Оглавление

Об авторе

Артур О'Двайр (Arthur O'Dwyer) использует современный язык C++ в своей повседневной работе около десяти лет – еще с тех пор, когда под «современным C++» подразумевался «классический C++». С 2006 по 2011 год он принимал участие в работе над компилятором Green Hills C++ Compiler. Начиная с 2014 г. организовывал еженедельные встречи пользователей C++ в Заливе Сан-Франциско и регулярно выступает, освещая темы, которые можно найти в этой книге. В 2018 году он во второй раз присутствовал на заседании комитета ISO C++.

Это его первая книга.

О научном редакторе

Уилл Бреннан (Will Brennan) – разработчик программ на C++/Python. Живет в Лондоне и имеет богатый опыт высокопроизводительных приложений обработки изображений и машинного обучения. Вы можете найти его на GitHub: https://github.com/WillBrennan.

Предисловие

Язык программирования C++ имеет долгую историю, уходящую корнями в 1980-е. Недавно он пережил возрождение благодаря появлению новых возможностей в стандартах 2011 и 2014 годов. Пока книга готовилась к печати, вышел новый стандарт C++17.

Стандарт C++11 практически удвоил объем стандартной библиотеки, добавив такие заголовки, как `<tuple>`, `<type_traits>` и `<regex>`. Стандарт C++17 снова удвоил объем библиотеки, добавив новые заголовки, такие как `<optional>`, `<any>` и `<filesystem>`. Программист, много времени тративший на разработку кода и не следивший за процессом стандартизации, вполне мог почувствовать себя отставшим от жизни – в стандартной библиотеке появилось так много нового, что он никогда не смог бы овладеть всеми нововведениями в одиночку или хотя бы отделить зерна от плевел. В конце концов, кому захочется провести месяц, читая техническую документацию о `std::locale` и `std::ratio`, только чтобы узнать, что в них нет ничего, что могло бы пригодиться ему в повседневной работе?

В этой книге я расскажу вам о наиболее важных особенностях стандартной библиотеки C++17. Для краткости я опущу некоторые разделы, такие как вышеупомянутый `<type_traits>`; но мы рассмотрим всю современную стандартную библиотеку шаблонов STL (каждый стандартный контейнер и каждый стандартный алгоритм), плюс такие важные темы, как умные указатели, случайные числа, регулярные выражения и новую для C++17 библиотеку `<filesystem>`.

Я буду знакомить вас с новинками на примерах. Вы научитесь создавать свои типы итераторов; свои диспетчеры памяти на основе `std::pmr::memory_resource`; свои пулы потоков выполнения с использованием `std::future`.

Я расскажу об идеях, которые вы не найдете в справочных руководствах. Вы узнаете, чем отличаются мономорфизм monomorphic, полиморфизм и обобщенные алгоритмы (глава 1 «*Классический полиморфизм и обобщенное программирование*»); что означает для `std::string` или `std::any` быть «словарным типом» (глава 5 «*Словарные типы*») и чего можно ожидать от грядущего стандарта C++20 и потом.

Я предполагаю, что вы уже достаточно хорошо знакомы с основами языка C++11; например, что вы уже понимаете, как писать шаблонные классы и функции, знаете, чем отличаются ссылки *lvalue* и *rvalue*, и т. д.

Содержание книги

Глава 1 «*Классический полиморфизм и обобщенное программирование*» охватывает классический полиморфизм (виртуальные функции-члены) и обобщенное программирование (шаблоны).

Глава 2 «*Итераторы и диапазоны*» объясняет идею представления итератора как обобщенного указателя и практическую пользу полуоткрытых диапазонов, выраженных в виде пары итераторов.

Глава 3 «Алгоритмы с парами итераторов» исследует широкое разнообразие обобщенных алгоритмов, оперирующих диапазонами, выраженными в виде пар итераторов.

Глава 4 «Зоопарк контейнеров» исследует не менее широкое разнообразие стандартных шаблонных классов контейнеров и рассказывает, какие контейнеры лучше подходят для тех или иных задач.

Глава 5 «Словарные типы» проведет вас через царство алгебраических типов, таких как `std::optional`, и ABI-совместимые стираемые типы, такие как `std::function`.

Глава 6 «Умные указатели» рассказывает о назначении и особенностях использования умных указателей.

Глава 7 «Конкуренция» охватывает атомы, мьютексы, условные переменные, потоки выполнения, объекты `future` и `promise`.

Глава 8 «Диспетчеры памяти» описывает новый заголовок `<memory_resource>`, появившийся в C++17.

Глава 9 «Потоки ввода/вывода» исследует развитие модели ввода/вывода в C++, от `<unistd.h>` до `<stdio.h>` и `<iostream>`.

Глава 10 «Регулярные выражения» научит вас пользоваться регулярными выражениями в C++.

Глава 11 «Случайные числа» описывает поддержку генераторов псевдослучайных чисел в C++.

Глава 12 «Файловая система» охватывает новую библиотеку `<filesystem>`, появившуюся в C++17.

Что потребуется для работы с книгой

Поскольку эта книга не является справочным руководством, вам может пригодиться такое руководство, как cppreference (`en.cppreference.com/w/cpp`), где вы сможете прояснить любые детали. Вам также определенно понадобится компилятор, поддерживающий стандарт C++17. На момент подготовки книги к печати существовало несколько компиляторов с более или менее полноценной поддержкой C++17, включая GCC, Clang и Microsoft Visual Studio. Вы можете установить их у себя локально или воспользоваться многочисленными бесплатными онлайн-службами, такими как Wandbox (`wandbox.org`), Godbolt (`gcc.godbolt.org`) и Rextester (`rextester.com`).

Кому адресована эта книга

Эта книга адресована разработчикам, желающим овладеть новыми особенностями библиотеки C++17 STL и в полной мере использовать ее компоненты. Знакомство с языком C++ является обязательным условием.

Типографские соглашения

В этой книге используется несколько разных стилей оформления текста с целью обеспечить визуальное отличие информации разных типов. Ниже приводится несколько примеров таких стилей оформления и краткое описание их назначения.

Программный код в тексте, имена таблиц баз данных, имена папок, имена файлов, расширения файлов, пути в файловой системе, адреса URL, ввод пользователя и ссылки в Twitter оформляются, как показано в следующем предложении: «Функция `buffer()` принимает аргументы типа `int`».

Блоки программного кода оформляются так:

```
try {
  none.get();
} catch (const std::future_error& ex) {
  assert(ex.code() == std::future_errc::broken_promise);
}
```

Новые термины и **важные определения** будут выделяться в обычном тексте жирным.

Таким значком будут оформляться предупреждения и важные примечания.

Таким значком будут оформляться советы и рекомендации.

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 прямо 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 напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы 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.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф в разделе «Читателям – Файлы к книгам».

Кроме того, примеры кода к книге доступны на сайте GitHub, по адресу: https://github.com/PacktPublishing/ Mastering-the-Cpp17-STL.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки всё равно случаются. Если вы найдёте ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии данной книги.

Если вы найдёте 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 Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли принять меры.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Глава 1

Классический полиморфизм и обобщенное программирование

Стандартная библиотека C++ преследует две разные, но одинаково важные цели. Первая цель – предоставить надежные реализации некоторых конкретных типов данных или функций, которые могут пригодиться в разных программах, но не являются частью базового синтаксиса языка. Именно поэтому стандартная библиотека включает `std::string`, `std::regex`, `std::filesystem::exists` и т. д. Другая цель – предоставить надежные реализации широко используемых абстрактных алгоритмов сортировки, поиска, обращения, сравнения и т. д. В этой главе мы узнаем, что подразумевается под словами «абстрактный код», и рассмотрим два подхода к определению абстрактного кода, которые используются в стандартной библиотеке: *классический полиморфизм* и *обобщенное программирование*.

В этой главе рассматриваются следующие темы:

- конкретные (мономорфные) функции, поведение которых не параметризуется;
- классический полиморфизм: базовые классы, виртуальные функции-члены и наследование;
- обобщенное программирование: концепции, требования и модели;
- практические достоинства и недостатки каждого из подходов.

Конкретные мономорфные функции

Что отличает абстрактный алгоритм от конкретной функции? Ответить на этот вопрос нам поможет пример. Напишем функцию, умножающую каждый элемент массива на 2:

```
class array_of_ints {
  int data[10] = {};
```

```
public:
  int size() const { return 10; }
  int& at(int i) { return data[i]; }
};

void double_each_element(array_of_ints& arr)
{
  for (int i=0; i < arr.size(); ++i) {
    arr.at(i) *= 2;
  }
}
```

Наша функция `double_each_element` работает *только* с объектами типа `array_of_int`; попытка передать объект другого типа не увенчается успехом (код просто не будет компилироваться). Функции, такие как наша `double_each_element`, называют *конкретными*, или *мономорфными*. Мы называем их *конкретными*, потому что они недостаточно *абстрактны* для наших целей. Просто представьте, насколько неудобно было бы, если бы стандартная библиотека C++ предоставляла конкретную процедуру `sort`, работающую только с определенным типом данных!

Классические полиморфные функции

Мы можем повысить уровень абстракции наших алгоритмов, применив приемы классического **объектно-ориентированного** (ОО) программирования, широко используемые в таких языках, как Java и C#. Суть подхода ОО состоит в том, чтобы решить, какие варианты поведения должны быть настраиваемыми, и затем объявить их в виде общедоступных виртуальных функций-членов *абстрактного базового класса*:

```
class container_of_ints {
public:
  virtual int size() const = 0;
  virtual int& at(int) = 0;
};

class array_of_ints : public container_of_ints {
  int data[10] = {};
public:
  int size() const override { return 10; }
  int& at(int i) override { return data[i]; }
};

class list_of_ints : public container_of_ints {
  struct node {
    int data;
    node *next;
  };
```

```
  node *head_ = nullptr;
  int size_ = 0;
public:
  int size() const override { return size_; }
  int& at(int i) override {
    if (i >= size_) throw std::out_of_range("at");
    node *p = head_;
    for (int j=0; j < i; ++j) {
      p = p->next;
    }
    return p->data;
  }
  ~list_of_ints();
};

void double_each_element(container_of_ints& arr)
{
  for (int i=0; i < arr.size(); ++i) {
    arr.at(i) *= 2;
  }
}

void test()
{
  array_of_ints arr;
  double_each_element(arr);

  list_of_ints lst;
  double_each_element(lst);
}
```

Два разных вызова `double_each_element` в `test` успешно компилируются, потому что с точки зрения классического ОО `array_of_ints` **ЯВЛЯЕТСЯ** `container_of_ints` (то есть наследует `container_of_ints` и реализует соответствующие виртуальные функции-члены) и `list_of_ints` также **ЯВЛЯЕТСЯ** `container_of_ints`. Однако поведение любого данного объекта `container_of_ints` параметризуется его *динамическим типом*; то есть определяется таблицей указателей на функции, связанной с конкретным объектом.

Поскольку теперь поведение функции `double_each_element` можно параметризовать, не меняя ее исходный код, а просто передавая ей объекты разных динамических типов, мы говорим, что функция является *полиморфной*.

Но такая полиморфная функция может работать только с типами, которые наследуют базовый класс `container_of_ints`. Например, у вас не получится передать этой функции вектор `std::vector<int>`; при попытке сделать это вы получите ошибку компиляции. Классический полиморфизм – удобный прием, но не дает нам полной обобщенности.

Преимущество классического (объектно-ориентированного) полиморфизма в том, что исходный код все еще однозначно соответствует машинному

коду, генерируемому компилятором. На уровне машинного кода у нас имеется только одна функция `double_each_element`, с одной сигнатурой и с одной, четко определенной точкой входа. Например, мы легко можем получить адрес функции `double_each_element` и сохранить его в указателе.

Обобщенное программирование с шаблонами

В современном C++ полностью обобщенные алгоритмы обычно записываются в виде *шаблонов*. Мы все еще должны реализовать шаблонную функцию в терминах общедоступных функций-членов `.size()` и `.at()`, но при этом отпадает требование к принадлежности аргумента `arr` определенному типу. Поскольку наша новая функция будет шаблонной, мы должны сообщить компилятору, что нас не интересует конкретный тип `arr`, что для каждого нового типа `arr` он должен сгенерировать совершенно новую функцию (то есть создать экземпляр шаблона) с параметром этого типа.

```
template<class ContainerModel>
void double_each_element(ContainerModel& arr)
{
  for (int i=0; i < arr.size(); ++i) {
    arr.at(i) *= 2;
  }
}

void test()
{
  array_of_ints arr;

  double_each_element(arr);

  list_of_ints lst;
  double_each_element(lst);

  std::vector<int> vec = {1, 2, 3};
  double_each_element(vec);
}
```

В большинстве случаев возможность точно выразить словами, какие операции должны поддерживаться шаблонным параметром типа `ContainerModel`, помогает создавать более совершенные программы. Такой набор операций определяет то, что в C++ называют *концепцией*; в этом примере мы можем сказать, что концепция `Container` заключается в «наличии функции-члена с именем `size`, которая возвращает размер контейнера в виде значения типа `int` (или другого, сравнимого с `int`); и в наличии функции-члена с именем `at`, которая принимает индекс типа `int` (или другого, который неявно может преобразовываться в тип `int`) и возвращает неконстантную ссылку на элемент в контейнере с этим *индексом*». Если некоторый класс `array_of_ints`

поддерживает операции, требуемые концепцией `Container`, так что объекты этого класса могут передаваться в вызов `double_each_element`, мы говорим, что конкретный класс `array_of_ints` является *моделью* концепции `Container`. Именно поэтому я дал имя `ContainerModel` параметру типа шаблона в предыдущем примере.

Более традиционно было бы использовать имя `Container` для параметра типа шаблона, и с этого момента я так и буду поступать; просто я не хотел начинать с путаницы между концепцией `Container` и конкретным параметром типа шаблона в этой конкретной шаблонной функции, которая выглядит так, будто ее аргумент принадлежит конкретному классу, моделирующему концепцию `Container`.

Реализация абстрактного алгоритма с использованием шаблонов, когда поведение алгоритма параметризуется на этапе компиляции типами, моделирующими соответствующие концепции, называется обобщенным программированием.

Обратите внимание, что в нашем описании концепции `Container` нигде не говорится о том, что элементы контейнера должны иметь тип `int`; и не случайно мы теперь можем использовать нашу обобщенную функцию `double_each_element` даже с контейнерами, содержащими значения других типов, отличных от `int`!

```
std::vector<double> vecd = {1.0, 2.0, 3.0};
double_each_element(vecd);
```

Этот дополнительный уровень обобщенности является одним из важнейших достоинств шаблонов C++ в обобщенном программировании, в отличие от классического полиморфизма. Классический полиморфизм скрывает разность поведений разных классов за неизменной сигнатурой (например, `.at(i)` всегда возвращает `int&`), но, как только появляется необходимость использовать разные сигнатуры, классический полиморфизм перестает быть хорошим инструментом для этой работы.

Другое достоинство обобщенного программирования – высокая скорость благодаря расширенным возможностям встраивания. Пример, реализованный в стиле классического полиморфизма, вынужден снова и снова обращаться к таблице виртуальных методов объекта `container_of_int`, чтобы получить адрес конкретного виртуального метода `at`, и, как правило, лишен возможности миновать этот поиск на этапе компиляции. Шаблонная функция `double_each_element<array_of_int>`, напротив, может скомпилировать непосредственный вызов `array_of_int::at` или даже встроить тело этой функции в код.

Поскольку обобщенное программирование с шаблонами легко справляется со сложными требованиями и обеспечивает гибкую поддержку разных типов – даже таких простых, как `int`, где классический полиморфизм терпит неудачу, – стандартная библиотека использует шаблоны для всех алгоритмов в ней и контейнеров, которыми эти алгоритмы оперируют. По этой причине часть стандартной библиотеки, имеющей отношение к алгоритмам и контейнерам, часто называют стандартной библиотекой шаблонов (Standard Template Library, STL).

Все верно – технически STL является лишь малой частью стандартной библиотеки C++! Но в этой книге, так же как в реальной жизни, мы можем иногда вольно использовать термин STL, подразумевая всю стандартную библиотеку, и наоборот.

Рассмотрим еще пару своих обобщенных алгоритмов, прежде чем углубиться в стандартные обобщенные алгоритмы, реализованные в STL. Вот шаблонная функция `count`, возвращающая общее количество элементов в контейнере:

```
template<class Container>
int count(const Container& container)
{
  int sum = 0;
  for (auto&& elt : container) {
    sum += 1;
  }
  return sum;
}
```

А вот функция `count_if`, возвращающая количество элементов, удовлетворяющих пользовательской функции-предикату:

```
template<class Container, class Predicate>
int count_if(const Container& container, Predicate pred)
{
  int sum = 0;
  for (auto&& elt : container) {
    if (pred(elt)) {
      sum += 1;
    }
  }
  return sum;
}
```

Эти функции можно использовать, как показано ниже:

```
std::vector<int> v = {3, 1, 4, 1, 5, 9, 2, 6};

assert(count(v) == 8);

int number_above =
  count_if(v, [](int e) { return e > 5; });
int number_below =
  count_if(v, [](int e) { return e < 5; });

assert(number_above == 2);
assert(number_below == 5);
```

Как много заключено в этом маленьком выражении `pred(elt)`! Попробуйте реализовать функцию `count_if` в терминах классического полиморфизма, просто чтобы понять, насколько быстро все начнет разваливаться. Под синтаксическим сахаром современного C++ скрыто огромное количество сигнатур. Например, синтаксис цикла `for` с диапазонами в нашей функции `count_if` преобразуется (или упрощается) компилятором в цикл `for`, действующий в терминах методов `container.begin()` и `container.end()`, каждый из которых должен возвращать итератор с типом, зависящим от типа контейнера. Другой пример: в обобщенной версии мы нигде не указываем – и нам нигде не нужно указывать, – будет ли `pred` принимать свой параметр `elt` по значению или по ссылке. Попробуйте реализовать то же самое в `virtual bool operator()`!

В продолжение темы итераторов: возможно, вы заметили, что все наши функции в этой главе (мономорфные, полиморфные или обобщенные) выражены в терминах контейнеров. В своей версии `count` мы подсчитывали элементы во всем контейнере. В `count_if` мы подсчитывали элементы во всем контейнере, соответствующие заданному условию. Как оказывается, это очень распространенный способ записи алгоритмов, особенно в современном C++; поэтому многое из того, что мы ожидаем увидеть в алгоритмах работы с контейнерами (или в их близких родственниках – алгоритмах работы с диапазонами), перекочует в C++20 или C++23. Однако STL зародилась в далеких 1990-х, до появления современного C++. Поэтому авторы STL предполагали, что обработка контейнеров будет обходиться особенно дорого (из-за всех этих дорогостоящих вызовов конструкторов копий – напомню, что семантика перемещения и конструкторы перемещения появились только в C++11) и проектировали STL в основном для работы с легковесной концепцией – итераторами. Это станет темой нашей следующей главы.

Итоги

Классический полиморфизм и обобщенное программирование призваны решить проблему параметризации поведения алгоритма: например, чтобы дать возможность написать функцию поиска, которая способна работать с произвольными операциями сопоставления.

Классический полиморфизм решает проблему за счет определения *базового класса* с закрытым списком *виртуальных функций-членов*, и реализации *полиморфных функций*, принимающих указатели или ссылки на экземпляры конкретных классов, *наследующих* базовый класс.

Обобщенное программирование решает ту же проблему за счет определения *концепции* с закрытым набором *требований* и создания экземпляров *шаблонных функций* с конкретными классами, *моделирующими* эту концепцию.

Классический полиморфизм испытывает проблемы с параметризацией высокого уровня (например, манипулирование объектами функций с любой сигнатурой) и с отношениями между типами (например, манипулирование элементами произвольных контейнеров). Поэтому в STL большое внимание уделяется обобщенным реализациям на основе шаблонов и почти полностью отсутствует классический полиморфизм.

Применение приемов обобщенного программирования помогает справиться с проблемами, если учитываются концептуальные требования к типам; но компилятор C++, по крайней мере соответствующий стандарту C++17, пока не в состоянии напрямую помочь с проверкой этих требований.