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.