

# Оглавление

<b>Об авторе .....</b>	<b>10</b>
<b>О научном редакторе .....</b>	<b>11</b>
<b>Предисловие.....</b>	<b>12</b>
Содержание книги.....	12
Что потребуется для работы с книгой.....	13
Кому адресована эта книга.....	13
Типографские соглашения.....	14
Отзывы и пожелания.....	14
Скачивание исходного кода примеров.....	15
Список опечаток.....	15
Нарушение авторских прав.....	15
<b>Глава 1. Классический полиморфизм и обобщенное программирование .....</b>	<b>16</b>
Конкретные мономорфные функции .....	16
Классические полиморфные функции.....	17
Обобщенное программирование с шаблонами .....	19
Итоги .....	22
<b>Глава 2. Итераторы и диапазоны .....</b>	<b>24</b>
Проблема целочисленных индексов .....	24
За границами указателей.....	25
Константные итераторы.....	28
Пара итераторов определяет диапазон .....	29
Категории итераторов .....	31
Итераторы ввода и вывода .....	33
Объединяем все вместе .....	36
Устаревший std::iterator.....	39
Итоги .....	42
<b>Глава 3. Алгоритмы с парами итераторов.....</b>	<b>44</b>
Замечание о заголовках.....	44
Диапазонные алгоритмы только для чтения.....	44
Манипулирование данными с std::copy.....	51
Вариации на тему: std::move и std::move_iterator .....	54
Непростое копирование с std::transform.....	57
Диапазонные алгоритмы только для записи.....	59

Алгоритмы, влияющие на жизненный цикл объектов.....	60
Наш первый перестановочный алгоритм: <code>std::sort</code> .....	62
Обмен местами, обратное упорядочение и разделение.....	63
Ротация и перестановка.....	67
Кучи и пирамидальная сортировка.....	69
Слияние и сортировка слиянием.....	71
Поиск и вставка в сортированный массив с <code>std::lower_bound</code> .....	71
Удаление из сортированного массива с <code>std::remove_if</code> .....	73
Итоги .....	77
<b>Глава 4. Зоопарк контейнеров .....</b>	<b>78</b>
Понятие владения .....	78
Простейший контейнер: <code>std::array&lt;T, N&gt;</code> .....	80
Рабочая лошадка: <code>std::vector&lt;T&gt;</code> .....	84
Изменение размера <code>std::vector</code> .....	85
Вставка и стирание в <code>std::vector</code> .....	89
Ловушки <code>vector&lt;bool&gt;</code> .....	90
Ловушки в конструкторах перемещения без <code>noexcept</code> .....	91
Быстрый гибрид: <code>std::deque&lt;T&gt;</code> .....	93
Особый набор возможностей: <code>std::list&lt;T&gt;</code> .....	94
Какие отличительные особенности имеет <code>std::list</code> ?.....	95
Список без удобств <code>std::forward_list&lt;T&gt;</code> .....	97
Абстракции с использованием <code>std::stack&lt;T&gt;</code> и <code>std::queue&lt;T&gt;</code> .....	98
Удобный адаптер: <code>std::priority_queue&lt;T&gt;</code> .....	99
Деревья: <code>std::set&lt;T&gt;</code> и <code>std::map&lt;K, V&gt;</code> .....	100
Замечание о прозрачных компараторах.....	104
Необычные <code>std::multiset&lt;T&gt;</code> и <code>std::multimap&lt;K, V&gt;</code> .....	105
Перемещение элементов без перемещения .....	107
Хеши: <code>std::unordered_set&lt;T&gt;</code> и <code>std::unordered_map&lt;K, V&gt;</code> .....	109
Фактор загрузки и списки в корзинах.....	111
Откуда берется память? .....	112
Итоги .....	113
<b>Глава 5. Словарные типы .....</b>	<b>114</b>
История <code>std::string</code> .....	114
Маркировка ссылочных типов с <code>reference_wrapper</code> .....	116
C++11 и алгебраические типы.....	117
Работа с <code>std::tuple</code> .....	118
Манипулирование значениями кортежа.....	120
Замечание об именованных классах.....	121
Выражение альтернатив с помощью <code>std::variant</code> .....	122
Чтение вариантов.....	123
О <code>make_variant</code> и семантике типа-значения.....	125

Задержка инициализации с помощью <code>std::optional</code> .....	127
И снова <code>variant</code> .....	131
Бесконечное число альтернатив с <code>std::any</code> .....	132
<code>std::any</code> и полиморфные типы.....	134
Коротко о стирании типа.....	135
<code>std::any</code> и копирование.....	137
И снова о стирании типов: <code>std::function</code> .....	138
<code>std::function</code> , копирование и размещение в динамической памяти.....	140
Итоги.....	141
<b>Глава 6. Умные указатели.....</b>	<b>142</b>
История появления умных указателей.....	142
Умные указатели никогда ничего не забывают.....	143
Автоматическое управление памятью с <code>std::unique_ptr&lt;T&gt;</code> .....	144
Почему в C++ нет ключевого слова <code>finally</code> .....	147
Настройка обратного вызова удаления.....	148
Управление массивами с помощью <code>std::unique_ptr&lt;T[]&gt;</code> .....	149
Подсчет ссылок с <code>std::shared_ptr&lt;T&gt;</code> .....	150
Не допускайте двойного управления!.....	153
Удерживание обнуляемых дескрипторов с помощью <code>weak_ptr</code> .....	153
Сообщение информации о себе с <code>std::enable_shared_from_this</code> .....	156
Странно рекурсивный шаблон проектирования.....	159
Заключительное замечание.....	160
Обозначение неисключительности с <code>observer_ptr&lt;T&gt;</code> .....	160
Итоги.....	162
<b>Глава 7. Конкуренция.....</b>	<b>163</b>
Проблемы с <code>volatile</code> .....	163
Использование <code>std::atomic&lt;T&gt;</code> для безопасного доступа в многопоточной среде.....	166
Атомарное выполнение сложных операций.....	168
Большие атомарные типы.....	170
Поочередное выполнение с <code>std::mutex</code> .....	171
Правильный порядок «приобретения блокировок».....	173
Всегда связывайте мьютекс с управляемыми данными.....	176
Специальные типы мьютексов.....	180
Повышение статуса блокировки для чтения/записи.....	183
Понижение статуса блокировки для чтения/записи.....	183
Ожидание условия.....	184
Обещания о будущем.....	187
Подготовка заданий для отложенного выполнения.....	190
Будущее механизма <code>future</code> .....	192
Поговорим о потоках.....	194
Идентификация отдельных потоков и текущего потока.....	196

Исчерпание потоков и <code>std::async</code> .....	198
Создание своего пула потоков.....	200
Оптимизация производительности пула потоков.....	204
Итоги.....	206
<b>Глава 8. Диспетчеры памяти.....</b>	<b>208</b>
Диспетчер памяти обслуживает ресурс памяти.....	209
Еще раз об интерфейсах и понятиях.....	210
Определение кучи с помощью <code>memory_resource</code> .....	212
Использование стандартных ресурсов памяти.....	215
Выделение из ресурса пулов.....	217
500-головый стандартный диспетчер памяти.....	218
Метаданные, сопровождающие причудливые указатели.....	222
Прикрепление контейнера к единственному ресурсу памяти.....	227
Использование диспетчеров памяти стандартных типов.....	229
Настройка ресурса памяти по умолчанию.....	230
Создание контейнера с поддержкой выбора диспетчера памяти.....	231
Передача вниз с <code>scoped_allocator_adaptor</code> .....	237
Передача разных диспетчеров памяти.....	240
Итоги.....	242
<b>Глава 9 Потоки ввода/вывода.....</b>	<b>244</b>
Проблемы ввода/вывода в C++.....	244
Буферизация и форматирование.....	246
POSIX API.....	247
Стандартный C API.....	250
Буферизация в стандартном C API.....	252
Форматирование с помощью <code>printf</code> и <code>snprintf</code> .....	257
Классическая иерархия потоков ввода/вывода.....	260
Потоки данных и манипуляторы.....	264
Потоки данных и обертки.....	267
Решение проблемы манипуляторов.....	269
Форматирование с <code>ostringstream</code> .....	270
Примечание о региональных настройках.....	271
Преобразование чисел в строки.....	273
Преобразование строк в числа.....	275
Чтение по одной строке или по одному слову.....	279
Итоги.....	281
<b>Глава 10. Регулярные выражения.....</b>	<b>282</b>
Что такое регулярное выражение?.....	283
Замечание об экранировании обратными слешами.....	284
Воплощение регулярных выражений в объектах <code>std::regex</code> .....	286
Сопоставление и поиск.....	287

Извлечение совпадений с подвыражениями .....	288
Преобразование совпадений в значения данных.....	292
Итерации по нескольким совпадениям.....	293
Использование регулярных выражений для замены строк .....	297
Грамматика регулярных выражений ECMAScript .....	299
Непоглощающие конструкции .....	302
Малопонятные особенности и ловушки ECMAScript .....	303
Итоги .....	305
<b>Глава 11. Случайные числа .....</b>	<b>306</b>
Случайные и псевдослучайные числа.....	306
Проблема функции rand() .....	308
Решение проблем с <random> .....	310
Генераторы .....	310
Истинно случайные биты и std::random_device .....	311
Псевдослучайные биты с std::mt19937 .....	311
Фильтрация вывода генераторов с помощью адаптеров.....	313
Распределения.....	316
Имитация броска игровой кости с uniform_int_distribution .....	316
Генерирование выборок с normal_distribution .....	318
Взвешенный выбор с discrete_distribution .....	319
Перемешивание карт с std::shuffle.....	320
Итоги .....	321
<b>Глава 12. Файловая система.....</b>	<b>323</b>
Примечание о пространствах имен .....	323
Очень длинное примечание об уведомлениях об ошибках.....	325
Использование <system_error> .....	327
Коды ошибок и условия ошибок.....	331
Возбуждение ошибок с std::system_error .....	334
Файловые системы и пути .....	336
Представление путей в C++ .....	338
Операции с путями .....	340
Получение информации о файлах с directory_entry.....	342
Обход каталогов с directory_iterator.....	343
Рекурсивный обход каталогов .....	343
Изменение файловой системы .....	344
Получение информации о диске .....	345
Итоги .....	346
<b>Предметный указатель .....</b>	<b>347</b>

# 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 монаморфизм `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 `srpreference` ([en.cppreference.com/w/cpp](http://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](http://wandbox.org)), Godbolt ([gcc.godbolt.org](http://gcc.godbolt.org)) и Rextester ([rextester.com](http://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 приводятся 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](http://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](mailto: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/](http://dmkpress.com/authors/publish_book/) или напишите в издательство по адресу [dmkpress@gmail.com](mailto:dmkpress@gmail.com).

## Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте [www.dmkpress.com](http://www.dmkpress.com) или [www.дмк.рф](http://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](mailto: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](mailto: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), перекоцует 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 – напомним, 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.