в этой точке не сделает программу ни более быстрой, ни менее расточительной (если программа делает так много работы в статических деструкторах, что отказ от скорейшего освобождения памяти

приводит к проблемам с производительностью, то, очевидно, настоящий корень зла следует искать в неудачном дизайне). Утекающий Одиночка с ленивой инициализацией очень похож на Одиночку Мейерса, только вместо локального статического объекта используется локальный статический указатель:

```
class Singleton {
    public:
    static Singleton& instance() {
        // Это единственное отличие от Одиночки Мейерса
        static Singleton* inst = new Singleton;
        return *inst;
    }

    int& get() { return value_; }

    private:
    Singleton() : value_(0) {
        std::cout << "Singleton::Singleton()" << std::endl;
    }
    ~Singleton() {
        std::cout << "Singleton::~Singleton()" << std::endl;
    }
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;
    private:
    int value_;
};
```

Если не считать отсутствия удаления, то эта реализация идентична Одиночке Мейерса. Она инициализируется лениво и гарантирует правильный порядок инициализации, она потокобезопасна, и ее производительность практически такая же. Как и Одиночке Мейерса, ей свойственны накладные расходы на проверку того, инициализирована ли статическая переменная, но эффективность можно повысить за счет хранения локальной ссылки для кеширования результата вызова `instance()`.

Альтернативная реализация, которую следует предпочесть, если это возможно, – явно запускать освобождение всех ресурсов, захваченных статическими объектами, в конце программы. Например, в классе `ErrorLogger` может быть метод `clear()`, который завершает все задачи протоколирования ошибок и возвращает всю память диспетчеру; тогда деструктору не остается делать ничего, кроме уничтожения самого статического экземпляра. Подобный дизайн полагается на добрую волю программиста и не проверяется компилятором. Тем не менее иногда это лучший из возможных вариантов.

Резюме

В этой главе мы узнали практически обо всем, что можно сказать о классическом объектно-ориентированном паттерне Одиночка. Мы обсудили, когда стоит подумать об использовании этого паттерна, а когда его следует избегать,

рассматривая как признак небрежного проектирования. Мы рассмотрели несколько возможных реализаций Одиночки; одни лениво инициализируются по запросу, другие – энергично, при запуске программы, в одних используется несколько объектов-описателей, в других программисту явно предоставляется единственный объект. Мы рассмотрели и сравнили разные реализации с точки зрения потокобезопасности и производительности и обсудили потенциальные проблемы, связанные с порядком конструирования и уничтожения.

При таком количестве различных реализаций можно извинить читателя, желающего получить более определенную рекомендацию – какого Одиночку использовать? С учетом всех «за» и «против» мы бы порекомендовали начать с Одиночки Мейерса. Если дополнительно требуется минимизировать зависимости компиляции или вам просто необходима оптимизация с кешированием ссылок, то мы рекомендуем вариант Одиночки Мейерса в сочетании с идиомой *pimpl*. Наконец, если порядок удаления является проблемой (а так и будет, если имеется несколько статических объектов, которые хранят ресурсы, полученные от других статических объектов), то мы настоятельно советуем явно очищать объекты-одиночки. Если это невозможно, то лучшим решением будет утекающий Одиночка Мейерса (который можно сочетать с идиомой *pimpl*).

Следующая глава велика – не только по объему, но и по важности. Она посвящена еще одному хорошо известному паттерну, Стратегии, который при использовании совместно с обобщенным программированием на C++ становится одним из самых мощных паттернов проектирования – проектированием на основе политик.

Вопросы

- Что такое паттерн Одиночка?
- Когда можно использовать паттерн Одиночка, а когда его следует избегать?
- Что такое ленивая инициализация и какие проблемы она решает?
- Как сделать инициализацию Одиночки потокобезопасной?
- В чем состоит проблема порядка удаления и какие есть способы ее решения?

Глава 16

Проектирование на основе политик

Проектирование на основе политик – один из самых хорошо известных паттернов в C++. С момента появления стандартной библиотеки шаблонов в 1998 году мало новых идей оказали такое сильное влияние на разработку программ на C++, как изобретение проектирования на основе политик.

Проектирование на основе политик – это о гибкости, расширяемости и настройке поведения. Это способ проектирования ПО, которое может эволюционировать и адаптироваться к изменяющимся потребностям, некоторые из которых даже предвидеть было нельзя на этапе разработки первоначального проекта. Хорошо спроектированная система на основе политик может оставаться структурно неизменной в течение многих лет и бескомпромиссно обслуживать новые требования. К сожалению, это еще и такой способ создания ПО, который позволил бы делать все вышеперечисленное, если бы хоть кто-нибудь мог разобраться, как программа работает. Цель настоящей главы – научить читателя понимать и самому проектировать системы первого рода, не впадая в крайности, способные привести к несчастьям второго рода.

В этой главе рассматриваются следующие вопросы:

- паттерн Стратегия и проектирование на основе политик;
- политики времени выполнения в C++;
- реализация классов на основе политик;
- рекомендации по использованию политик.

Технические требования

Примеры кода: **https://github.com/PacktPublishing/Hands-On-Design-Patterns-with-CPP/tree/master/Chapter16**.

Паттерн Стратегия и проектирование на основе политик

Классический паттерн Стратегия – это поведенческий паттерн проектирования, который позволяет во время выполнения выбрать конкретный алгоритм

реализации требуемого поведения, обычно из предопределенного семейства алгоритмов. Этот паттерн называется также *Политика*, и это название появилось раньше, чем паттерн стал применяться в обобщенном программировании на C++. Цель паттерна Стратегия – повысить гибкость проекта: в классическом объектно-ориентированном паттерне решение относительно выбора алгоритма откладывается до этапа выполнения.

Как и для многих других классических паттернов, обобщенное программирование на C++ позволяет применить тот же подход к выбору алгоритма на этапе компиляции, т. е. настроить некоторые аспекты поведения системы путем выбора из семейства взаимозаменяемых алгоритмов. Сначала мы рассмотрим основы реализации классов с политиками на C++, а затем перейдем к более сложным и разнообразным подходам к проектированию на основе политик.

Основы проектирования на основе политик

Паттерн Стратегия имеет смысл рассматривать при проектировании системы, которая выполняет некоторые операции, но точный способ их реализации заранее неизвестен, может изменяться, в том числе и после внедрения системы. Иными словами, когда мы знаем ответ на вопрос, *что система должна делать*, но не знаем, *как*. Аналогично стратегия времени компиляции, или политика, – это способ реализовать класс, имеющий определенную функцию (*что*), но более одного способа реализовать эту функцию (*как*).

В данной главе для иллюстрации различных способов применения политик мы спроектируем класс интеллектуального указателя. У интеллектуального указателя есть много обязательных и факультативных свойств, помимо политик, и мы не будем рассматривать их все – читателя, интересующегося полной реализацией интеллектуального указателя, отсылаем к примерам таких указателей в стандартной библиотеке C++ (`unique_ptr` и `shared_ptr`), а также в библиотеках Boost и Loki (`http://loki-lib.sourceforge.net/`). Материал, представленный в этой главе, поможет читателю понять, перед каким выбором стояли авторы этих библиотек, а также узнать, как проектировать собственные классы на основе политик.

Минимальная начальная реализация интеллектуального указателя может выглядеть следующим образом:

```
template <typename T>
class SmartPtr {
    public:
    explicit SmartPtr(T* p = nullptr)
        : p_(p) {}
    ~SmartPtr() {
        delete p_;
    }
    T* operator->() { return p_; }
    const T* operator->() const { return p_; }
    T& operator*() { return *p_; }
```

```
    const T& operator*() const { return *p_; }
    private:
    T* p_;
    SmartPtr(const SmartPtr&) = delete;
    SmartPtr& operator=(const SmartPtr&) = delete;
};
```

Для этого указателя определен конструктор из простого указателя того же типа и обычные для указателя операторы * и ->. Самая интересная часть здесь – деструктор: когда указатель уничтожается, он автоматически удаляет сам объект (необязательно проверять перед удалением, что указатель ненулевой, потому что, согласно стандарту, оператор delete обязан принимать нулевой указатель и при этом ничего не делать). Поэтому ожидается, что интеллектуальный указатель будет использоваться следующим образом:

```
Class C { ..... };
{
    SmartPtr<C> p(new C);
    ..... использовать p .....
}  // объект *p удаляется автоматически
```

Это типичный пример RAII-класса – RAII-объект, т. е. интеллектуальный указатель, владеет ресурсом (объектом, переданным конструктору) и освобождает (удаляет его), когда удаляется сам объект-владелец. Применения идиомы RAII, подробно рассмотренные в главе 5, призваны гарантировать, что объект, сконструированный в некоторой области видимости, будет удален, когда поток управления покидает эту область видимости любым способом (например, если где-то в коде возникло исключение, то деструктор RAII-объекта гарантирует, что объект будет уничтожен).

Отметим еще две функции-члена интеллектуального указателя – не за их реализацию, а за отсутствие таковой: указатель сделан некопируемым, поскольку копирующий конструктор и оператор присваивания подавлены. Эта деталь, о которой иногда забывают, критически важна для RAII-класса – поскольку деструктор указателя удаляет объект, принадлежавший владельцу, на него никогда не должно указывать два указателя, иначе каждый из них попытается удалить один и тот же объект.

Показанный выше указатель вполне работоспособен, но его реализация имеет ограничения. В частности, он может владеть (и удалять) только объектом, сконструированным стандартным оператором new, и притом только одним. Хотя он мог бы запомнить указатель, полученный от пользовательского оператора new, или указатель на массив элементов, правильно удалять их он не умеет.

Мы могли бы реализовать другой интеллектуальный указатель для объектов, созданных в определенной пользователем куче, и еще один для объектов, созданных в управляемой клиентом памяти, и т. д. – по одному для каждого способа конструирования объекта, вместе с соответствующим способом удаления. Большая часть кода таких указателей будет дублироваться – все они указатели,

и весь API объекта, похожего на указатель, нужно будет копировать в каждый класс. Легко видеть, что все эти различные классы по существу являются классами одного вида – ответом на вопрос «*что это за тип?*» всегда будет «*это интеллектуальный указатель*». Разница только в том, как реализовано удаление. Вот такое общее намерение с одним различающимся аспектом поведения и ведет к паттерну Стратегия. Мы можем реализовать более общий интеллектуальный указатель, делегировав детали удаления объекта различным политикам удаления, число которых не ограничено:

```
template <typename T, typename DeletionPolicy>
class SmartPtr {
    public:
    explicit SmartPtr(
        T* p = nullptr,
        const DeletionPolicy& deletion_policy = DeletionPolicy()
    ) : p_(p),
        deletion_policy_(deletion_policy)
    {}
    ~SmartPtr() {
        deletion_policy_(p_);
    }
    T* operator->() { return p_; }
    const T* operator->() const { return p_; }
    T& operator*() { return *p_; }
    const T& operator*() const { return *p_; }
    private:
    T* p_;
    DeletionPolicy deletion_policy_;
    SmartPtr(const SmartPtr&) = delete;
    SmartPtr& operator=(const SmartPtr&) = delete;
};
```

Политика удаления – дополнительный параметр шаблона, а объект этого типа передается конструктору интеллектуального указателя (если он не передан явно, то создается конструктором по умолчанию). Объект политики удаления хранится в интеллектуальном указателе и используется для удаления объекта, на который этот указатель указывает.

Единственное требование к типу политики удаления состоит в том, что он должен быть вызываемым – политика вызывается, как функция с одним аргументом – указателем на подлежащий удалению объект. Например, поведение нашего первоначального указателя, который вызывал для объекта оператор delete, можно реализовать с помощью такой политики удаления:

```
template <typename T>
struct DeleteByOperator {
    void operator()(T* p) const {
        delete p;
    }
};
```

Чтобы использовать эту политику, мы должны указать ее тип при конструировании интеллектуального указателя и факультативно передать объект этого типа конструктору, хотя в данном случае достаточно и объекта, сконструированного по умолчанию:

```
class C { ..... };
SmartPtr<C, DeleteByOperator<C>> p(new C);
```

Если политика удаления не соответствует типу объекта, то будет диагностирована синтаксическая ошибка – недопустимое обращение к `operator()`.

Для объектов, выделенных другими способами, нужны иные политики удаления. Например, если объект создан в пользовательской куче, интерфейс которой включает функции-члены `allocate()` и `deallocate()` соответственно для выделения и освобождения памяти, то мы можем воспользоваться следующей политикой удаления для кучи:

```
template <typename T>
struct DeleteHeap {
    explicit DeleteHeap(Heap& heap)
        : heap_(heap) {}
    void operator()(T* p) const {
        p->~T();
        heap_.deallocate(p);
    }
    private:
    Heap& heap_;
};
```

С другой стороны, если объект конструируется в памяти, которая независимо управляется вызывающей стороной, то нужно вызывать только деструктор объекта:

```
template <typename T>
struct DeleteDestructorOnly {
    void operator()(T* p) const {
        p->~T();
    }
};
```

Ранее мы упоминали, что поскольку политика используется как вызываемая сущность, `deletion_policy_(p_)`, она может быть любого типа, который можно вызывать как функцию. В том числе это может быть и просто функция:

```
typedef void (*delete_int_t)(int*);
void delete_int(int* p) { delete p; }
SmartPtr<int, delete_int_t> p(new int(42), delete_int);
```

Конкретизация шаблона также является функцией, поэтому ее можно использовать точно так же:

```
template <typename T> void delete_T(T* p) { delete p; }
SmartPtr<int, delete_int_t> p(new int(42), delete_T<int>);
```

Из всех возможных политик удаления одна обычно используется чаще других. В большинстве программ удаление, скорее всего, осуществляется функцией operator delete, подразумеваемой по умолчанию. Если это так, то имеет смысл не задавать политику при каждом использовании, а сделать ее параметром по умолчанию:

```
template <typename T, typename DeletionPolicy = DeleteByOperator<T>>
class SmartPtr {
    .....
};
```

Теперь наш интеллектуальный указатель на основе политик можно использовать точно так же, как первоначальный вариант, в котором был только один способ удаления:

```
SmartPtr<C> p(new C);
```

Тип второго параметра шаблона задается неявно по умолчанию, DeleteByOperator<C>, а в качестве второго аргумента конструктору передается объект этого типа, сконструированный по умолчанию.

Тут мы должны предостеречь читателя от тонкой ошибки, которую можно сделать при реализации таких классов на основе политик. Заметим, что объект политики запоминается в конструкторе интеллектуального указателя по константной ссылке:

```
explicit SmartPtr(
    T* p = nullptr,
    const DeletionPolicy& deletion_policy = DeletionPolicy());
```

Константность ссылки важна, потому что неконстантную ссылку нельзя связать с временным объектом (мы рассмотрим ссылки на r-значения ниже в этом разделе). Однако в самом объекте политика запоминается по значению, и, следовательно, необходимо создавать копию объекта политики:

```
template <typename T, typename DeletionPolicy = DeleteByOperator<T>>
class SmartPtr {
    .....
    private:
    DeletionPolicy deletion_policy_;
};
```

Возникает соблазн избежать копирования и запоминать политику в интеллектуальном указателе также по ссылке:

```
template <typename T, typename DeletionPolicy = DeleteByOperator<T>>
class SmartPtr {
    .....
    private:
    const DeletionPolicy& deletion_policy_;
};
```

В некоторых случаях это даже будет работать, например:

```
Heap h;
DeleteHeap<C> del_h(h);
SmartPtr<C, DeleteHeap<C>> p(new (&heap) C, del_h);
```

Однако это не работает для подразумеваемого по умолчанию способа создания объектов `SmartPtr` и вообще любых интеллектуальных указателей, инициализируемых временным объектом политики:

```
SmartPtr<C> p(new C, DeleteByOperator<C>());
```

Этот код компилируется. Но, к сожалению, он некорректен – временный объект `DeleteByOperator<C>` конструируется непосредственно перед вызовом конструктора `SmartPtr`, но уничтожается в конце предложения. Теперь ссылка внутри объекта `SmartPtr` оказалась висячей. На первый взгляд, тут нет ничего удивительного – разумеется, временный объект не может жить дольше предложения, в котором создан, и удаляется самое позднее по достижении завершающей его точки с запятой. Но читатель, более осведомленный в тонкостях языка, может спросить: «А разве стандарт не требует продлевать время жизни временного объекта, связанного с константной ссылкой?» Да, требует, например:

```
{
    const C& c = C();
    ..... c - не висячая ссылка! .....
}  // временный объект удаляется здесь
```

В этом фрагменте кода временный объект `C()` удаляется не в конце предложения, а только в конце времени жизни той ссылки, с которой связан. Так почему такой же трюк не работает для нашего объекта политики удаления? Да в каком-то смысле работает – временный объект, созданный в момент вычисления аргумента конструктора и связанный с аргументом, являющимся константной ссылкой, не был уничтожен на протяжении всего времени жизни этой ссылки, которое совпадает с протяженностью вызова конструктора. На самом деле он в любом случае не был бы уничтожен, т. к. все временные объекты, созданные в процессе вычисления аргументов функции, удаляются в конце предложения, содержащего вызов функции, т. е. по достижения завершающей его точки с запятой. В нашем случае функция – это конструктор объекта, поэтому время жизни временных объектов простирается до конца вызова конструктора. Однако оно не расширяется на время жизни объекта – константная ссылка-член объекта связана не с временным объектом, а с параметром конструктора, который сам по себе является константной ссылкой. Расширение времени жизни происходит только один раз – ссылка, связанная с временным объектом, продлевает его жизнь. Наличие еще одной ссылки, связанной с первой, не добавляет ничего нового, и эта ссылка может остаться висячей, если объект будет уничтожен. Поэтому если объект политики должен быть сохранен как член интеллектуального указателя, то его необходимо скопировать.

Обычно объекты политик малы, и их копирование – тривиальная операция. Но иногда политика может содержать нетривиальное внутреннее состояние, так что копирование обходится дорого. Можно также представить себе ситуацию, когда объект политики вообще не допускает копирования. В таких случаях имеет смысл переместить объект аргумента в член данных. Это легко сделать, если объявить перегруженный вариант, похожий на перемещающий конструктор:

```
template <typename T, typename DeletionPolicy = DeleteByOperator<T>>
class SmartPtr {
    public:
    explicit SmartPtr(T* p,
                      const DeletionPolicy& deletion_policy
    ) : p_(p),
        deletion_policy_(std::move(deletion_policy))
    {}
    explicit SmartPtr(T* p = nullptr,
                      DeletionPolicy&& deletion_policy = DeletionPolicy()
    ) : p_(p),
        deletion_policy_(std::move(deletion_policy))
    {}
    .....
    private:
    DeletionPolicy deletion_policy_;
};
```

Но, как мы сказали, объекты политик обычно невелики, так что их копирование редко составляет проблему.

Итак, у нас теперь есть класс интеллектуального указателя, который реализован только один раз, но так, что политику удаления можно настроить на этапе компиляции. Можно даже добавить новую политику удаления, которой не существовало в момент проектирования класса, и она будет работать при условии, что согласована с интерфейсом вызова. Далее мы рассмотрим другие способы реализации объектов политик.

Реализация политик

В предыдущем разделе мы узнали, как реализовать простейший объект политики. Политика может иметь любой тип, следующий соглашению об интерфейсе, и хранится в члене данных класса. Чаще всего объект политики генерируется по шаблону, однако он может быть и нешаблонным объектом, специфичным для определенного типа указателя, или даже функцией. Применение политики было ограничено конкретным аспектом поведения, например удалением объекта, которым владеет интеллектуальный указатель.

Существует несколько способов реализации и использования таких политик. Для начала еще раз приведем объявление интеллектуального указателя с политикой удаления:

```
template <typename T, typename DeletionPolicy = DeleteByOperator<T>>
class SmartPtr { ..... };
```

Теперь посмотрим, как можно сконструировать объект интеллектуального указателя:

```
class C { ..... };
SmartPtr<C, DeleteByOperator<C>> p(new C, DeleteByOperator<C>());
```

Один из недостатков такого дизайна сразу бросается в глаза – тип C четыре раза упомянут в определении объекта p – он должен быть одинаков во всех четырех местах, иначе код не откомпилируется. C++17 позволяет немного упростить это определение:

```
SmartPtr p(new C, DeleteByOperator<C>());
```

Здесь конструктор используется, чтобы вывести параметры шаблона класса из аргументов конструктора – так же, как это делается в шаблонах функций. Но остается еще два упоминания типа.

Альтернативная реализация, которая работает для политик без состояния, а также для объектов политик, чье внутреннее состояние не зависит от типов основного шаблона (в нашем случае от типа T в шаблоне SmartPtr), – сделать саму политику нешаблонным объектом, но снабдить ее шаблонной функцией-членом. Например, политика DeleteByOperator не содержит состояния (у объекта нет данных-членов) и может быть реализована без шаблона класса:

```
struct DeleteByOperator {
    template <typename T>
        void operator()(T* p) const {
        delete p;
    }
};
```

Это нешаблонный объект, поэтому ему не нужен параметр-тип. Шаблон функции-члена конкретизируется типом удаляемого объекта, и этот тип выводится компилятором. Поскольку тип объекта политики всегда один и тот же, нам не нужно думать о соответствии типов при создании объекта интеллектуального указателя:

```
SmartPtr<C, DeleteByOperator> p(new C, DeleteByOperator()); // до C++17
SmartPtr p(new C, DeleteByOperator());                      // C++17
```

Наш интеллектуальный указатель может использовать этот объект без внесения каких-либо изменений в шаблон SmartPtr, хотя можно было бы изменить аргумент шаблона по умолчанию:

```
template <typename T, typename DeletionPolicy = DeleteByOperator>
class SmartPtr { ..... };
```

Этот подход позволяет реализовать и более сложную политику, например политику удаления для кучи:

```
struct DeleteHeap {
  explicit DeleteHeap(SmallHeap& heap)
        : heap_(heap) {}
```

```
    template <typename T>
    void operator()(T* p) const {
        p->~T();
        heap_.deallocate(p);
    }
    private:
    Heap& heap_;
};
```

У этой политики имеется внутреннее состояние – ссылка на кучу, – но в объекте политики ничто не зависит от типа `T` удаляемого объекта, кроме функции-члена `operator()`. Таким образом, эту политику не нужно параметризировать типом объекта.

Поскольку главный шаблон `SmartPtr` не пришлось изменять при преобразовании наших политик из шаблона класса в нешаблонный класс с шаблонными функциями-членами, нет никаких причин, мешающих использовать оба типа политики с одним и тем же классом. Действительно, любая политика в виде шаблона класса из предыдущего раздела по-прежнему будет работать, поэтому мы можем использовать политики, реализованные как в виде классов, так и в виде шаблонов. Последнее может быть полезно, когда в политике имеются данные-члены, типы которых зависят от типа интеллектуального указателя.

Если политика реализована как шаблон класса, то для каждого класса на основе политики мы должны задать правильный тип, которым эта политика конкретизируется. Нередко данный процесс подразумевает много повторений – один и тот же тип используется для параметризации главного шаблона и его политик. Мы можем поручить эту работу компилятору, если будем использовать в качестве политики весь шаблон, а не какую-то его конкретизацию:

```
template <typename T, template <typename> class DeletionPolicy =
DeleteByOperator>
class SmartPtr {
    public:
    explicit SmartPtr(T* p = nullptr,
                      const DeletionPolicy<T>& deletion_policy =
                                               DeletionPolicy<T>()
    ) : p_(p),
        deletion_policy_(deletion_policy)
    {}
    ~SmartPtr() {
        deletion_policy_(p_);
    }
    .....
};
```

Обратите внимание на синтаксис второго параметра шаблона – `template <typename> class DeletionPolicy`. Это так называемый *шаблонный параметр шаблона* – параметр шаблона, сам являющийся шаблоном. Ключевое слово `class` необходимо в версии C++14 и более ранних; в C++17 его можно заменить на `typename`. Чтобы использовать этот параметр, его нужно конкретизировать

каким-то типом, в нашем случае это параметр-тип T главного шаблона. Это гарантирует согласованность типа объекта в главном шаблоне интеллектуального указателя и его политиках, хотя аргумент конструктора по-прежнему должен конструироваться с правильным типом:

```
SmartPtr<C, DeleteByOperator> p(new C, DeleteByOperator<C>());
```

Опять-таки, в C++17 параметры шаблона класса можно вывести из конструктора, и это работает также для шаблонных параметров шаблона:

```
SmartPtr p(new C, DeleteByOperator<C>());
```

Шаблонные параметры шаблона могут показаться привлекательной альтернативой обычным параметрам-типам, когда типы все равно конкретизируются по шаблону. Так почему же мы не пользуемся ими всегда? Оказывается, что у шаблонного параметра шаблона есть одно существенное ограничение – количество параметров шаблона должно в точности совпадать с заданным в спецификации, включая и аргументы по умолчанию. Иными словами, предположим, что имеется такой шаблон:

```
template <typename T, typename Heap = MyHeap> class DeleteHeap { ..... };
```

Этот шаблон нельзя использовать как параметр нашего интеллектуального указателя, т. к. в нем два параметра шаблона, а в объявлении SmartPtr мы указали только один (параметр, имеющий значение по умолчанию, все равно является параметром). С другой стороны, мы можем использовать конкретизацию этого шаблона для интеллектуального указателя с простым, а не шаблонным типом DeletionPolicy – нам просто нужен класс, и DeleteHeap<int, MyHeap> подойдет ничуть не хуже любого другого.

До сих пор мы всегда запоминали объект политики в члене данных класса. Этот подход – включение классов в более крупный класс – называется композицией. Но главный шаблон может получить доступ к алгоритмам, предоставляемым политиками, и другими способами, которые мы рассмотрим ниже.

Использование объектов политик

В рассмотренных до сих пор примерах объект политики хранился в члене данных класса. Обычно это предпочтительный способ хранения политик, но у него есть существенный недостаток – размер члена данных всегда больше нуля. Рассмотрим наш интеллектуальный указатель с одной из политик удаления:

```
template <typename T>
struct DeleteByOperator {
    void operator()(T* p) const {
        delete p;
    }
};

template <typename T, typename DeletionPolicy = DeleteByOperator<T>>
```

```
class SmartPtr {
    .....
    private:
    T* p_;
    DeletionPolicy deletion_policy_;
};
```

Заметим, что в объекте политики нет данных-членов. Однако его размер равен не нулю, а одному байту (в этом легко убедиться, напечатав значение `sizeof(DeleteByOperator<int>)`). Это необходимо, потому что в C++ каждый объект должен иметь уникальный адрес:

```
DeleteByOperator<int> d1;      // &d1 = ......
DeleteByOperator<long> d2;     // &d2 должно быть != &d1
```

Если два объекта расположены рядом в памяти, то разность между их адресами равна размеру первого объекта (плюс дополнительные байты для выравнивания на границу в памяти, если необходимо). Чтобы предотвратить размещение объектов `d1` и `d2` по одному адресу, стандарт требует, чтобы размер каждого объекта был не менее одного байта (это требование будет немного ослаблено в C++20, где появится способ использовать пустые объекты в качестве данных-членов без назначения каждому объекту уникального адреса).

Если объект используется как член данных другого класса, то он будет занимать по крайней мере столько памяти, каков его размер, в нашем случае – один байт. В предположении, что указатель занимает 8 байтов, размер всего объекта равен 9 байтам. Но размер объекта необходимо дополнить до ближайшего значения, удовлетворяющего требованиям выравнивания, – если адрес указателя должен быть выровнен на границу, кратную 8 байтам, то размер объекта может быть равен 8 или 16 байтам, промежуточное значение не годится. Таким образом, добавление пустого объекта политики в класс влечет за собой увеличение размера класса с 8 до 16 байтов. Это пустая и часто нежелательная трата памяти, особенно если объектов создается много, а именно так обстоит дело с указателями. И невозможно уговорить компилятор создать член данных нулевого размера, стандарт это запрещает. Но существует другой способ, позволяющий использовать политики без накладных расходов.

Альтернативой композиции является наследование – мы можем использовать политику в качестве базового класса для основного класса:

```
template <typename T, typename DeletionPolicy = DeleteByOperator<T>>
class SmartPtr : private DeletionPolicy {
    public:
    explicit SmartPtr(T* p = nullptr,
                      DeletionPolicy&& deletion_policy = DeletionPolicy()
    ) : DeletionPolicy(std::move(deletion_policy)),
        p_(p)
    {}
    ~SmartPtr() {
        DeletionPolicy::operator()(p_);
    }
```

```
    .....
    private:
    T* p_;
};
```

Этот подход опирается на известную оптимизацию *пустого базового класса* – если базовый класс пуст (в нем нет нестатических данных-членов), то при размещении в памяти его можно полностью исключить из производного класса. Она разрешена стандартом, но обычно не является обязательной (C++11 настаивает на такой оптимизации для некоторых классов, но не для тех, которые встречаются в этой главе). Даже если это и не требуется, современные компиляторы почти всегда это делают. Если выполнена оптимизация пустого базового класса, то размер производного класса `SmartPtr` ровно такой, какой необходим для хранения его данных-членов, т. е. 8 байтов.

Применяя для реализации политик наследование, мы должны выбрать, каким оно будет: открытым или закрытым. Обычно политики используются, чтобы предоставить реализацию какого-то конкретного аспекта поведения. Такое наследование ради реализации выражается закрытым наследованием. В некоторых случаях политику можно использовать для изменения открытого интерфейса класса, тогда следует выбирать открытое наследование. Что касается политики удаления, то мы не изменяем интерфейс класса – интеллектуальный указатель всегда удаляет объект в конце своей жизни, единственный вопрос – как. Поэтому для политики удаления следует использовать закрытое наследование.

Политика удаления, в которой используется оператор `delete`, не содержит состояния, но в некоторых политиках имеются данные-члены, которые следует сохранить при передаче объекта конструктору. Поэтому в общем случае политику, являющуюся базовым классом, следует инициализировать аргументом конструктора путем копирования или перемещения его в базовый класс, аналогично тому, как мы инициализируем данные-члены. Базовые классы всегда инициализируются с помощью списка инициализации членов раньше данных-членов производного класса. Наконец, для вызова функции-члена базового класса можно использовать синтаксическую конструкцию `base_type::function_name()`, в нашем случае – `DeletionPolicy::operator()(p_)`.

Наследование и композиция – два варианта включения класса политики в основной класс. В общем случае предпочтительнее композиция, если только нет веских причин использовать наследование. Одну такую причину мы уже видели – оптимизация пустого базового класса. Наследование также необходимо, если мы хотим повлиять на открытый интерфейс класса.

Пока что нашему интеллектуальному указателю недостает нескольких важных черт, которые присутствуют в большинстве реализаций интеллектуальных указателей. Одна такая черта – возможность освободить указатель, т. е. предотвратить автоматическое уничтожение объекта. Это может быть полезно, если объект иногда уничтожается другими средствами или если время жизни

объекта необходимо продлить, а владение им передать другому объекту – владельцу ресурсов. Такую возможность легко добавить в наш интеллектуальный указатель:

```
template <typename T, typename DeletionPolicy, typename ReleasePolicy>
class SmartPtr : private DeletionPolicy {
    public:
    ~SmartPtr() {
        DeletionPolicy::operator()(p_);
    }
    void release() { p_ = nullptr; }
    .....
    private:
    T* p_;
};
```

Теперь, если вызвать для нашего интеллектуального указателя метод `p.release()`, деструктор не будет делать ничего. Предположим, что такая функциональность необходима не всем интеллектуальным указателям, и мы решили сделать ее факультативной. Можно добавить параметр шаблона `ReleasePolicy`, который будет управлять наличием функции-члена `release()`, но что он должен делать? Конечно, мы могли бы перенести реализацию `SmartPtr::release()` в политику:

```
template <typename T>
class WithRelease {
    public:
    void release(T*& p) { p = nullptr; }
};
```

Теперь реализация `SmartPtr` должна просто вызвать `ReleasePolicy::release(p_)`, чтобы делегировать работу `release()` политике. Но что это должна быть за работа, если мы вообще не хотим поддерживать освобождение указателя? Политика без освобождения могла бы не делать ничего, но это только вводит в заблуждение пользователя, который ожидает, что раз он вызвал `release()`, то объект не будет уничтожен. Можно было бы завершить программу при обращении к этой функции. Но это превращает логическую ошибку программиста – попытку освободить интеллектуальный указатель, не допускающий освобождения, – в ошибку времени выполнения. Лучше всего, если бы в классе `SmartPtr` вообще не было функции-члена `release()`, если она не нужна. Тогда некорректный код не откомпилировался бы. Единственный способ добиться этого – заставить политику вставлять новую открытую функцию-член в открытый интерфейс основного шаблона. Это можно сделать с помощью открытого наследования:

```
template <typename T, typename DeletionPolicy, typename ReleasePolicy>
class SmartPtr : private DeletionPolicy, public ReleasePolicy { ..... };
```

Теперь, если политика освобождения имеет открытую функцию-член `release()`, то ее будет иметь и класс `SmartPtr`.

Это решает проблему интерфейса. Осталась небольшая проблема в части реализации. Функция-член `release()` теперь перемещена в класс политики, но должна оперировать членом `p_` производного класса. Один из способов решить этот вопрос – передать ссылку на данный указатель из производного класса в базовый класс политики в момент конструирования. Но это уродливая реализация – 8 байтов памяти расходуется впустую для хранения ссылки на данные, которые и так *почти здесь*, точнее хранятся в производном классе в непосредственной близости от базового. Гораздо лучше было бы привести базовый класс к правильному производному. Конечно, это возможно, только если базовый класс знает, что такое правильный производный класс. Решение проблемы дает паттерн Рекурсивный шаблон (Curiously Recurring Template Pattern – **CRTP**), который мы уже изучали в этой книге: политика должна быть шаблоном (поэтому нам понадобится шаблонный параметр шаблона), который конкретизируется типом производного класса. Таким образом, класс `SmartPtr` одновременно является производным классом политики освобождения и ее шаблонным параметром:

```
template <typename T,
          typename DeletionPolicy = DeleteByOperator<T>,
          template <typename> class ReleasePolicy = WithRelease>
class SmartPtr :
    private DeletionPolicy,
    public ReleasePolicy<SmartPtr<T, DeletionPolicy, ReleasePolicy>>
{ ..... };
```

Заметим, что шаблон `ReleasePolicy` специализирован конкретизацией шаблона `SmartPtr`, включающей все его политики, в т. ч. саму `ReleasePolicy`.

Теперь политика освобождения знает тип производного класса и может привести себя к этому типу. Это приведение всегда безопасно, потому что производный класс правилен по построению:

```
template <typename P> class WithRelease {
public:
    void release() { static_cast<P*>(this)->p_ = NULL; }
};
```

Вместо параметра шаблона `P` будет подставлен тип интеллектуального указателя. После того как интеллектуальный указатель открыто унаследует политику освобождения, будет унаследована и открытая функция-член `release()` политики, которая, следовательно, станет частью открытого интерфейса интеллектуального указателя.

Последняя деталь реализации политики освобождения касается доступа. До сих пор член `p_` класса `SmartPtr` был закрытым, так что к нему нельзя было напрямую обратиться из базовых классов. Решение – объявить соответствующий базовый класс другом производного:

```
template <typename T,
          typename DeletionPolicy = DeleteByOperator<T>,
          template <typename> class ReleasePolicy = WithRelease>
```

```
class SmartPtr :
    private DeletionPolicy,
    public ReleasePolicy<SmartPtr<T, DeletionPolicy, ReleasePolicy>>
{
    .....
    private:
    friend class ReleasePolicy<SmartPtr>;
    T* p_;
};
```

Отметим, что в теле класса `SmartPtr` нет нужды повторять все параметры шаблона. Краткая запись `SmartPtr` ссылается на текущий конкретизированный шаблон. Это не распространяется на часть объявления класса перед открывающей скобкой, поэтому приходится повторить параметры шаблона при задании политики в качестве базового класса.

Написать политику без освобождения столь же просто:

```
template <typename P> class NoRelease {
};
```

Здесь нет функции `release()`, поэтому попытка вызвать метод `release()` от имени интеллектуального указателя при такой политике не откомпилируется. Это удовлетворяет требованию включать открытую функцию-член `release()` только тогда, когда ее имеет смысл вызывать. Проектирование на основе политик – сложный паттерн, а ведь редко бывает так, что есть только один способ сделать нечто. Так и здесь – существует другой путь к достижению той же цели, и мы рассмотрим его далее в этой главе.

Иногда объекты политик используются еще одним способом. Это относится к политикам, ни одна версия которых, по определению, не имеет внутреннего состояния. Например, некоторые наши политики удаления обходятся без внутреннего состояния, но вот политика, которая хранит ссылку на кучу, созданную вызывающей стороной, к таковым не относится. Значит, эта политика не всегда лишена состояния. Политику освобождения всегда можно рассматривать как не имеющую состояния; не видно причин включать в нее данные-члены, но у нее есть другое ограничение – предполагается, что она используется посредством открытого наследования, т. к. ее основное назначение – включить новую открытую функцию-член. Рассмотрим еще один аспект поведения, который иногда требуется настраивать, – отладку или протоколирование. Для отладки удобно выводить сообщения о том, что объектом завладел интеллектуальный указатель, и о том, что объект удален. Чтобы поддержать это требование, можно добавить к интеллектуальному указателю политику отладки. У этой политики только одно назначение – печатать что-нибудь, когда интеллектуальный указатель конструируется или уничтожается. Ей не понадобится доступ к интеллектуальному указателю, если мы будем передавать значение указателя функции печати. Поэтому функцию печати можно сделать статической в политике отладки и вообще не хранить ее в классе интеллектуального указателя:

```
template <typename T,
          typename DeletionPolicy,
          typename DebugPolicy = NoDebug>
class SmartPtr : private DeletionPolicy {
    public:
    explicit SmartPtr(T* p = nullptr,
                      DeletionPolicy&& deletion_policy = DeletionPolicy()
        ) : DeletionPolicy(std::move(deletion_policy)),
            p_(p)
    {
        DebugPolicy::constructed(p_);
    }
    ~SmartPtr() {
        DebugPolicy::deleted(p_);
        DeletionPolicy::operator()(p_);
    }
    .....
    private:
    T* p_;
};
```

Реализация политики отладки тривиальна:

```
struct Debug {
    template <typename T> static void constructed(const T* p) {
        std::cout << "Сконструирован SmartPtr для объекта "
                  << static_cast<const void*>(p) << std::endl;
    }
    template <typename T> static void deleted(const T* p) {
        std::cout << "Уничтожен SmartPtr для объекта "
                  << static_cast<const void*>(t) << std::endl;
    }
};
```

Мы решили реализовать политики как нешаблонный класс с шаблонными статическими функциями-членами. Можно было бы вместо этого реализовать ее в виде шаблона, параметризованного типом объекта `T`. Версия без отладки, подразумеваемая по умолчанию, еще проще. В ней должны быть определены те же функции, но они ничего не делают:

```
struct NoDebug {
    template <typename T> static void constructed(const T* p) {}
    template <typename T> static void deleted(const T* p) {}
};
```

Можно ожидать, что компилятор встроит пустые шаблонные функции в точке вызова и уберет это обращение целиком, т. к. никакой код генерировать не надо.

Заметим, что, выбрав эту реализацию политик, мы приняли несколько ограничительное проектное решение – все версии политики отладки не должны содержать состояния. Возможно, мы еще пожалеем об этом решении, если, например, понадобится хранить в политике отладки заданный пользователем

поток вывода, а не использовать `std::cout`. Но даже в этом случае изменить придется только реализацию класса интеллектуального указателя, а клиентский код будет работать, как работал.

Мы рассмотрели три способа внедрить объекты политик в класс – путем композиции, путем наследования (открытого или закрытого) и исключительно на этапе компиляции, когда объект политики не нужно хранить в главном объекте во время выполнения. Теперь перейдем к более сложным приемам проектирования на основе политик.

Продвинутое проектирование на основе политик

Приемы, описанные в предыдущем разделе, образуют фундамент проектирования на основе политик – политики могут быть классами, конкретизациями шаблонов или шаблонами (в последнем случае для их использования нужны шаблонные параметры шаблонов). Классы политик можно внедрять с помощью композиции, наследования или использовать статически на этапе компиляции. Если политика должна знать тип основного класса, в который внедрена, то можно использовать паттерн CRTP. Все остальное – вариации на ту же тему, а также различные способы сочетания нескольких техник для получения чего-то нового. Некоторые из этих продвинутых приемов мы сейчас и рассмотрим.

Политики для конструкторов

Политики можно использовать для настройки почти любого аспекта реализации, а также для изменения интерфейса класса. Однако попытка настроить поведение конструкторов класса с помощью политик наталкивается на уникальные сложности.

В качестве примера рассмотрим еще одно ограничение на наш интеллектуальный указатель. До сих пор объект, которым владел интеллектуальный указатель, удалялся при уничтожении последнего. Если интеллектуальный указатель поддерживает освобождение, то мы можем вызвать функцию-член `release()` и взять на себя всю ответственность за удаление объекта. Но как гарантировать, что объект будет удален? Скорее всего, мы передадим владение им другому интеллектуальному указателю:

```
SmartPtr<C> p1(new C);
SmartPtr<C> p2(&*p1);    // теперь два указателя владеют одним объектом
p1.release();
```

Этот подход многословен и чреват ошибками – мы временно позволяем двум указателям владеть одним и тем же объектом. Если в этот момент произойдет нечто такое, что заставит удалить оба указателя, то мы уничтожим один объект дважды. Кроме того, нужно помнить о том, что требуется удалить один из указателей, но только один. Можно взглянуть на проблему на более высоком уровне – мы пытаемся передать владение объектом от одного интеллектуального указателя другому.

Правильнее было бы переместить первый указатель во второй:

```
SmartPtr<C> p1(new C);
SmartPtr<C> p2(std::move(p1));
```

Теперь первый указатель остался в состоянии «перемещен из», которое мы можем определить (единственное требование – что вызов деструктора по-прежнему должен быть допустим). Мы определим его как состояние, в котором указатель не владеет никаким объектом, т. е. освобожден. Второй указатель получает объект во владение и в должное время удалит его.

Для поддержки этой функциональности необходимо реализовать перемещающий конструктор. Однако иногда могут существовать причины для предотвращения передачи владения. Поэтому указатели могут быть перемещаемыми и неперемещаемыми. Это наводит на мысль ввести еще одну политику, которая будет управлять перемещаемостью:

```
template <typename T,
          typename DeletionPolicy = DeleteByOperator<T>,
          typename MovePolicy = MoveForbidden
>
class SmartPtr .....;
```

Для простоты мы вернулись к единственной политике – удаления. Остальные рассмотренные выше политики можно добавить наряду с новой политикой `MovePolicy`. Политику удаления можно реализовать любым из изученных нами способов. Поскольку она выигрывает от оптимизации пустого базового класса, оставим реализацию на основе наследования. Политику перемещения можно реализовать несколькими разными способами, но, пожалуй, самый простой – наследование. Мы будем предполагать, что освобождение всегда доступно, и вернем функцию-член `release()` в класс интеллектуального указателя:

```
template <typename T,
          typename DeletionPolicy = DeleteByOperator<T>,
          typename MovePolicy = MoveForbidden
         >
class SmartPtr : private DeletionPolicy,
                 private MovePolicy
{
    public:
    explicit SmartPtr(T* p = nullptr,
                      DeletionPolicy&& deletion_policy = DeletionPolicy()
    ) : DeletionPolicy(std::move(deletion_policy)),
        MovePolicy(),
        p_(p)
    {}
    SmartPtr(SmartPtr&& other)
        : DeletionPolicy(std::move(other)),
          MovePolicy(std::move(other)),
          p_(other.p_)
    {
```

```
        other.release();
    }
    ~SmartPtr() {
        DeletionPolicy::operator()(p_);
    }
    void release() { p_ = NULL; }
    T* operator->() { return p_; }
    const T* operator->() const { return p_; }
    T& operator*() { return *p_; }
    const T& operator*() const { return *p_; }
    private:
    T* p_;
    SmartPtr(const SmartPtr&) = delete;
    SmartPtr& operator=(const SmartPtr&) = delete;
};
```

Коль скоро обе политики включены с помощью закрытого наследования, теперь у нас имеется производный объект с несколькими базовыми классами. Такое множественное наследование довольно часто встречается в проектировании на основе политик в C++, так что тревожиться тут не о чем. Иногда эту технику называют *подмешиванием* (mix-in), потому что реализация производного класса *смешивается* с ингредиентами, предоставленными базовыми классами. В C++ термин *подмешивание* также применяется для совершенно другой схемы наследования, относящейся к CRTP, поэтому его употребление часто вносит путаницу (в большинстве объектно-ориентированных языков *подмешивание* всегда обозначает то применение множественного наследования, которое мы здесь видим).

Новым в нашем классе интеллектуального указателя является перемещающий конструктор, который присутствует в классе `SmartPtr` безусловно. Однако для его реализации необходимо, чтобы все базовые классы были перемещаемыми. Это открывает возможность отключить поддержку перемещения с помощью политики, запрещающей перемещение:

```
struct MoveForbidden {
    MoveForbidden() = default;
    MoveForbidden(MoveForbidden&&) = delete;
    MoveForbidden(const MoveForbidden&) = delete;
    MoveForbidden& operator=(MoveForbidden&&) = delete;
    MoveForbidden& operator=(const MoveForbidden&) = delete;
};
```

Политика с перемещением гораздо проще:

```
struct MoveAllowed {
};
```

Теперь можно сконструировать перемещаемый и неперемещаемый указатели:

```
class C { ..... };
SmartPtr<C, DeleteByOperator<C>, MoveAllowed> p = .....;
```

```
auto p1(std::move(p));              // OK
SmartPtr<C, DeleteByOperator<C>, MoveForbidden> q = .....;
auto q1(std::move(q));              // не компилируется
```

Попытка переместить неперемещаемый указатель не компилируется, потому что один из базовых классов, MoveForbidden, неперемещаемый (в нем нет перемещающего конструктора). Заметим, что указатель p из предыдущего примера, находящийся в состоянии «перемещен из», можно безопасно удалить, но никаким другим способом использовать нельзя. В частности, его нельзя разыменовать.

Раз уж мы имеем дело с перемещаемыми указателями, имеет смысл предоставить и перемещающий оператор присваивания:

```
template <typename T,
          typename DeletionPolicy = DeleteByOperator<T>,
          typename MovePolicy = MoveForbidden
         >
class SmartPtr : private DeletionPolicy,
                 private MovePolicy
{
    public:
    SmartPtr(SmartPtr&& other)
        : DeletionPolicy(std::move(other)),
          MovePolicy(std::move(other)),
          p_(other.p_)
    {
        other.release();
    }
    SmartPtr& operator=(SmartPtr&& other) {
        if (this == &other) return *this;
        DeletionPolicy::operator=(std::move(other));
        MovePolicy::operator=(std::move(other));
        p_ = other.p_;
        other.release();
        return *this;
    }
    .....
};
```

Обратите внимание на проверку присваивания себе – хотя идут споры по поводу перемещения в себя, и вполне возможно внесение дополнений в стандарт, по общепринятому соглашению такое «самоперемещение» всегда должно оставлять объект в корректном состоянии (примером может служить состояние «перемещен из»). Необязательно, чтобы перемещение в себя ничего не делало, но это тоже допустимо. Отметим также способ реализации перемещающего присваивания в базовых классах – проще всего вызывать оператор перемещающего присваивания каждого базового класса напрямую. Нет необходимости приводить производный класс other к каждому базовому типу – это приведение производится неявно. Не забудем освободить перемещенный ука-

затель вызовом release(), иначе объект, которым владеют оба этих указателя, будет удален дважды.

Для простоты мы проигнорировали все рассмотренные ранее политики. Это нормально – не в каждом проекте всем подряд должны управлять политики, да и в любом случае комбинировать несколько политик элементарно. Однако это хорошая возможность подчеркнуть, что иногда различные политики взаимосвязаны – например, если используется одновременно политика освобождения и политика перемещения, то применение действующей (а не пустой) политики перемещения подразумевает, что объект должен поддерживать освобождение. Воспользовавшись метапрограммированием шаблонов, мы сможем гарантировать наличие такой зависимости между политиками.

Заметим, что политику, которая должна включать или исключать конструкторы, необязательно использовать в качестве базового класса – перемещающее конструирование или присваивание перемещает также все данные-члены, и потому наличие неперемещаемого члена с тем же успехом запретит операции перемещения. Здесь более важная причина использовать наследование – оптимизация пустого базового класса.

Мы рассмотрели, как сделать наши указатели перемещаемыми. А как насчет копирования? До сих пор копирование было вообще запрещено – копирующие конструктор и оператор присваивания помечены квалификатором delete с самого начала. И это разумно, т. к. мы не хотим, чтобы два интеллектуальных указателя владели одним и тем же объектом и удаляли его дважды. Но есть еще один тип владения, для которого копирование очень даже имеет смысл, – совместное владение, реализованное, например, разделяемым указателем с подсчетом ссылок. Для указателя такого типа копирование разрешено, и оба указателя обладают равными правами владения на объект. Счетчик ссылок показывает, сколько указателей на данный объект существует в программе. Когда последний указатель, владеющий объектом, удаляется, вместе с ним удаляется и объект, потому что никаких ссылок на него больше не осталось.

Существует несколько способов реализовать разделяемый указатель с подсчетом ссылок, но начнем с проектирования класса и его политик. Нам попрежнему нужна политика удаления, и имеет смысл завести одну политику для управления операциями копирования и перемещения. Для простоты снова ограничимся только политиками, которые сейчас являются предметом изучения:

```
template <typename T,
          typename DeletionPolicy = DeleteByOperator<T>,
          typename CopyMovePolicy = NoMoveNoCopy
         >
class SmartPtr : private DeletionPolicy,
                 public CopyMovePolicy
{
    public:
    explicit SmartPtr(T* p = nullptr,
```

```
                    DeletionPolicy&& deletion_policy = DeletionPolicy()
    ) : DeletionPolicy(std::move(deletion_policy)),
        p_(p)
    {}
    SmartPtr(SmartPtr&& other)
        : DeletionPolicy(std::move(other)),
          CopyMovePolicy(std::move(other)),
          p_(other.p_)
    {
        other.release();
    }
    SmartPtr(const SmartPtr& other)
        : DeletionPolicy(other),
          CopyMovePolicy(other),
          p_(other.p_)
    {
    }
    ~SmartPtr() {
        if (CopyMovePolicy::must_delete()) DeletionPolicy::operator()(p_);
    }
    void release() { p_ = NULL; }
    T* operator->() { return p_; }
    const T* operator->() const { return p_; }
    T& operator*() { return *p_; }
    const T& operator*() const { return *p_; }
    private:
    T* p_;
};
```

Операции копирования больше не удаляются безусловно. Включены копирующий и перемещающий конструкторы (оба оператора присваивания для краткости опущены, но должны быть реализованы так же, как раньше).

Удаление объекта в деструкторе интеллектуального указателя больше не является безусловным – в случае указателя с подсчетом ссылок политика копирования ведет счетчик ссылок и знает, когда осталась только одна копия указателя на объект.

Сам класс интеллектуального указателя предъявляет требования к классам политик. Политика, не подразумевающая перемещения и копирования, должна запретить все операции перемещения и копирования:

```
class NoMoveNoCopy {
    protected:
    NoMoveNoCopy() = default;
    NoMoveNoCopy(NoMoveNoCopy&&) = delete;
    NoMoveNoCopy(const NoMoveNoCopy&) = delete;
    NoMoveNoCopy& operator=(NoMoveNoCopy&&) = delete;
    NoMoveNoCopy& operator=(const NoMoveNoCopy&) = delete;
    constexpr bool must_delete() const { return true; }
};
```

Кроме того, некопируемый интеллектуальный указатель всегда удаляет объект, которым владеет, в деструкторе, поэтому функция-член `must_delete()`

должна возвращать true. Заметим, что эту функцию должны реализовывать все политики копирования, даже если она тривиальна, иначе класс интеллектуального указателя не откомпилируется. Однако мы вправе ожидать, что компилятор уберет обращение к ней и будет безусловно вызывать деструктор, если используется такая политика.

Политика, допускающая только перемещение, аналогична реализованной ранее, но теперь мы должны явно разрешить операции перемещения и запретить операции копирования:

```
class MoveNoCopy {
    protected:
    MoveNoCopy() = default;
    MoveNoCopy(MoveNoCopy&&) = default;
    MoveNoCopy(const MoveNoCopy&) = delete;
    MoveNoCopy& operator=(MoveNoCopy&&) = default;
    MoveNoCopy& operator=(const MoveNoCopy&) = delete;
    constexpr bool must_delete() const { return true; }
};
```

И здесь удаление безусловное (указатель внутри объекта интеллектуального указателя может быть нулевым, если объект был перемещен, но это не мешает вызову для него оператора delete). Эта политика разрешает компилировать перемещающие конструктор и оператор присваивания; класс SmartPtr предоставляет корректную реализацию этих операций, и никакой дополнительной поддержки со стороны политики не требуется.

Политика копирования с подсчетом ссылок гораздо сложнее. Здесь мы должны определиться с реализацией разделяемого указателя. Простейшая реализация выделяет память для счетчика путем отдельного обращения к распределителю, и это управляется политикой копирования. Начнем с политики копирования с подсчетом ссылок, которая не разрешает операций перемещения:

```
class NoMoveCopyRefCounted {
    protected:
    NoMoveCopyRefCounted() : count_(new size_t(1)) {}
    NoMoveCopyRefCounted(const NoMoveCopyRefCounted& other)
        : count_(other.count_)
    {
        ++(*count_);
    }
    NoMoveCopyRefCounted(NoMoveCopyRefCounted&&) = delete;
    ~NoMoveCopyRefCounted() {
        --(*count_);
        if (*count_ == 0) {
            delete count_;
        }
    }
    bool must_delete() const { return *count_ == 1; }
    private:
    size_t* count_;
};
```

Когда конструируется интеллектуальный указатель с этой политикой копирования, выделяется память для нового счетчика ссылок, и счетчик инициализируется значением 1 (у нас имеется один интеллектуальный указатель на объект – тот, который мы сейчас конструируем). Когда интеллектуальный указатель копируется, копируются и все его базовые классы, включая политику копирования. Копирующий конструктор этой политики просто увеличивает счетчик ссылок на единицу. При удалении интеллектуального указателя счетчик ссылок уменьшается на единицу. Когда удаляется последний интеллектуальный указатель, вместе с ним удаляется и счетчик. Политика копирования управляет также тем, когда удаляется объект, на который указывает указатель, – это происходит, когда счетчик ссылок равен 1, т. е. мы собираемся удалить последний оставшийся указатель на объект. Конечно, очень важно следить за тем, чтобы счетчик не удалялся раньше вызова функции `must_delete()`. Выполнение этого условия гарантируется, потому что деструкторы базовых классов вызываются после деструктора производного – производный класс последнего интеллектуального указателя увидит, что счетчик равен 1, и удалит объект; затем деструктор политики копирования еще раз уменьшит счетчик на 1, увидит, что он обратился в ноль, и удалит сам счетчик.

Имея такую политику, мы можем реализовать совместное владение объектом:

```
SmartPtr<C, DeleteByOperator<C>, NoMoveCopyRefCounted> p1{new C};
auto p2(p1);
```

Теперь на один объект указывают два указателя, и счетчик ссылок равен 2. Объект удаляется вместе с последним из этих двух указателей в предположении, что больше копий не создавалось. Интеллектуальный указатель копируемый, но не перемещаемый:

```
SmartPtr<C, DeleteByOperator<C>, NoMoveCopyRefCounted> p1{new C};
auto p2(std::move(p1));        // не компилируется
```

Вообще говоря, если уж поддерживается копирование с подсчетом ссылок, то не видно причин запрещать операции перемещения, если только они действительно не нужны (а тогда реализация без поддержки перемещения может оказаться немного эффективнее). Для поддержки перемещения необходимо подумать о состоянии «перемещено из» политики с подсчетом ссылок – очевидно, что в этом состоянии не следует уменьшать счетчик ссылок при удалении объекта, поскольку указатель уже не владеет объектом. Проще всего сбросить указатель на счетчик ссылок, чтобы он стал недоступен политике копирования, но тогда политика копирования должна учитывать специальный случай нулевого указателя на счетчик:

```
class MoveCopyRefCounted {
    protected:
    MoveCopyRefCounted() : count_(new size_t(1)) {}
    MoveCopyRefCounted(const MoveCopyRefCounted& other)
        : count_(other.count_)
```

```
    {
        if (count_) ++(*count_);
    }
    ~MoveCopyRefCounted() {
        if (!count_) return;
        --(*count_);
        if (*count_ == 0) {
            delete count_;
        }
    }
    MoveCopyRefCounted(MoveCopyRefCounted&& other)
        : count_(other.count_)
    {
        other.count_ = nullptr;
    }
    bool must_delete() const { return count_ && *count_ == 1; }
    private:
    size_t* count_;
};
```

Наконец, политика копирования с подсчетом ссылок должна также поддерживать операции присваивания. Они реализуются по аналогии с копирующим или перемещающим конструктором.

Как видим, некоторые реализации политики могут оказаться довольно сложными, а их взаимодействия – тем более. К счастью, проектирование на основе политик прекрасно подходит для написания тестопригодных объектов. Это применение настолько важно, что заслуживает отдельного упоминания.

Применение политик для тестирования

Теперь мы покажем, как с помощью проектирования на основе политики улучшить тесты. В частности, политики можно использовать, чтобы сделать код более пригодным для автономного тестирования. Для этого следует подставить специальную тестовую политику вместо обычной. Продемонстрируем это на примере политики с подсчетом ссылок из предыдущего раздела.

Конечно, для этой политики главная проблема – правильно подсчитывать ссылки. Легко придумать тесты, которые будут проверять все граничные случаи подсчета:

```
// Тест 1: только один указатель
{
    SmartPtr<C, .....> p(new C);
} // C должен быть удален здесь

// Тест 2: одна копия
{
    SmartPtr<C, .....> p(new C);
    {
        auto p1(p);    // счетчик ссылок должен быть равен 2
    } // C не должен быть удален здесь
}     // C должен быть удален здесь
```

Трудная часть – проверить, что код действительно работает, как предполагается. Мы знаем, каким должен быть счетчик ссылок, но нет никакого способа проверить его истинное значение. Мы знаем, когда объект должен быть удален, но трудно убедиться, что он и вправду удален. Программа, вероятно, «грохнется», если удалить объект дважды, но даже в этом нет уверенности. А еще труднее отловить случай, когда объект так и не был удален.

По счастью, можно использовать политики, чтобы у тестов появилась возможность взглянуть на работу объекта изнутри. Например, можно создать тестопригодную обертку для политики с подсчетом ссылок:

```
class NoMoveCopyRefCounted {
    .....
    protected:
    size_t* count_;
};
class NoMoveCopyRefCountedTest : public NoMoveCopyRefCounted {
    public:
    using NoMoveCopyRefCounted::NoMoveCopyRefCounted;
    size_t count() const { return *count_; }
};
```

Заметим, что нам пришлось сделать закрытый член данных `count_` защищенным в главной политике копирования. Можно было бы вместо этого объявить тестовую политику другом, но тогда это пришлось бы делать для каждой новой политики. Вот теперь можно перейти к реализации тестов:

```
// Тест 1: только один указатель
{
    SmartPtr<C, ..... NoMoveCopyRefCountedTest> p(new C);
    assert(p.count() == 1);
} // C должен быть удален здесь

// Тест 2: одна копия
{
    SmartPtr<C, ..... NoMoveCopyRefCountedTest> p(new C);
    {
        auto p1(p); // счетчик ссылок должен быть равен 2
        assert(p.count() == 2);
        assert(p1.count() == 2);
        assert(&*p == &*p1);
    } // C не должен быть удален здесь
    assert(p.count == 1);
} // C должен быть удален здесь
```

Аналогично можно создать оснащенную политику удаления, которая проверяет, будет ли объект удален, или записывать в какой-то внешний объект-регистратор, что объект действительно был удален, а затем проверять, что удаление запротоколировано.

Читатель, наверное, уже заметил, что объявления объектов на основе политик могут быть довольно длинными:

```
SmartPtr<C, DeleteByOperator<T>, MoveNoCopy, WithRelease, Debug> p( .....);
```

Это одна из проблем, на которые чаще всего жалуются, и следует рассмотреть некоторые средства ее смягчения.

Адаптеры и псевдонимы политик

Пожалуй, самый очевидный недостаток проектирования на основе политик – способ объявления конкретных объектов, точнее, длинный список политик, который нужно тащить за собой повсюду. Рациональное использование параметров по умолчанию позволяет упростить наиболее распространенные случаи применения. Рассмотрим, к примеру, следующее длинное объявление:

```
SmartPtr<C, DeleteByOperator<T>, MoveNoCopy, WithRelease, NoDebug> p( ..... );
```

Его можно сократить до:

```
SmartPtr<C> p( ..... );
```

Это можно сделать, если умолчания представляют наиболее распространенный случай перемещаемого указателя без отладки, в котором используется оператор `delete`. Но какой смысл добавлять политики, если мы не собираемся их использовать? Продуманный порядок параметров шаблона позволит сделать наиболее востребованные комбинации политик короче. Например, если чаще всего используется политика удаления, то новый указатель с другой политикой удаления и остальными политиками по умолчанию можно будет объявить, не повторяя политики, которые мы не хотим изменять:

```
SmartPtr<C, DeleteHeap<T>> p( ..... );
```

Но проблема редко используемых политик все равно остается. К тому же политики часто добавляются впоследствии, как дополнительные возможности. И почти всегда они помещаются в конец списка параметров, иначе потребовалось бы переписывать весь код, в котором объявлен класс с политиками, чтобы изменить порядок параметров. Однако добавленные позже политики не обязательно используются редко, поэтому такая эволюция дизайна может привести к необходимости явно выписывать много аргументов-политик, даже если они имеют значения по умолчанию, – только чтобы изменить один из последних аргументов.

У этой проблемы нет общего решения в рамках традиционного проектирования на основе политик, но на практике нередко бывает так, что есть совсем немного широкоупотребительных групп политик и еще некоторые частые вариации. Например, большая часть наших интеллектуальных указателей, наверное, использует оператор `delete` и поддерживает перемещение и освобождение, но часто приходится переключаться между отладочной и производственной версиями. Тогда можно создать адаптеры, которые преобразуют класс с большим количеством политик в новый интерфейс, который раскрывает только часто изменяемые политики, а для остальных фиксирует типичные

значения. В большом проекте, вероятно, понадобится несколько таких адаптеров, потому что употребительные наборы политик могут изменяться.

Создать такой адаптер можно, например, с помощью наследования:

```
template <typename T,
          typename DebugPolicy = NoDebug
         >
class SmartPtrAdapter :
public SmartPtr<T, DeleteByOperator<T>, MoveNoCopy, WithRelease,
DebugPolicy>
{.....};
```

При этом создается шаблон производного класса, в котором некоторые параметры шаблона базового класса фиксированы, а остальные оставлены параметризуемыми. Наследуется весь открытый интерфейс базового класса, но в отношении конструкторов базового класса необходима осторожность. По умолчанию они не наследуются, поэтому в новом производном классе будут только конструкторы по умолчанию, сгенерированные компилятором. Скорее всего, это не то, что нам нужно. В C++11 проблема легко решается – объявление `using` включает все конструкторы базового класса в производный:

```
template <typename T,
          typename DebugPolicy = NoDebug
         >
class SmartPtrAdapter :
public SmartPtr<T, DeleteByOperator<T>, MoveNoCopy,
                  WithRelease, DebugPolicy>
{
    using SmartPtr<T, DeleteByOperator<T>, MoveNoCopy,
                     WithRelease, DebugPolicy>::SmartPtr;
};
```

К сожалению, в C++03 не существует простого способа повторить все необходимые конструкторы.

Теперь новый адаптер можно использовать, когда нужен интеллектуальный указатель с предустановленными политиками, кроме политики отладки, которую желательно легко изменять:

```
SmartPtrAdapter<C, Debug> p1{new C); // отладочный указатель
SmartPtrAdapter<C> p2{new C);        // указатель без отладки
```

В C++11 есть даже более простой способ решить задачу – псевдонимы шаблонов (иногда они называются *шаблонными typedef*):

```
template <typename T,
          typename DebugPolicy = NoDebug
         >
using SmartPtrAdapter =
    SmartPtr<T, DeleteByOperator<T>, MoveNoCopy, WithRelease, DebugPolicy>;
```

Это предложение делает примерно то же, что обычный `typedef`, – новых типов или шаблонов не появляется, а создается лишь псевдоним для вызова по

новому имени существующего шаблона, в котором некоторым параметрам присвоены заданные значения.

Как было сказано в самом начале, самое распространенное применение политик – выбор конкретной реализации некоторого аспекта поведения класса. Иногда такие вариации отражаются еще и в открытом интерфейсе класса – некоторые операции имеют смысл для одних реализаций и не имеют для других, а лучший способ гарантировать, что операция, несовместимая с реализацией, никогда не будет затребована, – просто не предоставлять ее.

Теперь мы вернемся к вопросу об избирательной активации частей открытого интерфейса с помощью политик.

Применение политик для управления открытым интерфейсом

Выше мы использовали политики для управления открытым интерфейсом одним из двух способов. Во-первых, мы могли внедрить открытую функцию-член, унаследовав политику. Однако у этого довольно гибкого и мощного подхода есть два недостатка. Первый заключается в том, что, открыто наследуя политике, мы уже не можем контролировать, какая часть интерфейса внедряется, – все открытые функции-члены политики становятся частью интерфейса производного класса. Есть и второй недостаток – чтобы реализовать нечто полезное таким способом, мы должны позволить классу политики приводить себя к типу производного класса и, кроме того, он должен иметь доступ ко всем данным-членам и, возможно, другим политикам класса. Второй рассмотренный нами подход опирался на специальное свойство конструкторов – чтобы скопировать или переместить класс, мы должны скопировать или переместить все его базовые классы и данные-члены. Если хотя бы один из них не допускает копирования или перемещения, то и весь конструктор не откомпилируется. К сожалению, обычно при этом сообщается о какой-то неочевидной синтаксической ошибке – ничего похожего на бесхитростное «в этом объекте не найден копирующий конструктор». Эту технику можно распространить и на другие функции-члены, например операторы присваивания, но она становится менее элегантной.

Сейчас мы узнаем о более прямом способе манипулировать открытым интерфейсом класса на основе политик. Прежде всего будем различать условный запрет существующих функций-членов и добавление новых. Первое разумно и, вообще говоря, безопасно: если реализация не может поддержать некоторые операции, предлагаемые интерфейсом, то их не следует и предлагать. Второе опасно, потому что допускает по существу произвольное и неконтролируемое расширение открытого интерфейса класса. Поэтому мы выберем предоставление интерфейса для всех предполагаемых применений класса на основе политик, а затем будем запрещать те части, которые не имеют смысла при некотором наборе политик. В C++ уже есть средство для избирательного разрешения и запрета функций-членов. Чаще всего оно реализуется с помощью шаблона `std::enable_if`, но в основе лежит идиома SFINAE, которую мы изучали в главе 7.

Чтобы продемонстрировать использование SFINAE для избирательного разрешения функции-члена с помощью политик, мы дадим возможность по желанию запрещать функцию operator->() в нашем классе интеллектуального указателя. Это сделано в основном в иллюстративных целях – оператор operator->() действительно имеет смысл не всегда, а только для классов, имеющих данные-члены, но обычно это не проблема, потому что при неправильном использовании код просто не откомпилируется. Однако для демонстрации нескольких важных приемов этот пример полезен.

Прежде всего рассмотрим, как применяется шаблон std::enable_if для разрешения или запрета конкретной функции-члена. В общем случае выражение std::enable_if<value, type> компилируется и дает указанный тип type, если значение value равно true (оно должно быть булевой константой времени компиляции, constexpr). Если value равно false, то подстановка типа завершается неудачно (не порождается никакой результирующий тип). Эту шаблонную метафункцию следует использовать в SFINAE-контексте, где неудавшаяся подстановка типа не приводит к ошибке компиляции, а просто запрещает функцию, вызвавшую ошибку (точнее, удаляет ее из множества перегруженных вариантов).

Поскольку для разрешения или запрета функций-членов с помощью SFINAE нам не нужно ничего, кроме константы времени выполнения, мы можем определить наши политики, используя только constexpr-значение:

```
struct WithArrow {
    static constexpr bool have_arrow = true;
};

struct WithoutArrow {
    static constexpr bool have_arrow = false;
};
```

Теперь мы сможем воспользоваться политикой, чтобы управлять включением operator->() в открытый интерфейс класса:

```
template <typename T,
          typename DeletionPolicy = DeleteByOperator<T>,
          typename ArrowPolicy = WithArrow
         >
class SmartPtr : private DeletionPolicy
{
    public:
    std::enable_if_t<ArrowPolicy::have_arrow, T*> operator->() {
        return p_;
    }
    .....
    private:
    T* p_;
};
```

Здесь мы использовали std::enable_if, чтобы сгенерировать тип, возвращаемый функцией-членом operator->(), – если мы хотим, чтобы эта функция су-

ществовала, то она должна возвращать тип T*, как положено, иначе выведение возвращаемого типа завершится неудачно. К сожалению, все не так просто – показанный выше код работает, когда ArrowPolicy установлена равной WithArrow, но не компилируется для политики WithoutArrow, даже если operator->() нигде не используется. Складывается впечатление, что иногда неудавшаяся подстановка является ошибкой, несмотря ни на какую SFINAE. Причина этой ошибки в том, что SFINAE работает в шаблонном контексте, поэтому сама функция-член должна быть шаблоном. Того факта, что все происходит внутри шаблона класса, недостаточно. Довольно просто преобразовать наш operator->() в шаблонную функцию-член, но как быть с параметром-типом шаблона? Функция operator->() не принимает аргументов, поэтому выведение типа из аргументов не подойдет. По счастью, есть другой способ – параметры шаблона могут иметь значения по умолчанию:

```
template <typename T,
          typename DeletionPolicy = DeleteByOperator<T>,
          typename ArrowPolicy = WithArrow
         >
class SmartPtr : private DeletionPolicy
{
    public:
    .....
    template <typename U = T>
        std::enable_if_t<ArrowPolicy::have_arrow, U*>
        operator->() { return p_; }
    template <typename U = T>
        std::enable_if_t<ArrowPolicy::have_arrow, const U*>
        operator->() const { return p_; }
    private:
    T* p_;
};
```

Здесь используются средства C++14. В C++11 пришлось бы употребить чуть больше слов, потому что шаблона std::enable_if_t в этой версии нет:

```
template <typename U = T>
    typename std::enable_if<ArrowPolicy::have_arrow, U*>::type
    operator->() { return p_; }
```

Мы определили шаблонную функцию-член operator->() с параметром шаблона U. И результатом выведения этого параметра может быть только значение по умолчанию, т. е. T. Теперь SFINAE работает, как положено, – если ArrowPolicy::have_arrow равно false, то тип, возвращаемый operator->(), определить невозможно, и вся функция просто исключается из открытого интерфейса класса.

Один из случаев, когда operator->() можно исключить наверняка, – если тип T не является классом, поскольку синтаксис p->x допустим, только если в типе T имеется член x, а для этого тип должен быть классом. Поэтому мы можем по умолчанию задавать политику WithArrow для всех типов, являющихся классами,

и политику WithoutArrow – для типов, не являющихся классами (снова применяются средства C++14):

```
template <typename T,
          typename DeletionPolicy = DeleteByOperator<T>,
          typename ArrowPolicy =
            std::conditional_t<std::is_class<T>::value,
                               WithArrow, WithoutArrow>
         >
class SmartPtr : private DeletionPolicy
```

Теперь, располагая способом избирательно запрещать функции-члены, мы можем вернуться к условному разрешению конструкторов. Конструкторы тоже можно разрешать и запрещать, а единственная сложность заключается в том, что у конструкторов нет типа возвращаемого значения, так что проверку SFINAE нужно будет спрятать где-то еще. Кроме того, нам придется сделать конструктор шаблоном, а распространенный способ сокрытия проверки SFINAE, например std::enable_if, состоит в том, чтобы добавить в шаблон дополнительный параметр, который не используется и имеет тип по умолчанию. Именно при выведении типа по умолчанию подстановка и может условно завершиться неудачей:

```
struct MoveForbidden {
    static constexpr bool can_move = false;
};

struct MoveAllowed {
    static constexpr bool can_move = true;
};

template <typename T,
          typename DeletionPolicy = DeleteByOperator<T>,
          typename MovePolicy = MoveForbidden
         >
class SmartPtr : private DeletionPolicy
{
    public:
    template <typename U,
              typename V = std::enable_if_t<MovePolicy::can_move &&
                           std::is_same<U, SmartPtr>::value, U>> SmartPtr(U&& other)
        : DeletionPolicy(std::move(other)),
          p_(other.p_)
    {
        other.release();
    }
    .....
};
```

Теперь наш перемещающий конструктор является шаблоном, который принимает произвольный тип U (это ненадолго), а не ограничивается типом интеллектуального указателя SmartPtr. Он условно разрешен, если политика перемещения допускает это и (момент настал) если тип U действительно совпадает

с самим типом SmartPtr. Тем самым мы избежали расширения интерфейса путем включения перемещающего конструктора из произвольного типа, но при этом создали шаблон, необходимый для применения SFINAE. C++17 допускает более компактную форму выражения is_same – вместо std::is_same<U, SmartPtr>::value мы можем написать std::is_same_v<U, SmartPtr>.

Теперь, увидев совершенно общий способ разрешения и запрещения отдельных функций-членов, который работает и для конструкторов, читатель, наверное, недоумевает, а зачем было приводить первый способ. Главным образом для простоты – выражение enable_if нужно использовать в правильном контексте, а ошибки, которые компилятор выдает в случае малейшей неточности, выглядят, мягко говоря, несимпатично. С другой стороны, идея о том, что некопируемый базовый класс делает некопируемым и весь производный класс, очень проста и работает всегда и всюду. Эту технику можно использовать даже в C++03, где идиома SFINAE гораздо более ограничена и заставить ее правильно работать куда труднее. Еще одна причина знать о том, как можно внедрить открытые функции-члены с помощью политик, заключается в том, что иногда для применения альтернативы, enable_if, требуется объявить весь набор возможных функций в основном классе, а затем избирательно запрещать некоторые из них. Бывает, что полный набор функций противоречив и не может быть объявлен одновременно. Примером может служить набор операторов преобразования. На данный момент наш интеллектуальный указатель нельзя преобразовать обратно в простой. Но можно было бы разрешить такие преобразования, потребовав, чтобы они были явными, или допустить неявные преобразования:

```
void f(C*);
SmartPtr<C> p(.....);
f((C*)(p));          // явное преобразование
f(p);                // неявное преобразование
```

Операторы преобразования определяются следующим образом:

```
template <typename T, .....>
class SmartPtr ..... {
    public:
    explicit operator T*() { return p_; }   // явное преобразование
    operator T*() { return p; }             // неявное преобразование
    .....
    private:
    T* p_;
};
```

Можно было бы разрешить один из этих операторов, воспользовавшись политикой преобразования на основе std::enable_if и SFINAE. Проблема в том, что невозможно объявить явное и неявное преобразования в один и тот же тип, даже если впоследствии один из операторов будет запрещен. Эти операторы изначально не могут находиться в одном и том же множестве перегруженных вариантов. Если мы хотим иметь возможность внедрить один из них в класс,

то должны прибегнуть к базовому классу политики. Поскольку политика должна знать о типе интеллектуального указателя, то снова придется использовать паттерн CRTP. Ниже приведен набор политик для управления преобразованием из интеллектуального в простой указатель:

```
template <typename P, typename T>
struct NoRaw {
};

template <typename P, typename T>
struct ExplicitRaw {
    explicit operator T*() { return static_cast<P*>(this)->p_; }
    explicit operator const T*() const {
        return static_cast<const P*>(this)->p_;
    }
};

template <typename P, typename T>
struct ImplicitRaw {
    operator T*() { return static_cast<P*>(this)->p_; }
    operator const T*() const { return static_cast<const P*>(this)->p_; }
};
```

Эти политики добавляют желаемые открытые функции-члены в производный класс. Поскольку это шаблоны, которые необходимо конкретизировать типом производного класса, политика преобразования является шаблонным параметром шаблона и используется в соответствии с CRTP:

```
template <typename T,
          .....
          template <typename, typename>
             class ConversionPolicy = ExplicitRaw
         >
class SmartPtr : .....,
    public ConversionPolicy<SmartPtr<T, ....., ConversionPolicy>, T>
{
    public:
    .....
    private:
    template<typename, typename> friend class ConversionPolicy;
    T* p_;
};
```

Выбранная политика преобразования добавляет свой открытый интерфейс, если он имеется, в интерфейс производного класса. Одна политика добавляет набор операторов явного преобразования, другая – неявного преобразования. Как и в примере с CRTP выше, базовому классу необходим доступ к данным-членам производного класса. Мы можем сделать другом либо весь шаблон (и все его конкретизации), либо определенную конкретизацию, используемую в качестве базового класса для любого интеллектуального указателя (получается длиннее):

```
friend class ConversionPolicy<SmartPtr<T, ....., ConversionPolicy>, T>;
```

Мы изучили несколько способов реализации новых политик. Но иногда проблема возникает при попытке повторно использовать уже имеющиеся политики. В следующем разделе показано, как это можно сделать.

Перепривязка политики

Мы уже видели, что список политики может оказаться весьма длинным. Часто требуется изменить только одну политику и создать класс *такой же, как другой, но чуть-чуть отличающийся*. Сделать это можно по меньшей мере двумя способами.

Первый способ очень общий, но несколько многословный. На первом шаге мы раскрываем параметры шаблона в виде typedef'ов, или псевдонимов в главном шаблоне. Это в любом случае полезная практика, поскольку без псевдонимов очень трудно на этапе компиляции определить, каким был параметр шаблона, если мы захотим использовать его вне шаблона. Например, у нас есть интеллектуальный указатель, и мы хотим знать, какая политика удаления была задана. Проще всего это сделать, попросив помощи у самого класса интеллектуального указателя:

```
template <typename T,
          typename DeletionPolicy = DeleteByOperator<T>,
          typename CopyMovePolicy = NoMoveNoCopy,
          template <typename, typename>
             class ConversionPolicy = ExplicitRaw
         >
class SmartPtr : private DeletionPolicy,
                 public CopyMovePolicy,
                 public ConversionPolicy<SmartPtr<T, DeletionPolicy,
                                                  CopyMovePolicy,
                                                  ConversionPolicy>,
                                         T>
{
    public:
    using value_t = T;
    using deletion_policy_t = DeletionPolicy;
    using copy_move_policy_t = CopyMovePolicy;
    template <typename P, typename T1> using conversion_policy_t =
            ConversionPolicy<P, T1>;
    .....
};
```

Обратите внимание, что здесь используется два разных вида псевдонимов – для обычных параметров шаблона, например DeletionPolicy, можно использовать typedef или эквивалентный псевдоним using. Для шаблонного параметра шаблона необходимо использовать псевдоним шаблона, который иногда называют шаблонным typedef: чтобы воспроизвести ту же самую политику с другим интеллектуальным указателем, нам нужно знать сам шаблон, а не его конкретизацию, например ConversionPolicy<SmartPtr, T>. Для единообразия мы всюду используем синтаксис псевдонимов. Теперь, если потребуется создать другой

интеллектуальный указатель, в котором некоторые политики будут такими же, можно просто запросить политики исходного объекта:

```
SmartPtr<int, DeleteByOperator<int>, MoveNoCopy, ImplicitRaw>
p_original(new int(42));
using ptr_t = decltype(p_original);      // точный тип p_original
SmartPtr<ptr_t::value_t, ptr_t::deletion_policy_t,
         ptr_t::copy_move_policy_t, ptr_t::conversion_policy_t> p_copy;
SmartPtr<double, ptr_t::deletion_policy_t,
         ptr_t::copy_move_policy_t, ptr_t::conversion_policy_t> q;
```

Сейчас типы `p_copy` и `p_original` в точности совпадают. Есть, конечно, и более простой способ добиться той же цели. Но штука в том, что мы могли бы изменить любой тип в списке, оставив остальные в неприкосновенности, и получить указатель, который *во всем похож на* `p_original`, *кроме одного отличия*. Например, указатель `q` имеет те же самые политики, но указывает на значение типа `double`.

Последний случай встречается довольно часто, и существует простой способ *перепривязать* шаблон к другому типу, не изменяя остальных аргументов. Для этого главный шаблон и все его политики должны поддерживать такую перепривязку:

```
template <typename T>
struct DeleteByOperator {
    void operator()(T* p) const {
        delete p;
    }
    template <typename U> using rebind_type = DeleteByOperator<U>;
};

template <typename T,
          typename DeletionPolicy = DeleteByOperator<T>,
          typename CopyMovePolicy = NoMoveNoCopy,
          template <typename, typename>
             class ConversionPolicy = ExplicitRaw
         >
class SmartPtr : private DeletionPolicy,
    public CopyMovePolicy,
    public ConversionPolicy<SmartPtr<T, DeletionPolicy,
                                        CopyMovePolicy,
                                        ConversionPolicy>,
                            T>
{
    public:
    .....
    template <typename U> using rebind_type =
        SmartPtr<U, typename DeletionPolicy::template rebind_type<U>,
                    CopyMovePolicy, ConversionPolicy>;
};
```

Псевдоним `rebind_type` определяет новый шаблон всего с одним параметром – типом, который мы собираемся изменить. Остальные параметры берутся из самого главного шаблона. Некоторые из них являются типами, также

зависящими от главного типа T, и сами нуждаются в перепривязке (в нашем примере такова политика удаления). Решив не перепривязывать политику копирования/перемещения, мы налагаем ограничение – ни одна из этих политик не может зависеть от главного типа, иначе ее пришлось бы также перепривязать. Наконец, политика преобразования типов не нуждается в перепривязке – тут мы имеем доступ ко всему шаблону, поэтому она будет конкретизирована новым главным типом. Теперь можно использовать механизм перепривязки для создания *похожего* типа указателя:

```
SmartPtr<int, DeleteByOperator<int>, MoveNoCopy, ImplicitRaw> p(new
int(42));
using dptr_t = decltype(p)::rebind_type<double>;
dptr_t q(new double(4.2));
```

Если у нас имеется прямой доступ к типу интеллектуального указателя, то его можно использовать для перепривязки (например, в шаблонном контексте). В противном случае мы можем получить тип из переменной этого типа, воспользовавшись `decltype()`. Указатель `q` обладает такими же политиками, что и `p`, но указывает на `double`, а зависящие от типа политики, в частности политика удаления, соответственно обновлены.

Мы рассмотрели основные способы реализации политик и их использования для настройки классов с политиками. Настало время подвести итог тому, что мы изучили, и сформулировать общие рекомендации по проектированию на основе политик.

Рекомендации и указания

Проектирование на основе политик придает исключительную гибкость процессу создания классов, допускающих тонкую настройку. Иногда избыточная гибкость и мощность становятся врагами хорошего дизайна. В этом разделе мы обсудим слабые и сильные стороны проектирования на основе политик и сформулируем ряд общих рекомендаций.

Достоинства проектирования на основе политик

Главные достоинства проектирования на основе политик – гибкость и расширяемость проекта. На верхнем уровне это те же достоинства, что предлагает паттерн Стратегия, только реализованные на этапе компиляции. Проектирование на основе политик позволяет программисту на этапе компиляции выбрать один из нескольких алгоритмов решения определенной задачи или выполнения некоторой операции системой. Поскольку единственными ограничениями на алгоритмы являются требования к их интерфейсу с остальной системой, то ничто не мешает расширять систему за счет новых политик для настраиваемых операций.

На верхнем уровне проектирование на основе политик позволяет строить программную систему из компонентов. Эта идея, конечно, не нова и уж точ-

но не ограничивается проектированием на основе политик. Смысл проектирования на основе политик заключается в использовании компонентов для определения поведения и реализации отдельных классов. Имеется некоторое сходство между политиками и обратными вызовами – те и другие позволяют предпринять заданное пользователем действие в ответ на возникновение определенного события. Однако политики – более общий инструмент, чем обратные вызовы; если обратный вызов – это функция, то политиками могут быть целые классы, содержащие несколько функций и, быть может, нетривиальное внутреннее состояние.

Эти общие концепции выливаются в не имеющий аналогов набор достоинств для проектирования, в основном относящихся к гибкости и расширяемости. После того как общая структура системы и компоненты верхнего уровня определены в процессе проектирования, политики позволяют производить низкоуровневую настройку в рамках ограничений первоначального проекта. Политики способны расширять интерфейс класса (добавлять открытые функции-члены), реализовывать или расширять состояние класса (добавлять данные-члены) и определять реализацию (добавлять закрытые функции-члены). Первоначальный проект, устанавливая структуру и взаимодействия классов, по существу наделяет политики правом играть одну или несколько из указанных ролей.

В результате получается расширяемая система, которую можно модифицировать в соответствии с изменяющимися требованиями, в т. ч. и такими, которые нельзя было предвидеть на этапе проектирования системы. Общая архитектура остается стабильной, тогда как выбор возможных политик и ограничений на их интерфейсы предлагает систематический, дисциплинированный способ модификации и расширения системы.

Недостатки проектирования на основе политик

Сразу приходит на ум проблема, с которой мы уже сталкивались, – объявления классов с конкретными наборами политик уж очень многословные, особенно если нужно изменить политики, находящиеся в конце списка параметров. Вот объявление интеллектуального указателя, в котором присутствуют все реализованные в этой главе политики:

```
SmartPtr<int, DeleteByOperator<int>, NoMoveNoCopy, ExplicitRaw,
        WithoutArrow, NoDebug> p;
```

И это всего лишь интеллектуальный указатель – класс с довольно простым интерфейсом и ограниченной функциональностью. И хотя маловероятно, что кому-то понадобится указатель со всеми этими возможностями настройки, в общем случае множество политик в подобных классах имеет тенденцию к разрастанию. Эта проблема, пожалуй, самая очевидная, но не худшая из всех. Псевдонимы шаблонов позволяют дать краткие имена небольшому числу комбинаций политик, используемых в конкретном приложении. В шаблонном

контексте типы интеллектуальных указателей, используемые в качестве аргументов функции, выводятся и могут не указываться явно. В обычном коде ключевое слово auto позволяет вводить гораздо меньше текста, а заодно сделать программу более надежной – если сложные объявления типов, которые встречаются в разных местах и должны совпадать, заменяются автоматически, то исчезают ошибки из-за небольших отличий в наборе кода (вообще, пользуйтесь любой возможностью заставить компилятор генерировать код, правильный по построению).

Существует куда более серьезная, хотя и не так заметная проблема – все типы с разными политиками на самом деле являются разными типами. Два интеллектуальных указателя, указывающих на один объект, но объявленных с разными политиками удаления, – это разные типы. В чем же тут проблема? Рассмотрим функцию, которая должна работать с объектом, переданным ей по интеллектуальному указателю. Эта функция не копирует интеллектуальный указатель, поэтому ей безразлично, какая там политика копирования, – она же не используется. А все-таки, каким должен быть тип аргумента? Не существует одного типа, который подошел бы для всех интеллектуальных указателей, как бы похожи они ни были.

Тут есть несколько решений. Самое простое – превратить все функции, работающие с типами на основе политик, в шаблоны. Это позволит упростить кодирование и сократить дублирование кода (по крайней мере, исходного), но у такого подхода свои минусы – машинный код становится больше, поскольку содержит по нескольку копий каждой функции. К тому же весь шаблонный код должен находиться в заголовочных файлах.

Другой вариант – стереть типы политик. Мы познакомились с техникой стирания типа в главе 6. Она решает проблему наличия большого числа похожих типов – мы могли бы сделать так, что все интеллектуальные указатели, независимо от политик, будут иметь одинаковый тип (но, конечно, лишь до такой степени, чтобы политики определяли реализацию, а не открытый интерфейс). Однако это обойдется очень дорого. Одно из основных достоинств шаблонов вообще и проектирования на основе политик в частности – тот факт, что шаблоны предлагают абстракцию с нулевыми издержками – мы можем выражать программы в терминах удобных абстракций и концепций высокого уровня, но компилятор все это отсекает, встраивает все шаблоны и генерирует минимально необходимый код. Стирание типа не только сводит на нет это преимущество, но и дает противоположный эффект – добавляет высокие накладные расходы на выделение памяти и косвенные вызовы функций.

И последний вариант – избегать использования типов на основе политик, по крайней мере для некоторых операций. Иногда это несет с собой небольшие дополнительные затраты; например, функция, которой нужно работать с объектом, но не удалять его и не принимать во владение, должна получать объект по ссылке, а не по интеллектуальному указателю (см. главу 3). Помимо явного выражения того факта, что функция не собирается владеть объектом, это

изящно решает проблему типа аргумента – тип ссылки не зависит от того, из какого указателя она получена. Но этот подход ограничен – все-таки чаще нам необходимо работать с объектами на основе политик целиком, а они обычно гораздо сложнее простого указателя (например, нестандартные контейнеры часто реализуются с помощью политик).

И последний недостаток – сложность типов на основе политики вообще, хотя такие заявления нужно делать с осторожностью и задаваться важным вопросом: сложно по сравнению с чем? Проекты на основе политик обычно затеваются для решения сложных проблем проектирования, в которых семейство типов служит единой общей цели (*что*), но достигает ее по-разному (*как*). Это подводит нас к рекомендациям относительно использования политик.

Рекомендации по проектированию на основе политик

Рекомендации по проектированию на основе политик сводятся к управлению сложностью и соблюдению принципа «цель оправдывает средства» – гибкость дизайна и элегантность получающегося решения должны оправдывать сложность реализации и использования.

Поскольку причиной сложности является прежде всего увеличение количества политик, большинство рекомендаций посвящено именно этому вопросу. Некоторые политики в итоге сводят воедино очень разные типы, у которых оказались схожие реализации. Цель такого типа на основе политики – уменьшить дублирование кода. Хотя это достойная цель, обычно она не является достаточно веской причиной для вываливания всего многообразия разнородных политик на конечного пользователя типа. Если два разных типа или семейства типов, по стечению обстоятельств, имеют схожую реализацию, то следует вычленить эту реализацию. В закрытой, невидимой части проекта, содержащей только реализацию, тоже можно использовать политики, если это упрощает реализацию. Но клиент не должен задавать эти скрытые политики, клиенту следует оставить только задание типов, имеющих смысл для приложения, и политик, настраивающих видимое поведение. Зная эти типы и политики, реализация может вывести дополнительные типы по мере необходимости. Тут можно провести аналогию с вызовом общей функции, которая, скажем, ищет минимальный элемент последовательности, из разных не связанных между собой алгоритмов, которым почему-то нужна эта операция. Общий код не дублируется, но и не раскрывается пользователю.

Итак, когда тип на основе политик следует разбить на две или более частей? Для ответа на этот вопрос есть хорошая методика: попробовать придумать для главного типа с конкретным набором политик подходящее имя, описывающее его назначение. Например, некопируемый владеющий указатель, не важно, перемещаемый или нет, естественно назвать *уникальным указателем* (unique pointer) – в любой момент времени для любого объекта может существовать только один такой указатель. Это справедливо при любой политике удаления или преобразования. С другой стороны, указатель с подсчетом ссылок является

разделяемым, опять-таки при любом выборе политик. Это наводит на мысль, что наш интеллектуальный указатель, который был призван заменить собой все указатели, стоило бы разделить на два – некопируемый уникальный указатель и копируемый разделяемый указатель. Мы по-прежнему обеспечиваем частичное повторное использование, потому, например, что политика удаления общая для обоих типов, и ее не нужно реализовывать дважды. Именно такое решение и принято в стандарте C++. У типа `std::unique_ptr` есть только одна политика – удаления. У типа `std::shared_ptr` тоже имеется такая политика, и он может использовать те же самые объекты политик, но ее тип стерт, поэтому все разделяемые указатели на конкретный объект имеют один и тот же тип.

Но как насчет других политик? Тут мы подходим ко второй рекомендации – политики, ограничивающие применение класса, должны быть оправданы стоимостью возможных ошибок, вызванных некорректным использованием, которое эти политики призваны предотвратить. Например, так ли нам нужна политика, запрещающая перемещение? С одной стороны, она могла бы предотвратить программную ошибку в случае, когда владение объектом нельзя передавать ни при каком раскладе. С другой стороны, во многих случаях программист просто может внести в код изменения с целью использовать перемещаемый указатель.

Аналогично, хотя, быть может, и желательно предотвратить неявное приведение к простому указателю с целью повысить дисциплину программирования, всегда существует способ сделать это явно – уж `&*p` точно должно работать. Опять-таки, преимущества со всех сторон ограниченного интерфейса могут и не оправдывать добавления такой политики. С другой стороны, это был прекрасный компактный учебный пример, демонстрирующий приемы, которые можно использовать для разработки более сложных и полезных политик, так что время, потраченное на изучение данной политики, не выброшено на ветер.

Другой взгляд на вопрос о том, как должен выглядеть правильный набор политик и как разбить его на отдельные группы, подразумевает возврат к фундаментальному достоинству проектирования на основе политик – компонуемости поведений, выраженных различными политиками. Если имеется класс с четырьмя разными политиками, у каждой из которых четыре разные реализации, то всего получается 256 вариантов класса. Конечно, маловероятно, что нам понадобятся все 256. Но важно то, что в момент реализации класса мы не знаем, какие варианты действительно понадобятся впоследствии. Мы могли бы высказать некоторую гипотезу и реализовывать лишь самые вероятные варианты. Если мы ошибемся, то придется расплачиваться излишним дублированием кода и копированием-вставкой. Применяя проектирование на основе политик, мы получаем возможность реализовать любую комбинацию поведений, не выписывая каждую из них заранее.

Понимая, в чем сила проектирования на основе политик, мы можем использовать ее для оценки конкретного набора политик – должны ли они компоноваться? Понадобится ли нам когда-нибудь комбинировать их разными способами? Если некоторые политики всегда встречаются в определенных

комбинациях или группах, то напрашивается решение автоматически выводить эти политики из одной главной, заданной пользователем. С другой стороны, набор практически независимых друг от друга политик, которые можно сочетать произвольным образом, наверное, можно счесть хорошим набором.

Еще один способ борьбы с минусами проектирования на основе политик – попробовать добиться той же цели другими средствами. Заменить все возможности, предлагаемые политиками, не получится – в конце концов, паттерн Стратегия появился не без причины. Но существуют альтернативные паттерны, обладающие поверхностным сходством, которые все же можно применить для решения некоторых из тех же проблем. С одной такой альтернативой мы познакомимся в главе 17, когда будем говорить о декораторах. И еще одно решение, которое *выглядит как политика* в ограниченной предметной области, мы покажем в следующем разделе.

Почти политики

Сейчас мы рассмотрим одну альтернативу проектированию на основе политик. Она не такая общая, но в тех случаях, когда работает, может предложить все преимущества политики, в частности компонуемость, но без некоторых присущих им проблем. Для иллюстрации будем рассматривать задачу о проектировании нестандартного типа-значения.

Если говорить по-простому, то типом-значением называется тип, который ведет себя в основном как `int`. Часто в этой роли выступают числа. Конечно, у нас имеется набор встроенных типов для этой цели, но иногда требуется также работать с рациональными или комплексными числами, с тензорами, матрицами или числами, с которыми связана единица измерения (метры, граммы и т. д.). Типы-значения поддерживают множество операций: арифметические операции, операции сравнения, присваивание и копирование. В зависимости от представляемого значения иногда нужно только подмножество операций, например для матриц – сложение и умножение, но не деление. Да и сравнение матриц на что-либо, кроме равенства, в большинстве случаев не имеет смысла. Точно так же мы не хотим сравнивать метры с граммами.

Вообще, часто возникает желание иметь числовой тип с ограниченным интерфейсом – операции, которые мы не хотим предоставлять для таких чисел, не должны компилироваться. Тогда программу, содержащую недопустимую операцию, просто нельзя будет написать.

К этой задаче можно подойти с позиций политик:

```
template <typename T,
          typename AdditionPolicy, typename ComparisonPolicy,
          typename OrderPolicy, typename AssignmentPolicy, ..... >
class Value { ..... };
```

Такая реализация грешит всеми недостатками проектирования на основе политик: длинный список политик, все политики нужно задавать явно, не су-

ществует хороших умолчаний. Поскольку параметры-политики позиционные, в объявлении типа нужно внимательно считать запятые, а при добавлении новых политик всякое подобие осмысленного порядка параметров исчезает. Кстати, мы не упомянули тот факт, что разные наборы политик порождают разные типы, – в данном случае это не недостаток, а цель проектирования. Если нам нужен тип, поддерживающий сложение, и похожий тип без поддержки сложения, то понятно, что они должны быть различны.

В идеале мы хотели бы просто перечислить свойства, которыми должно обладать значение, – я хочу иметь тип-значение, основанный на целых числах, который поддерживает сложение, умножение и присваивание, а больше ничего. И как выясняется, это можно сделать.

Для начала подумаем, как могла бы выглядеть такая политика. Например, политика, разрешающая сложение, должна внедрить в открытый интерфейс класса функцию `operator+()` (и, быть может, `operator+=()`). Политика, обеспечивающая присваивание значения, должна внедрить `operator=()`. Мы видели достаточно таких политик и знаем, как они реализуются – они должны быть открыто наследуемыми базовыми классами, знать о своем производном классе и приводиться к его типу, т. е. в них должен использоваться паттерн CRTP:

```
template <typename T,       // T - примитивный тип (например, int)
          typename V>       // V - производный класс
struct Incrementable
{
    V operator++() {
        V& v = static_cast<V&>(*this);
        ++v.value_;         // это фактическое значение в производном классе
        return v;
    }
};
```

Теперь подумаем, как использовать эти политики в главном шаблоне. Во-первых, мы хотим поддержать заранее неизвестное количество политик в любом порядке. Это наводит на мысль о шаблонах с переменным числом аргументов. Однако чтобы можно было использовать CRTP, параметры шаблона сами должны быть шаблонами. Во-вторых, мы хотим унаследовать реализации всех этих шаблонов, сколько бы их не было. Итак, нам нужен шаблон с переменным числом аргументов, которые являются шаблонными параметрами шаблона:

```
template <typename T, template <typename, typename> class ... Policies>
class Value : public Policies<T, Value<T, Policies ... >> ...
{ ..... };   // Не три точки!
```

Дальше нужно осторожно подходить к многоточию (...) – раньше мы использовали ..... для обозначения *дополнительного кода, который мы уже видели и не хотим повторять*. Но, начиная с этого места, три точки (...) – это часть синтаксиса C++, применяемого при работе с шаблонами с переменным числом аргументов (именно поэтому в оставшейся части этой главы дополнительный код обозначается пятью, а не тремя точками). Выше объявлен шаблон класса

Value, имеющий по меньшей мере один параметр-тип и ноль или более шаблонных политик, каждая из которых имеет два параметра-типа (напомним, что в C++17 можно писать также typename ... Policies вместо class ... Policies). Класс Value конкретизирует эти шаблоны типом T и самим собой и открыто наследует каждому из них.

Шаблон класса Value должен включать интерфейс, общий для всех типов-значений. Недостающее обеспечат политики. Будем считать, что по умолчанию все значения должны допускать копирование, присваивание и печать:

```
template <typename T, template <typename, typename> class ... Policies>
class Value : public Policies<T, Value<T, Policies ... >> ...
{
    public:
    typedef T value_type;
    explicit Value() : val_(T()) {}
    explicit Value(T v) : val_(v) {}
    Value(const Value& rhs) : val_(rhs.val_) {}
    Value& operator=(Value rhs) { val_ = rhs.val_; return *this; }
    Value& operator=(T rhs) { val_ = rhs; return *this; }
    friend std::ostream& operator<<(std::ostream& out, Value x) {
        out << x.val_; return out;
    }
    friend std::istream& operator>>(std::istream& in, Value& x) {
        in >> x.val_; return in;
    }

    private:
    T val_;
};
```

Операторы ввода из потока и вывода в поток, как обычно, должны быть свободными функциями. Для их порождения мы будем использовать *Фабрику друзей*, описанную в главе 12.

Прежде чем с головой окунуться в реализацию этих политик, нужно устранить еще одно препятствие. Значение val_ – закрытый член класса Value, и нас это вполне устраивает. Однако политики должны иметь возможность читать и изменять его. Ранее мы решали эту проблему, делая все политики, которым нужен такой доступ, друзьями. Но на этот раз мы даже не знаем имен политик, которыми располагаем. Поломав голову над показанным выше объявлением расширения пакета параметров как множества базовых классов, читатель уже, наверное, ждет, что мы вытащим кролика из шляпы и каким-то образом объявим дружбу со всем этим пакетом. Увы, мы не знаем такого способа. Лучшее, что мы можем предложить, – предоставить набор функций доступа, которые, по идее, должны вызываться только из политик, но как это гарантировать, не понятно (можно было бы пойти по пути придумывания имен типа policy_accessor_do_not_call(), предупреждающих, что эта функция не для пользователей, но изобретательность программистов не знает границ, и на такого рода намеки никто не обращает внимания):

```
template <typename T, template <typename, typename> class ... Policies>
class Value : public Policies<T, Value<T, Policies ... >> ...
{
    public:
    .....
    T get() const { return val_; }
    T& get() { return val_; }
    private:
    T val_;
};
```

Чтобы создать значение-тип с ограниченным набором операций, мы должны всего лишь конкретизировать этот шаблон списком нужных нам политик:

```
using V = Value<int, Addable, Incrementable>;
V v1(0), v2(1);
v1++;                // Incrementable - OK
V v3(v1 + v2);       // Addable - OK
v3 *= 2;             // политики умножения нет - не откомпилируется
```

Сколько и каких политик мы можем реализовать, зависит в основном от потребностей (или воображения), но ниже приведено несколько примеров, демонстрирующих добавление различных операций в класс.

Для начала реализуем вышеупомянутую политику `Incrementable`, которая предоставляет два оператора ++, постфиксный и префиксный:

```
template <typename T, typename V>
struct Incrementable
{
    V operator++() {
        V& v = static_cast<V&>(*this);
        ++(v.get());
        return v;
    }
    V operator++(int) {
        V& v = static_cast<V&>(*this);
        return V(v.get()++);
    }
};
```

Можно создать отдельную политику `Decrementable` для операторов -- или объединить обе политику в одну, если для нашего типа это имеет смысл. Кроме того, если мы хотим иметь возможность прибавлять не только единицу, то понадобятся также операторы +=:

```
template <typename T, typename V>
struct Incrementable
{
    V& operator+=(V val) {
        V& v = static_cast<V&>(*this);
        v.get() += val.get();
        return v;
    }
```

```
    V& operator+=(T val) {
        V& v = static_cast<V&>(*this);
        v.get() += val;
        return v;
    }
};
```

Показанная выше политика предоставляет два варианта функции `operator+=()`: один принимает приращение того же типа `Value`, а другой – примитивного типа `T`. Это требование необязательно, и при желании мы могли бы реализовать инкремент на значения других типов. Можно даже завести несколько версий политики инкремента, при условии что используется только одна (компилятор даст знать, если мы попробуем создать несовместимые перегруженные варианты одного и того же оператора).

Точно так же можно добавить операторы `*=` и `/=`. Бинарные операторы, например сложения и умножения, добавляются несколько иначе; эти операторы должны быть свободными функциями, чтобы были допустимы преобразования типа первого аргумента. И снова на помощь приходит фабрика друзей. Начнем с операторов сравнения:

```
template <typename T, typename V>
struct ComparableSelf
{
    friend bool operator==(V lhs, V rhs) { return lhs.get() == rhs.get(); }
    friend bool operator!=(V lhs, V rhs) { return lhs.get() != rhs.get(); }
};
```

Будучи конкретизирован, этот шаблон порождает две свободные нешаблонные функции, т. е. операторы сравнения для переменных типа конкретного класса `Value` – того, который конкретизируется. Можно также разрешить сравнение с примитивным типом (например, `int`):

```
template <typename T, typename V>
struct ComparableValue
{
    friend bool operator==(V lhs, T rhs) { return lhs.get() == rhs; }
    friend bool operator==(T lhs, V rhs) { return lhs == rhs.get(); }
    friend bool operator!=(V lhs, T rhs) { return lhs.get() != rhs; }
    friend bool operator!=(T lhs, V rhs) { return lhs != rhs.get(); }
};
```

Скорее всего, мы захотим иметь оба типа сравнения одновременно. Можно было бы просто поместить оба в одну и ту же политику и не пытаться их разделить, а можно было бы создать комбинированную политику из двух уже имеющихся:

```
template <typename T, typename V>
struct Comparable : public ComparableSelf<T, V>,
                    public ComparableValue<T, V>
{
};
```

Операторы сложения и умножения создаются с помощью похожих политик. Это также дружественные нешаблонные свободные функции. Разница только в типе возвращаемого значения – они возвращают сам объект, например:

```
template <typename T, typename V>
struct Addable
{
    friend V operator+(V lhs, V rhs) { return V(lhs.get() + rhs.get()); }
    friend V operator+(V lhs, T rhs) { return V(lhs.get() + rhs); }
    friend V operator+(T lhs, V rhs) { return V(lhs + rhs.get()); }
};
```

Можно также добавить явные или неявные операторы преобразования; политика очень похожа на ту, что мы видели для указателей:

```
template <typename T, typename V>
struct ExplicitConvertible
{
    explicit operator T() {
        return static_cast<V*>(this)->get();
    }
    explicit operator const T() const {
        return static_cast<const V*>(this)->get();
    }
};
```

На первый взгляд, этот подход устраняет большинство недостатков, свойственных типам на основе политик (кроме того что все они, конечно, отдельные типы). Порядок политик не играет роли – мы можем задать только те, что нам нужны, а об остальных даже не думать. Так что же не нравится? Есть два фундаментальных ограничения. Первое заключается в том, что класс, основанный на политиках, не может ссылаться ни на какую политику по имени. Нет больше позиции для `DeletionPolicy` или `AdditionPolicy`. Не существует принимаемых по соглашению интерфейсов политик, скажем, что политика удаления должна быть вызываемой. Весь процесс связывания политик в один тип неявный, это просто суперпозиция интерфейсов.

Таким образом, имеются ограничения на то, что можно делать с помощью подобных политик. Мы можем внедрить открытые функции-члены и свободные функции и даже добавить закрытые данные-члены, но не можем предоставить реализацию аспекта поведения, которое определено и лимитировано основным классом. Таким образом, это не реализация паттерна Стратегия – мы компонуем интерфейс (и, по необходимости, реализацию) по своему желанию, но не настраиваем конкретный алгоритм.

Второе, тесно связанное ограничение заключается в отсутствии политик по умолчанию. Отсутствующие политики отсутствуют – и точка. Вместо них ничего нет. Поведение по умолчанию – это всегда отсутствие поведения. При традиционном проектировании на основе политик каждая позиция в списке политик должна быть заполнена. Если имеется разумное умолчание, его можно задать. Тогда оно и будет принимаемой политикой, если пользователь ее

не переопределит (например, в политике удаления по умолчанию подразумевается оператор delete). Если умолчание не определено, то компилятор не позволит опустить политику – шаблону нужен аргумент.

Последствия этих ограничений гораздо серьезнее, чем может показаться на первый взгляд. Например, может возникнуть соблазн использовать технику на основе enable_if вместо внедрения открытых функций-членов через базовый класс. Тогда можно было бы иметь поведение по умолчанию, активируемое, когда нет других вариантов. Но тут это не сработает. Мы, конечно, можем создать политику, предназначенную для использования совместно с enable_if:

```
template <typename T, typename V> struct Addable
{
    constexpr bool adding_enabled = true;
};
```

Только использовать ее никак не получится – мы не можем написать AdditionPolicy::adding_enabled, потому что нет никакой AdditionPolicy – все позиции в списке политик безымянные. Можно было бы попробовать вместо этого Value::adding_enabled; политика сложения – базовый класс Value, и, следовательно, все ее данные-члены видны в классе Value. Увы, и это не работает – в точке, где это выражение вычисляется компилятором (в определении типа Value как параметра шаблона для CRTP-политик), тип Value неполон, и мы еще не можем обращаться к его данным-членам. Мы могли бы вычислить policy_name::adding_enabled, если бы знали имя политики policy_name. Но именно этим знанием мы и пожертвовали в обмен на возможность не задавать список политик целиком.

Хотя эта альтернатива проектированию на основе политик, строго говоря, не является применением паттерна Стратегия, она может оказаться привлекательной, когда политики используются прежде всего для управления множеством поддерживаемых операций. Предлагая рекомендации по проектированию на основе политик, мы отметили, что редко имеет смысл резервировать позицию в списке политик только для обеспечения дополнительной безопасности ограниченного интерфейса. Вот для таких ситуаций только что описанный альтернативный подход следует иметь в виду.

Резюме

В этой главе мы подробно изучили применения паттерна Стратегия (известного также под названием Политика) к обобщенному программированию на C++. Сочетание того и другого рождает одно из самых действенных средств в арсенале программиста на C++ – проектирование классов на основе политик. Гибкость этого подхода заключается в том, что он позволяет компоновать поведение класса из многих строительных блоков, или политик, каждая из которых отвечает за свой аспект поведения.

Мы изучили различные способы реализации политик – это могут быть шаблоны, классы с шаблонными функциями-членами, классы со статическими

функциями и даже классы с константными значениями. Не менее разнообразны способы использования политик посредством композиции, наследования или прямого доступа к статическим членам. Параметрами политик могут быть типы или шаблоны, у каждого есть свои достоинства и ограничения.

Столь мощный инструмент, как проектирование на основе политик, легко использовать во вред или по неразумию. Часто такие ситуации возникают в процессе постепенной эволюции программы в сторону все большей и большей сложности. Чтобы свести к минимуму такие промашки, мы включили набор рекомендаций, в которых уделили внимание важнейшим достоинствам проектирования на основе политик с точки зрения программиста и предложили приемы и ограничения, позволяющие извлечь из этих достоинств максимум.

Мы также рассмотрели более ограниченный паттерн проектирования, который иногда можно использовать для имитации проектирования на основе политик, но без некоторых его недостатков. Еще более ограниченные альтернативы будут описаны в главе 17 «Адаптеры и декораторы». Оба паттерна – Декоратор и более общий Адаптер – заставляют объект казаться тем, чем он на самом деле не является.

Вопросы

- Что такое паттерн Стратегия?
- Как паттерн Стратегия реализуется в C++ на этапе компиляции с помощью обобщенного программирования?
- Какие типы можно использовать в качестве политик?
- Как можно интегрировать политики с главным шаблоном?
- Каковы основные недостатки проектирования на основе политик?

Глава 17

Адаптеры и Декораторы

Эта глава посвящена двум классическим паттернам объектно-ориентированного программирования (ООП): Адаптеру и Декоратору. Это лишь два из двадцати трех паттернов, описанных в книге Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides «Design Patterns – Elements of Reusable Object-Oriented Software». В C++ этими паттернами можно воспользоваться, как и в любом другом объектно-ориентированном языке. Но, как часто бывает, обобщенное программирование привносит в классические паттерны свои преимущества, вариации, а вместе с ними и новые проблемы.

В этой главе рассматриваются следующие вопросы:

- что такое паттерны Адаптер и Одиночка;
- какая между ними разница;
- какие проблемы проектирования помогают решить эти паттерны;
- как эти паттерны используются в C++;
- как обобщенное программирование помогает проектировать адаптеры и декораторы;
- какие еще паттерны предлагают решение похожих проблем.

Технические требования

Примеры кода: **https://github.com/PacktPublishing/Hands-On-Design-Patterns-with-CPP/tree/master/Chapter17**.

Паттерн Декоратор

Начнем с определения обоих классических паттернов. Как мы увидим, на бумаге сами паттерны и различия между ними вполне понятны. Но потом приходит C++ и размывает границы, так что становятся возможны проектные решения, лежащие где-то посередине. Тем не менее кристальная ясность простых примеров окажется полезной, пусть даже она замутняется по мере усложнения. Итак, начнем с простого.

Декоратор – это также структурный паттерн; он позволяет наделять объект новым поведением. Классический декоратор расширяет поведение сущест-

вующего класса. Он *декорирует* класс новым поведением и создает объект нового декорированного типа. Декоратор реализует интерфейс исходного класса и переадресует этому классу запросы, адресованные его собственному интерфейсу, но, кроме того, выполняет дополнительные действия до и после переадресованных запросов – это и есть *декорации*. Иногда такие декораторы называют **обертками классов**.

Основной паттерн Декоратор

Для начала реализуем на C++ паттерн Декоратор, стараясь не отклоняться от классического определения. Представим себе игру в стиле фэнтези, разворачивающуюся во времена Средневековья (близко к жизни, но со всякими драконами, эльфами и прочими созданиями). Ну а какое Средневековье без войн? Поэтому у игрока будет выбор персонажей, соответствующих его стороне, которых он сможет бросить в поединок. Вот как выглядит базовый класс `Unit`, по крайней мере его часть, относящаяся к поединкам:

```
class Unit {
    public:
    Unit(double strength, double armor) : strength_(strength),
                                          armor_(armor) {}
    virtual bool hit(Unit& target) { return attack() > target.defense(); }
    virtual double attack() = 0;
    virtual double defense() = 0;
    protected:
    double strength_;
    double armor_;
};
```

У персонажа есть сила `strength`, определяющая его атакующие возможности, и доспехи `armor` для защиты. Фактические значения атаки и защиты вычисляются производными классами – конкретными персонажами, – но механизм поединка встроен прямо сюда: если атака сильнее защиты, то персонаж успешно поражает цель (конечно, это крайне упрощенный подход к игре, но мы хотим, чтобы примеры были как можно короче).

А каковые основные персонажи игры? Столпом человеческих армий является доблестный рыцарь `Knight`. У этого персонажа крепкие доспехи и острый меч, что дает ему преимущества как в атаке, так и в защите:

```
class Knight : public Unit {
    public:
    using Unit::Unit;
    double attack() { return strength_ + sword_bonus_; }
    double defense() { return armor_ + plate_bonus_; }
    protected:
    static constexpr double sword_bonus_ = 2;
    static constexpr double plate_bonus_ = 3;
};
```

С рыцарями сражаются зверообразные огры. Огры размахивают простыми деревянными дубинками и одеты в потрепанные кожаные доспехи. Ни то, ни другое не назовешь великими военными достижениями, что уменьшает их боевые возможности:

```
class Ogre : public Unit {
  public:
    using Unit::Unit;
    double attack() { return strength_ + club_penalty_; }
    double defense() { return armor_ + leather_penalty_; }
    protected:
    static constexpr double club_penalty_ = -1;
    static constexpr double leather_penalty_ = -1;
};
```

С другой стороны, огры поразительно сильны, что дает им начальное преимущество:

```
Knight k(10, 5);
Ogre o(12, 2);
k.hit(o);        // Есть!
```

Здесь рыцарь благодаря своему преимуществу в атаке и слабой защите противника успешно поразил огра. Но игра далека от завершения. В ходе сражений выжившие персонажи набираются опыта и в конце концов становятся ветеранами. Ветеран – это все тот же персонаж, но с увеличенными атакующими и защитными возможностями, которые отражают его боевой опыт. В данном случае мы не собираемся изменять интерфейсы классов, но хотим модифицировать поведение функций `attack()` и `defense()`. Это задача паттерна Декоратор, и ниже показана классическая реализация декоратора `VeteranUnit`:

```
class VeteranUnit : public Unit {
    public:
    VeteranUnit(Unit& unit, double strength_bonus, double armor_bonus) :
        Unit(strength_bonus, armor_bonus), unit_(unit) {}
    double attack() { return unit_.attack() + strength_; }
    double defense() { return unit_.defense() + armor_; }
    private:
    Unit& unit_;
};
```

Отметим, что этот класс наследует напрямую классу `Unit`, поэтому в иерархии классов он находится *сбоку* от конкретных классов `Knight` и `Ogre`. Мы попрежнему имеем исходного персонажа, который декорирован и стал ветераном – декоратор `VeteranUnit` содержит ссылку на него. Этот класс декорирует персонажа и используется вместо него, но исходный персонаж не удаляется:

```
Knight k(10, 5);
Ogre o(12, 2);
VeteranUnit vk(k, 7, 2);
VeteranUnit vo(o, 1, 9);
vk.hit(vo);              // И снова попал!
```

Здесь оба старых врага достигли ветеранского уровня, и победа снова досталась рыцарю. Но опыт – лучший учитель, и наш огр перешел на следующий уровень, где ему достались заговоренные, покрытые рунами доспехи с большим защитным потенциалом:

```
VeteranUnit vvo(vo, 1, 9);
vk.hit(vvo);              // И ничего!
```

Отметим, что такой дизайн позволяет декорировать уже декорированный объект! Это сделано намеренно, чтобы бонусы росли по мере того, как персонаж переходит с уровня на уровень. На этот раз защита опытного бойца оказалась непробиваемой для рыцаря.

Как мы уже говорили, это классический паттерн Декоратор, прямо из учебника. Он работает в C++, но с некоторыми ограничениями. Первое довольно очевидно – хотя, получив декорированного персонажа, мы хотим использовать именно его, исходного персонажа нужно хранить и тщательно следить за временем его жизни. У таких практических проблем есть практические же решения, но эта книга посвящена комбинированию паттернов проектирования с обобщенным программированием и новым проектным возможностям, которые открываются вследствие такой синергии. Поэтому наш путь ведет в другом направлении.

Вторая проблема в большей степени специфична для C++. Ее лучше проиллюстрировать на примере. Разработчики игры добавили персонажу `Knight` специальную способность – он может стремглав броситься на врага, получив краткосрочное преимущество в атаке. Данное преимущество действует только для следующей атаки, но в пылу сражения этого может оказаться достаточно:

```
class Knight : public Unit {
    public:
    Knight(double strength, double armor) :
    Unit(strength, armor), charge_bonus_(0) {}
    double attack() {
        double res = strength_ + sword_bonus_ + charge_bonus_;
        charge_bonus_ = 0;
        return res;
    }
    double defense() { return armor_ + plate_bonus_; }
    void charge() { charge_bonus_ = 1; }
    protected:
    double charge_bonus_;
    static constexpr double sword_bonus_ = 2;
    static constexpr double plate_bonus_ = 3;
};
```

Бонус за бросок активируется путем вызова функции-члена `charge()` и действует только для одной атаки, а затем сбрасывается. Когда игрок активирует бросок, игра выполняет примерно такой код:

```
Knight k(10, 5);
gre o(12, 2);
```

```
k.charge();
k.hit(o);
```

Разумеется, мы ожидаем, что рыцарь-ветеран тоже способен броситься вперед, но тут возникает проблема – код не компилируется:

```
VeteranUnit vk(k, 7, 2);
vk.charge();   // не компилируется!
```

Корень проблемы в том, что `charge()` является частью интерфейса класса `Knight`, но декоратор `VeteranUnit` наследует классу `Unit`. Мы могли бы перенести функцию `charge()` в базовый класс `Unit`, однако это неудачное решение, поскольку `Ogre` тоже наследует `Unit`, но огры не умеют бросаться вперед, и потому в их интерфейсе не должно быть такой функции (это нарушает принцип *является* открытого наследования).

Данная проблема внутренне присуща выбранному нами способу реализации паттерна Декоратор – `Knight` и `VeteranUnit` наследуют одному и тому же классу `Unit`, но ничего не знают друг о друге. Существуют уродливые обходные пути, но вообще это фундаментальное ограничение C++: в этом языке плохо поддерживается *перекрестное приведение* (приведение к типу в другой ветви иерархии). Но то, что язык одной рукой отбирает, он другой рукой дает – у нас есть гораздо лучшие средства решения этой проблемы, и далее мы рассмотрим их.

Декораторы на манер C++

В процессе реализации классического декоратора на C++ мы столкнулись с двумя проблемами. Первая заключалась в том, что декорированный объект не принимает владения исходным, поэтому нужно сохранять оба (это можно рассматривать как преимущество, а не как проблему, если впоследствии декорации нужно будет сбросить, и это одна из причин реализации декоратора подобным образом). Вторая проблема связана с тем, что декорированный `Knight` – в действительности не `Knight`, а `Unit`. Эту проблему можно решить, если наследовать декоратор от декорируемого класса. Тогда у класса `VeteranUnit` не было бы фиксированного базового класса – базовым всегда был бы декорируемый класс. Это описание точь-в-точь соответствует паттерну Рекурсивный шаблон (**CRTP**) – идиоме C++, описанной выше. Для применения CRTP мы должны превратить декоратор в шаблон и унаследовать параметру шаблона:

```
template <typename U> class VeteranUnit : public U {
    public:
    VeteranUnit(U&& unit, double strength_bonus, double armor_bonus) :
        U(unit), strength_bonus_(strength_bonus), armor_bonus_(armor_bonus)
    {}
    double attack() { return U::attack() + strength_bonus_; }
    double defense() { return U::defense() + armor_bonus_; }
    private:
    double strength_bonus_;
    double armor_bonus_;
};
```

Теперь, чтобы перевести персонажа в статус ветерана, мы должны преобразовать его в декорированную версию конкретного подкласса `Unit`:

```
Knight k(10, 5);
Ogre o(12, 2);
k.hit(o);          // Попал!
VeteranUnit<Knight> vk(std::move(k), 7, 2);
VeteranUnit<Ogre> vo(std::move(o), 1, 9);
vk.hit(vo);        // Попал!
VeteranUnit<VeteranUnit<Ogre>> vvo(std::move(vo), 1, 9);
vk.hit(vvo);       // Мимо...
vk.charge();       // Теперь компилируется, vk - тоже Knight
vk.hit(vvo);       // Попал с бонусом на бросок!
```

Это тот же сценарий, который мы видели в конце предыдущего раздела, но теперь в нем используется шаблонный декоратор. Обратите внимание на различия. Во-первых, `VeteranUnit` – класс, производный от конкретного класса персонажа, например `Knight` или `Ogre`. Поэтому он имеет доступ к интерфейсу базового класса: например, рыцарь-ветеран `VeteranUnit<Knight>` одновременно является и рыцарем `Knight` и, стало быть, имеет функцию-член `charge()`, унаследованную от `Knight`. Во-вторых, декорированный персонаж явно принимает владение исходным персонажем – чтобы создать ветерана, мы должны переместить в него исходный персонаж (базовый класс персонажа-ветерана конструируется перемещением из исходного персонажа). Исходный объект остается в неспецифицированном состоянии «перемещено из», в котором единственное безопасное действие – вызов деструктора. Заметим, что по крайней мере для этой простой реализации классов персонажей операция перемещения – это просто копирование, так что исходный объект все еще пригоден к использованию, но полагаться на это не следует; делать какие-либо предположения о состоянии «перемещено из» – значит нарываться на ошибку.

Стоит отметить, что наше объявление конструктора `VeteranUnit` требует такой передачи владения и навязывает ее. Попытка сконструировать персонажа-ветерана, не переместив исходный персонаж, не откомпилируется.

```
VeteranUnit<Knight> vk(k, 7, 2); // не компилируется
```

Предоставляя только конструктор, принимающий r-значение, т. е. `Unit&&`, мы требуем, чтобы вызывающая сторона согласилась передать владение.

До сих пор в целях демонстрации мы создавали все объекты персонажей в стеке как локальные переменные. В любой нетривиальной программе этого было бы недостаточно – нам нужно, чтобы объекты продолжали существовать еще долго после того, как создавшая их функция завершилась. Мы можем объединить объекты-декораторы с механизмом владения памятью и гарантировать удаление исходных, уже перемещенных объектов после создания декорированной версии.

Предположим, что во всей программе для управления владением используются уникальные указатели (в любой момент времени у каждого объекта есть

только один владелец). Вот как этого можно добиться. Прежде всего удобно объявить псевдонимы нужных нам указателей:

```
using Unit_ptr = std::unique_ptr<Unit>;
using Knight_ptr = std::unique_ptr<Knight>;
```

Указатель Unit_ptr может владеть любым персонажем, но через него нельзя вызвать функции-члены конкретных персонажей, например charge(), поэтому нам могут понадобиться также указатели на конкретные классы. Далее мы увидим, что необходимо уметь перемещать объект между этими классами. Переместить из указателя на производный класс в указатель на базовый класс просто:

```
Knight_ptr k(new Knight(10, 5));
Unit_ptr u(std::move(k));       // теперь k равно null
```

Перемещение в обратном направлении немного сложнее: std::move не будет работать неявно – точно так же мы не смогли преобразовать указатель Unit* в Knight* без явного приведения. Нам необходимо *перемещающее приведение*:

```
template <typename To, typename From>
std::unique_ptr<To> move_cast(std::unique_ptr<From>& p) {
    return std::unique_ptr<To>(static_cast<To*>(p.release()));
}
```

Здесь мы воспользовались оператором static_cast для приведения к производному классу, он будет работать в предположении, что такое отношение действительно существует (т. е. базовый объект действительно принадлежит ожидаемому производному классу), в противном случае результат не определен. При желании можно проверить это предположение во время выполнения с помощью оператора dynamic_cast. В следующей версии такая проверка производится, но только когда включены утверждения (можно было бы вместо assert возбуждать исключение):

```
template <typename To, typename From>
std::unique_ptr<To> move_cast(std::unique_ptr<From>& p) {
#ifndef NDEBUG
    auto p1 = std::unique_ptr<To>(dynamic_cast<To*>(p.release()));
    assert(p1);
    return p1;
#else
    return std::unique_ptr<To>(static_cast<To*>(p.release()));
#endif
}
```

Если всеми объектами владеют экземпляры уникального указателя, то конструктор декоратора VeteranUnit должен принимать указатель и перемещать объект из этого указателя:

```
template <typename U> class VeteranUnit : public U {
    public:
```

```
    template <typename P>
    VeteranUnit(P&& p, double strength_bonus, double armor_bonus) :
        U(std::move(*move_cast<U>(p))),
        strength_bonus_(strength_bonus), armor_bonus_(armor_bonus) {}
    double attack() { return U::attack() + strength_bonus_; }
    double defense() { return U::defense() + armor_bonus_; }
    private:
    double strength_bonus_;
    double armor_bonus_;
};
```

Здесь нетривиальная часть – инициализация базового класса `U` класса `VeteranUnit<U>`; мы должны переместить персонаж из уникального указателя на базовый класс в перемещающий конструктор производного класса (невозможно просто переместить объект из одного уникального указателя в другой, нужно обернуть его производным классом). И сделать это необходимо, не допустив утечки памяти. Исходный уникальный указатель освобождается, поэтому его деструктор не будет делать ничего, но наша функция `move_cast` возвращает новый уникальный указатель, который с этого момента владеет тем же объектом. Этот уникальный указатель является временной переменной и будет удален в конце инициализации нового объекта, но не раньше, чем мы воспользуемся его объектом для конструирования нового производного объекта класса `VeteranUnit` (в нашем случае инициализация объекта персонажа перемещением сама по себе не экономит время по сравнению с копированием, но это хорошая практика, которая может дать эффект, если более тяжеловесный объект персонажа предоставляет оптимизированный перемещающий конструктор).

Вот как этот новый декоратор используется в программе, которая управляет ресурсами (в нашем случае – персонажами) с помощью уникальных указателей:

```
Knight_ptr k(new Knight(10, 5)); // Knight_ptr, чтобы вызывать call charge()
Unit_ptr o(new Ogre(12, 2));     // при необходимости мог бы быть Orge_ptr
Knight_ptr vk(new VeteranUnit<Knight>(k, 7, 2));
Unit_ptr vo(new VeteranUnit<Ogre>(o, 1, 9));
Unit_ptr vvo(new VeteranUnit<VeteranUnit<Ogre>>(vo, 1, 9));
vk->hit(*vvo);  // мимо
vk->charge();   // работает, потому что vk имеет тип Knight_ptr
vk->hit(*vvo);  // попал
```

Заметим, что мы не переопределили функцию `hit()` – она по-прежнему принимает объект по ссылке. Это правильно, потому что эта функция не берет на себя владение объектом, а просто работает с ним. Нет никакой необходимости передавать ей владеющий указатель, т. к. это подразумевало бы передачу владения.

Обратите внимание, что, строго говоря, между первым и вторым примерами разница очень незначительна – к персонажу в состоянии «перемещено из» обращаться все равно не следует. Но на практике разница велика – перемещенный указатель больше не владеет объектом. Его значение равно `null`, поэтому

ошибочность любой попытки поработать с исходным персонажем, после того как он перешел на более высокую ступень, станет очевидна очень скоро (программа разыменует нулевой указатель и «грохнется»).

Мы уже видели, что можно декорировать уже декорированный класс, так что эффекты декораторов суммируются. Можно также применить два разных декоратора к одному классу. Каждый декоратор добавит к классу новое поведение. В нашей игре мы могли бы печатать результаты каждой атаки – достигла она цели или нет. Но если результат не совпал с ожидаемым, то мы не будем знать, почему. Для отладки было бы полезно напечатать значения атаки и защиты. Но если мы хотим делать это не каждый раз для всех персонажей, а только в интересующей нас части программы, то можем воспользоваться отладочным декоратором, который добавляет персонажам новое поведение – печать промежуточных результатов вычислений.

В классе `DebugDecorator` используется та же идея, что в предыдущем декораторе, – это шаблон класса, который порождает класс, производный от класса декорируемого объекта. Его виртуальные функции `attack()` и `defense()` переадресуют вызовы базовому классу и печатают результаты:

```
template <typename U> class DebugDecorator : public U {
    public:
    using U::U;
    template <typename P> DebugDecorator(P&& p) :
        U(std::move(*move_cast<U>(p))) {}
    double attack() {
        double res = U::attack();
        cout << "Attack: " << res << endl;
        return res;
    }
    double defense() {
        double res = U::defense();
        cout << "Defense: " << res << endl;
        return res;
    }
};
```

При реализации декораторов нужна осторожность, чтобы случайно не изменить поведение класса неожиданным образом. Рассмотрим, к примеру, такую реализацию `DebugDecorator`:

```
template <typename U> class DebugDecorator : public U {
    double attack() {
        cout << "Attack: " << U::attack() << endl;
        return U::attack();
    }
};
```

Здесь имеется тонкая ошибка – декорированный объект, помимо ожидаемого нового поведения – печати, – скрытно изменяет поведение исходного класса: он дважды вызывает метод `attack()` базового класса. Мало того, что мо-

жет печататься неверная величина, если два обращения к attack() возвращают разные значения, так еще могут сгореть одноразовые атакующие бонусы, например на бросок рыцаря.

DebugDecorator добавляет очень похожее поведение к каждой декорируемой функции-члену. В C++ имеется развитый инструментарий, специально предназначенный для улучшения повторного использования кода и уменьшения дублирования. Посмотрим, нельзя ли получить более универсальный декоратор, допускающий повторное использование.

Полиморфные декораторы и их ограничения

Некоторые декораторы рассчитаны только на классы, которые модифицируют, и имеют узкую направленность. Другие очень общие, по крайней мере в принципе. Например, отладочный декоратор, который протоколирует вызовы функций и печатает возвращенные ими значения, можно было бы использовать с любой функцией, если бы только мы смогли правильно реализовать его.

Такую реализацию легко написать на C++14 и выше с помощью шаблонов с переменным числом аргументов, пакетов параметров и идеальной передачи:

```
template <typename Callable> class DebugDecorator {
    public:
    DebugDecorator(const Callable& c, const char* s) : c_(c), s_(s) {}
    template <typename ... Args> auto operator()(Args&& ... args) const {
        cout << "Вызывается " << s_ << endl;
        auto res = c_(std::forward<Args>(args) ...);
        cout << "Результат: " << res << endl;
        return res;
    }

    private:
    const Callable& c_;
    const std::string s_;
};
```

Этим декоратором можно обернуть любой вызываемый объект или функцию (все, что допускает вызов со скобками ()) с любым числом аргументов. Он печатает заданную строку и результат вызова. Однако зачастую выписать тип вызываемого объекта нелегко, и было бы лучше, чтобы это сделал компилятор с помощью выведения аргументов шаблона:

```
template <typename Callable>
auto decorate_debug(const Callable& c, const char* s) {
    return DebugDecorator<Callable>(c, s);
}
```

Эта шаблонная функция выводит тип Callable и декорирует его отладочной оберткой. Теперь ее можно применить к любой функции или объекту. Вот как выглядит декорированная функция:

```
int g(int i, int j) { return i - j; }   // какая-то функция
auto g1 = decorate_debug(g, "g()");     // декорированная функция
g1(5, 2);                               // печатается "Вызывается g()" и "Результат: 3"
```

Можно так же декорировать вызываемый объект:

```
struct S {
    double operator()() const {
        return double(rand() + 1)/double(rand() + 1);
    }
};
S s;                                      // вызываемый объект
auto s1 = decorate_debug(s, "rand/rand"); // декорированный вызываемый объект
s1(); s1();                               // результат печатается дважды
```

Заметим, что наш декоратор не принимает владение вызываемым объектом (при желании можно было бы написать его и по-другому).

Декорировать можно даже лямбда-выражение, которое есть не что иное, как неявный вызываемый объект. Выражение ниже – это вызываемый объект с двумя аргументами:

```
auto f2 = decorate_debug([](int i, int j) { return i + j; }, "i+j");
f2(5, 3);    // печатается "Вызывается i+j" и "Результат: 8"
```

У нашего декоратора есть некоторые ограничения. Во-первых, с его помощью не удастся декорировать функцию, которая ничего не возвращает, например следующее лямбда-выражение, которое просто инкрементирует свой аргумент:

```
auto incr = decorate_debug([](int& x) { ++x; }, "++x");
int i;
incr(i);    // не компилируется
```

Проблема в выражении `void res`, которое возникает из строки `auto res = ...` в определении `DebugDecorator`. Действительно, нельзя же объявлять переменные типа `void`. Да и автоматический возвращаемый тип нашего декоратора выводится правильно лишь *в большинстве случаев*; например, если функция возвращает `double&`, то декорированная функция вернет просто `double`. Наконец, обернуть вызовы функции-члена можно, но это требует несколько иного синтаксиса.

Впрочем, механизм шаблонов в C++ настолько мощный, что существуют способы сделать наш обобщенный декоратор еще более общим. А также более сложным. Подобному коду место в библиотеке, скажем стандартной, а в большинстве практических приложений отладочный декоратор не стоит таких усилий.

Второе ограничение состоит в том, что чем более общим становится декоратор, тем меньше он может делать. И так уже есть совсем немного осмысленных действий, которые стоило бы делать для вызова любой функции или функции-члена. Можно было бы добавить отладочную печать и печатать результат при условии, что для него определен оператор вывода в поток. Можно

было бы захватить мьютекс для защиты вызова потоконебезопасной функции в многопоточной программе. Быть может, найдется еще несколько более общих действий. Но в общем случае не гонитесь за все более общим кодом просто из принципа.

Не важно, что нам нужно – общие или очень специфические декораторы, – часто приходится добавлять к объекту несколько поведений. Один такой пример мы уже видели. Теперь рассмотрим задачу о применении нескольких декораторов более систематически.

Компонуемые декораторы

Интересующее нас свойство декораторов имеет название – компонуемость. Поведения называются компонуемыми, если их можно применить к одному объекту по отдельности. В нашем случае, если имеется два декоратора `A` и `B`, `A(B(object))` должен обладать обоими поведениями. Альтернатива компонуемости – явное создание комбинированного поведения: чтобы получить оба поведения без композиции, мы должны написать новый декоратор `AB`. Поскольку писать новый код для каждой комбинации нескольких декораторов было бы невозможно, даже если количество декораторов относительно невелико, компонуемость является очень важным свойством.

По счастью, добиться компонуемости с помощью нашего подхода не так уж трудно. CRTP-декораторы, которые мы использовали в проекте игры, естественно допускают композицию:

```
template <typename U> class VeteranUnit : public U { ... };
template <typename U> class DebugDecorator : public U { ... };
Unit_ptr o(new DebugDecorator<Ogre>(12, 2));
Unit_ptr vo(new DebugDecorator<VeteranUnit<Ogre>>(o, 1, 9));
```

Каждый декоратор наследует декорируемому классу и, стало быть, сохраняет его интерфейс, добавляя при этом новое поведение. Заметим, что порядок декораторов имеет значение, поскольку новое поведение добавляется до или после декорированного вызова. `DebugDecorator` применяется к декорируемому объекту и предоставляет ему средства отладки, поэтому `VeteranUnit<DebugDecorator<Ogre>>` будет отлаживать базовую часть объекта (`Ogre`), что само по себе может быть полезно.

Наши (в какой-то мере) универсальные декораторы тоже можно компоновать. У нас уже есть отладочный декоратор, который умеет работать со многими вызываемыми объектами, и мы упоминали о потенциальной потребности защищать эти вызовы мьютексами. Теперь можно реализовать блокирующий декоратор примерно так же (и с похожими ограничениями), как полиморфный отладочный декоратор:

```
template <typename Callable> class LockDecorator {
    public:
    LockDecorator(const Callable& c, std::mutex& m) : c_(c), m_(m) {}
    template <typename ... Args> auto operator()(Args&& ... args) const {
```

```
        std::lock_guard<std::mutex> l(m_);
        return c_(std::forward<Args>(args) ...);
    }
    private:
    const Callable& c_;
    std::mutex& m_;
};

template <typename Callable>
auto decorate_lock(const Callable& c, std::mutex& m) {
    return LockDecorator<Callable>(c, m);
}
```

Как и раньше, мы используем вспомогательную функцию `decorate_lock()`, чтобы поручить компилятору утомительную работу по выведению правильного типа вызываемого объекта. Теперь мы можем воспользоваться мьютексом, чтобы защитить вызов потоконебезопасной функции:

```
std::mutex m;
auto safe_f = decorate_lock([](int x) { return unsafe_f(x); }, m);
```

Если мы хотим защитить функцию мьютексом и выводить отладочную печать при ее вызове, то не придется писать новый *блокирующий отладочный декоратор*, а можно просто последовательно применить оба декоратора:

```
auto safe_f = decorate_debug(
                decorate_lock(
                   [](int x) { return unsafe_f(x); },
                   m
                ),
                "f(x)");
```

Этот пример демонстрирует преимущества компонуемости – не нужно писать специальный декоратор для каждой комбинации поведений (подумайте, сколько декораторов пришлось бы написать, чтобы получить все комбинации пяти основных декораторов, если бы они не допускали композиции!).

Компонуемости легко достичь в наших декораторах, потому что они сохраняют интерфейс исходного объекта, по крайней мере ту его часть, что нас интересует, – поведение изменяется, а интерфейс – нет. Если декоратор используется в роли исходного объекта для другого декоратора, то сохраненный интерфейс снова сохраняется и т. д.

Это сохранение интерфейса – фундаментальная особенность паттерна Декоратор. Но оно же является одним из самых серьезных его ограничений. Наш блокирующий декоратор совсем не так полезен, как может показаться на первый взгляд (так что не стоит пересматривать весь свой код, ставя блокировки на каждый вызов, который должен быть потокобезопасным). Как мы увидим далее, не каждый интерфейс можно сделать потокобезопасным, какой бы хорошей ни была реализация. И вот тогда возникает необходимость изменить интерфейс вдобавок к модификации поведения.

Паттерн Адаптер

Мы закончили предыдущий раздел замечанием о том, что у паттерна Декоратор имеются определенные преимущества, проистекающие из сохранения декорированного интерфейса, и что иногда эти преимущества обращаются ограничениями. В таких случаях можно использовать более общий паттерн Адаптер.

У паттерна Адаптер очень общее определение – это структурный паттерн, который позволяет использовать интерфейс класса как другой, отличный от него интерфейс. Он позволяет применить существующий класс в контексте, где ожидается другой интерфейс, не внося изменений в исходный класс. Иногда такие адаптеры называются **обертками классов**. Вспомните, что и декораторы иногда так называют – по той же причине.

Однако Адаптер – очень общий паттерн широкого назначения. Его можно использовать для реализации нескольких других, более узких паттернов, в частности Декоратора. С паттерном Декоратор проще разобраться, поэтому мы начали с него. А теперь перейдем к общему случаю.

Основной паттерн Адаптер

Продолжим с последнего примера из предыдущего раздела – блокирующего декоратора. Он вызывает произвольную функцию под защитой мьютекса, так что никакую другую функцию, защищенную тем же мьютексом, нельзя вызвать в том же потоке, пока не завершится первая. В некоторых случаях этого достаточно, чтобы сделать весь код потокобезопасным. Но чаще – нет.

Для демонстрации мы реализуем объект потокобезопасной очереди. Очередь – довольно сложная структура данных даже без всякой потокобезопасности, но, к счастью, нам не нужно начинать с нуля, потому что в стандартной библиотеке C++ имеется шаблон `std::queue`. Мы можем помещать объекты в очередь и извлекать их оттуда в порядке «первым пришел, первым обслужен», но только в одном потоке. Одновременно помещать два объекта в одну очередь из разных потоков небезопасно. Но у нас есть решение – мы можем реализовать блокирующую очередь в виде класса, декорирующего основную. Поскольку в данном случае можно не беспокоиться об оптимизации пустого базового класса (`std::queue` – не пустой класс) и мы должны переадресовывать вызов каждой функции-члена, можно обойтись без наследования и воспользоваться композицией. Наш декоратор будет содержать очередь и мьютекс. Обернуть метод `push()` легко. В `std::queue` есть два варианта `push()`: один перемещает объект, другой копирует. Мы должны защитить мьютексом оба:

```
template <typename T> class locking_queue {
    using mutex = std::mutex;
    using lock_guard = std::lock_guard<mutex>;
    using value_type = typename std::queue<T>::value_type;
    void push(const value_type& value) {
        lock_guard l(m_);
```

```
        q_.push(value);
    }
    void push(value_type&& value) {
        lock_guard l(m_);
        q_.push(value);
    }
    private:
    std::queue<T> q_;
    mutex m_;
};
```

Теперь обратимся к получению элементов из очереди. В стандартной очереди для этой цели есть три функции-члена. Первая, `front()`, дает доступ к элементу в начале очереди, но не удаляет его. Функция `pop()` удаляет элемент из начала очереди, но ничего не возвращает (она не предоставляет доступа к первому элементу, а просто удаляет его). Обе эти функции не следует вызывать, если очередь пуста – контроля ошибок нет, но результат не определен.

Наконец, имеется еще функция `empty()`; она возвращает `false`, если очередь не пуста, и в этом случае мы можем вызывать `front()` и `pop()`. Если декорировать эти функции блокировкой, то мы сможем написать такой код:

```
locking_queue<int> q;
q.push(5);
... где-то позже ...
if (!q.empty()) {
    int i = q.front();
    q.pop();
}
```

Сама по себе каждая функция потокобезопасна, но их комбинация таковой не является. Важно понимать, почему. Сначала мы вызываем `q.empty()`. Предположим, что она вернула `false`, так что в очереди заведомо есть хотя бы один элемент. В следующей строке мы можем обратиться к нему, вызвав `q.front()`, которая вернет `5`. Но ведь это лишь один поток из многих работающих в программе. В то же самое время другой поток выполняет тот же самый код (как этого добиться – упражнение для читателя). Этот поток тоже вызывает `q.empty()` и тоже получает `false` – как мы уже сказали, в очереди есть элемент, и пока что мы еще ничего не сделали, чтобы его удалить. Второй поток тоже вызывает `q.front()` и тоже получает `5`. Проблема уже налицо – два потока пытались выбрать элемент из очереди, но получили один и тот же. Однако на деле все еще хуже – теперь первый поток вызывает `q.pop()` и удаляет элемент `5` из очереди. Очередь пуста, но второй поток об этом не знает – он ведь вызывал раньше `q.empty()`. Поэтому второй поток тоже вызывает `q.pop()`, но уже для пустой очереди. В лучшем случае программа «грохнется» сразу.

Мы только что продемонстрировали частный случай общей проблемы – последовательность действий, каждое из которых по отдельности потокобезопасно, в целом может быть потоконебезопасной. На самом деле эта *блокирующая очередь* абсолютно бесполезна, с ее помощью потокобезопасный код не

напишешь. В действительности нам нужна одна потокобезопасная функция, которая выполняет всю транзакцию под защитой одного мьютекса как единое непрерываемое действие (такие транзакции называются **атомарными**). В нашем случае транзакцией является удаление первого элемента, если он существует, и выдача какой-то диагностики, если его нет. Интерфейс `std::queue` не предоставляет такого транзакционного API.

Таким образом, нам необходим новый паттерн, который преобразовывал бы существующий интерфейс класса в другой интерфейс в соответствии с нашими требованиями. Декоратор этого сделать не может, но именно эту задачу решает паттерн Адаптер. Согласившись, что нужен другой интерфейс, надо решить, как он должен выглядеть. Все должна делать новая функция-член `pop()` – если очередь не пуста, то она должна удалить первый элемент и вернуть его вызывающей стороне путем копирования или перемещения. Если же очередь пуста, то функция не должна изменять ее состояние, а лишь как-то уведомить вызывающую сторону. Например, можно было бы вернуть два значения – сам элемент (если он существует) и булев флаг, сообщающий, была ли очередь пуста. Ниже показана функция `pop()` блокирующей очереди, которая теперь является адаптером, а не декоратором:

```
template <typename T> class locking_queue {
    ... push() не изменилась ...
    bool pop(value_type& value) {
        lock_guard l(m_);
        if (q_.empty()) return false;
        value = std::move(q_.front());
        q_.pop();
        return true;
    }
    private:
    std::queue<T> q_;
    mutex m_;
};
```

Заметим, что нет необходимости изменять `push()` – один вызов функции уже делает все, что надо, так что эта часть интерфейса просто переадресуется нашим адаптером один в один. Новая версия `pop()` возвращает `true`, если удалила элемент из очереди, и `false` в противном случае. Если возвращено `true`, то элемент сохранен в предоставленном аргументе, а если `false` – то этот аргумент не изменяется. Если тип `T` элемента допускает присваивание перемещением, то будет использовано перемещение, а не копирование.

Разумеется, это не единственный возможный интерфейс такой атомарной функции `pop()`. Можно было бы возвращать элемент и булево значение в виде пары. Существенное отличие заключается в том, что невозможно будет оставить переданный элемент неизменным, – это возвращаемое значение, и оно должно быть чему-то равно. Естественный способ – сконструировать элемент по умолчанию, если его нет в очереди:

```
template <typename T> class locking_queue {
    ... push() не изменилась ...
    std::pair<value_type, bool> pop() {
        lock_guard l(m_);
        if (q_.empty()) return { value_type(), false };
        value_type value = std::move(q_.front());
        q_.pop();
        return { value, true };
    }
    private:
    std::queue<T> q_;
    mutex m_;
};
```

Теперь имеется ограничение на тип элемента `T` – в нем должен присутствовать конструктор по умолчанию. Какой интерфейс предпочесть, зависит от приложения, в котором используется очередь, к тому же есть и другие способы спроектировать его. Но в любом случае остаются две функции-члена, `push()` и `pop()`, защищенные одним и тем же мьютексом. Теперь любую комбинацию этих операций можно одновременно выполнять из любого числа потоков, и результат будет корректно определен. Поэтому объект `locking_queue` потокобезопасен.

Преобразование текущего интерфейса объекта в нужный приложению без переписывания самого объекта и есть основная цель паттерна Адаптер. В преобразовании может нуждаться любой вид интерфейса, поэтому существует много видов адаптеров. О некоторых из них мы узнаем в следующем разделе.

Адаптеры функций

Только что мы видели адаптер класса, изменяющий интерфейс класса. Другой вид интерфейса – функция (член или свободная). У функции имеются определенные аргументы, но иногда мы хотим вызвать ее с другим набором аргументов. Для этого нужен адаптер. Одно из распространенных применений таких адаптеров – каррирование (currying) одного или нескольких аргументов функции. Это попросту означает, что значение одного или нескольких аргументов функции фиксируется и в дальнейшем не указывается при вызове. Например, пусть имеется функция `f(int i, int j)`, а нам нужна функция `g(i)`, которая делает то же самое, что `f(i, 5)`, только мы не хотим каждый раз писать `5`.

Приведем более интересный пример, который подробно проработаем, когда будем реализовывать адаптер. Функция `std::sort` принимает диапазон итераторов (последовательность, подлежащую сортировке), но ее также можно вызвать с тремя аргументами – третьим будет объект сравнения (по умолчанию используется `std::less`, который, в свою очередь, вызывает `operator<()` для сравниваемых объектов).

Но нам нужно нечто иное – мы хотим сравнивать числа с плавающей точкой *неточно*, с некоторым допуском. Если два числа `x` и `y` достаточно близки, то мы не будем считать, что одно меньше другого. Лишь если `x` намного меньше `y`,

мы позаботимся о том, чтобы в отсортированной последовательности x предшествовало y.

Вот как выглядит наш функтор (вызываемый объект) для сравнения:

```
struct much_less {
    template <typename T>
    bool operator()(T x, T y) {
        return x < y &&
            std::abs(x - y) > tolerance*std::max(std::abs(x), std::abs(y));
    }
    static constexpr double tolerance = 0.2;
};
```

Этот объект сравнения можно использовать в сочетании со стандартной сортировкой:

```
std::vector<double> v;
std::sort(v.begin(), v.end(), much_less());
```

Однако если такая сортировка бывает нужна часто, то лучше бы каррировать последний аргумент, разработав адаптер, который принимает только два итератора, а функция сортировки подразумевается. Вот этот адаптер – очень простой:

```
template<typename RandomIt>
void sort_much_less(RandomIt first, RandomIt last) {
    std::sort(first, last, much_less());
}
```

Теперь можно вызывать функцию сортировки с двумя аргументами:

```
std::vector<double> v;
sort_much_less(v.begin(), v.end());
```

Но если мы часто вызываем sort таким образом для сортировки всего контейнера, то возникает желание еще раз изменить интерфейс и создать еще один адаптер:

```
template<typename Container> void sort_much_less(Container& c) {
    std::sort(c.begin(), c.end(), much_less());
}
```

Теперь код будет выглядеть еще проще:

```
std::vector<double> v;
sort_much_less(v);
```

Важно отметить, что C++14 предлагает альтернативный способ написания таких простых адаптеров, который мы всячески рекомендуем: можно использовать лямбда-выражение:

```
auto sort_much_less = [](auto first, auto last) {
    return std::sort(first, last, much_less());
};
```

Написать адаптер контейнера тоже нетрудно:

```
auto sort_much_less = [](auto& container) {
    return std::sort(container.begin(), container.end(), much_less());
};
```

Заметим, что в одной программе нельзя иметь два таких выражения с одинаковым именем – лямбда-выражения так не перегружаются, поскольку они и не функции вовсе, а объекты.

Возвращаясь к вопросу о вызове функций с частично фиксированными (или связанными с константами) аргументами, следует сказать, что это настолько распространенная вещь, что в стандартной библиотеке C++ есть даже стандартный настраиваемый адаптер для этой цели: `std::bind`. Вот пример его использования:

```
using namespace std::placeholders; // для _1, _2 и т. д.
int f3(int i, int j, int k) { return i + j + k; }
auto f2 = std::bind(f3, _1, _2, 42);
auto f1 = std::bind(f3, 5, _1, 7);
f2(2, 6);           // возвращает 50
f1(3);              // возвращает 15
```

В этом стандартном адаптере применяется специальный *мини-язык* – первый аргумент `std::bind` – связываемая функция, остальные – ее аргументы, по порядку. Аргументы, подлежащие связыванию, заменяются маркерами `_1`, `_2` и т. д. (необязательно в таком порядке, т. е. разрешается изменять порядок аргументов). Тип возвращаемого значения не специфицируется, вместо него следует указывать `auto`.

Каким бы полезным ни был адаптер `std::bind`, он не освобождает нас от необходимости писать собственные адаптеры функций. Главное ограничение `std::bind` состоит в том, что он не позволяет связывать шаблонные функции. Нельзя написать так:

```
auto sort_much_less = std::bind(std::sort, _1, _2, much_less()); // Нет!
```

Этот код не откомпилируется. Внутри шаблона можно произвести связывание с какой-то его конкретизацией, но, по крайней мере в примере с сортировкой, это нам ничего не дает:

```
template<typename RandomIt> void sort_much_less(RandomIt first,
                                                RandomIt last) {
    auto f = std::bind(std::sort<RandomIt, much_less>, _1, _2,
    much_less());
    f(first, last, much_less());
}
```

До сих пор мы рассматривали только адаптеры, преобразующие интерфейсы времени выполнения, т. е. интерфейсы, которые вызываются во время выполнения программы. Однако в C++ имеются также интерфейсы времени компиляции – одним из основных примеров, рассмотренных в предыдущей главе,

было проектирование на основе политик. Эти интерфейсы не всегда в точности таковы, какими мы хотели бы их видеть, поэтому далее мы научимся писать адаптеры времени компиляции.

Адаптеры времени компиляции

В главе 16 мы узнали о политиках, которые служат строительными блоками для классов, – они позволяют программисту настроить конкретное поведение реализации. Например, мы можем реализовать на основе политик интеллектуальный указатель, автоматически удаляющий объект, которым владеет. В данном случае политика является конкретной реализацией удаления:

```
template <typename T,
          template <typename> class DeletionPolicy = DeleteByOperator>
class SmartPtr {
    public:
    explicit SmartPtr(
        T* p = nullptr,
        const DeletionPolicy<T>& deletion_policy = DeletionPolicy<T>()
    ) : p_(p),
    deletion_policy_(deletion_policy)
    {}
    ~SmartPtr() {
        deletion_policy_(p_);
    }
    ... интерфейс указателя ...
    private:
    T* p_;
    DeletionPolicy<T> deletion_policy_;
};
```

Заметим, что политика удаления сама является шаблоном – это *шаблонный параметр*. Политика удаления по умолчанию вызывает оператор `delete`:

```
template <typename T>
struct DeleteByOperator {
    void operator()(T* p) const {
        delete p;
    }
};
```

Однако для объектов, выделенных из пользовательской кучи, нам нужна другая политика удаления, которая возвращала бы память в эту кучу:

```
template <typename T>
struct DeleteHeap {
    explicit DeleteHeap(MyHeap& heap)
        : heap_(heap) {}
    void operator()(T* p) const {
        p->~T();
        heap_.deallocate(p);
    }
```

```
    private:
    MyHeap& heap_;
};
SmartPtr<int, DeleteHeap<int>> p; // в этом указателе используется политика DeleteHeap
```

Но эта политика недостаточно гибкая – она может работать только с кучами типа `MyHeap` и никакими другими. Ее можно обобщить, сделав тип кучи вторым параметром шаблона. При условии что в классе кучи есть функция-член `deallocate()`, которая возвращает память в кучу, этот класс можно будет использовать вместе с такой политикой:

```
template <typename T, typename Heap>
struct DeleteHeap {
    explicit DeleteHeap(Heap& heap)
        : heap_(heap) {}
    void operator()(T* p) const {
        p->~T();
        heap_.deallocate(p);
    }
    private:
    Heap& heap_;
};
```

Разумеется, если имеется класс кучи, в котором эта функция-член называется по-другому, мы сможем написать адаптер класса, чтобы заставить и его работать с нашей политикой. Но перед нами стоит более серьезная проблема – эта политика не работает с нашим интеллектуальным указателем. Следующий код не компилируется:

```
SmartPtr<int, DeletelHeap> p; // не компилируется
```

Причина опять-таки в несогласованности интерфейсов, только теперь речь идет о другом интерфейсе – шаблон `template <typename T, template <typename> class DeletionPolicy> class SmartPtr {};` ожидает, что второй аргумент будет шаблоном с одним параметром-типом. А вместо этого мы подсунули шаблон `DeleteHeap` с двумя параметрами-типами. Это все равно, что пытаться вызвать с двумя аргументами функцию, имеющую всего один параметр, – работать не будет. Нам нужен адаптер, который преобразует наш шаблон с двумя параметрами в шаблон с одним параметром, а второй аргумент зафиксирует, сделав его конкретным типом кучи (но нам не придется переписывать политику под каждый тип кучи, нужно будет только написать несколько адаптеров). Для создания такого адаптера, `DeleteMyHeap`, можно воспользоваться псевдонимом шаблона:

```
    template <typename T> using DeleteMyHeap = DeleteHeap<T, MyHeap>;
```

Можно было бы также использовать наследование и включить конструкторы базового класса в производный класс адаптера:

```
template <typename T>
struct DeleteMyHeap : public DeleteHeap<T, MyHeap> {
    using DeleteHeap<T, MyHeap>::DeleteHeap;
};
```

Второй вариант, очевидно, гораздо длиннее. Однако мы должны знать оба способа написания адаптеров шаблонов, потому что у псевдонима шаблон есть одно важное ограничение. Чтобы проиллюстрировать его, рассмотрим еще один пример, в котором нужен адаптер. Начнем с реализации оператора вывода в поток для любого STL-совместимого последовательного контейнера, в элементах которого определен такой оператор. Это простой шаблон функции:

```
template <template <typename> class Container, typename T>
std::ostream& operator<<(std::ostream& out, const Container<T>& c) {
    bool first = true;
    for (auto x : c) {
        if (!first) out << ", ";
        first = false;
        out << x;
    }
    return out;
}
```

У этой шаблонной функции два параметра-типа: тип контейнера и тип элемента. Контейнер сам является шаблоном с одним параметром-типом. Компилятор выводит тип контейнера и тип элемента из второго аргумента функции (первый аргумент любого оператора `operator<<()` – всегда поток). Протестируем наш оператор вывода на простом контейнере:

```
template <typename T> class Buffer {
    public:
    explicit Buffer(size_t N) : N_(N), buffer_(new T[N_]) {}
    ~Buffer() { delete [] buffer_; }
    T* begin() const { return buffer_; }
    T* end() const { return buffer_ + N_; }
    ...
    private:
    const size_t N_;
    T* const buffer_;
};

Buffer<int> buffer(10);
... заполнить буфер ...
cout << buffer;     // печатаются все элементы буфера
```

Но это игрушечный контейнер, не очень-то и полезный. А мы хотим напечатать элементы настоящего контейнера, такого как `std::vector`:

```
std::vector<int> v;
... поместить элементы в v ...
cout << v;
```

Увы, этот код не компилируется. Причина в том, что `std::vector` на самом деле не является шаблоном с одним параметром-типом, хотя использовали мы его именно так. У него два параметра-типа, второй – тип распределителя памяти. Для распределителя имеется значение по умолчанию, поэтому мы и можем написать `std::vector<int>`, не оскорбляя компилятор. Но все равно это шаблон

с двумя параметрами, а в объявлении нашего оператора вывода в поток разрешается принимать шаблоны только с одним параметром. Эту проблему снова можно решить с помощью адаптера (кстати, у большинства STL-контейнеров есть параметр по умолчанию – распределитель памяти). Проще всего написать такой адаптер, воспользовавшись псевдонимом:

```
template <typename T> using vector1 = std::vector<T>;
vector1<int> v;
...
cout << v;    // тоже не компилируется
```

К сожалению, и этот код не компилируется, и теперь мы можем объяснить, в чем состоит вышеупомянутое ограничение псевдонимов шаблонов – их нельзя использовать при выведении типа аргумента шаблона. Когда компилятор пытается вывести типы аргументов шаблона для вызова `operator<<()` с аргументами `cout` и `v`, он *не видит* псевдонима шаблона `vector1`. В таком случае придется использовать адаптер в виде производного класса:

```
template <typename T> struct vector1 : public std::vector<T> {
    using std::vector<T>::vector;
};
vector1<int> v;
...
cout << v;
```

Итак, мы видели, как реализовать декораторы для добавления нового поведения в интерфейсы классов и функций и как создать адаптеры в случае, если существующий интерфейс не подходит. Декоратор и Адаптер, последний особенно, – очень общие и гибкие паттерны, применимые для решения многих задач. Поэтому неудивительно, что задачу зачастую можно решить несколькими способами, выбирая тот или иной паттерн. В следующем разделе мы рассмотрим один такой случай.

Адаптер и Политика

Паттерны Адаптер и Политика (или Стратегия) относятся к числу наиболее общих, а C++ добавил к ним возможности программирования шаблонов. Это расширяет сферы их применения и иногда размывает границу между паттернами. Сами по себе паттерны определены совершенно по-разному – Политики предоставляют пользовательские реализации, а Адаптеры изменяют интерфейс и добавляют функциональность в существующий интерфейс (последнее – характерная особенность декораторов, но, как мы видели, большинство декораторов реализуется как адаптеры). В предыдущей главе мы также видели, что C++ расширяет возможности проектирования на основе политик; в частности политики в C++ могут добавлять и удалять части интерфейса, а также управлять его реализацией. Таким образом, хотя паттерны и различны, между задачами, к которым они применимы, есть значительное перекрытие. В пре-

дыдущей главе мы видели пример основанного на политиках ограниченного типа-значения: типа, который, по крайней мере концептуально, является числом *наподобие int*, но имеет интерфейс, который можно определять по частям, например: допускает сравнение, упорядоченный, допускает сложение, но не допускает ни умножения, ни деления. Теперь, когда мы узнали об адаптерах, возникает искушение применить их к решению той же задачи. Мы начнем с простого типа-значения, интерфейс которого почти ничего не поддерживает, а затем добавим желаемые возможности по одной.

Вот как выглядит наш первоначальный вариант шаблона класса `Value`:

```
template <typename T> class Value {
    public:
    typedef T basic_type;
    typedef Value value_type;
    explicit Value() : val_(T()) {}
    explicit Value(T v) : val_(v) {}
    Value(const Value& rhs) : val_(rhs.val_) {}
    Value& operator=(Value rhs) { val_ = rhs.val_; return *this; }
    Value& operator=(basic_type rhs) { val_ = rhs; return *this; }
    friend std::ostream& operator<<(std::ostream& out, Value x) {
        out << x.val_; return out;
    }
    friend std::istream& operator>>(std::istream& in, Value& x) {
        in >> x.val_; return in;
    }
    protected:
    T val_;
};
```

Он допускает копирование, присваивание и печать (некоторые из этих возможностей можно было бы также перенести в адаптеры). Больше ничего делать с этим классом нельзя – нет ни сравнения на равенство или неравенство, ни арифметических операций. Однако мы можем создать адаптер, добавляющий интерфейс сравнения:

```
template <typename V> class Comparable : public V {
    public:
    using V::V;
    typedef typename V::value_type value_type;
    typedef typename value_type::basic_type basic_type;
    Comparable(value_type v) : V(v) {}
    friend bool operator==(Comparable lhs, Comparable rhs) {
        return lhs.val_ == rhs.val_;
    }
    friend bool operator!=(Comparable lhs, Comparable rhs) {
        return lhs.val_ != rhs.val_;
    }
    friend bool operator==(Comparable lhs, basic_type rhs) {
        return lhs.val_ == rhs;
    }
```

```
    friend bool operator==(basic_type lhs, Comparable rhs) {
        return lhs == rhs.val_;
    }
    friend bool operator!=(Comparable lhs, basic_type rhs) {
        return lhs.val_ != rhs;
    }
    friend bool operator!=(basic_type lhs, Comparable rhs) {
        return lhs != rhs.val_;
    }
};
```

Это адаптер класса – он наследует классу, чьи возможности расширяет, а значит, наследует весь его интерфейс и добавляет кое-что свое: полный набор операторов сравнения. Мы знаем, как используются такие адаптеры:

```
using V = Comparable<Value<int>>;
V i(3), j(5);
i == j;    // false
i == 3;    // true
5 == j;    // также true
```

Это одна возможность. А еще что-нибудь? Нет проблем – адаптер `Ordered` можно написать аналогично, только он будет предоставлять операторы `<`, `<=`, `>` и `>=`:

```
template <typename V> class Ordered : public V {
    public:
    using V::V;
    typedef typename V::value_type value_type;
    typedef typename value_type::basic_type basic_type;
    Ordered(value_type v) : V(v) {}
    friend bool operator<(Ordered lhs, Ordered rhs) {
        return lhs.val_ < rhs.val_;
    }
    friend bool operator<(basic_type lhs, Ordered rhs) {
        return lhs < rhs.val_;
    }
    friend bool operator<(Ordered lhs, basic_type rhs) {
        return lhs.val_ < rhs;
    }
    ... то же самое для других операторов ...
};
```

Оба адаптера можно сочетать – как мы знаем, они допускают композицию и могут задаваться в любом порядке:

```
using V = Ordered<Comparable<Value<int>>>; // или Comparable<Ordered<...>
V i(3), j(5);
i == j;    // false
i <= 3;    // true
```

Ничуть не сложнее реализовать адаптеры для сложения или умножения:

```
template <typename V> class Addable : public V {
    public:
```

```
    using V::V;
    typedef typename V::value_type value_type;
    typedef typename value_type::basic_type basic_type;
    Addable(value_type v) : V(v) {}
    friend Addable operator+(Addable lhs, Addable rhs) {
        return Addable(lhs.val_ + rhs.val_);
    }
    friend Addable operator+(Addable lhs, basic_type rhs) {
        return Addable(lhs.val_ + rhs);
    }
    friend Addable operator+(basic_type lhs, Addable rhs) {
        return Addable(lhs + rhs.val_);
    }
    ... то же самое для - ...
};

template <typename V> class Multipliable : public V {
    public:
    using V::V;
    typedef typename V::value_type value_type;
    typedef typename value_type::basic_type basic_type;
    Multipliable(value_type v) : V(v) {}
    friend Multipliable operator*(Multipliable lhs, Multipliable rhs) {
        return Multipliable(lhs.val_ * rhs.val_);
    }
    ... то же самое для других вариантов * и / ...
};
```

И это предел того, что можно сделать, по крайней мере без труда, с помощью паттерна Адаптер:

```
using V = Multipliable<Addable<Value<int>>>;
V i(5), j(3), k(7);
i + j;              // правильно
i * j;              // правильно
(i + j) * (k + 3);  // неправильно
```

Проблема здесь в том, что в выражении `i + j` используется `operator +`, определенный в адаптере `Addable`, а этот оператор возвращает объект типа `Addable<Value<int>>`. Оператор умножения ожидает получить тип `Multipliable<Addable<Value<int>>>` и не принимает *частичный* тип (не существует неявного преобразования из базового класса в производный). Эту проблему можно было бы решить, изменив порядок `Multipliable` и `Addable`, но тогда перестало бы компилироваться выражение `(i * j) + (i / k)` – по той же самой причине.

Это ограничение компонуемых адаптеров – они прекрасно работают, до тех пор пока добавленный ими интерфейс не оказывается вынужден вернуть адаптированный тип. У нас не было никаких проблем с операторами сравнениями, возвращающими `bool`, но как только понадобилось вернуть сам адаптированный тип, компонуемости настал конец. Есть несколько способов обойти эту проблему, но все они очень сложны и имеют побочные эффекты. Если мы

спотыкаемся на этом месте, то решение на основе политик из главы 16 выглядит гораздо проще:

```
template <typename T, template <typename, typename> class ... Policies>
class Value : public Policies<T, Value<T, Policies ... >> ...
{ ..... };
    using V = Value<int, Addable, Multipliable, Ordered>; // работает в любом порядке
```

Впрочем, как было показано в конце главы 16, у этого решения есть свои недостатки. Такова уж природа проблем, которые приходится решать программистам, – если задача достаточно сложна, то ее можно решить, и часто несколькими разными способами, но у каждого из них будут свои достоинства и ограничения. Нет никакой возможности сравнить каждые два паттерна, с помощью которых создаются очень разные проекты, направленные на достижение одной и той же цели. По крайней мере, не в книге конечного размера, сколь бы велик он ни был. Мы представили и проанализировали эти примеры в надежде вооружить читателя знаниями и пониманием, которые окажутся полезными при оценке столь же сложных и многообразных вариантов проектирования, но уже в реальных задачах.

Резюме

Мы изучили два самых распространенных паттерна – не только в C++, но и в проектировании программного обеспечения вообще. Паттерн Адаптер предлагает подход к решению широкого класса задач проектирования. Эти задачи разделяют только одну, но очень общую черту – дан класс, функция или программный компонент, предоставляющий некую функциональность, а требуется решить другую, хотя и родственную, задачу. Паттерн Декоратор во многих отношениях является подмножеством паттерна Адаптер, но он может только пополнять существующий интерфейс класса или функции новым поведением.

Мы видели, что адаптеры и декораторы, преобразующие интерфейс, могут применяться на любом этапе жизни программы – чаще всего с их помощью модифицируют интерфейс во время выполнения, чтобы класс можно было использовать в другом контексте, но существуют также адаптеры времени компиляции для обобщенного кода, которые позволяют использовать класс как компонент более крупного и сложного класса.

Паттерн Адаптер применим для решения многих совершенно разных проблем проектирования. Разнообразие задач и общность самого паттерна часто приводят к существованию альтернативных решений. Нередко в них используются абсолютно разные подходы – разные паттерны проектирования, – но в итоге получается похожее поведение. Отличаются компромиссы, дополнительные условия и ограничения, налагаемые выбранным подходом на дизайн системы, а также возможности расширения решения в тех или иных направлениях. С этой точки зрения, настоящая и предыдущая главы позволяют срав-

нить два очень разных подхода к проектированию решения одной и той же задачи и содержат оценку сильных и слабых сторон каждого подхода.

В следующей, и последней, главе мы познакомимся с большим и сложным паттерном, состоящим из нескольких взаимодействующих компонент. Этот паттерн – Посетитель – станет достойным финалом нашей оперы.

Вопросы

- Что такое паттерн Адаптер?
- Что такое паттерн Декоратор и чем он отличается от паттерна Адаптер?
- Классическая объектно-ориентированная реализация паттерна Декоратор обычно не рекомендуется в C++. Почему?
- Когда в декораторе класса в C++ следует использовать наследование, а когда композицию?
- Когда в адаптере класса в C++ следует использовать наследование, а когда композицию?
- C++ предлагает общий адаптер функции для каррирования аргументов, `std::bind`. Каковы его ограничения?
- C++11 предлагает псевдонимы шаблонов, которые можно использовать как адаптеры. Каковы их ограничения?
- Оба паттерна, Адаптер и Политика, можно использовать для расширения или модификации открытого интерфейса класса. Приведите несколько причин, по которым один паттерн следует предпочесть другому.

Глава 18

Паттерн Посетитель и множественная диспетчеризация

Паттерн Посетитель – еще один классический объектно-ориентированный паттерн проектирования, перечисленный среди 23 паттернов в книге Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides «Design Patterns – Elements of Reusable Object-Oriented Software». Он был одним из самых популярных паттернов в золотом веке объектно-ориентированного программирования, поскольку позволяет упростить сопровождение больших иерархий классов. В последние годы увлечение паттерном Посетитель в C++ пошло на спад, поскольку большие иерархии стали встречаться реже, а сам этот паттерн реализовать довольно трудно. Обобщенное программирование – в частности, новые языковые средства, добавленные в стандартах C++11 и C++14, – облегчает реализацию и сопровождение классов Посетителя, а благодаря новым применениям угасший было интерес к старому паттерну вспыхнул с новой силой.

В этой главе рассматриваются следующие вопросы:

- паттерн Посетитель;
- реализации Посетителя в C++;
- использование обобщенного программирования для упрощения классов Посетителя;
- использование Посетителя в составных объектах;
- Посетитель времени компиляции и отражение.

Технические требования

Примеры кода: **https://github.com/PacktPublishing/Hands-On-Design-Patterns-with-CPP/tree/master/Chapter18**.

Паттерн Посетитель

Паттерн Посетитель выделяется на фоне других классических объектно-ориентированных паттернов своей сложностью. С одной стороны, сама базовая структура паттерна Посетитель весьма сложна и включает много классов, которые должны координировать свою работу. С другой стороны, даже описание паттерна Посетитель вызывает трудности – есть несколько очень разных способов описать один и тот же паттерн. Многие паттерны можно применить к решению нескольких видов задач, но Посетитель и тут стоит особняком – описать его действие можно различными способами, в которых используется совершенно разная лексика, говорится о, на первый взгляд, не связанных между собой проблемах, да и вообще ничего общего не просматривается. Однако все они описывают один и тот же паттерн. Мы начнем с исследования разных ликов паттерна Посетитель, а затем перейдем к его реализации.

Что такое паттерн Посетитель?

Паттерн Посетитель отделяет алгоритм от структуры объекта, который содержит данные для этого алгоритма. Благодаря Посетителю мы можем добавить новую операцию в иерархию классов, не изменяя сами классы. Применения паттерна Посетитель – пример следования **принципу открытости-закрытости** в проектировании программного обеспечения: класс (или иная единица кода, например модуль) должен быть закрыт для модификации; после того как класс объявил свой интерфейс клиентам, клиенты становятся зависимы от этого интерфейса и предоставляемой им функциональности. Интерфейс должен оставаться стабильным, не должно возникать необходимости в модификации классов при сопровождении и развитии ПО. В то же время класс должен быть открыт для расширений – для удовлетворения новых требований разрешено добавлять новую функциональность. Как и для всех общих принципов, можно отыскать контрпример, когда строго следовать правилу хуже, чем нарушить его. И, как и все общие принципы, ценен он не тем, что требует неукоснительного соблюдения во всех случаях, а тем, что предлагает правило *по умолчанию*, рекомендацию, которой стоит следовать в отсутствие веских причин для нарушения. Реальность же такова, что в повседневной работе *нет ничего особенного*, и результат будет лучше, если этот принцип соблюдать.

При таком взгляде паттерн Посетитель позволяет добавить функциональность в класс или целую иерархию классов, не требуя модифицировать класс. Эта возможность особенно полезна при работе с открытыми API – пользователи API могут расширять его, добавляя новые операции, без необходимости изменять исходный код.

Совсем другой, более технический способ описать паттерн Посетитель – сказать, что он реализует **двойную диспетчеризацию**. Тут не обойтись без пояснений. Начнем с обычных вызовов виртуальных функций:

```
class Base {
    virtual void f() = 0;
};
class Derived1 : public Base {
    void f() override;
};
class Derived2 : public Base {
    void f() override;
};
```

Если вызвать виртуальную функцию `b->f()` через указатель на базовый класс `b`, то вызов диспетчеризуется к функции `Derived1::f()` или `Derived2::f()` в зависимости от истинного типа объекта. Это **одиночная диспетчеризация** – какую функцию вызывать, определяется одним фактором, типом объекта.

Теперь предположим, что функция `f()` принимает еще и аргумент, который также является указателем на базовый класс:

```
class Base {
    virtual void f(Base* p) = 0;
};
class Derived1 : public Base {
    void f(Base* p) override;
};
class Derived2 : public Base {
    void f(Base* p) override;
};
```

Фактический тип объекта `*p` также совпадает с типом одного из производных классов. Теперь вызов `b->f(p)` может быть разрешен четырьмя способами, т. к. каждый из объектов `*b` и `*p` может принадлежать одному из двух производных типов. Хотелось бы, чтобы действие функции было различным в каждом случае. Это было бы двойной диспетчеризацией – выбор выполняемого кода определяется двумя разными факторами. Виртуальные функции не позволяют реализовать двойную диспетчеризацию непосредственно, а вот паттерн Посетитель именно это и делает.

При таком изложении не очевидно, как связаны паттерн Посетитель для **двойной диспетчеризации** с паттерном Посетитель для **добавления операции**. Однако это в точности один и тот же паттерн, а требования в действительности совпадают. Убедиться в этом поможет такое соображение – добавление операции во все классы иерархии эквивалентно добавлению виртуальной функции, т. е. мы имеем один фактор, управляющий конечным назначением каждого вызова, – тип объекта. Но если мы умеем добавлять виртуальные функции, то можем добавить и несколько – по одной для каждой операции, которую необходимо поддержать. Тип операции – второй фактор, управляющий диспетчеризацией, аналогичный аргументу функции в предыдущем примере. Наоборот, если бы мы знали, как реализовать двойную диспетчеризацию, то могли бы сделать то, что делает паттерн Посетитель, – добавить виртуальную функцию для каждой интересующей нас операции.

Теперь, когда мы знаем, что делает Посетитель, самое время задать вопросы: «*Зачем* это может понадобиться?», «*Как* применяется двойная диспетчеризация?» и «Зачем нужен *еще один способ* добавления квазивиртуальной функции в класс, когда мы умеем добавлять *настоящую* виртуальную функцию?». Оставляя в стороне случай, когда исходный код открытого API недоступен, зачем добавлять операцию *внешним образом*, если ее можно реализовать в каждом классе? Рассмотрим задачу о сериализации и десериализации. Сериализацией называется операция, которая преобразует объект в форму, допускающую хранение на внешнем носителе или передачу (например, запись в файл). Десериализация – это обратная операция – она конструирует новый объект из его сериализованного хранимого образа. Чтобы поддержать сериализацию и десериализацию естественным объектно-ориентированным способом, у каждого класса в иерархии должно быть два метода, по одному для каждой операции. Но что, если способов сохранить объект несколько? Например, может понадобиться записать объект в буфер памяти для последующей передачи по сети и десериализации на другой машине. А можно сохранить объект на диске или преобразовать все объекты в контейнере в некоторый формат разметки, например JSON. Прямолинейный подход означал бы, что в каждый объект нужно добавить методы сериализации и десериализации для каждого механизма сериализации. Если появится новый подход к сериализации, то придется пройтись по всей иерархии классов и добавить поддержку для него.

Альтернатива – реализовать всю операцию сериализации-десериализации в отдельной функции, которая умеет работать со всеми классами. В результате получается цикл, который обходит все объекты и содержит внутри себя большое решающее дерево. Код должен опросить каждый объект и определить его тип, скажем, с помощью динамического приведения. Когда в иерархию добавляется новый класс, все реализации сериализации и десериализации необходимо обновить с учетом новых объектов.

Обе реализации очень трудно сопровождать для больших иерархий. Паттерн Посетитель предлагает решение, он позволяет реализовать новую операцию – в данном случае сериализацию – вне классов и без их модификации, но также и без гигантского решающего дерева в цикле. (Отметим, что Посетитель – не единственное решение проблемы сериализации, C++ предлагает и другие подходы, но в этой главе нас интересует паттерн Посетитель.)

Как было сказано в начале главы, Посетитель – сложный паттерн со сложным описанием. Понять его будет легче на конкретных примерах, и в следующем разделе мы начнем с очень простых.

Простой Посетитель на C++

Единственный способ по-настоящему разобраться в том, как работает паттерн Посетитель, – проработать пример. Начнем с очень простого. Прежде всего нам понадобится иерархия классов:

```
class Pet {
    public:
    virtual ~Pet() {}
    Pet(const std::string& color) : color_(color) {}
    const std::string& color() const { return color_; }
    private:
    std::string color_;
};

class Cat : public Pet {
    public:
    Cat(const std::string& color) : Pet(color) {}
};

class Dog : public Pet {
    public:
    Dog(const std::string& color) : Pet(color) {}
};
```

В этой иерархии имеется базовый класс домашнего питомца Pet и несколько производных для различных животных. Мы хотим добавить во все классы некоторые операции, например: *покормить питомца* или *поиграть с питомцем*. Реализация зависит от типа питомца, поэтому, если бы мы добавляли эти операции непосредственно в класс, то они должны были бы быть виртуальными функциями. Для такой простой иерархии классов это не проблема, но мы предвидим, что в будущем придется сопровождать гораздо более крупную систему, и тогда модификация каждого класса в иерархии обошлась бы дорого и заняла бы много времени. Нам нужен способ получше, и начнем мы с создания нового класса PetVisitor, который будет применяться к каждому объекту Pet (посещать его) и выполнять необходимые операции. Сначала объявим класс:

```
class Cat;
class Dog;
class PetVisitor {
    public:
    virtual void visit(Cat* c) = 0;
    virtual void visit(Dog* d) = 0;
};
```

Мы вынуждены сделать опережающие объявления всех классов в иерархии Pet, поскольку PetVisitor должен быть объявлен раньше конкретных классов Pet. Далее мы должны сделать так, чтобы иерархия Pet допускала посещение. Это значит, что модифицировать ее все-таки придется, но только один раз – независимо от того, сколько операций мы добавим впоследствии. Мы должны добавить в каждый класс виртуальную функцию, которая будет принимать Посетителя:

```
class Pet {
    public:
    virtual void accept(PetVisitor& v) = 0;
    .....
};
```

```
class Cat : public Pet {
    public:
    void accept(PetVisitor& v) override { v.visit(this); }
    .....
};

class Dog : public Pet {
    public:
    void accept(PetVisitor& v) override { v.visit(this); }
    .....
};
```

Теперь наша иерархия `Pet` стала посещаемой, и мы имеем абстрактный класс `PetVisitor`. Все готово для реализации новых операций с классами. (Заметим, что ничего из сделанного до сих пор не зависит от того, какие операции мы будем добавлять; мы только подготовили инфраструктуру посещения, которую нужно реализовать однократно.) Для добавления операций нужно реализовать классы конкретных Посетителей, производные от `PetVisitor`:

```
class FeedingVisitor : public PetVisitor {
    public:
    void visit(Cat* c) override {
        std::cout << "Покормить тунцом " << c->color() << " кошку"
                  << std::endl;
    }
    void visit(Dog* d) override {
        std::cout << "Покормить стейком " << d->color() << " собаку"
                  << std::endl;
    }
};

class PlayingVisitor : public PetVisitor {
    public:
    void visit(Cat* c) override {
        std::cout << "Поиграть в перышко с " << c->color() << " кошкой"
                 << std::endl;
    }
    void visit(Dog* d) override {
        std::cout << "Поиграть в брось-принеси с " << d->color() << " собакой"
                  << std::endl;
    }
};
```

В предположении, что инфраструктура посещения уже встроена в иерархию классов, новую операцию можно добавить, реализовав производный от Посетителя класс со всеми переопределенными виртуальными функциями `visit()`. Чтобы вызвать операцию для объекта из иерархии классов, нужно создать посетителя и посетить объект:

```
Cat c("orange");
FeedingVisitor fv;
c.accept(fv);     // Покормить тунцом рыжую кошку
```

Этот пример слишком прост в одном важном отношении – в точке вызова посетителя известен точный тип посещаемого объекта. Чтобы сделать пример более реалистичным, объект нужно посещать полиморфно:

```
std::unique_ptr<Pet> p(new Cat("orange"));
.....
FeedingVisitor fv;
p->accept(fv);
```

Здесь в момент компиляции фактический тип объекта, на который указывает p, неизвестен; в точке, где принимается посетитель, p мог бы происходить из разных источников. Самого посетителя тоже можно использовать полиморфно, хотя это встречается реже:

```
std::unique_ptr<Pet> p(new Cat("orange"));
.....
std::unique_ptr<PetVisitor> v(new FeedingVisitor);
.....
p->accept(*v);
```

В таком виде код подчеркивает ту сторону паттерна Посетитель, которая связана с двойной диспетчеризацией, – вызов accept() диспетчеризуется к конкретной функции visit() в зависимости от двух факторов: типа посещаемого объекта *p и типа посетителя *v. Если мы захотим подчеркнуть этот аспект Посетителя, то можем вызывать посетителей с помощью вспомогательной функции:

```
void dispatch(Pet* p, PetVisitor* v) { p->accept(*v); }
Pet* p = .....;
PetVisitor* v = .....;
dispatch(p, v);    // двойная диспетчеризация
```

Теперь у нас есть примитивный пример классического объектно-ориентированного посетителя в C++. Несмотря на простоту, он содержит все необходимые компоненты; реализация для большой реальной иерархии классов и нескольких операций посетителя потребовала бы написания гораздо большего объема кода, но ничего принципиально нового нет, просто более масштабное повторение уже сделанного. Этот пример демонстрирует оба аспекта паттерна Посетитель; с одной стороны, функциональность кода с подготовленной инфраструктурой посещения позволяет добавлять новые операции, не внося изменений в сами классы. С другой стороны, если смотреть только на способ вызова операции, обращение к accept(), то мы реализовали двойную диспетчеризацию.

Привлекательность паттерна Посетитель уже очевидна – мы можем добавить сколько угодно новых операций без необходимости модифицировать каждый класс в иерархии. Если в иерархию Pet добавлен новый класс, то забыть о его обработке невозможно – если с посетителем вообще ничего не делать, то вызов accept() в новом классе не откомпилируется, потому что не существует соответствующей функции visit(). Добавив новый перегруженный

вариант visit() в базовый класс PetVisitor, мы вынуждены добавить его и во все производные классы, в противном случае компилятор сообщит, что чисто виртуальная функция не переопределена. Последнее также является одним из главных недостатков паттерна Посетитель – если в иерархию добавлен новый класс, то необходимо обновить всех посетителей, пусть даже класс не собирается поддерживать некоторые операции. По этой причине иногда рекомендуют использовать Посетитель только для *относительно стабильных* иерархий, в которые новые классы добавляются нечасто. Существует также альтернативная реализация Посетителя, которая несколько смягчает эту проблему; мы рассмотрим ее ниже.

Пример в этом разделе очень прост – новая операция не принимает аргументов и не возвращает результата. Далее мы выясним, являются ли эти ограничения существенными и как их снять.

Обобщения и ограничения паттерна Посетитель

Наш первый посетитель, описанный в предыдущем разделе, позволил, по сути дела, добавить виртуальную функцию в каждый класс иерархии. У этой виртуальной функции не было ни параметров, ни возвращаемого значения. С первым легко разобраться, нет никаких причин, мешающих функциям visit() принимать параметры. Расширим нашу иерархию классов, разрешив домашним питомцам иметь котят и щенков. Для этого недостаточно одного лишь паттерна Посетитель – мы хотим добавить в классы иерархии не только новые операции, но и новые данные-члены. Для первого хватит и Посетителя, но для второго нужно вносить изменения в код. Проектирование на основе политик позволило бы вынести это изменение в новую реализацию существующей политики, если бы мы заранее побеспокоились о подходящей политике. В этой книге политикам посвящена отдельная глава, так что здесь мы не станем смешивать разные паттерны, а просто добавим новые данные-члены:

```
class Pet {
    public:
    ....
    void add_child(Pet* p) { children_.push_back(p); }
    virtual void accept(PetVisitor& v, Pet* p = nullptr) = 0;
    private:
    std::vector<Pet*> children_;
};
```

Каждый питомец-родитель Pet хранит своих отпрысков (заметьте, что контейнер – это вектор указателей, а не вектор уникальных указателей, поэтому объект не владеет своими детьми, а просто имеет доступ к ним). Мы также добавили новую функцию-член add_child() для добавления объектов в вектор. Можно было бы сделать это с помощью посетителя, но эта функция невиртуальная, поэтому ее нужно добавить только в базовый класс, а не в каждый производный, так что посетитель здесь ни при чем. Функция accept() изменена

и теперь принимает дополнительный параметр, который придется добавить во все производные классы, которые просто передают его функции visit():

```
class Cat : public Pet {
    public:
    Cat(const std::string& color) : Pet(color) {}
    void accept(PetVisitor& v, Pet* p = nullptr) override {
        v.visit(this, p);
    }
};
class Dog : public Pet {
    public:
    Dog(const std::string& color) : Pet(color) {}
    void accept(PetVisitor& v, Pet* p = nullptr) override {
        v.visit(this, p);
    }
};
```

Функцию visit() тоже придется модифицировать, чтобы она принимала дополнительный аргумент, даже в тех посетителях, которым он не нужен. Таким образом, изменение состава параметров функции accept() – дорогая глобальная операция, которую не стоит делать часто, а лучше вообще не делать. Заметим, что все переопределенные виртуальные функции в иерархии обязаны иметь одинаковые параметры. Паттерн Посетитель распространяет это ограничение на все операции, добавляемые посредством одного и того же базового класса Посетителя. Стандартное обходное решение этой проблемы – передавать параметры в агрегатах (классах или структурах, объединяющих несколько параметров). В объявлении функции visit() указано, что она принимает указатель на базовый класс агрегата, а каждый посетитель принимает указатель на производный класс, в котором могут быть дополнительные поля, и использует их по своему усмотрению.

Теперь наш дополнительный аргумент передается по цепочке вызовов виртуальных функций посетителю, который может как-то использовать его. Давайте создадим посетителя, который регистрирует рождение питомцев и добавляет новые объекты питомцев в качестве потомков родительских объектов:

```
class BirthVisitor : public PetVisitor {
    public:
    void visit(Cat* c, Pet* p) override {
        assert(dynamic_cast<Cat*>(p));
        c->add_child(p);
    }
    void visit(Dog* d, Pet* p) override {
        assert(dynamic_cast<Dog*>(p));
        d->add_child(p);
    }
};
```

Если мы хотим гарантировать отсутствие биологической невозможности в нашем генеалогическом древе, то проверку придется делать во время вы-

полнения – на этапе компиляции фактические типы полиморфных объектов неизвестны. Нового посетителя так же легко использовать, как описанных в предыдущем разделе:

```
Pet* parent;   // кошка
BirthVisitor bv;
Pet* child(new Cat("calico"));
parent->accept(bv, child);
```

После того как отношения родитель–потомок установлены, мы, возможно, захотим исследовать семейства наших питомцев. Это еще одна операция, и для нее нужен отдельный посетитель:

```
class FamilyTreeVisitor : public PetVisitor {
    public:
    void visit(Cat* c, Pet*) override {
        std::cout << "Котята: ";
        for (auto k : c->children_) {
            std::cout << k->color() << " ";
        }
        std::cout << std::endl;
    }
    void visit(Dog* d, Pet*) override {
        std::cout << "Щенки: ";
        for (auto p : d->children_) {
            std::cout << p->color() << " ";
        }
        std::cout << std::endl;
    }
};
```

Мы столкнулись с небольшой проблемой – в таком виде код не откомпилируется. Дело в том, что класс `FamilyTreeVisitor` пытается получить доступ к члену `Pet::children_`, а тот закрыт. Это еще одно слабое место паттерна Посетитель – с нашей точки зрения, посетители добавляют в класс новые операции как виртуальные функции, но, с точки зрения компилятора, это вполне самостоятельные классы, а совсем не функции-члены класса `Pet`, и никакого специального доступа у них нет. Для применения паттерна Посетитель обычно требуется ослабить инкапсуляцию одним из двух способов: либо предоставить открытый доступ к данным (напрямую или с помощью функций-акцессоров), либо объявить классы Посетителя друзьями (что потребует внесения изменений в исходный код). В нашем примере мы выберем второй путь:

```
class Pet {
    .....
    friend class FamilyTreeVisitor;
};
```

Теперь посетитель генеалогического древа работает, как ожидается:

```
Pet* parent;   // кошка
.....
```

```
FamilyTreeVisitor tv;
parent->accept(tv);   // печатаются окрасы котят
```

В отличие от BirthVisitor, посетителю FamilyTreeVisitor не нужен дополнительный аргумент. А как насчет возвращаемых значений? Технически не требуется, чтобы функции visit() и accept() возвращали void. Они могут возвращать что угодно. Однако ограничение, требующее, чтобы все они возвращали значение одного и того же типа, обычно делает эту возможность бесполезной. Виртуальные функции могут иметь ковариантные возвращаемые типы, когда виртуальная функция базового класса возвращает объект некоторого класса, а в производных классах она переопределена и возвращает объект, производный от этого класса, но обычно даже эта возможность чрезмерно ограничена. Есть другое, более простое решение – функции visit() каждого объекта-посетителя имеют полный доступ к данным-членам этого объекта. Так почему бы не сохранить возвращаемое значение в самом классе Посетителя и не обращаться к нему позже? Это хорошо ложится на типичное использование, когда каждый посетитель добавляет свою операцию и, по всей вероятности, имеет уникальный возвращаемый тип, но сама операция возвращает значение одного и того же типа для всех классов иерархии. Например, можно поручить посетителю FamilyTreeVisitor подсчет общего числа потомков и возвращать это значение через объект Посетителя:

```
class FamilyTreeVisitor : public PetVisitor {
    public:
    FamilyTreeVisitor() : child_count_(0) {}
    void reset() { child_count_ = 0; }
    size_t child_count() const { return child_count_; }
    private:
    size_t child_count_;
};
FamilyTreeVisitor tv;
parent->accept(tv);
std::cout << "всего котят: " << tv.child_count() << std::endl;
```

Этот подход имеет ограничения в многопоточных программах – посетитель теперь не является потокобезопасным, поскольку несколько потоков не может использовать один и тот же объект Посетителя для посещения различных объектов питомцев. Типичное решение – завести по одному объекту Посетителя в каждом потоке, обычно в виде локальной переменной, созданной в стеке функции, вызывающей посетителя. Если это невозможно, то есть и более сложные варианты, например наделить каждого посетителя поточно-локальным состоянием, но их анализ выходит за рамки этой книги. С другой стороны, иногда мы хотим аккумулировать результаты нескольких посещений, и тогда описанная выше техника хранения результата в объекте Visitor работает идеально. Отметим еще, что такое же решение можно использовать для передачи аргументов операциям Посетителя вместо задания их при вызове функций visit(); мы можем хранить аргументы в самом объекте Посетителя, и тогда для

доступа к ним из посетителя не нужно ничего специально делать. Особенно хорошо эта техника работает, когда аргументы не изменяются при каждом вызове посетителя, но могут изменяться при вызовах разных посетителей.

Вернемся ненадолго к реализации класса `FamilyTreeVisitor`. Заметим, что мы в цикле обходим дочерние объекты родительского объекта и для каждого по очереди вызываем одну и ту же операцию. Но потомки дочерних объектов не обрабатываются – наше генеалогическое древо одноуровневое. Задача о посещении объектов, содержащих другие объекты, очень общая и встречается довольно часто. Пример с сериализацией, приведенный в самом начале главы, убедительно демонстрирует эту потребность – для сериализации составного объекта нужно одну за другой сериализовать его компоненты, которые, в свою очередь, реализуются точно так же, до тех пор пока мы не дойдем до встроенных типов `int`, `double` и т. д., которые уже умеем читать и записывать. В следующем разделе мы систематически займемся посещением сложных объектов.

Посещение сложных объектов

В предыдущем разделе мы видели, что паттерн Посетитель позволяет добавлять новые операции в существующую иерархию. В одном из примеров мы посетили сложный объект, который содержал указатели на другие объекты. Посетитель обошел эти указатели, но с ограничениями. Теперь мы рассмотрим общую задачу о посещении объектов, составленных из других объектов, и в конце представим демонстрацию работоспособного подхода к сериализации и десериализации.

Посещение составных объектов

Общая идея посещения составных объектов проста: посещая сам объект, мы обычно не знаем всех деталей обработки каждого из его компонентов. Но есть кое-кто, кто знает – посетитель для объекта-компонента специально написан для обработки его класса и ничего более. Следовательно, для правильной обработки объектов-компонентов нужно просто посетить каждый и таким образом делегировать проблему кому-то другому (это весьма действенная техника в программировании, и не только).

Сначала продемонстрируем эту идею на примере простого контейнерного класса, например `Shelter`, который может содержать любое число объектов-питомцев, ожидающих, когда их заберут из приюта:

```
class Shelter {
    public:
    void add(Pet* p) {
        pets_.emplace_back(p);
    }
    void accept(PetVisitor& v) {
        for (auto& p : pets_) {
            p->accept(v);
```

```
        }
    }
    private:
    std::vector<std::unique_ptr<Pet>> pets_;
};
```

Этот класс по существу является адаптером, который делает вектор питомцев пригодным для посещения (паттерн Адаптер мы подробно обсуждали в главе 17). Заметим, что объекты этого класса владеют хранящимися в нем объектами питомцев – когда объект `Shelter` уничтожается, вместе с ним уничтожаются все находящиеся в нем объекты. Любой контейнер уникальных указателей владеет содержащимися в нем объектами; именно так должны храниться в контейнере, например `std::vector`, полиморфные объекты (неполиморфные объекты можно хранить без обертки, но в нашем случае это не пройдет, потому что объекты, производные от `Pet`, принадлежат разным типам).

К нашей задаче, естественно, относится метод `Shelter::accept()`, который определяет, как происходит посещение объекта `Shelter`. Как видите, мы не вызываем посетителя для самого объекта `Shelter`, а делегируем посещение каждому содержащемуся в нем объекту. Поскольку наши посетители уже написаны для обработки объектов типа `Pet`, то больше ничего делать не надо. Если `Shelter` посещается, например, посетителем `FeedingVisitor`, то каждое животное в приюте будет накормлено, и специальный код для этого действия писать не надо.

Посещение составных объектов производится аналогично – если объект состоит из нескольких меньших объектов, то мы должны посетить каждый из них. Рассмотрим объект, представляющий семью с двумя питомцами, собакой и кошкой (люди, обслуживающие питомцев, не включены в показанный ниже код, но мы предполагаем, что они тоже существуют):

```
class Family {
    public:
    Family(const char* cat_color, const char* dog_color) :
        cat_(cat_color), dog_(dog_color)
    {}
    void accept(PetVisitor& v) {
        cat_.accept(v);
        dog_.accept(v);
    }
    private:   // другие члены семьи для краткости не показаны
    Cat cat_;
    Dog dog_;
};
```

И снова посещение семьи посетителем из иерархии `PetVisitor` делегируется таким образом, чтобы посетить все объекты, производные от `Pet`, а сами посетители уже имеют все необходимое для обработки этих объектов (разумеется, объект `Family` мог бы принимать и посетителей других видов, но для них нужно было бы написать отдельные методы `accept()`).

Вот теперь, наконец, у нас есть все, что нужно для решения задачи о сериализации и десериализации произвольных объектов. В следующем разделе мы покажем, как это сделать с помощью паттерна Посетитель.

Сериализация и десериализация с помощью Посетителя

Сама задача была подробно описана в предыдущем разделе – для сериализации каждый объект необходимо преобразовать в последовательность битов, которые нужно будет сохранить, скопировать или отправить. Первая часть работы зависит от объекта (каждый объект преобразуется по-разному), а вторая – от конкретного применения сериализации (сохранение на диск – не то же самое, что передача по сети). Реализация зависит от обоих факторов, отсюда и необходимость в двойной диспетчеризации, которую как раз и предлагает паттерн Посетитель. Кроме того, если мы уже знаем, как сериализовать некоторый объект, а затем десериализовать его (восстановить из последовательности битов), то можем применить этот метод и в случае, когда этот объект является частью другого.

Чтобы продемонстрировать сериализацию и десериализацию иерархии классов с помощью паттерна Посетитель, нам понадобится более сложная иерархия, чем те игрушечные примеры, с которыми мы имели дело до сих пор. Рассмотрим иерархию двумерных геометрических объектов:

```
class Geometry {
    public:
    virtual ~Geometry() {}
};

class Point : public Geometry {
    public:
    Point() = default;
    Point(double x, double y) : x_(x), y_(y) {}
    private:
    double x_;
    double y_;
};

class Circle : public Geometry {
    public:
    Circle() = default;
    Circle(Point c, double r) : c_(c), r_(r) {}
    private:
    Point c_;
    double r_;
};

class Line : public Geometry {
    public:
    Line() = default;
    Line(Point p1, Point p2) : p1_(p1), p2_(p2) {}
    private:
```

```
    Point p1_;
    Point p2_;
};
```

Все объекты наследуют абстрактному базовому классу `Geometry`, но более сложные объекты содержат один или несколько более простых. Например, прямая `Line` определяется двумя точками `Point`. Заметим, что в конечном итоге все объекты состоят из чисел типа `double`, поэтому результатом их сериализации является последовательность чисел. Штука в том, чтобы узнать, какое число `double` какому полю объекта соответствует, без этого мы не сможем правильно восстановить исходные объекты.

Для сериализации этих объектов с помощью паттерна Посетитель мы применим тот же процесс, что в предыдущем разделе. Сначала объявим базовый класс Посетителя:

```
class Visitor {
    public:
    virtual void visit(double& x) = 0;
    virtual void visit(Point& p) = 0;
    virtual void visit(Circle& c) = 0;
    virtual void visit(Line& l) = 0;
};
```

Тут есть одна дополнительная деталь – мы можем посещать также числа типа `double`, и каждый посетитель должен правильно обрабатывать их (записывать, читать и т. д.). Посещение любого геометрического объекта в конечном итоге сводится к посещению чисел, из которых он составлен.

Наш базовый класс `Geometry` и все производные от него должны принимать этого посетителя:

```
class Geometry {
    public:
    virtual ~Geometry() {}
    virtual void accept(Visitor& v) = 0;
};
```

Конечно, нет никакой возможности добавить функцию-член `accept()` в тип `double`, но нам это и не нужно. Функции-члены `accept()` в производных классах, каждый из которых состоит из одного или более чисел, будут посещать каждый член данных по порядку:

```
void Point::accept(Visitor& v) {
    v.visit(x_); // double
    v.visit(y_); // double
}
void Circle::accept(Visitor& v) {
    v.visit(c_); // Point
    v.visit(r_); // double
}
void Point::accept(Visitor& v) {
```

```
    v.visit(p1_); // Point
    v.visit(p2_); // Point
}
```

Конкретные классы Посетителей, производные от `Visitor`, отвечают за механизмы сериализации и десериализации. Порядок разложения объекта на составные части, вплоть до чисел, контролируется самим объектом, но что делать с этими числами, решают посетители. Например, мы можем сериализовать все объекты в строку, применяя форматный ввод-вывод (по аналогии с тем, что мы получаем, когда выводим числа в `cout`):

```
class StringSerializeVisitor : public Visitor {
    public:
    void visit(double& x) override { S << x << " "; }
    void visit(Point& p) override { p.accept(*this); }
    void visit(Circle& c) override { c.accept(*this); }
    void visit(Line& l) override { l.accept(*this); }
    std::string str() const { return S.str(); }
    private:
    std::stringstream S;
};
```

Строка накапливается в `stringstream`, пока не будут сериализованы все необходимые объекты:

```
Line l(.....);
Circle c(.....);
StringSerializeVisitor serializer;
serializer.visit(l);
serializer.visit(c);
std::string s(serializer.str());
```

После того как объекты выведены в строку `s`, мы можем восстановить их из строки, возможно, на другой машине (если организовали передачу туда строки). Сначала понадобится десериализующий Посетитель:

```
class StringDeserializeVisitor : public Visitor {
    public:
    StringDeserializeVisitor(const std::string& s) { S.str(s); }
    void visit(double& x) override { S >> x; }
    void visit(Point& p) override { p.accept(*this); }
    void visit(Circle& c) override { c.accept(*this); }
    void visit(Line& l) override { l.accept(*this); }
    private:
    std::stringstream S;
};
```

Этот Посетитель читает числа из строки и сохраняет их в переменных, которые передает ему посещаемый объект. Ключ к успешной десериализации – читать числа в том порядке, в каком они сохранялись, – например, если мы начинали с записи координат X и Y точки, то должны сконструировать точку из первых двух прочитанных чисел, интерпретируемых как координаты X и Y.

Если первая записанная точка была концом отрезка, то следует использовать только что сконструированную точку как конец нового отрезка. Красота паттерна Посетитель заключается в том, что функции, которые занимаются чтением и записью, не должны делать ничего специального для сохранения порядка – порядок определяется каждым объектом, и гарантируется, что он будет одинаков для всех посетителей (объект не делает различия между конкретными посетителями и даже не знает, кто его посещает). Нужно только посещать объекты в том порядке, в каком они были сериализованы:

```
Line l1;
Circle c1;
StringDeserializeVisitor deserializer(s); // строка, созданная сериализатором
deserializer.visit(l1);                   // восстановлена прямая l
deserializer.visit(c1);                   // восстановлена окружность c
```

До сих пор мы знали, какие объекты были сериализованы и в каком порядке. Поэтому и десериализовать их мы можем в том же порядке. Но в более общем случае неизвестно, каких объектов ожидать на этапе десериализации, – объекты сохраняются в допускающем посещение контейнере, аналогичном `Shelter` из примера выше, который должен гарантировать, что сериализация и десериализация производятся в одном и том же порядке. Например, рассмотрим следующий класс, который представляет пересечение двух геометрических объектов:

```
class Intersection : public Geometry {
    public:
    Intersection() = default;
    Intersection(Geometry* g1, Geometry* g2) : g1_(g1), g2_(g2) {}
    void accept(Visitor& v) override {
        g1_->accept(v);
        g2_->accept(v);
    }
    private:
    std::unique_ptr<Geometry> g1_;
    std::unique_ptr<Geometry> g2_;
};
```

Сериализовать этот объект тривиально – мы сериализуем оба составляющих его объекта по очереди, делегируя детали самим объектам. Мы не можем вызвать `v.visit()` напрямую, потому что не знаем типов `*g1_` и `*g2_`, но можем поручить объектам диспетчеризовать вызов, как они сочтут нужным. Однако в таком виде десериализация не пройдет, т. к. указатели на геометрические объекты равны `null` – память для объектов еще не выделена, и мы не знаем, под какие типы ее выделять. Необходимо сначала каким-то образом закодировать типы объектов в сериализованном потоке, а затем конструировать их в соответствии с хранящимися типами. Существует еще один паттерн, предлагающий стандартное решение этой проблемы, – Фабрика (при построении сложной системы применение нескольких паттернов – обычное дело).

Сделать это можно несколькими способами, но все они сводятся к преобразованию типов в числа и сериализации этих чисел. В нашем случае полный перечень типов геометрических объектов нужно знать при объявлении базового класса Visitor, так что мы можем заодно определить и перечисление этих типов:

```
class Geometry {
    public:
    enum type_tag { POINT = 100, CIRCLE, LINE, INTERSECTION };
    virtual type_tag tag() const = 0;
};

class Visitor {
    public:
    static Geometry* make_geometry(Geometry::type_tag tag);
    virtual void visit(Geometry::type_tag& tag) = 0;
    .....
};
```

Необязательно, чтобы перечисление type_tag было определено внутри класса Geometry или чтобы фабричный конструктор make_geometry был статической функцией-членом класса Visitor. Их можно объявить и вне любого класса, но виртуальный метод tag(), который возвращает правильный тег для любого производного геометрического типа, должен быть объявлен точно так, как показано. Функция tag() должна быть переопределена в каждом классе, производном от Geometry, например в классе Point:

```
class Point : public Geometry {
    public:
    .....
    type_tag tag() const override { return POINT; }
};
```

Другие производные классы нужно модифицировать аналогично.

Затем определим фабричный конструктор:

```
Geometry* Visitor::make_geometry(Geometry::type_tag tag) {
    switch (tag) {
        case Geometry::POINT: return new Point;
        case Geometry::CIRCLE: return new Circle;
        case Geometry::LINE: return new Line;
        case Geometry::INTERSECTION: return new Intersection;
    }
}
```

Эта фабричная функция конструирует тот или иной производный объект в зависимости от заданного тега типа. Объекту Intersection осталось только сериализовать и десериализовать теги двух пересекающихся геометрических объектов:

```
class Intersection : public Geometry {
    public:
```

```
    void accept(Visitor& v) override {
        Geometry::type_tag tag;
        if (g1_) tag = g1_->tag();
        v.visit(tag);
        if (!g1_) g1_.reset(Visitor::make_geometry(tag));
        g1_->accept(v);
        if (g2_) tag = g2_->tag();
        v.visit(tag);
        if (!g2_) g2_.reset(Visitor::make_geometry(tag));
        g2_->accept(v);
    }
    .....
};
```

Сначала посетителю передаются теги. Сериализующий посетитель должен записывать теги вместе с остальными данными:

```
class StringSerializeVisitor : public Visitor {
    public:
    void visit(Geometry::type_tag& tag) override {
        S << size_t(tag) << " ";
    }
    .....
};
```

Десериализующий посетитель должен прочитать тег (фактически он читает число типа `size_t` и преобразует его в тег):

```
class StringDeserializeVisitor : public Visitor {
    public:
    void visit(Geometry::type_tag& tag) override {
        size_t t;
        S >> t;
        tag = Geometry::type_tag(t);
    }
    .....
};
```

После того как тег восстановлен десериализующим посетителем, объект `Intersection` может вызвать фабричный конструктор, чтобы сконструировать нужный геометрический объект. После этого можно десериализовать объект из потока, и `Intersection` восстанавливается точно в таком виде, в каком был сериализован. Заметим, что существуют и другие способы упаковать посещение тегов и вызовы фабричного конструктора; оптимальное решение зависит от ролей различных объектов в системе – например, конструировать объекты на основе тегов может десериализующий посетитель, а не владеющий ими составной объект. Но последовательность событий остается такой же.

До сих пор мы изучали классический объектно-ориентированный паттерн Посетитель. Но прежде чем познакомиться с тем, во что превратил классический паттерн C++, нам нужно будет узнать еще об одном типе посетителя, устраняющем некоторые неудобства паттерна Посетитель.

Ациклический посетитель

Паттерн Посетитель, рассмотренный выше, делает все, что мы от него хотели. Он отделяет реализацию алгоритма от объекта, представляющего собой данные для этого алгоритма, и позволяет выбирать подходящую реализацию в зависимости от двух факторов, известных во время выполнения, – типа объекта и конкретной выполняемой операции, причем то и другое выбирается из соответствующих иерархий классов. Однако в этой бочке меда есть ложка дегтя – мы хотели уменьшить сложность и упростить сопровождение программы, и этой цели мы добились, но теперь мы вынуждены сопровождать две параллельные иерархии классов, посещаемые объекты и посетители, зависимости между которыми нетривиальны. И хуже всего, что они образуют цикл – объект Посетителя зависит от типов посещаемых объектов (имеются перегруженные методы `visit()` для каждого посещаемого типа), а базовый посещаемый тип зависит от базового типа Посетителя. Наибольшее зло несет в себе первая часть этой зависимости. Всякий раз как в иерархию добавляется новый объект, необходимо обновить каждого посетителя. Вторая часть не требует действий от программиста, поскольку новых посетителей можно добавлять в любой момент, не внося никаких других изменений, – в этом и состоит идея паттерна Посетитель. Но по-прежнему остается зависимость времени компиляции базового посещаемого класса и, стало быть, всех производных классов от базового класса Посетителя. Посетители по большей части стабильны в части интерфейса и реализации, за исключением одного случая – добавления нового посещаемого класса. Следовательно, на практике цикл выглядит следующим образом: новый класс добавляется в иерархию посещаемых объектов. Классы Посетителей необходимо обновить с учетом нового типа. Поскольку базовый класс Посетителя изменился, необходимо перекомпилировать базовый посещаемый класс и весь зависящий от него код, включая и код, в котором новый посещаемый класс вообще не используется, а используются только старые. Даже вставка опережающих объявлений всюду, где возможно, не помогает – если добавлен новый посещаемый класс, придется перекомпилировать все старые.

Еще одна проблема традиционного паттерна Посетитель состоит в том, что нужно обрабатывать все возможные комбинации типа объекта с типом посетителя. Часто некоторые комбинации не имеют смысла, а иные объекты никогда не посещаются посетителями определенных типов. Но нам это ничего не дает, потому что для каждой комбинации должно быть определено действие (оно может быть очень простым, но все равно в каждом классе Посетителя должен быть полный набор функций-членов `visit()`).

Паттерн Ациклический посетитель – это вариант паттерна Посетитель, специально разработанный, чтобы разорвать циклическую зависимость и разрешить частичное посещение. Базовый посещаемый класс в паттерне Ациклический посетитель такой же, как для обычного Посетителя:

```
class Pet {
    public:
    virtual ~Pet() {}
    virtual void accept(PetVisitor& v) = 0;
    .....
};
```

Однако на этом сходство и заканчивается. В базовом классе Посетителя нет перегруженных вариантов visit() для каждого посещаемого класса. Вообще никаких функций-членов visit() нет:

```
class PetVisitor {
    public:
    virtual ~PetVisitor() {}
};
```

Но кто же тогда осуществляет посещение? Для каждого производного класса в исходной иерархии мы также объявляем класс соответствующего Посетителя, и вот там-то и появляется функция visit():

```
class Cat;
class CatVisitor {
    public:
    virtual void visit(Cat* c) = 0;
};

class Cat : public Pet {
    public:
    Cat(const std::string& color) : Pet(color) {}
    void accept(PetVisitor& v) override {
        if (CatVisitor* cv = dynamic_cast<CatVisitor*>(&v))
            cv->visit(this);
        else { // обработать ошибку
            assert(false);
        }
    }
};
```

Заметим, что каждый посетитель может посещать только тот класс, который ему предназначен: CatVisitor посещает только объекты класса Cat, DogVisitor – только объекты класса Dog и т. д. Все волшебство кроется в новой функции accept() – когда класс просят принять посетителя, он первым делом с помощью dynamic_cast проверяет, того ли типа посетитель. Если да, то посетитель принимается, иначе имеет место ошибка, которую мы должны обработать (механизм обработки ошибок зависит от приложения, например можно возбудить исключение). Таким образом, конкретный класс Посетителя должен наследовать как общему базовому классу PetVisitor, так и специфическим базовым классам Посетителей (например, CatVisitor, DogVisitor), соответствующим питомцам, которых он должен посещать:

```
class FeedingVisitor : public PetVisitor, public CatVisitor, public DogVisitor {
    public:
```

```
    void visit(Cat* c) override {
        std::cout << "Покормить тунцом " << c->color() << " кошку"
                  << std::endl;
    }
    void visit(Dog* d) override {
        std::cout << "Покормить стейком " << d->color() << " собаку"
                  << std::endl;
    }
};
```

С другой стороны, если посетитель не предназначен для посещения некоторых классов иерархии, то мы просто опускаем соответствующий базовый класс и не должны переопределять его виртуальную функцию:

```
class BathingVisitor : public PetVisitor,
                       public DogVisitor { // но не CatVisitor
    public:
    void visit(Dog* d) override {
        std::cout << "Искупать " << d->color() << " собаку" << std::endl;
    }
    // Здесь нет visit(Cat*)!
};
```

Паттерн Ациклический посетитель вызывается точно так же, как обычный Посетитель:

```
std::unique_ptr<Pet> c(new Cat("orange"));
std::unique_ptr<Pet> d(new Dog("brown"));

FeedingVisitor fv;
c->accept(fv);
d->accept(fv);

BathingVisitor bv;
//c->accept(bv); // ошибка
d->accept(bv);
```

При попытке посетить объект, не поддерживаемый данным Посетителем, возникает ошибка. Таким образом, проблему частичного посещения мы решили. А как обстоит дело с циклическими зависимостями? Мы позаботились и об этом – в общем базовом классе `PetVisitor` не нужно перечислять всю иерархию посещаемых объектов, а конкретные посещаемые классы зависят только от ассоциированных с ними посетителей, но не от посетителей для других типов. Следовательно, когда в иерархию добавляется новый посещаемый класс, существующие не придется перекомпилировать.

Паттерн Ациклический посетитель выглядит настолько хорошо, что не понятно, почему бы не использовать его всегда, отказавшись от обычного Посетителя. Тому есть несколько причин. Прежде всего в Ациклическом посетителе применяется оператор `dynamic_cast` для приведения одного базового класса к другому (иногда это называют перекрестным приведением). В типичном случае эта операция гораздо дороже вызова виртуальной функции, поэтому

Ациклический посетитель медленнее обычного. Кроме того, в паттерне Ациклический посетитель необходим класс Посетителя для каждого посещаемого класса, т. е. количество классов удваивается. К тому же используется множественное наследование от большого количества базовых классов. Для большинства современных компиляторов вторая проблема не обременительна, но многие программисты с трудом воспринимают множественное наследование. Составляет ли проблему стоимость динамического приведения на этапе выполнения, зависит от приложения, но знать об этом необходимо. С другой стороны, паттерн Ациклический посетитель начинает по-настоящему сверкать в ситуациях, когда иерархия посещаемых объектов часто изменяется или когда стоимость перекомпиляции всего исходного кода высока.

Вероятно, вы обратили внимание еще на один недостаток Ациклического посетителя – в нем много трафаретного кода. Несколько строк кода приходится копировать для каждого посещаемого класса. На самом деле эта проблема присуща и обычному Посетителю, поскольку реализация каждого посетителя подразумевает ввод одного и того же кода. Но в C++ имеются специальные средства, позволяющие заменить повторный ввод кода его повторным использованием, именно для этого и предназначено обобщенное программирование. Далее мы рассмотрим, как паттерн Посетитель адаптировался к современному C++.

Посетители в современном C++

Как мы только что видели, паттерн Посетитель способствует разделению обязанностей; например, порядок сериализации и механизм сериализации независимы, за них отвечают разные классы. Этот паттерн также упрощает сопровождение кода благодаря тому, что весь код для решения данной задачи собран в одном месте. А вот чему паттерн Посетитель никак не способствует, так это повторному использованию кода без дублирования. Но это недостаток объектно-ориентированного Посетителя, до пришествия C++. Посмотрим, что можно сделать с помощью средств обобщенного программирования C++, и начнем с обычного паттерна Посетитель.

Обобщенный посетитель

Мы собираемся сократить объем трафаретного кода в реализации паттерна Посетитель. Начнем с функции-члена `accept()`, которую необходимо скопировать в каждый посещаемый класс. Выглядит она всегда одинаково:

```
class Cat : public Pet {
    void accept(PetVisitor& v) override { v.visit(this); }
};
```

Эту функцию нельзя перенести в базовый класс, потому что мы должны передавать посетителю фактический, а не базовый тип – `visit()` принимает `Cat*`, `Dog*` и т. д., но не `Pet*`. Но мы можем генерировать эту функцию по шаблону, если введем промежуточный шаблонный базовый класс:

```
class Pet { // то же, что и раньше
    public:
    virtual ~Pet() {}
    Pet(const std::string& color) : color_(color) {}
    const std::string& color() const { return color_; }
    virtual void accept(PetVisitor& v) = 0;
    private:
    std::string color_;
};

template <typename Derived>
class Visitable : public Pet {
    public:
    using Pet::Pet;
    void accept(PetVisitor& v) override {
        v.visit(static_cast<Derived*>(this));
    }
};
```

Шаблон параметризован производным классом. В этом отношении он похож на паттерн **Рекурсивный шаблон** (**CRTP**), только здесь мы не наследуем параметру шаблона, а используем его для приведения указателя this к типу указателя на правильный производный класс. Теперь нужно только унаследовать класс каждого питомца от правильной конкретизации шаблона, и мы получим функцию accept() автоматически:

```
class Cat : public Visitable<Cat> {
    using Visitable<Cat>::Visitable;
};

class Dog : public Visitable<Dog> {
    using Visitable<Dog>::Visitable;
};
```

Тем самым мы убрали половину трафаретного кода – внутри производных посещаемых классов. Осталась еще одна половина – та, что находится в классах Посетителей, где мы вынуждены снова и снова набирать одно и то же объявление для каждого посещаемого класса. С конкретными посетителями мы мало что можем сделать, ведь именно здесь и производится настоящая работа и, надо полагать, разная для разных посещаемых классов (иначе зачем вообще нужно было затеваться с двойной диспетчеризацией?).

Однако мы можем упростить объявление базового класса Посетителя, если введем такой обобщенный шаблон Посетителя:

```
template <typename ... Types>
class Visitor;

template <typename T>
class Visitor<T> {
    public:
    virtual void visit(T* t) = 0;
};
```

```
template <typename T, typename ... Types>
class Visitor<T, Types ...> : public Visitor<Types ...> {
    public:
    using Visitor<Types ...>::visit;
    virtual void visit(T* t) = 0;
};
```

Заметим, что этот шаблон нужно реализовать только один раз: не по разу для каждой иерархии классов, а раз и навсегда (по крайней мере, до тех пор, пока не понадобится изменить сигнатуру функции `visit()`, например добавить аргументы). Это хороший кандидат в библиотеку обобщенных классов. При его наличии объявление базового класса Посетителя для конкретной иерархии классов становится таким тривиальным делом, что даже как-то скучно:

```
using PetVisitor = Visitor<class Cat, class Dog>;
```

Обратите внимание на несколько необычное синтаксическое использование ключевого слова `class` – оно объединяет список аргументов шаблона с опережающим объявлением и эквивалентно такой последовательности предложений:

```
class Cat;
class Dog;
using PetVisitor = Visitor<Cat, Dog>;
```

Как работает базовый Обобщенный посетитель? В нем используется шаблон с переменным числом аргументов для запоминания произвольного количества аргументов-типов, но главный шаблон лишь объявлен, но не определен. Все остальное – специализации. Сначала идет частный случай одного аргумента-типа. Мы объявляем чисто виртуальную функцию-член `visit()` для этого типа. Затем идет специализация для случая, когда аргументов-типов больше одного, причем первый аргумент выделен явно, а остальные собраны в пакет параметров. Мы генерируем функцию `visit()` для явно заданного типа, а остальные наследуем от конкретизации того же шаблона, но с меньшим на единицу числом аргументов. Конкретизации порождаются рекурсивно, пока мы не дойдем до шаблона с одним аргументом-типом, а тогда используется первая специализация.

Теперь, когда у нас есть трафаретный код посетителя, сгенерированный по шаблонам, мы можем упростить и определение конкретных посетителей.

Лямбда-посетитель

Большая часть определения конкретного посетителя сводится к написанию кода обработки каждого посещаемого объекта. В классе конкретного посетителя не так много трафаретного кода. Но иногда не хочется объявлять сам класс. Вспомним о лямбда-выражениях – все, что можно сделать с помощью лямбда-выражения, можно также сделать и с помощью явно объявленного класса, допускающего вызов, поскольку лямбды и есть (анонимные) вызываемые классы. Тем не менее мы считаем лямбда-выражения очень полезными

для написания одноразовых вызываемых объектов. И точно так же хотелось бы написать посетитель, не именуя его явно, – лямбда-посетитель. Он мог бы выглядеть как-то так:

```
auto v(lambda_visitor<PetVisitor>(
    [](Cat* c) {
        std::cout << "Выпустить " << c->color() << " кошку"
                  << std::endl;
    },
    [](Dog* d) {
        std::cout << "Вывести " << d->color() << " собаку на прогулку"
                  << std::endl;
    }
));
pet->accept(v);
```

Тут надо решить две проблемы: как создать класс, который умеет обрабатывать список типов и соответствующих объектов (в нашем случае – посещаемые типы и соответствующие им лямбда-выражения), и как сгенерировать набор перегруженных функций с помощью лямбда-выражений.

Для решения первой проблемы мы должны будем рекурсивно конкретизировать шаблон, отщепляя по одному аргументу от пакета параметров. Решение второй проблемы аналогично множеству перегруженных вариантов лямбда-выражения, которое было описано в главе о шаблонах классов. Можно было бы воспользоваться множеством перегруженных вариантов из той главы, а можно применить рекурсивную конкретизацию шаблона, которая нам все равно нужна, для построения этого множества непосредственно.

В этой реализации нам предстоит столкнуться еще с одной проблемой – нужно обрабатывать не один, а два списка типов. В первом списке находятся все посещаемые типы: `Cat`, `Dog` и т. д., а во втором – типы лямбда-выражений, по одному для каждого посещаемого типа. Нам еще не встречались шаблоны с переменным числом аргументов, имеющие два пакета параметров, и не без причины – невозможно просто объявить `template<typename ... A, typename ... B>`, поскольку компилятор не мог бы узнать, где заканчивается первый список и начинается второй. Хитрость в том, чтобы скрыть один или оба списка типов внутри других шаблонов. В нашем случае уже имеется шаблон `Visitor`, конкретизированный списком посещаемых типов:

```
using PetVisitor = Visitor<class Cat, class Dog>;
```

Мы можем извлечь этот список из шаблона `Visitor` и сопоставить каждому типу его лямбда-выражение. Синтаксис частичной специализации, использованный для параллельной обработки двух пакетов параметров, заковыристый, поэтому разберем его по частям. Прежде всего мы должны объявить общий шаблон класса `LambdaVisitor`:

```
template <typename Base, typename ... >
    class LambdaVisitor;
```

Заметим, что здесь имеется всего один пакет параметров плюс базовый класс для посетителя (в нашем случае это будет PetVisitor). Этот шаблон необходимо объявить, но использовать мы его никогда не будем, а предоставим специализацию для каждого случая, нуждающегося в обработке. Первая специализация используется, когда есть только один посещаемый тип и одно соответственное лямбда-выражение:

```
template <typename Base, typename T1, typename F1>
class LambdaVisitor<Base, Visitor<T1>, F1> : private F1, public Base {
    public:
    LambdaVisitor(F1&& f1) : F1(std::move(f1)) {}
    LambdaVisitor(const F1& f1) : F1(f1) {}
    void visit(T1* t) override { return F1::operator()(t); }
};
```

Эта специализация применяется не только для обработки случая одного посещаемого типа, но и как последняя конкретизация в любой цепочке рекурсивных конкретизаций шаблона. Поскольку она всегда является первым базовым классом в рекурсивной иерархии конкретизаций LambdaVisitor, она единственная прямо наследует базовому классу Посетителя, например PetVisitor. Заметим, что даже в случае единственного посещаемого типа T1 мы используем шаблон Visitor как обертку для него. Это делается в преддверии общего случая, где у нас будет список типов неизвестной длины. Оба конструктора сохраняют лямбда-выражение f1 в классе LambdaVisitor, используя перемещение вместо копирования. Наконец, переопределенная виртуальная функция visit(T1*) просто переадресует вызов лямбда-выражению. На первый взгляд, может показаться проще открыто унаследовать F1 и согласиться на использование функционального синтаксиса вызова (иными словами, переименовать все обращения к visit() в обращения к operator()). Но это работать не будет; нам необходима косвенность, потому что оператор operator() лямбда-выражения сам не может быть переопределенной виртуальной функцией. Кстати, ключевое слово override может оказать бесценную помощь при отладке ошибок в коде, где шаблон наследуется не от того базового класса или объявления виртуальных функций не точно совпадают.

Общему случаю произвольного количества посещаемых типов и лямбда-выражений соответствует следующая частичная специализация, которая явно обрабатывает первые типы в обоих списках, а затем рекурсивно конкретизирует себя хвостами списков:

```
template <typename Base,
          typename T1, typename ... T,
          typename F1, typename ... F>
class LambdaVisitor<Base, Visitor<T1, T ...>, F1, F ...> :
    private F1, public LambdaVisitor<Base, Visitor<T ...>, F ...>
{
    public:
    LambdaVisitor(F1&& f1, F&& ... f) :
```

```
        F1(std::move(f1)),
        LambdaVisitor<Base, Visitor<T ...>, F ...>(std::forward<F>(f) ...)
    {}
    LambdaVisitor(const F1& f1, F&& ... f) :
        F1(f1),
        LambdaVisitor<Base, Visitor<T ...>, F ...>(std::forward<F>(f) ...)
    {}
    void visit(T1* t) override { return F1::operator()(t); }
};
```

И снова мы имеем два конструктора, которые сохраняют первое лямбда-выражение в классе и переадресуют остаток следующей конкретизации. На каждом шаге рекурсии генерируется одна переопределенная виртуальная функция, всегда для первого типа в оставшемся списке посещаемых классов. Затем этот тип удаляется из списка, и так продолжается до тех пор, пока мы не дойдем до последней конкретизации – для одного посещаемого типа.

Поскольку невозможно явно поименовать типы лямбда-выражений, мы также не можем явно объявить тип лямбда-посетителя. Вместо этого типы лямбда-выражений должны быть выведены из аргументов шаблона, поэтому нам необходима шаблонная функция `lambda_visitor()`, которая принимает несколько лямбда-выражений в качестве аргументов и конструирует из них объект `LambdaVisitor`:

```
template <typename Base, typename ... F>
auto lambda_visitor(F&& ... f) {
    return LambdaVisitor<Base, Base, F ...>(std::forward<F>(f) ...);
}
```

Теперь, когда у нас имеется класс, который сохраняет произвольное количество лямбда-выражений и связывает с каждым соответствующую переопределенную функцию `visit()`, мы можем писать лямбда-посетителей так же просто, как пишем лямбда-выражения:

```
void walk(Pet& p) {
    auto v(lambda_visitor<PetVisitor>(
        [](Cat* c) {
            std::cout << "Выпустить " << c->color() << " кошку"
                      << std::endl; },
        [](Dog* d) {
            std::cout << "Вывести " << d->color() << " собаку на прогулку"
                      << std::endl; }
));
p.accept(v);
```

Заметим, что поскольку мы объявляем функцию `visit()` в том же классе, который наследует соответствующему лямбда-выражению, порядок лямбда-выражений в списке аргументов функции `lambda_visitor()` должен совпадать с порядком классов в списке типов в определении `PetVisitor`. Это ограничение при желании можно снять ценой еще большего усложнения реализации.

Теперь мы видели, как общие фрагменты кода посетителя можно превратить в повторно используемые шаблоны и как это, в свою очередь, позволя-

ет создать лямбда-посетителя. Но не будем забывать о другой рассмотренной в этой главе реализации, паттерне Ациклический посетитель. Посмотрим, как в ней могут пригодиться языковые возможности современного C++.

Обобщенный Ациклический посетитель

Паттерн Ациклический посетитель не нуждается в базовом классе, содержащем список всех посещаемых типов. Однако в нем есть свой трафаретный код. Во-первых, в каждом посещаемом типе должна присутствовать функция-член `accept()`, которая содержит больше кода, чем аналогичная функция в обычном паттерне Посетитель:

```
class Cat : public Pet {
    public:
    void accept(PetVisitor& v) override {
        if (CatVisitor* cv = dynamic_cast<CatVisitor*>(&v))
            cv->visit(this);
        else {   // обработать ошибку
            assert(false);
      }
    }
};
```

В предположении, что все ошибки обрабатываются единообразно, эта функция повторяется в разных типах посетителей, соответствующих посещаемым типам (в данном случае – `CatVisitor`). Затем имеются сами классы Посетителей, по одному на каждый тип, например:

```
class CatVisitor {
    public:
    virtual void visit(Cat* c) = 0;
};
```

Этот код также копируется из одного места программы в другое с минимальными модификациями. Давайте преобразуем это чреватое ошибками дублирование кода в повторно используемый код, удобный для сопровождения.

Как и раньше, сначала необходимо создать инфраструктуру. В паттерне Ациклический посетитель иерархия основана на общем базовом классе для всех посетителей:

```
class PetVisitor {
    public:
    virtual ~PetVisitor() {}
};
```

Заметим, что ничего специфического для иерархии `Pet` здесь нет. Если выбрать более подходящее имя, то этот класс может служить базовым для любой иерархии посетителей:

```
class VisitorBase {
    public:
    virtual ~VisitorBase() {}
};
```

Нам также необходим шаблон для генерации всех базовых классов посетителей, ассоциированных с посещаемыми типами, который позволил бы заменить почти идентичные классы `CatVisitor`, `DogVisitor` и т. д. Поскольку от этих классов требуется только одно – объявление чисто виртуального метода `visit()`, мы можем параметризовать шаблон посещаемым типом:

```
template <typename Visitable>
class Visitor {
    public:
    virtual void visit(Visitable* p) = 0;
};
```

Базовый посещаемый класс для любой иерархии классов теперь принимает посетителей по ссылке на общий базовый класс `VisitorBase`:

```
class Pet {
    .....
    virtual void accept(VisitorBase& v) = 0;
};
```

Вместо того чтобы наследовать каждый посещаемый класс напрямую от `Pet` и вставлять копию метода `accept()`, мы введем промежуточный шаблонный класс, который умеет генерировать этот метод с правильными типами:

```
template <typename Visitable>
class PetVisitable : public Pet {
    public:
    using Pet::Pet;
    void accept(VisitorBase& v) override {
        if (Visitor<Visitable>* pv = dynamic_cast<Visitor<Visitable>*>(&v))
            pv->visit(static_cast<Visitable*>(this));
        else {   // обработать ошибку
            assert(false);
        }
    }
};
```

Это единственная копия функции `accept()`, которую нам надо написать, она содержит предпочтительную реализацию обработки ошибки из-за того, что посетитель не принимается базовым классом (напомним, что Ациклический посетитель допускает частичное посещение, при котором некоторые комбинации посетителя и посещаемого не поддерживаются).

Конкретные посещаемые классы наследуют общему базовому классу `Pet` не напрямую, а через промежуточный класс `PetVisitable`, который наделяет их интерфейсом посещаемого объекта. Аргументом шаблона `PetVisitable` является сам производный класс (вот еще один пример CRTP в действии):

```
class Cat : public PetVisitable<Cat> {
    using PetVisitable<Cat>::PetVisitable;
};
class Dog : public PetVisitable<Dog> {
    using PetVisitable<Dog>::PetVisitable;
};
```

Конечно, необязательно использовать одни и те же конструкторы базового класса во всех производных классах, при необходимости в каждом классе можно определить свои конструкторы.

Осталось только реализовать класс Посетителя. Напомним, что любой конкретный посетитель в паттерне Ациклический посетитель наследует общему базовому классу посетителя и всем классам посетителей, ассоциированным с поддерживаемыми посещаемыми типами. В этом отношении ничего не изменится, но теперь у нас есть способ генерировать эти классы посетителей по запросу:

```
class FeedingVisitor :
    public VisitorBase, public Visitor<Cat>, public Visitor<Dog>
{
  public:
    void visit(Cat* c) override {
        std::cout << "Покормить тунцом " << c->color() << " кошку"
                  << std::endl;
    }
    void visit(Dog* d) override {
        std::cout << "Покормить стейком " << d->color() << " собаку"
                  << std::endl;
    }
};
```

Оглянемся на проделанную работу – параллельную иерархию классов посетителей больше не нужно создавать явно, вместо этого типы генерируются по мере необходимости. Повторяющиеся функции `accept()` свелись к одному шаблону класса `PetVisitable`. Но все-таки мы должны писать этот шаблон для каждой новой иерархии посещаемых классов. Можно пойти дальше по пути обобщения и создать шаблон, допускающий повторное использование для всех иерархий, который будет параметризован базовым посещаемым классом:

```
template <typename Base, typename Visitable>
class VisitableBase : public Base {
    public:
    using Base::Base;
    void accept(VisitorBase& vb) override {
        if (Visitor<Visitable>* v = dynamic_cast<Visitor<Visitable>*>(&vb))
            v->visit(static_cast<Visitable*>(this));
        else {   // обработать ошибку
            assert(false);
        }
    }
};
```

Теперь для каждой иерархии посещаемых классов мы должны только создать псевдоним шаблона:

```
template <typename Visitable>
using PetVisitable = VisitableBase<Pet, Visitable>;
```

Мы можем сделать еще одно упрощение и разрешить программисту задавать список посещаемых классов в виде списка типов вместо наследования классам `Visitor<Cat>`, `Visitor<Dog>` и т. д., как было раньше. Для хранения списка типов потребуется шаблон с переменным числом аргументов. Реализация похожа на показанную выше реализацию `LambdaVisitor`:

```
template <typename ... V> struct Visitors;

template <typename V1>
struct Visitors<V1> : public Visitor<V1>
{};

template <typename V1, typename ... V>
struct Visitors<V1, V ...> : public Visitor<V1>, public Visitors<V ...>
{};
```

Этот шаблон-обертку можно использовать, чтобы сократить объявления конкретных посетителей:

```
class FeedingVisitor :
public VisitorBase, public Visitors<Cat, Dog>
{
    .....
};
```

При желании мы можем даже скрыть `VisitorBase` в определении специализации шаблона `Visitors` для одного аргумента-типа.

Итак, мы видели как классический объектно-ориентированный паттерн Посетитель, так и его повторно используемые реализации, ставшие возможными благодаря средствам обобщенного программирования в C++. В предыдущих главах мы также видели, как некоторые паттерны применяются целиком на этапе компиляции. Посмотрим теперь, нельзя ли сделать то же самое с паттерном Посетитель.

Посетитель времени компиляции

В этом разделе мы проанализируем возможность использования паттерна Посетитель на этапе компиляции по аналогии с паттерном Стратегия, для которого этот путь привел к проектированию на основе политик.

Прежде всего аспект паттерна Посетитель, связанный со множественной диспетчеризацией, в шаблонном контексте становится тривиальным:

```
template <typename T1, typename T2> auto f(T1 t1, T2 t2);
```

Для каждой комбинации типов `T1` и `T2` шаблонная функция может выполнять разные алгоритмы. В отличие от полиморфизма времени выполнения,

реализованного с помощью виртуальных функций, диспетчеризация вызова по-разному в зависимости от двух и более типов вообще ничего не стоит (если, конечно, не считать затрат на написание кода для всех подлежащих обработке комбинаций). Воспользовавшись этим наблюдением, мы легко можем имитировать классический паттерн Посетитель на этапе выполнения:

```
class Pet {
    std::string color_;
    public:
    Pet(const std::string& color) : color_(color) {}
    const std::string& color() const { return color_; }
    template <typename Visitable, typename Visitor>
        static void accept(Visitable& p, Visitor& v) { v.visit(p); }
};
```

Теперь функция `accept()` является шаблоном и статической функцией-членом – фактический тип первого аргумента, посещаемого объекта, производного от класса `Pet`, будет выведен во время компиляции. Конкретные посещаемые классы наследуют базовому обычным образом:

```
class Cat : public Pet {
    public:
    using Pet::Pet;
};

class Dog : public Pet {
    public:
    using Pet::Pet;
};
```

Посетители не обязаны наследовать общему базовому классу, поскольку теперь типы разрешаются во время компиляции:

```
class FeedingVisitor {
    public:
    void visit(Cat& c) {
        std::cout << "Покормить тунцом " << c.color() << " кошку"
                  << std::endl;
    }
    void visit(Dog& d) {
        std::cout << "Покормить стейком " << d.color() << " собаку"
                  << std::endl;
    }
};
```

Посещаемые классы могут принять любого посетителя, имеющего правильный интерфейс, т. е. перегруженные варианты функции `visit()` для всех классов иерархии:

```
Cat c("orange");
Dog d("brown");
FeedingVisitor fv;
Pet::accept(c, fv);
Pet::accept(d, fv);
```

Разумеется, любая функция, которая принимает посетителей в качестве аргументов и должна поддерживать нескольких посетителей, тоже обязана быть шаблонной (больше недостаточно иметь общий базовый класс, поскольку это позволяет определить фактический тип только во время выполнения).

Посетитель времени компиляции решает ту же задачу, что и классический посетитель, т. е. по существу позволяет добавлять новые функции-члены в класс, не изменяя само определение класса. Однако выглядит он далеко не так интригующе, как вариант времени выполнения.

Более интересные возможности открываются, если объединить паттерны Посетитель и Композиция. Один раз мы это уже делали, когда обсуждали посещение сложных объектов, особенно в контексте проблемы сериализации. Особый интерес связан с тем, как эта комбинация связана с одним из немногих *существенных* средств, отсутствующих в C++, – отражением. В программировании отражением (reflection) называется способность программы исследовать собственный исходный код и порождать новое поведение в зависимости от результатов такой *интроспекции*. В некоторых языках, например в Delphi и Python, механизм отражения встроен, в C++ – нет. Отражение полезно при решении многих задач. Например, проблему сериализации было бы легко решить, если бы мы могли заставить компилятор обойти все данные-члены объекта; тогда мы рекурсивно сериализовывали бы их, пока не дошли бы до встроенных типов. Нечто подобное можно реализовать с помощью паттерна Посетитель времени компиляции.

Мы снова будем рассматривать иерархию геометрических объектов. Поскольку теперь все происходит на этапе компиляции, нас не интересует полиморфная природа классов (в них могли бы присутствовать виртуальные функции для каких-то операций во время выполнения, просто в этом разделе мы таких писать не будем). Вот, например, как выглядит класс `Point`:

```
class Point {
    public:
    Point() = default;
    Point(double x, double y) : x_(x), y_(y) {}
    template <typename This, typename Visitor>
    static void accept(This& t, Visitor& v) {
        v.visit(t.x_);
        v.visit(t.y_);
    }
    private:
    double x_;
    double y_;
};
```

Посещение, как и раньше, обеспечивает функция `accept()`, но теперь она зависит от класса. Единственная причина для включения первого параметра шаблона, `This`, – поддержать константные и неконстантные операции: `This` может иметь тип `Point` или `const Point`. Любому посетителю этого класса предлагается посетить оба значения, определяющих точку: `x_` и `y_`. Этот посетитель

должен иметь подходящий интерфейс, а именно функцию-член visit(), принимающую аргумент типа double. Как и в большинстве библиотек шаблонов на C++, включая **стандартную библиотеку шаблонов** (**STL**), этот код скрепляется соглашениями – здесь нет ни виртуальных функций, которые можно было бы переопределить, ни базовых классов, которым можно было бы унаследовать, лишь требования к интерфейсу каждого класса, участвующего в системе. Сложные классы составлены из более простых, как, например, класс Line:

```
class Line {
    public:
    Line() = default;
    Line(Point p1, Point p2) : p1_(p1), p2_(p2) {}
    template <typename This, typename Visitor>
    static void accept(This& t, Visitor& v) {
        v.visit(t.p1_);
        v.visit(t.p2_);
    }
    private:
    Point p1_;
    Point p2_;
};
```

Класс прямой Line составлен из двух точек. На этапе компиляции посетителю предлагается посетить каждую точку. На этом участие класса Line заканчивается; класс Point сам решает, что сделать при его посещении (как мы видели выше, он также делегирует работу другому посетителю). Поскольку мы больше не пользуемся полиморфизмом времени выполнения, контейнерные классы, способные содержать геометрические объекты разных типов, теперь обязаны быть шаблонами:

```
template <typename G1, typename G2>
class Intersection {
    public:
    Intersection() = default;
    Intersection(G1 g1, G2 g2) : g1_(g1), g2_(g2) {}
    template <typename This, typename Visitor>
    static void accept(This& t, Visitor& v) {
        v.visit(t.g1_);
        v.visit(t.g2_);
    }
    private:
    G1 g1_;
    G2 g2_;
};
```

Теперь у нас есть посещаемые типы. С этим интерфейсом можно использовать и другие виды посетителей, не только сериализующие. Но нас сейчас интересует сериализация. Ранее мы видели посетителя, который преобразует объекты в ASCII-строки. А теперь сериализуем объекты в виде двоичных данных, т. е. непрерывного потока бит. Сериализующий посетитель имеет доступ

к буферу определенного размера и записывает объекты в этот буфер, по одному числу типа double за раз:

```
class BinarySerializeVisitor {
    public:
    BinarySerializeVisitor(char* buffer, size_t size) :
        buf_(buffer), size_(size)
    {}
    void visit(double x) {
        if (size_ < sizeof(x))
            throw std::runtime_error("Buffer overflow");
        memcpy(buf_, &x, sizeof(x));
        buf_ += sizeof(x);
        size_ -= sizeof(x);
    }
    template <typename T> void visit(const T& t) { T::accept(t, *this); }
    private:
    char* buf_;
    size_t size_;
};
```

Десериализующий посетитель читает из буфера и копирует в данные-члены восстанавливаемого объекта:

```
class BinaryDeserializeVisitor {
    public:
    BinaryDeserializeVisitor(const char* buffer, size_t size) :
        buf_(buffer), size_(size)
    {}
    void visit(double& x) {
        if (size_ < sizeof(x))
            throw std::runtime_error("Buffer overflow");
        memcpy(&x, buf_, sizeof(x));
        buf_ += sizeof(x);
        size_ -= sizeof(x);
    }
    template <typename T> void visit(T& t) { T::accept(t, *this); }
    private:
    const char* buf_;
    size_t size_;
};
```

Оба посетителя обрабатывают встроенные типы непосредственно, копируя их в буфер и из буфера, а более сложным типам предоставляют самим решать, как обрабатывать объекты. В обоих случаях посетитель возбуждает исключение, если превышен размер буфера. Теперь мы можем использовать посетителей, например, для того, чтобы передать объекты через сокет на другую машину:

```
// На машине-отправителе:
Line l = .....;
Circle c = .....;
```

```
Intersection<Circle, Circle> x = .....;
char buffer[1024];
BinarySerializeVisitor serializer(buffer, sizeof(buffer));
serializer.visit(l);
serializer.visit(c);
serializer.visit(x);

..... отправить буфер получателю.....

// На машине-получателе:
Line l;
Circle c;
Intersection<Circle, Circle> x;
BinaryDeserializeVisitor deserializer(buffer, sizeof(buffer));
deserializer.visit(l);
deserializer.visit(c);
deserializer.visit(x);
```

Хотя реализовать универсальный механизм отражения без поддержки со стороны языка невозможно, мы можем заставить классы в какой-то, ограниченной, степени отражать свое содержимое – например, как в этом паттерне посещения составных объектов. Можно рассмотреть и другие вариации на ту же тему.

Для начала примем соглашение – делать вызываемыми объекты, имеющие только одну *важную* функцию-член; иными словами, вместо того чтобы вызывать функцию-член, мы вызываем сам объект, применяя синтаксис вызова функции. В силу этого соглашения функцию-член `visit()` следует назвать `operator()`:

```
class BinarySerializeVisitor {
    public:
    void operator()(double x);
    template <typename T> void operator()(const T& t);
    .....
};
```

Теперь посещаемые классы вызывают посетителей как функции:

```
class Point {
    public:
    static void accept(This& t, Visitor& v) {
        v(t.x_);
        v(t.y_);
    }
.....
};
```

Иногда удобно также реализовать функции-обертки, которые вызывают посетителей сразу для нескольких объектов:

```
SomeVisitor v;
Object1 x; Object2 y; .....
visitation(v, x, y, z);
```

Также нетрудно реализовать шаблон с переменным числом аргументов:

```
template <typename V, typename T>
void visitation(V& v, T& t) {
    v(t);
}
template <typename V, typename T, typename ... U>
void visitation(V& v, T& t, U& ... u) {
    v(t);
    visitation(v, u ...);
}
```

В общем случае посетителей времени компиляции реализовать проще, потому что не нужно ничего изобретать для получения множественной диспетчеризации, т. к. шаблоны это уже умеют. Нужно только найти интересные применения этого паттерна, как, например, рассмотренная нами задача о сериализации и десериализации.

Резюме

Мы узнали о паттерне Посетитель и о различных способах его реализации в C++. Классический объектно-ориентированный паттерн Посетитель позволяет добавить новую виртуальную функцию в целую иерархию классов, не изменяя исходный код самих классов. Иерархия должна быть сделана посещаемой, но после того как это один раз сделано, можно добавить сколько угодно операций, и их реализация будет отделена от самих объектов. В классической реализации паттерна Посетитель исходный код классов посещаемой иерархии не нужно изменять, но его необходимо перекомпилировать при добавлении нового класса в иерархию. Паттерн Ациклический посетитель решает эту проблему, но ценой дополнительного динамического приведения. С другой стороны, Ациклический посетитель поддерживает еще и частичное посещение – позволяет игнорировать некоторые комбинации посетитель–посещаемый, тогда как классический Посетитель требует, чтобы все комбинации были по крайней мере объявлены.

Для всех вариантов посетителя платой за расширяемость является ослабление инкапсуляции и зачастую предоставление внешним классам посетителей доступа к данным-членам, которые должны были бы быть закрытыми.

Паттерн Посетитель часто сочетают с другими паттернами проектирования, в частности с паттерном Композиция, для создания сложных объектов, допускающих посещение. Составной объект делегирует посещение своим компонентам. Этот комбинированный паттерн особенно полезен, когда объект необходимо разложить на мельчайшие структурные компоненты, например для сериализации.

Классический паттерн Посетитель реализует двойную диспетчеризацию на этапе выполнения – в процессе работы программа выбирает, какой код выполнять, в зависимости от двух факторов: типа посетителя и типа посещаемо-

го объекта. Аналогичный паттерн можно использовать на этапе компиляции, и тогда она дает ограниченную возможность отражения.

Глава о паттерне Посетитель завершает эту книгу, посвященную идиомам и паттернам проектирования в C++. Но, как и рождение звезд, появление новых паттернов никогда не прекращается – вслед за расширением границ и новыми идеями появляются и новые задачи, изобретаются методы их решения, они эволюционируют и обогащаются, пока сообщество программистов не достигнет точки, в которой можно с уверенностью сказать: «*Обычно это является хорошим способом решения данной проблемы*». Мы развиваем сильные стороны нового подхода, осмысливаем его недостатки, придумываем название, чтобы можно было лаконично сослаться на новое знание о проблеме, ее решениях и подводных камнях. После этого новый паттерн становится частью нашего инструментария и входит в лексикон программистов.

Вопросы

- Что такое паттерн Посетитель?
- Какую проблему решает паттерн Посетитель?
- Что такое двойная диспетчеризация?
- Каковы преимущества паттерна Ациклический посетитель?
- Как паттерн Посетитель помогает реализовать сериализацию?

Ответы на вопросы

Глава 1

- В чем важность объектов в C++?
  Объекты и классы – строительные блоки программы на C++. Объединяя данные и алгоритм (*код*) в единое целое, программа на C++ представляет компоненты моделируемой системы, а также их взаимодействия.
- Какое отношение выражает открытое наследование?
  Открытое наследование представляет отношение *является* между объектами – объект производного класса можно использовать так, будто он является объектом базового класса. Из этого отношения вытекает, что интерфейс базового класса со всеми его инвариантами и ограничениями также является интерфейсом производного класса.
- Какое отношение выражает закрытое наследование?
  В отличие от открытого наследования, закрытое наследование ничего не говорит об интерфейсах. Оно выражает отношение *содержит* или *реализован в терминах*. Производный класс повторно использует реализацию, предоставленную базовым классом. Обычно того же результата можно достичь с помощью композиции. Если возможно, следует остановиться на композиции, однако оптимизация пустого базового класса и (реже) переопределение виртуальных методов – веские основания для использования закрытого наследования.
- Что такое полиморфный объект?
  Полиморфным объектом в C++ называется объект, поведение которого зависит от его типа, а тип неизвестен на этапе компиляции (по крайней мере, в точке, где опрашивается интересующее поведение). Объект, к которому обращаются как к объекту базового класса, может демонстрировать поведение производного класса, если таков его истинный тип. В C++ полиморфное поведение реализуется с помощью виртуальных функций.

Глава 2

- В чем разница между типом и шаблоном?
  Шаблон не является типом, это *фабрика* по изготовлению различных типов с похожей структурой. Шаблон пишется в терминах обобщенных типов; подстановка конкретных типов вместо обобщенных дает порождаемый по шаблону тип.
- Какие виды шаблонов имеются в C++?
  Существуют шаблоны классов, функций и переменных. Каждый вид шаблона генерирует соответствующие сущности – функции по шабло-

ну функции, классы (типы) по шаблону класса, переменные по шаблону переменной.

- Какие виды параметров могут быть у шаблонов C++?
  У шаблонов могут быть параметры-типы и параметры-нетипы. Параметрами-нетипами могут быть целые числа или перечислимые значения, а также шаблоны (в случае шаблонов с переменным числом аргументов подстановочные маркеры также являются параметрами-нетипами).
- В чем разница между специализацией и конкретизацией шаблона?
  Конкретизация шаблона – это код, сгенерированный по шаблону. Обычно конкретизация производится неявно, в точке использования шаблона. Возможна также явная конкретизация, без использования; при этом генерируется тип или функция для последующего использования. В случае явной специализации шаблона задаются все обобщенные типы; это не конкретизация, и никакой код не генерируется, пока шаблон не будет использован. Это лишь альтернативный рецепт генерации кода для этих и только этих типов.
- Как осуществляется доступ к пакету параметров шаблона с переменным числом аргументов?
  Обычно пакет параметров обходится с помощью рекурсии. Как правило, компилятор встраивает код, сгенерированный в процессе рекурсии, поэтому рекурсия существует только во время компиляции (а также в голове программиста, читающего код). В C++17 (и редко в C++14) можно оперировать всем пакетом без рекурсии.
- Для чего применяются лямбда-выражения?
  Лямбда-выражения – это по существу компактный способ объявления локальных классов, которые можно вызывать как функции. Они используются, чтобы сохранить фрагмент кода в переменной (а точнее, ассоциировать код с переменной), так чтобы этот код можно было вызвать впоследствии.

Глава 3

- Почему так важно четко выражать, кто владеет памятью в программе?
  Ясное выражение того, кто владеет памятью и вообще любым ресурсом, – один из ключевых признаков хорошего проекта. Если владелец четко определен, то гарантируется, что ресурс создан и доступен к моменту, когда в нем возникает необходимость, существует на протяжении всего времени использования и освобожден или очищен, когда необходимость в нем отпадает.
- Каковы типичные проблемы, возникающие из-за нечеткого указания владельца памяти?
  Наиболее типичные проблемы – утечка ресурсов, в т. ч. утечка памяти; висячие описатели (например, указатели, ссылки или итераторы, указы-

вающие на ресурсы, которые уже не существуют); многократные попытки освободить один и тот же ресурс; многократные попытки сконструировать один и тот же ресурс.

- Какие виды владения памятью можно выразить на C++?
  Невладение, монопольное владение, совместное владение, а также преобразование между различными типами владения и передачу владения.
- Как писать функции и классы, не владеющие памятью?
  Безразличные к владению функции и классы должны обращаться к объектам по простым указателям и ссылкам.
- Почему монопольное владение памятью предпочтительнее совместного?
  Монопольное владение памятью проще понять и проследить по потоку управления в программе. Оно также более эффективно.
- Как выразить монопольное владение памятью в C++?
  Предпочтительно путем создания объекта в стеке или как члена данных владеющего класса (в т. ч. и контейнерного). Если необходима семантика ссылки или перемещения, то следует использовать уникальный указатель.
- Как выразить совместное владение памятью в C++?
  Для выражения совместного владения следует использовать разделяемый указатель, например `std::shared_ptr`.
- Каковы потенциальные недостатки совместного владения памятью?
  В большой системе совместным владением трудно управлять, из-за него ресурсы могут освобождаться с задержкой без всякой на то необходимости. Кроме того, по сравнению с монопольным владением, у совместного владения нетривиальные накладные расходы. Для потокобезопасного управления совместным владением в конкурентной программе реализация должна быть написана очень аккуратно.

Глава 4

- Что делает операция обмена?
  Обменивает состояния двух объектов. После нее объекты должны остаться неизменными, за исключением имен, по которым к ним обращаются.
- Как обмен используется в программах, безопасных относительно исключений?
  Обмен обычно используется в программах, предоставляющих семантику фиксации или отката; сначала создается временная копия результата, а затем, если не было ошибок, она обменивается с окончательным результатом.
- Почему функция `swap` не должна возбуждать исключений?
  Использование обмена для предоставления семантики фиксации или отката подразумевает, что сама операция обмена не может возбуждать

исключений или как-то иначе завершаться аномально, оставив обмениваемые объекты в неопределенном состоянии.

- Какую реализацию `swap` следует предпочесть: в виде функции-члена или свободной функции?

  Свободную функцию `swap` следует предоставлять всегда и гарантировать, что обращения к ней выполняются корректно. Функцию-член также можно предоставить по двум причинам: во-первых, это единственный способ обменять объект с временным объектом, а во-вторых, для реализации обмена обычно нужен доступ к закрытым данным-членам класса. Если предоставлено то и другое, то свободная функция должна вызывать функцию-член от имени одного из двух своих параметров.

- Как обмен реализован в классах из стандартной библиотеки?

  Все контейнеры STL и некоторые другие классы из стандартной библиотеки представляют функцию-член `swap()`. Кроме того, свободная шаблонная функция `std::swap()` имеет перегруженные варианты для всех типов из STL.

- Почему свободную функцию `swap` следует вызывать без квалификатора `std::`?

  Квалификатор `std::` отключает механизм поиска, зависящего от аргументов, и заставляет компилятор вызвать конкретизацию шаблона `std::swap` по умолчанию, даже если в классе реализована собственная функция `swap`. Чтобы избежать этой проблемы, рекомендуется также предоставлять явную специализацию шаблона `std::swap`.

Глава 5

- Что понимается под *ресурсами*, которыми может управлять программа?

  Память – наиболее распространенный ресурс, но вообще ресурсом может быть любой объект. Любая виртуальная или физическая сущность, которой оперирует программа, является ресурсом.

- Каковы основные проблемы управления ресурсами в программе на C++?

  Ресурсы не должны теряться (утекать). Если для доступа к ресурсу используется описатель, например указатель или идентификатор, то описатель не должен оставаться висячим (ссылаться на уже не существующий ресурс). Ресурсы следует освобождать, когда в них отпадает необходимость, причем способом, соответствующим их захвату.

- Что такое RAII?

  Идиома «захват ресурса есть инициализация» (RAII) – основной подход к управлению ресурсами в C++. Она означает, что ресурсом владеет некоторый объект, причем захват ресурса производится в конструкторе, а освобождение – в деструкторе этого объекта.

- Как RAII решает проблему утечки ресурсов?

  RAII-объект всегда должен создаваться в стеке или как член данных другого объекта. Когда программа покидает область видимости, охватываю-

щую RAII-объект или содержащий его объект, вызывается деструктор RAII-объекта. Это происходит вне зависимости от того, как именно программа покидает область видимости.

- Как RAII решает проблему висячих описателей ресурсов?
  Если каждым ресурсом владеет RAII-объект и RAII-объект не раскрывает простые описатели (или пользователь ведет себя осторожно и не клонирует простые описатели), то описатель можно получить только от RAII-объекта, и ресурс не освобождается, пока этот объект существует.
- Какие RAII-объекты предоставляет стандартная библиотека C++?
  Чаще всего используется `std::unique_ptr` для управления памятью; объект `std::lock_guard` предназначен для управления мьютексами.
- О каких предосторожностях следует помнить при написании RAII-объектов?
  Как правило, RAII-объекты не должны допускать копирование. Перемещение RAII-объекта передает владение ресурсом; классический паттерн RAII этого не поддерживает, поэтому обычно RAII-объекты следует делать неперемещаемыми (различайте `std::unique_ptr` и `const std::unique_ptr`).
- Что происходит, когда освобождение ресурса завершается неудачно?
  RAII испытывает трудности с обработкой ошибок освобождения, потому что исключения не могут распространяться наружу из деструкторов, а значит, нет хорошего способа сообщить об ошибке вызывающей стороне. Поэтому неудачное освобождение ресурса часто приводит к неопределенному поведению (иногда так поступает и стандарт C++).

Глава 6

- Что собой представляет стирание типа?
  Стирание типа – это техника программирования, при которой программа не показывает явной зависимости от некоторых используемых в ней типов.
- Как стирание типа реализуется в C++?
  Реализация всегда подразумевает наличие полиморфного объекта и вызов виртуальной функции или динамического приведения. Обычно это сочетается с обобщенным программированием для конструирования таких полиморфных объектов.
- В чем разница между сокрытием типа за ключевым словом `auto` и его стиранием?
  Программа может быть написана так, что большинство типов вообще не упоминается. Типы выводятся шаблонными функциями и объявляются как `auto` или `typedef`'ы, выведенные из шаблона. Однако фактические типы объектов, скрытые за словом `auto`, все равно зависят от всех типов, которыми оперирует объект (например, типа ликвидатора `deleter` для указателя). Стертый тип вообще не запоминается в типе объекта. Иными словами,

если бы можно было узнать у компилятора, что стоит за конкретным словом `auto`, то все типы вылезли бы наружу. Но если тип стерт, то даже самое детальное объявление объемлющего объекта не раскрыло бы тип (например, `std::shared_ptr<int>` – это весь тип, и типа ликвидатора в нем нет).

- Как материализуется конкретный тип, когда у программы возникает в нем необходимость?

  Одним из двух способов – программист может указать конкретный тип, основываясь на априорных знаниях (например, контексте или значении некоторой переменной во время выполнения), или тип можно использовать полиморфно через единый интерфейс.

- Каковы издержки стирания типа?

  Всегда существует дополнительное косвенное обращение и ассоциированный с ним указатель. Почти во всех реализациях применяется полиморфизм времени выполнения (виртуальные функции или динамическое приведение), что приводит к увеличению как времени работы (косвенные вызовы функций), так и потребления памяти (виртуальные указатели). Наибольшие издержки обычно связаны с дополнительными операциями выделения памяти, необходимой для конструирования полиморфных объектов, размер которых неизвестен на этапе компиляции. Если такие операции удается минимизировать и включить дополнительную память в сам объект, то увеличение времени работы может быть совсем невелико (но при этом объем дополнительной памяти сохраняется и часто даже увеличивается).

Глава 7

- Что такое множество перегруженных вариантов?

  Для каждого вызова функции это множество всех функций с данным именем, доступных в точке вызова (на доступность могут влиять пространства имен, вложенные области видимости и т. д.).

- Что такое разрешение перегрузки?

  Это процесс выбора той функции из множества перегруженных вариантов, которую следует вызвать при известном количестве и типах аргументов.

- Что такое выведение типов и подстановка типов?

  Для шаблонных функций и функций-членов (а также конструкторов в C++17) механизм выведения типов определяет типы параметров шаблона по типам аргументов функции. Иногда тип параметра можно вывести из нескольких аргументов. В таком случае результаты выведения должны совпадать, иначе процесс выведения завершается ошибкой.

  После того как типы параметров шаблона выведены, для каждого аргумента, типа возвращаемого значения и аргументов по умолчанию подставляются конкретные типы. Это называется подстановкой типов.

- Что такое SFINAE?

  Описанная выше подстановка типов может приводить к некорректным типам, например к появлению указателя на функцию-член для типа, не имеющего функций-членов. Такие неудачные подстановки не считаются ошибками компиляции, просто соответствующий вариант функции удаляется из множества перегруженных вариантов.

- В каких контекстах потенциально недопустимый код не приводит к ошибке компиляции, если только он не понадобится в действительности?

  Только в объявлении функций (возвращаемого типа, типов параметров и значений по умолчанию). Неудавшаяся подстановка в теле функции, выбранной в результате разрешения перегрузки, всегда считается ошибкой компиляции.

- Как можно определить, какой перегруженный вариант был выбран, не вызывая его?

  Если все перегруженные варианты возвращают разные типы, то эти типы можно исследовать во время компиляции. Необходим какой-то способ различить типы, например разный размер или разные значения включенных в них констант.

- Как SFINAE применяется для управления условной компиляцией?

  Аккуратно и осторожно. Намеренно вызывая ошибки подстановки, мы можем направить процесс разрешения перегрузки в нужном нам направлении. В общем случае выбирается желательный перегруженный вариант, если только он не приводит к неудаче перегрузки, а иначе остается только вариант с переменным числом аргументов, что свидетельствует о недопустимости выражения, которое мы хотели проверить. Различая перегруженные варианты по типу возвращаемого значения, мы можем генерировать константы времени компиляции (`constexpr`), которые можно использовать для условной компиляции.

Глава 8

- Насколько дорого обходится вызов виртуальной функции и почему?

  В абсолютных единицах измерения недорого (не более нескольких наносекунд), но все равно вызов виртуальной функции в несколько раз дороже вызова невиртуальной и на порядок или более дороже вызова встраиваемой функции. Накладные расходы связаны с косвенностью: виртуальная функция всегда вызывается через указатель на функцию, поэтому какая именно функция будет вызвана, компилятору неизвестно, и встроить ее вызов он не может.

- Почему у вызова аналогичной функции, разрешаемого во время компиляции, нет таких накладных расходов?

  Если компилятор знает, какую именно функцию предстоит вызвать, он может исключить косвенность в процессе оптимизации и встроить функцию.

- Как реализовать вызовы полиморфных функций на этапе компиляции?
  Как полиморфные вызовы во время выполнения производятся через указатель на базовый класс, так и статические полиморфные вызовы можно выполнить через указатель или ссылку на базовый класс. В случае CRTP и статического полиморфизма базовый тип – это на самом деле целый набор типов, генерируемых по шаблону базового класса, по одному для каждого производного класса. Чтобы сделать полиморфный вызов, мы должны использовать шаблон функции, который можно конкретизировать любым из этих базовых типов.
- Как использовать CRTP для расширения интерфейса базового класса?
  Когда производный класс вызывается напрямую, использование CRTP принципиально отличается от эквивалента виртуальных функций на этапе компиляции. Это становится техникой реализации, при которой общая функциональность предоставляется нескольким производным классам, и каждый расширяет и настраивает интерфейс шаблона базового класса.

Глава 9

- Почему наличие функций с большим количеством аргументов одного или родственных типов приводит к хрупкости кода?
  Легко ошибиться при подсчете аргументов, изменить не тот аргумент или использовать аргумент неверного типа, который по чистой случайности допускает преобразование в тип параметра. Кроме того, при добавлении нового параметра необходимо изменить сигнатуры всех функций, которые должны передавать этот параметр дальше.
- Как агрегатные объекты в качестве аргументов повышают удобство сопровождения и надежность кода?
  У значений аргументов внутри агрегата имеются явные имена. При добавлении нового значения не требуется изменять сигнатуры функций. Классы, созданные для передачи разных групп аргументов, имеют разные типы, их нельзя случайно перепутать.
- Что такое идиома именованных аргументов и чем она отличается от агрегатного объекта-аргумента?
  Идиома именованных аргументов разрешает использовать временные агрегатные объекты. Вместо того чтобы изменять члены данных по имени, мы пишем метод для установки значения каждого аргумента. Все такие методы возвращают ссылку на сам объект и могут сцепляться в одном предложении.
- В чем разница между сцеплением и каскадированием методов?
  Каскадирование методов – это техника применения нескольких методов к одному и тому же объекту. В случае сцепления методов каждый метод возвращает, вообще говоря, новый объект, и следующий метод применяется уже к нему. Часто сцепление применяется как способ каскадирова-

ния методов, в этом случае все сцепленные методы возвращают ссылку на исходный объект.

Глава 10

- Как измерить производительность небольшого фрагмента кода?

  Эталонные микротесты позволяют измерить производительность небольших фрагментов кода, выполняемых изолированно. Для измерения производительности того же фрагмента в контексте программы необходим профилировщик.

- Почему частое выделение небольших блоков памяти особенно вредит производительности?

  Обработка небольших объемов данных обычно требует столь же небольшого объема вычислений и потому производится очень быстро. Выделение памяти добавляет постоянные накладные расходы, не пропорциональные объему данных. Относительный эффект тем больше, чем короче время обработки. Кроме того, при выделении памяти иногда требуется глобальная блокировка или иной механизм сериализации нескольких потоков.

- Что такое оптимизация локального буфера и как она работает?

  Оптимизация локального буфера заменяет выделение внешней памяти буфером, который является частью самого объекта. Это устраняет затраты на выделение дополнительной памяти.

- Почему выделение памяти для дополнительного буфера внутри объекта обходится по существу *бесплатно*?

  Объект должен быть сконструирован, и память для него необходимо выделить независимо от того, производится выделение дополнительной памяти или нет. У этого выделения есть цена – больше, если память выделяется в куче, меньше, если в стеке, – но эту цену так или иначе придется уплатить до того, как объект можно будет использовать. Оптимизация локального буфера увеличивает размер объекта, а значит, объем выделенной для него области, но обычно это не сильно влияет на стоимость выделения.

- Что такое оптимизация короткой строки?

  Оптимизация короткой строки сводится к хранению символов строки в локальном буфере внутри объекта строки, если длина строки не превышает некоторый порог.

- Что такое оптимизация короткого вектора?

  Оптимизация короткого вектора подразумевает хранение нескольких элементов вектора в локальном буфере внутри объекта вектора.

- Почему оптимизация локального буфера особенно эффективна для вызываемых объектов?

  Вызываемые объекты обычно малы, поэтому требуют небольшого локального буфера. Кроме того, вызов внешнего вызываемого объекта тре-

бует дополнительного косвенного обращения, что часто сопровождается промахом кеша.

- Какие компромиссы необходимо рассмотреть при использовании оптимизации локального буфера?

  Оптимизация локального буфера увеличивает размер каждого объекта класса с локальным буфером независимо от того, используется буфер или нет. Если в большинстве случаев размер требуемой памяти превышает размер локального буфера, то память расходуется впустую. Увеличение размера локального буфера позволяет хранить в нем больше данных, но одновременно увеличивает общее потребление памяти.

- Когда объект не следует помещать в локальный буфер?

  Локальные буферы и находящиеся в них данные необходимо копировать или перемещать всякий раз, как содержащий буфер объект копируется или перемещается. С другой стороны, внешние данные, доступные по указателю, никогда не нужно перемещать и, возможно, не нужно и копировать, если владеющий ими объект поддерживает совместное владение (например, с помощью механизма подсчета ссылок). Копирование или перемещение данных делает недействительными все направленные на них указатели или итераторы и может возбуждать исключения. Это ограничивает гарантии, которые класс может дать касательно копирования или перемещения. Если требуются более строгие гарантии, то оптимизация локального буфера может оказаться невозможной или будет иметь ограничения.

Глава 11

- Что такое программа, безопасная относительно ошибок или исключений?

  Безопасная относительно ошибок программа сохраняет корректное состояние (набор инвариантов), даже если возникает ошибка. Безопасность относительно исключений – частный случай безопасности относительно ошибок, когда программа получает уведомление об ошибке в виде исключения. Программа не должна переходить в неопределенное состояние, если возбуждается (разрешенное) исключение. В безопасной относительно исключений программе некоторые операции не должны возбуждать исключений.

- Как можно сделать безопасной относительно ошибок процедуру, выполняющую несколько взаимосвязанных действий?

  Если на протяжении нескольких действий, каждое из которых может оказаться неудачным, требуется поддерживать согласованное состояние, то предыдущие действия должны быть отменены, когда последующее завершается ошибкой. Часто для этого необходимо, чтобы действие не фиксировалось окончательно, пока не будет успешно достигнут конец транзакции. Операция окончательной фиксации не должна приводить

к ошибкам (например, возбуждать исключение), в противном случае гарантировать безопасность невозможно. Операция отката также не должна приводить к ошибкам.

- Как идиома RAII помогает писать программы, безопасные относительно ошибок?

  RAII-классы гарантируют, что некоторое действие обязательно выполняется, когда программа покидает область видимости, например выходит из функции. Действие при выходе из области видимости нельзя ни пропустить, ни обойти, даже если функция выходит не через закрывающую скобку, а выполняя досрочный возврат или возбуждая исключение.

- Как паттерн ScopeGuard обобщает идиому RAII?

  Классическая идиома RAII нуждается в специальном классе для каждого действия. Паттерн ScopeGuard автоматически генерирует RAII-класс по произвольному фрагменту кода (по крайней мере, если поддерживаются лямбда-выражения).

- Как программа может автоматически определить, когда функция завершилась успешно, а когда – неудачно?

  Если информация о состоянии возвращается с помощью кодов ошибок, то не может. Если о любой ошибке в программе сигнализирует исключение, а нормальный возврат из функции означает, что она завершилась успешно, то мы можем во время выполнения определить, было ли возбуждено исключение. Затруднение состоит в том, что сама охраняемая операция может случиться во время раскрутки стека, вызванной другим исключением. Это исключение распространяется, когда класс-охранник должен решить, была операция успешной или неудачной, но его присутствие не означает ошибку в охраняемой операции (оно может указывать на то, что ошибка произошла где-то в другом месте). Надежный механизм обнаружения исключений должен отслеживать, сколько исключений распространялось в начале охраняемой области видимости, а сколько – в конце, но это возможно только в C++17 (или с помощью расширений компилятора).

- Каковы достоинства и недостатки паттерна ScopeGuard со стертым типом?

  Классы, устроенные в соответствии с паттерном ScopeGuard, обычно являются конкретизациями шаблона. Это означает, что фактический тип класса неизвестен программисту или, по крайней мере, его трудно задать явно. Для управления сложностью ScopeGuard опирается на продление времени жизни и выведение аргументов шаблона. ScopeGuard со стертым типом – это конкретный тип, он не зависит от кода, который хранит. Недостаток состоит в том, что для стирания типа требуется полиморфизм времени выполнения и в большинстве случаев выделение памяти.

Глава 12

- Каков эффект объявления функции *другом*?
  Свободная дружественная функция имеет такой же доступ к членам класса, как функция-член.
- В чем разница между предоставлением дружественного доступа функции и шаблону функции?
  Предоставление дружественного доступа шаблону делает другом любую конкретизацию шаблона, включая конкретизации другими, никак не связанными типами.
- Почему бинарные операторы обычно реализуются как свободные функции?
  Бинарные операторы, реализованные как функции-члены, всегда вызываются от имени левого операнда, к которому не применяются никакие преобразования типа. Но преобразования применяются к правому операнду в соответствии с его типом. Это создает асимметрию между такими выражениями, как `x+2` и `2+x`, – второе невозможно обработать функцией-членом, потому что в типе операнда `2` (`int`) таковой нет.
- Почему оператор вывода в поток всегда реализуется в виде свободной функции?
  Первым операндом оператора вывода в поток всегда является поток, а не печатаемый объект. Поэтому оператор должен быть функцией-членом класса этого потока, который, однако, является частью стандартной библиотеки и не может быть расширен пользователем.
- В чем основная разница между преобразованиями аргументов шаблонных и нешаблонных функций?
  Детали сложны, но основная разница заключается в том, что при вызове нешаблонных функций рассматриваются определенные пользователем преобразования типов (неявные конструкторы и операторы преобразования), тогда как при вызове шаблонной функции типы аргументов должны (почти) точно совпадать с типами параметров, и никакие пользовательские преобразования не допускаются.
- Как сделать так, чтобы при конкретизации шаблона всегда генерировалась также уникальная нешаблонная свободная функция?
  Определение дружественной функции по месту (когда определение сразу следует за объявлением) в шаблоне класса приводит к тому, что любая конкретизация этого шаблона генерирует в объемлющей области видимости одну нешаблонную свободную функцию с заданным именем и типами параметров.

Глава 13

- Почему в C++ не разрешены виртуальные конструкторы?
  Причин несколько, но самая простая в том, что память должна выделяться блоком размера `sizeof(T)`, где `T` – фактический тип объекта, а оператор `sizeof()` – это `constexpr` (константа времени компиляции).

- Что такое паттерн Фабрика?
  Это порождающий паттерн, который решает проблему создания объектов без явного задания типа объекта.
- Как паттерн Фабрика используется для создания эффекта виртуального конструктора?
  Хотя в C++ в точке конструирования необходимо указать фактический тип, паттерн Фабрика позволяет отделить точку конструирования от места, где программа должна решить, какой объект конструировать, и указать тип с помощью какого-то альтернативного идентификатора – числа или значения другого типа.
- Как добиться эффекта виртуального копирующего конструктора?
  Виртуальный копирующий конструктор – это частный случай Фабрики, которая конструирует объект с типом, определяемым типом другого, уже имеющегося объекта. Типичная реализация сводится к виртуальному методу `clone()`, который переопределяется в каждом производном классе.
- Как паттерны Шаблонный метод и Фабрика используются совместно?
  Паттерн Шаблонный метод описывает дизайн, в котором общий поток управления определяется базовым классом, а производные классы настраивают реализации в предопределенных точках. В нашем случае общий поток управления – это поток фабричного конструирования, а точка настройки – акт конструирования объекта (выделение памяти и вызов конструктора).

Глава 14

- Что такое поведенческий паттерн проектирования?
  Поведенческий паттерн проектирования описывает способ решения типичной задачи посредством определенной организации взаимодействия различных объектов.
- Что такое паттерн Шаблонный метод?
  Паттерн Шаблонный метод – это стандартный способ реализовать алгоритм, который имеет жесткую *структуру*, т. е. общий поток управления, но допускает одну или несколько точек настройки для решения конкретных задач.
- Почему Шаблонный метод считается поведенческим паттерном?
  Шаблонный метод позволяет подклассам (производным типам) по-своему реализовать отдельные аспекты поведения общего в остальных отношениях алгоритма. Ключ к этому паттерну – способ взаимодействия базового и производных типов.
- Что такое инверсия управления и каким образом она применима к Шаблонному методу?
  В более распространенном иерархическом подходе к проектированию низкоуровневый код предоставляет *строительные блоки*, из которых

высокоуровневый код составляет конкретный алгоритм, объединяя их в определенном порядке – потоке управления. В паттерне Шаблонный метод высокоуровневый код не определяет общий алгоритм и не контролирует общий поток. Низкоуровневый код управляет алгоритмом и определяет, когда вызвать высокоуровневый код, чтобы тот настроил определенные аспекты поведения.

- Что такое невиртуальный интерфейс?

  Это паттерн, в котором открытый интерфейс иерархии классов реализован невиртуальными открытыми методами базового класса, а производные классы содержат только виртуальные закрытые методы (а также необходимые данные и вспомогательные невиртуальные методы).

- Почему в C++ рекомендуется делать все виртуальные функции закрытыми?

  Открытая виртуальная функция выполняет две разные задачи: предоставляет интерфейс (поскольку она открытая) и модифицирует реализацию. Для лучшего разделения обязанностей правильнее использовать виртуальные функции только для настройки реализации, а общий интерфейс описывать с помощью невиртуальных функций базового класса.

- Когда следует делать виртуальные функции защищенными?

  Если принята идиома невиртуального интерфейса, то виртуальные функции обычно делаются закрытыми. Исключение составляет случай, когда производному классу нужно вызвать виртуальную функцию базового класса, чтобы делегировать ей часть реализации. Тогда эту функцию следует сделать защищенной.

- Почему Шаблонный метод нельзя использовать для деструкторов?

  Деструкторы вызываются в порядке *вложенности*, начиная с «самого производного» класса. После того как деструктор производного класса закончил работу, он вызывает деструктор своего базового класса. К этому моменту *дополнительная* информация, содержавшаяся в производном классе, уже уничтожена, а осталась только часть, принадлежащая базовому классу. Если бы деструктор базового класса вызвал виртуальную функцию, то была бы вызвана функция базового класса (т. к. производного класса уже нет). Никаким способом деструктор базового класса не может вызвать виртуальные функции производного класса.

- Что такое проблема хрупкого базового класса и как избежать ее при использовании Шаблонного метода?

  Проблема хрупкого базового класса проявляется, когда изменение базового класса неожиданно нарушает работу производного. Это не уникальная особенность паттерна Шаблонный метод, потенциально она может затронуть любой объектно-ориентированный проект. В простейшем случае изменение невиртуальной открытой функции базового класса таким образом, что изменяются имена виртуальных функций, вызываемых для

настройки поведения алгоритма, приводит к неработоспособности всех существующих производных классов, поскольку модификации, реализованные в них с помощью виртуальных функций со старыми именами, внезапно перестали работать. Чтобы избежать этой проблемы, не следует изменять существующие точки настройки.

Глава 15

- Что такое паттерн Одиночка?

  Паттерн Одиночка гарантирует единственность объекта; в программе может существовать только один экземпляр данного класса.

- Когда можно использовать паттерн Одиночка, а когда его следует избегать?

  В плохо спроектированной программе Одиночка иногда используется как замена глобальной переменной. Чтобы оправдать его использование, нужны причины, по которым объект должен быть единственным. Эти причины могут отражать природу реальности, моделируемой программой (один двигатель в автомобиле, одно солнце в Солнечной системе), или искусственно наложенное проектное ограничение (один центральный источник памяти для всей программы). В любом случае программист должен подумать, насколько вероятно, что в будущем требования изменятся и понадобится несколько экземпляров, и сопоставить это с объемом работы по сопровождению более сложного кода, в котором несколько экземпляров поддерживается еще до того, как в них возникла необходимость.

- Что такое ленивая инициализация и какие проблемы она решает?

  Лениво инициализируемый объект конструируется при первом использовании. Ленивую инициализацию можно отложить надолго, и, возможно, она вообще не потребуется, если в конкретном прогоне программы некоторый объект так и не понадобился. Противоположностью является энергичная инициализация, которая происходит в предопределенном порядке независимо от того, нужен объект или нет.

- Как сделать инициализацию Одиночки потокобезопасной?

  В версии C++11 и более поздних это очень просто: гарантируется, что инициализация локальной статической переменной потокобезопасна.

- В чем состоит проблема порядка удаления и какие есть способы ее решения?

  Даже после решения проблемы порядка инициализации может остаться проблема порядка удаления – статический объект может быть удален, хотя еще существует другой объект, ссылающийся на него по указателю или ссылке. У этой проблемы нет общего решения. Явная очистка предпочтительна, но если это сопряжено с трудностями, то утекание ресурсов, связанных со статическими объектами, может быть сочтено меньшим из зол.

Глава 16

- Что такое паттерн Стратегия?

  Стратегия – это поведенческий паттерн, который позволяет пользователю настроить некоторый аспект поведения класса путем выбора алгоритма, реализующего этот аспект, из набора предоставленных альтернатив или путем предоставления новой реализации.

- Как паттерн Стратегия реализуется в C++ на этапе компиляции с помощью обобщенного программирования?

  Традиционный объектно-ориентированный паттерн Стратегия применяется на этапе выполнения, но в C++ сочетание обобщенного программирования со Стратегией вылилось в так называемое проектирование на основе политик. При таком подходе главный шаблон класса делегирует некоторые аспекты своего поведения типам политик, определенным пользователем.

- Какие типы можно использовать в качестве политик?

  В общем случае на тип политики не налагается почти никаких ограничений, хотя есть соглашения, ограничивающие конкретные способы объявления и использования политик. Например, если политика вызывается как функция, то можно использовать любую вызываемую сущность. С другой стороны, если вызывается определенная функция-член политики, то политика обязана быть классом и предоставлять такую функцию-член. Шаблонные политики тоже можно использовать, но количество параметров шаблона должно точно совпадать с объявленным.

- Как можно интегрировать политики с главным шаблоном?

  Основные два способа – композиция и наследование. В общем случае лучше предпочесть композицию, однако на практике политики часто являются пустыми классами баз данных-членов, так что к ним применима оптимизация пустого базового класса. Закрытое наследование следует предпочесть, если только политика не должна модифицировать также открытый интерфейс главного класса. Политики, которые должны оперировать самим главным классом, часто вынуждены использовать паттерн CRTP. В остальных случаях, когда объект политики не зависит от типов, использованных при конструировании главного шаблона, поведение политики можно раскрывать с помощью статической функции-члена.

- Каковы основные недостатки проектирования на основе политик?

  Главным недостатком является сложность в различных проявлениях. Основанные на политиках типы с разными политиками, вообще говоря, представляют собой разные типы (единственная альтернатива, стирание типа, обычно сопровождается неприемлемыми накладными расходами во время выполнения). Поэтому может возникнуть необходимость преобразовать значительные участки кода в шаблоны. Длинные списки по-

литик трудно сопровождать и правильно использовать. Поэтому следует избегать создания ненужных или плохо обоснованных политик. Иногда тип с двумя слабо связанными наборами политики лучше разбить на два отдельных типа.

Глава 17

- Что такое паттерн Адаптер?

  Адаптер – очень общий паттерн, который модифицирует интерфейс класса или функции (или шаблона в C++), так чтобы ее можно было использовать в контексте, подразумевающем другой интерфейс, но схожее поведение.

- Что такое паттерн Декоратор и чем он отличается от паттерна Адаптер?

  Декоратор – более узкий паттерн, он модифицирует интерфейс, добавляя или удаляя некоторое поведение, но не преобразует интерфейс в совершенно непохожий.

- Классическая объектно-ориентированная реализация паттерна Декоратор обычно не рекомендуется в C++. Почему?

  В классической объектно-ориентированной реализации декорированный класс и класс Декоратора наследуют общему базовому классу. У такого решения два ограничения; самое важное состоит в том, что декорированный объект сохраняет полиморфное поведение декорированного класса, однако не может сохранить интерфейс, который был добавлен в конкретный (производный) декорированный класс, но не присутствовал в базовом. Второе ограничение заключается в том, что Декоратор специфичен для конкретной иерархии. Оба ограничения можно снять с помощью инструментов обобщенного программирования в C++.

- Когда в декораторе класса в C++ следует использовать наследование, а когда композицию?

  В общем случае Декоратор сохраняет максимально возможную часть интерфейса декорированного класса. Те функции, поведение которых не изменено, остаются в прежнем виде. Поэтому обычно применяется открытое наследование. Если Декоратор должен явно переадресовывать большинство вызовов декорированному классу, то преимущества наследования не так важны и можно использовать композицию или закрытое наследование.

- Когда в адаптере класса в C++ следует использовать наследование, а когда композицию?

  В противоположность декораторам, адаптеры обычно предоставляют интерфейс, сильно отличающийся от интерфейса исходного класса. В этом случае предпочтительнее композиция. Исключение составляют адаптеры времени компиляции, которые модифицируют параметры шаблона, но в остальном остаются по существу тем же шаблоном класса

(как псевдонимы шаблонов). Для таких адаптеров необходимо использовать открытое наследование.

- C++ предлагает общий адаптер функции для каррирования аргументов, `std::bind`. Каковы его ограничения?

  Главное ограничение – невозможность применить к шаблонным функциям. Кроме того, он не позволяет заменить аргументы функции выражениями, содержащими эти аргументы.

- C++11 предлагает псевдонимы шаблонов, которые можно использовать как адаптеры. Каковы их ограничения?

  Псевдонимы шаблонов не рассматриваются при выведении типов аргументов во время конкретизации шаблонов функций.

- Оба паттерна, Адаптер и Политика, можно использовать для расширения или модификации открытого интерфейса класса. Приведите несколько причин, по которым один паттерн следует предпочесть другому.

  Адаптеры легко составлять (компоновать) для построения сложного интерфейса. Те средства, которые не активируются, вообще не нуждаются в специальном внимании; если соответствующий адаптер не используется, то и средство не включено. В традиционном паттерне Политика для каждой политики должна быть зарезервирована позиция. За исключением аргументов по умолчанию, следующих за последним заданным, все политики, даже имеющие значения по умолчанию, должны быть явно указаны. С другой стороны, адаптеры в середине стека не имеют доступа к окончательному типу объекта, что ограничивает интерфейс. Класс на основе политик – всегда окончательный тип, и с помощью CRTP этот тип можно распространить до политик, которые в нем нуждаются.

Глава 18

- Что такое паттерн Посетитель?

  Паттерн Посетитель предоставляет способ отделить реализацию алгоритма от объектов, с которыми он работает. Иными словами, он позволяет добавлять операции в классы, не модифицируя их посредством написания новых функций-членов.

- Какую проблему решает паттерн Посетитель?

  Паттерн Посетитель позволяет расширять функциональность иерархий классов. Его можно использовать, когда исходный код класса недоступен для модификации или когда такие модификации трудно сопровождать.

- Что такое двойная диспетчеризация?

  Двойная диспетчеризация – это процесс диспетчеризации вызова функции (выбора подлежащего выполнению алгоритма) в зависимости от двух факторов. Двойную диспетчеризацию можно реализовать на этапе выполнения с помощью паттерна Посетитель (виртуальные функции

обеспечивают одиночную диспетчеризацию) или на этапе компиляции с помощью шаблонов либо посетителей времени компиляции.

- Каковы преимущества паттерна Ациклический посетитель?
  В классическом Посетителе имеется циклическая зависимость между иерархией классов посетителей и иерархией посещаемых классов. Хотя посещаемые классы не изменяются при добавлении нового посетителя, их все равно приходится перекомпилировать после каждого изменения иерархии посетителей. А это неизбежно при каждом добавлении нового посещаемого класса, отсюда и циклическая зависимость. Паттерн Ациклический посетитель разрывает цикл благодаря перекрестному приведению и множественному наследованию.
- Как паттерн Посетитель помогает реализовать сериализацию?
  Естественный способ принять посетителя объектом, составленным из меньших объектов, – посетить все компоненты по очереди. Если реализовать эту схему рекурсивно, то мы закончим посещением каждого входящего в объект члена данных встроенного типа, причем в однозначно определенном порядке. Но эта схема идеально соответствует требованиям сериализации и десериализации, когда мы должны разложить объект в совокупность встроенных типов, а затем восстановить его по этой совокупности.

Предметный указатель

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Федор Г. Пикус

Идиомы и паттерны проектирования в современном C++

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 36,73. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**