

## Как улучшить свои программы с помощью приемов функционального программирования

Хорошо написанный код легче тестировать и использовать повторно, его проще распараллелить, и он меньше подвержен ошибкам. Владение приемами функционального программирования поможет вам писать код, соответствующий требованиям к современным приложениям, проще выражать сложную логику программ, изящно обрабатывать ошибки и элегантно оформлять параллельные алгоритмы. Язык C++ поддерживает функциональное программирование с использованием шаблонов, лямбда-выражений и других своих базовых возможностей, а также алгоритмов из библиотеки STL.

В этой книге вы найдете десятки примеров, диаграмм и иллюстраций, описывающих идеи функционального программирования, которые вы сможете применять в C++, включая ленивые вычисления, объекты-функции и вызываемые объекты, алгебраические типы данных и многое другое.

### Что внутри:

- как писать безопасный код без ущерба для производительности;
- явная обработка ошибок через систему типов;
- добавление в C++ новых управляющих структур;
- решение задач с использованием предметно-ориентированных языков (DSL).

**Иван Чукич** – один из основных разработчиков KDE; программирует на C++ с 1998 года. Преподает курс современного C++ и функционального программирования на факультете математики в Белградском университете.

«Книга предлагает читателям новый способ создания качественного программного обеспечения и новый способ мышления».

Джан Лоренцо Меокки, CommProve

«Это издание особенно ценно для разработчиков на C++ среднего и выше среднего уровня, желающих научиться писать программы в реактивном стиле».

Марко Массенцио, Apple

**Книга предназначена для опытных разработчиков на C++.**

Интернет-магазин:  
www.dmkpress.ru

Оптовая продажа:  
КТК «Галактика»  
e-mail: books@aliants-kniga.ru

MANNING

DMK  
ИЗДАТЕЛЬСТВО

www.dmk.pf

ISBN 978-5-97060-781-7



9 785970 607817 >

Функциональное программирование на C++

Иван Чукич

# Функциональное программирование

# на C++

DMK  
ИЗДАТЕЛЬСТВО

---

Иван Чукич



# Функциональное программирование на языке C++





# *Functional Programming in C++*

How to improve your C++ programs  
using functional techniques

IVAN ČUKIĆ



MANNING  
Shelter Island

---



# *Функциональное программирование на языке C++*

Как сделать свои программы изящными  
с помощью технологии функционального программирования



**ИВАН ЧУКИЧ**



Москва, 2020

УДК 004.4  
ББК 32.973.202  
Ч88



**Чукич И.**

Ч88 **Функциональное программирование на языке C++ / пер. с англ. В. Ю. Винника, А. Н. Киселева. – М.: ДМК Пресс, 2020. – 360 с.: ил.**

**ISBN 978-5-97060-781-7**

Язык C++ обычно ассоциируется с объектно-ориентированным программированием. Автор книги доказывает, что на C++ так же удобно создавать программы и в функциональном стиле. Это дает ряд преимуществ, повышая удобство кода и снижая вероятность возникновения ошибок.

Книга разделена на две части. В первой читатель знакомится с азами функционального программирования: основными идиомами и способами их воплощения в языке C++. Вторая часть затрагивает более сложные аспекты и посвящена собственно разработке программ с использованием функционального подхода.

Издание предназначено для опытных разработчиков на C++, желающих расширить границы использования этого языка и повысить эффективность работы.

УДК 004.4  
ББК 32.973.202

Original English language edition published by Manning Publications USA, USA. Copyright © 2019 by Manning Publications Co. Russian-language edition copyright © 2020 by DMK Press. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-617-29381-8 (англ.)  
ISBN 978-5-97060-781-7 (рус.)

Copyright © 2019 by Manning Publications Co.  
© Оформление, издание, перевод, ДМК Пресс, 2020



---

# Оглавление

---

1	■ Введение в функциональное программирование .....	22
2	■ Первые шаги в функциональном программировании .....	47
3	■ Функциональные объекты .....	75
4	■ Средства создания новых функций из имеющихся .....	107
5	■ Чистота функций: как избежать изменяемого состояния .....	141
6	■ Ленивые вычисления .....	167
7	■ Диапазоны .....	191
8	■ Функциональные структуры данных .....	209
9	■ Алгебраические типы данных и сопоставление с образцом .....	226
10	■ Монады .....	254
11	■ Метапрограммирование на шаблонах .....	284
12	■ Функциональный дизайн параллельных систем .....	309
13	■ Тестирование и отладка .....	338



# Содержание

Предисловие .....	12
Благодарности .....	14
Об этой книге .....	15
Об авторе .....	21

<b>1 Введение в функциональное программирование .....</b>	<b>22</b>
1.1 Что такое функциональное программирование .....	23
1.1.1 Соотношение функционального программирования с объектно-ориентированным .....	25
1.1.2 Сравнение императивной и декларативной парадигм на конкретном примере .....	25
1.2 Чистые функции .....	31
1.2.1 Устранение изменяемого состояния .....	34
1.3 Функциональный стиль мышления .....	36
1.4 Преимущества функционального программирования .....	38
1.4.1 Краткость и удобочитаемость кода .....	39
1.4.2 Параллельная обработка и синхронизация .....	41
1.4.3 Непрерывная оптимизация .....	42
1.5 Эволюция C++ как языка функционального программирования .....	42
1.6 Что узнает читатель из этой книги .....	44
Итоги .....	45

<b>2 Первые шаги в функциональном программировании .....</b>	<b>47</b>
2.1 Функции с аргументами-функциями .....	48
2.2 Примеры из библиотеки STL .....	50
2.2.1 Вычисление средних .....	51

2.2.2	Свертки.....	53
2.2.3	Удаление лишних пробелов.....	58
2.2.4	Разделение коллекции по предикату.....	60
2.2.5	Фильтрация и преобразование.....	62
2.3	Проблема композиции алгоритмов из библиотеки STL.....	64
2.4	Создание своих функций высшего порядка.....	66
2.4.1	Передача функции в качестве аргумента.....	66
2.4.2	Реализация на основе циклов.....	67
2.4.3	Рекурсия и оптимизация хвостового вызова.....	68
2.4.4	Реализация на основе свертки.....	72
	Итоги.....	74

<b>3</b>	<b>Функциональные объекты.....</b>	<b>75</b>
3.1	Функции и функциональные объекты.....	76
3.1.1	Автоматический вывод возвращаемого типа.....	76
3.1.2	Указатели на функции.....	80
3.1.3	Перегрузка операции вызова.....	81
3.1.4	Обобщенные функциональные объекты.....	84
3.2	Лямбда-выражения и замыкания.....	86
3.2.1	Синтаксис лямбда-выражений.....	88
3.2.2	Что находится у лямбда-выражений «под капотом».....	89
3.2.3	Создание лямбда-выражений с произвольными переменными-членами.....	92
3.2.4	Обобщенные лямбда-выражения.....	94
3.3	Как сделать функциональные объекты еще лаконичнее.....	95
3.3.1	Объекты-обертки над операциями в стандартной библиотеке.....	98
3.3.2	Объекты-обертки над операциями в сторонних библиотеках.....	100
3.4	Обертка над функциональными объектами – класс <code>std::function</code> .....	103
	Итоги.....	105

<b>4</b>	<b>Средства создания новых функций из имеющихся.....</b>	<b>107</b>
4.1	Частичное применение функций.....	108
4.1.1	Универсальный механизм превращения бинарных функций в унарные.....	110
4.1.2	Использование функции <code>std::bind</code> для фиксации значений некоторых аргументов функции.....	114
4.1.3	Перестановка аргументов бинарной функции.....	116
4.1.4	Использование функции <code>std::bind</code> с функциями большего числа аргументов.....	118
4.1.5	Использование лямбда-выражений вместо функции <code>std::bind</code> .....	121
4.2	Карринг – необычный взгляд на функции.....	124

4.2.1	Простой способ создавать каррированные функции .....	125
4.2.2	Использование карринга для доступа к базе данных.....	127
4.2.3	Карринг и частичное применение функций .....	130
4.3	Композиция функций .....	132
4.4	Повторное знакомство с подъемом функций .....	136
4.4.1	Переворачивание пар – элементов списка .....	138
	Итоги .....	140

<b>5</b>	<b>Чистота функций: как избежать изменяемого состояния .....</b>	<b>141</b>
5.1	Проблемы изменяемого состояния.....	142
5.2	Чистые функции и референциальная прозрачность .....	145
5.3	Программирование без побочных эффектов .....	148
5.4	Изменяемые и неизменяемые состояния в параллельных системах.....	152
5.5	О важности констант .....	156
5.5.1	Логическая и внутренняя константность.....	159
5.5.2	Оптимизированные функции-члены для временных объектов.....	161
5.5.3	Недостатки константных объектов .....	163
	Итоги .....	165

<b>6</b>	<b>Ленивые вычисления .....</b>	<b>167</b>
6.1	Ленивые вычисления в языке C# .....	168
6.2	Ленивые вычисления как средство оптимизации программ.....	172
6.2.1	Ленивая сортировка коллекций .....	172
6.2.2	Отображение элементов в пользовательском интерфейсе .....	174
6.2.3	Подрезка дерева рекурсивных вызовов за счет запоминания результатов функции.....	175
6.2.4	Метод динамического программирования как разновидность ленивого вычисления.....	178
6.3	Универсальная мемоизирующая обертка .....	180
6.4	Шаблоны выражений и ленивая конкатенация строк.....	184
6.4.1	Чистота функций и шаблоны выражений.....	188
	Итоги .....	190

<b>7</b>	<b>Диапазоны .....</b>	<b>191</b>
7.1	Введение в диапазоны .....	193
7.2	Создание представлений данных, доступных только для чтения.....	194
7.2.1	Функция filter для диапазонов .....	194

7.2.2	Функция <i>transform</i> для диапазонов .....	196
7.2.3	Ленивые вычисления с диапазоном значений .....	197
7.3	Изменение значений с помощью диапазонов .....	199
7.4	Ограниченные и бесконечные диапазоны .....	201
7.4.1	Использование ограниченных диапазонов для оптимизации обработки входных диапазонов .....	201
7.4.2	Создание бесконечного диапазона с помощью ограничителя .....	203
7.5	Использование диапазонов для вычисления частоты слов .....	204
	Итоги .....	208

<b>8</b>	<b>Функциональные структуры данных</b> .....	209
8.1	Неизменяемые связанные списки .....	210
8.1.1	Добавление и удаление элемента в начале списка .....	210
8.1.2	Добавление и удаление элемента в конце списка .....	212
8.1.3	Добавление и удаление элемента в середине списка .....	213
8.1.4	Управление памятью .....	213
8.2	Неизменяемые векторы .....	216
8.2.1	Поиск элементов в префиксном дереве .....	218
8.2.2	Добавление элементов в конец префиксного дерева .....	220
8.2.3	Изменение элементов в префиксном дереве .....	223
8.2.4	Удаление элемента из конца префиксного дерева .....	223
8.2.5	Другие операции и общая эффективность префиксных деревьев .....	223
	Итоги .....	225

<b>9</b>	<b>Алгебраические типы данных и сопоставление с образцом</b> .....	226
9.1	Алгебраические типы данных .....	227
9.1.1	Определение типов-сумм через наследование .....	229
9.1.2	Определение типов-сумм с использованием объединений и <i>std::variant</i> .....	232
9.1.3	Реализация конкретных состояний .....	235
9.1.4	Особый тип-сумма: необязательные значения .....	237
9.1.5	Типы-суммы для обработки ошибок .....	240
9.2	Моделирование предметной области с алгебраическими типами .....	245
9.2.1	Простейшее решение .....	246
9.2.2	Более сложное решение: проектирование сверху вниз .....	247
9.3	Алгебраические типы и сопоставление с образцом .....	248
9.4	Сопоставление с образцом с помощью библиотеки <i>Mach7</i> .....	251
	Итоги .....	253

<b>10</b>	<b>Монады</b> .....	254
10.1	Функторы .....	255
10.1.1	Обработка необязательных значений .....	256
10.2	Монады: расширение возможностей функторов .....	259
10.3	Простые примеры .....	262
10.4	Генераторы диапазонов и монад .....	265
10.5	Обработка ошибок .....	268
10.5.1	<code>std::optional&lt;T&gt;</code> как монада .....	268
10.5.2	<code>expected&lt;T, E&gt;</code> как монада .....	270
10.5.3	Исключения и монады .....	271
10.6	Обработка состояния с помощью монад .....	273
10.7	Монады, продолжения и конкурентное выполнение .....	275
10.7.1	Тип <code>future</code> как монада .....	277
10.7.2	Реализация типа <code>future</code> .....	279
10.8	Композиция монад .....	281
	Итоги .....	283
<b>11</b>	<b>Метапрограммирование на шаблонах</b> .....	284
11.1	Манипулирование типами во время компиляции .....	285
11.1.1	Проверка правильности определения типа .....	288
11.1.2	Сопоставление с образцом во время компиляции .....	290
11.1.3	Получение метаинформации о типах .....	293
11.2	Проверка свойств типа во время компиляции .....	294
11.3	Каррирование функций .....	296
11.3.1	Вызов всех вызываемых объектов .....	299
11.4	Строительные блоки предметно-ориентированного языка .....	302
	Итоги .....	307
<b>12</b>	<b>Функциональный дизайн параллельных систем</b> .....	309
12.1	Модель акторов: мышление в терминах компонентов .....	310
12.2	Простой источник сообщений .....	314
12.3	Моделирование реактивных потоков данных в виде монад .....	318
12.3.1	Создание приемника для сообщений .....	319
12.3.2	Преобразование реактивных потоков данных .....	323
12.3.3	Создание потока заданных значений .....	325
12.3.4	Объединение потоков в один поток .....	326
12.4	Фильтрация реактивных потоков .....	327
12.5	Обработка ошибок в реактивных потоках .....	328
12.6	Возврат ответа клиенту .....	331
12.7	Создание акторов с изменяемым состоянием .....	335
12.8	Распределенные системы на основе акторов .....	336
	Итоги .....	337

<b>13</b>	<b>Тестирование и отладка</b> .....	338
13.1	Программа, которая компилируется, – правильная? .....	339
13.2	Модульное тестирование и чистые функции .....	341
13.3	Автоматическое генерирование тестов .....	343
13.3.1	Генерирование тестовых случаев .....	343
13.3.2	Тестирование на основе свойств .....	345
13.3.3	Сравнительное тестирование .....	347
13.4	Тестирование параллельных систем на основе монад .....	348
	Итоги .....	352
	<i>Предметный указатель</i> .....	353



---

# Предисловие

---

Программирование – одна из тех немногих дисциплин, что позволяют творить нечто буквально из ничего. Программист может творить целые миры, которые ведут себя в точности как задумал автор. Для этого нужен лишь компьютер.

Когда я учился в школе, курс программирования был в основном сфокусирован на императивном программировании – сначала это было процедурное программирование на языке C, затем объектно-ориентированное на языках C++ и Java. С поступлением в университет почти ничего не изменилось – основной парадигмой по-прежнему оставалось объектно-ориентированное программирование (ООП).

Поэтому тогда я едва не попался в мыслительную ловушку, убедив себя в том, что все языки программирования, в сущности, одинаковы, различия между ними – чисто синтаксические и программисту довольно изучить такие основополагающие понятия, как ветвление и цикл, в одном языке, чтобы запрограммировать (с небольшими поправками) на всех остальных языках.

С языками функционального программирования я впервые познакомился в университете, когда в рамках одной из дисциплин понадобилось изучить язык LISP. Моей первой реакцией было смоделировать средствами языка LISP условный оператор `if-then-else` и оператор цикла `for`, чтобы сделать язык пригодным для работы. Вместо того чтобы привести свое восприятие в соответствие с языком, я решил доработать язык напильником, чтобы он позволял мне и дальше писать программы таким же образом, каким я привык писать на языке C. Нетрудно догадаться, что в те дни я не увидел никакого смысла в функциональном программировании, ведь все, чего я мог добиться, используя LISP, можно было гораздо проще сделать на языке C.

Прошло немало времени, прежде чем я снова стал поглядывать в сторону функционального программирования. Подтолкнула меня к этому неудовлетворенность слишком медленной эволюцией одного языка, который мне необходимо было использовать для нескольких проектов. В язык наконец добавили оператор цикла `for-each`, и это подавалось как громадное достижение. Программисту оставалось лишь загрузить новый компилятор, и жизнь должна была заиграть новыми красками.

Это заставило меня задуматься. Чтобы получить в свое распоряжение новую языковую конструкцию наподобие цикла `for-each`, мне нужно было дождаться новой версии языка и новой версии компилятора. Но на языке LISP я мог самостоятельно реализовать цикл `for` в виде обыкновенной функции. Никакого обновления компилятора при этом не требовалось.

Именно это склонило меня к функциональному программированию: возможность расширить язык без необходимости менять компилятор. Я по-прежнему оставался в плену объектно-ориентированного мировоззрения, но уже научился использовать конструкции, заимствованные из функционального стиля, чтобы упростить работу по созданию объектно-ориентированного кода.

Тогда я стал посвящать много времени исследованию функциональных языков программирования, таких как Haskell, Scala и Erlang. Меня поразило, что многие проблемы, заставляющие страдать объектно-ориентированных программистов, удается легко решить, посмотрев на них под другим углом – функциональным.

Поскольку большая часть моей работы связана с языком C++, я решил найти способ, как использовать в этом языке приемы функционального программирования. Оказалось, что я в этом не одинок: в мире полно других людей с похожими идеями. Некоторых из них мне посчастливилось встретить на различных конференциях по языку C++. Всякий раз это была превосходная возможность обменяться идеями, научиться чему-то новому и поделиться своим опытом применения идиом функционального программирования на языке C++.

На большей части таких встреч мы с коллегами сходились на том, что было бы замечательно, если бы кто-то написал книгу о функциональном программировании на C++. Вот только каждый из нас хотел, чтобы эту книгу написал кто-то другой, поскольку каждый искал источник готовых идей, пригодных для собственных проектов.

Когда издательство Manning предложило мне стать автором такой книги, это меня поначалу обескуражило: я считал, что мне следовало бы прочесть об этом книгу, а не написать ее. Однако затем я подумал, что если каждый будет рассуждать подобным образом, никто так и не увидит книгу о функциональном программировании на языке C++. Я решил принять предложение и отправиться в это путешествие. То, что из этого получилось, вы сейчас и читаете.

---

# Благодарности

---



Я хотел бы поблагодарить всех, чье участие сделало эту книгу возможной: профессора Сашу Малкова за то, что привил мне любовь к языку C++; Ако Самарджича за то, что научил меня, насколько важно писать легкочитаемый код; моего друга Николу Йелича, который убедил меня, что функциональное программирование – это здорово; Золтана Порколаба, поддержавшего мою догадку, что функциональное программирование и язык C++ образуют хорошую смесь; Мирьяну Малькович за помощь в обучении наших студентов тонкостям современного языка C++, включая и элементы функционального программирования.

Почет и уважение Сергею Платонову и Йенсу Веллеру за организацию превосходных конференций по языку C++ для тех из нас, кто все еще живет старыми традициями. Можно смело сказать, что без всех перечисленных людей эта книга бы не состоялась.

Я хотел бы поблагодарить своих родителей, сестру Сою и свою вторую половинку Милицу за то, что всегда поддерживали меня в смелых начинаниях наподобие этого. Кроме того, я благодарен своим давним товарищам по проекту KDE, которые помогли мне вырасти как разработчику за прошедшее десятилетие, – прежде всего Марко Мартину, Аарону Сеиго и Себастиану Кюглеру.

Огромная благодарность команде редакторов, которую для меня организовало издательство Manning: Майклу (или просто Майку) Стивенсу за самое непринужденное из всех моих собеседований; замечательным ответственным редакторам Марине Майклс и Лесли Трайтс, которые научили меня писать книгу (благодаря им я научился гораздо большему, чем мог ожидать); техническому редактору Марку Эльстону за то, что заставлял меня держаться ближе к практике; блестящему Юнвю Ву, который не только проделал работу по вычитке книги, но и помог улучшить рукопись множеством различных способов. Надеюсь, что доставил всем им не слишком много хлопот.

Также выражаю свою признательность всем, кто предоставил свой отзыв на рукопись: Андреасу Шабусу, Биннуру Курту, Дэвиду Кернсу, Димитрису Пападопулосу, Дрору Хелперу, Фредерику Флейолу, Георгу Эрхардту, Джанлуиджи Спануоло, Глену Сиракавиту, Жану Франсуа Морену, Керти Шетти, Марко Массенцио, Нику Гидео, Никосу Атанасиу, Нитину Годе, Ольве Маудалу, Патрику Регану, Шону Липпи, а в особенности Тимоти Театро, Ади Шавиту, Суманту Тамбе, Джану Лоренцо Меоччи и Николе Гиганте.

---

# Об этой книге

---



Эта книга предназначена не для того, чтобы научить читателя языку программирования C++. Она повествует о функциональном программировании и о том, как воплотить его в языке C++. Функциональное программирование предлагает непривычный взгляд на разработку программ и иной подход к программированию, по сравнению с императивным и объектно-ориентированным стилем, который обычно используется совместно с языком C++.

Многие, увидев заглавие этой книги, могут счесть его странным, поскольку C++ часто по ошибке принимают за объектно-ориентированный язык. Однако хотя язык C++ и впрямь хорошо поддерживает объектно-ориентированную парадигму, он на самом деле выходит далеко за ее границы. Так, он поддерживает еще и процедурную парадигму, а в поддержке обобщенного программирования большинству других языков с ним не сравниться. Кроме того, язык C++ весьма хорошо поддерживает большинство, если не все, из идиом функционального программирования, в чем читатель сможет вскоре убедиться. С каждой новой версией язык C++ пополнялся новыми средствами, делающими функциональное программирование на нем все более удобным.

## Кому стоит читать эту книгу

Эта книга предназначена в первую очередь для профессиональных разработчиков на языке C++. Предполагается, что читатель умеет самостоятельно настраивать среду для сборки программ, устанавливать и использовать сторонние библиотеки. Кроме того, от читателя потребуются хотя бы начальное знакомство со стандартной библиотекой шаблонов, автоматическим выводом типов – параметров шаблона и примитивами параллельного программирования – например, двоичными семафорами.

Впрочем, книга не окажется полностью непонятной для читателя, не искушенного в разработке на языке C++. В конце каждой главы приведен список статей, разъясняющих языковые конструкции, которые могли бы показаться читателю малознакомыми.

## Последовательность чтения

Данную книгу лучше всего читать последовательно, так как каждая последующая глава опирается на понятия, разобранные в предыдущих. Если что-либо остается непонятным после первого прочтения, лучше перечитать сложный раздел еще раз, прежде чем двигаться дальше, по-

сколько сложность материала возрастает с каждой главой. Единственное исключение здесь составляет глава 8, которую читатель может пропустить, если не интересуется методами долговременного хранения структур данных.

Книга разделена на две части. Первая часть охватывает идиомы функционального программирования и способы их воплощения в языке C++.

- Глава 1 вкратце знакомит читателя с функциональным программированием и преимуществами, которые оно может принести в мир C++.
- Глава 2 посвящена функциям высшего порядка – функциям, которые принимают другие функции в качестве аргументов или возвращают функции в качестве значений. Данное понятие иллюстрируется некоторыми из множества полезных функций высшего порядка, включенных в стандартную библиотеку языка программирования C++.
- Глава 3 повествует обо всех разнообразных сущностях, которые в языке C++ рассматриваются как функции, – от обычных функций в смысле языка C до функциональных объектов и лямбда-функций.
- Глава 4 содержит изложение различных способов создания новых функций из имеющихся. В этой главе рассказано о частичном применении функций с использованием операции `std::bind` и лямбда-выражений, а также представлен необычный взгляд на функции, известный как *карринг*.
- В главе 5 говорится о важности неизменяемых данных, то есть объектов данных, которые невозможно модифицировать после создания. Здесь освещены проблемы, возникающие при наличии изменяемого состояния, и методы создания программ без присваивания переменным новых значений.
- Глава 6 посвящена подробному разбору понятия ленивых вычислений. В ней показано, как ленивый порядок вычислений можно использовать для оптимизации, начиная с простых задач наподобие конкатенации строк и до алгоритмов оптимизации, основанных на методе динамического программирования.
- В главе 7 рассказано о диапазонах – современном дополнении к алгоритмам стандартной библиотеки, призванном повысить удобство восприятия кода и его производительность.
- Глава 8 содержит изложение неизменяемых структур данных, т. е. структур данных, которые хранят историю своих предыдущих состояний, пополняя ее при каждой модификации.

Во второй части книги речь пойдет о более сложных понятиях, по большей части относящихся к разработке программ в функциональном стиле.

- В главе 9 речь идет о том, как избавить программу от недопустимых состояний с помощью суммы типов. Показано, как реализовать сумму типов на основе наследования и шаблона `std::variant` и как

для их обработки использовать перегруженные функциональные объекты.

- Глава 10 посвящена функторам и монадам абстракциям, способным существенно помочь при работе с обобщенными типами и, в частности, позволяющим создавать функции для работы с векторами, значениями типа `optional` и фьючерсами.
- В главе 11 содержится объяснение техник метапрограммирования на шаблонах, способствующих функциональному программированию на языке C++. В частности, разобраны статическая интроспекция, вызываемые объекты и техники метапрограммирования на шаблонах, предназначенные для создания встроенных предметно-ориентированных языков.
- В главе 12 весь предшествующий материал книги собран воедино, чтобы продемонстрировать функциональный подход к разработке параллельных программных систем. В этой главе рассказано, как монаду продолжения можно применить для создания реактивных программ.
- Глава 13 знакомит читателя с функциональным подходом к тестированию и отладке программ.

Автор советует читателю по мере чтения книги реализовывать все изложенные в ней понятия и запускать все встречающиеся примеры кода. Большую часть подходов, описанных в книге, можно использовать и в старых версиях языка C++, однако это потребовало бы написания большого объема дублирующегося кода; поэтому в книге упор сделан главным образом на языковые стандарты C++14 и C++17.

Примеры кода созданы в предположении, что у читателя есть работающий компилятор с поддержкой стандарта C++17. Можно, например, пользоваться компилятором GCC, как автор этой книги, или Clang. Последние выпущенные версии обоих этих компиляторов одинаково хорошо поддерживают все элементы стандарта C++17, использованные в данной книге. Все примеры кода протестированы с версиями<sup>1</sup> GCC 7.2 и Clang 5.0.

Для работы с примерами можно пользоваться обычным текстовым редактором и запускать компилятор вручную из командной строки через посредство утилиты GNU `make` (к каждому примеру приложен несложный сценарий `Makefile`); также можно воспользоваться полновесной интегрированной средой разработки наподобие QtCreator ([www.qt.io](http://www.qt.io)), Eclipse ([www.eclipse.org](http://www.eclipse.org)), Kdevelop ([www.kdevelop.org](http://www.kdevelop.org)) или CLion ([www.jetbrains.com/clion](http://www.jetbrains.com/clion)) и импортировать в нее примеры. Тем, кто намерен пользоваться средой Microsoft Visual Studio, рекомендуется установить наиболее свежую версию, какую возможно загрузить, и настроить ее на применение компилятора Clang вместо используемого по умолчанию компилятора Microsoft Visual C++ (MSVC), которому на момент написания этого введения не хватало некоторых важных мелочей.

---

<sup>1</sup> На момент перевода этого введения актуальны версии GCC 9.1 и Clang 8.0.0. Качество поддержки стандарта C++17 более не должно составлять предмета для беспокойства. – *Прим. перев.*

Хотя для компиляции большей части примеров не нужны никакие внешние зависимости, некоторым примерам все же нужны сторонние библиотеки, такие как `range-v3`, `catch` и `JSON`, клоны которых доступны вместе с кодом примеров в директории `common/3rd-party`, а также коллекция библиотек `Boost`, которую можно загрузить с сайта [www.boost.org](http://www.boost.org).

## Оформление кода и загрузка примеров

Исходный код примеров к этой книге можно загрузить на сайте издательства по адресу [www.manning.com/books/functional-programming-in-c-plus-plus](http://www.manning.com/books/functional-programming-in-c-plus-plus) и с системы GitLab по адресу <https://gitlab.com/manning-fp-cpp-book>.

В книге содержится много примеров исходного кода – как в виде нумерованных листингов, так и короткими вставками в обычный текст. В обоих случаях код набран моноширинным шрифтом наподобие этого, чтобы его легко было отличить от текста на естественном языке.

Во многих случаях первоначальный текст примеров пришлось изменить, добавив разрывы строк и отступы, чтобы улучшить размещение исходного кода на книжной странице. В редких случаях даже этого оказалось недостаточно, и тогда в листинг пришлось ввести знаки продолжения строки `\`. Кроме того, комментарии из исходного кода часто удалялись, если этот код сопровождается подробными пояснениями в основном тексте. Пояснения прилагаются ко многим листингам, помогая понять важные детали.

Спорить о стилях оформления исходного кода – превосходная возможность впустую потратить уйму времени. Это особенно характерно для языка C++, где едва ли не каждый проект претендует на собственный стиль.

Автор старался следовать соглашениям, используемым в ряде других книг по языку C++. Два стилистических правила стоит оговорить здесь особо:

- классы, моделирующие сущности из реального мира, например людей или домашних животных, получают имена с суффиксом `_t`. Благодаря этому становится проще понять в каждом конкретном случае, идет речь о реальном объекте (например, человеке) или о типе – имя `person_t` читается проще, чем «тип «человек»»;
- имена закрытых переменных-членов имеют приставку `m_`. Это отличает их от статических переменных-членов, чьи имена начинаются с префикса `s_`.

## Форум книги

Покупка этой книги дает бесплатный доступ к закрытому форуму, функционирующему под управлением издательства Manning Publications, где читатели могут оставлять комментарии о книге, задавать технические вопросы и получать помощь от автора и других пользователей. Чтобы

получить доступ к форуму, нужно зайти на страницу <https://forums.manning.com/forums/functional-programming-in-c-plus-plus>. Больше о форумах издательства Manning и о принятых там правилах поведения можно узнать на странице <https://forums.manning.com/forums/about>.

Издательство Manning берет на себя обязательство перед своими читателями обеспечить им удобную площадку для общения читателей как между собой, так и с автором книги. Это, однако, не предполагает какого-либо определенного объема участия со стороны автора, чье общение на форуме остается исключительно добровольным и неоплачиваемым. Задавая автору интересные вопросы, читатели могут освежать и подогревать его интерес к теме книги. Форум и архивы предыдущих обсуждений будут доступны на сайте издательства все время, пока книга остается в продаже.

## 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 оставить 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) на странице с описанием соответствующей книги.

## Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете 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 этой книги.

Если вы найдете 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 Manning 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 применить санкции.

Пожалуйста, свяжитесь с нами по адресу [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.





Иван Чукич преподает современные методы программирования на языке C++ и функциональное программирование на факультете математики в Белграде. Он использует язык C++ с 1998 г. Он исследовал функциональное программирование на языке C++ перед и во время подготовки своей диссертации, а также применяет методы функционального программирования в реальных проектах, которыми пользуются сотни миллионов человек по всему миру. Иван – один из ключевых разработчиков среды KDE, крупнейшего в мире проекта с открытым кодом на языке C++.



# Введение в функциональное программирование

# 1

## О чем говорится в этой главе:

- понятие о функциональном программировании;
- рассуждение в терминах предназначения вместо шагов алгоритма;
- понятие о чистых функциях;
- преимущества функционального программирования;
- эволюция C++ в язык функционального программирования.



Каждому программисту приходится изучить за свою жизнь целый ряд языков программирования. Как правило, программист останавливается на двух или трех языках, на которых лично ему удобнее всего работать. Часто можно услышать, что изучить очередной язык программирования просто – что различия между языками в основном касаются лишь синтаксиса и что все языки предоставляют примерно одинаковые возможности. Тот, кто знает язык C++, наверняка легко выучит языки Java и C#, и наоборот.

В этом утверждении есть доля истины. Однако, берясь за изучение нового языка, мы невольно пытаемся имитировать на нем тот стиль программирования, который выработался в связи с предыдущим языком. Когда я впервые начал применять функциональный язык во время учебы в университете, то сразу начал с попыток определить на нем привычные операторы цикла `for` и `while` и оператор ветвления `if-then-else`. Это именно тот подход, которым пользуется большинство из нас, чтобы просто сдать экзамен и никогда больше не возвращаться к изученному.

Широко известен афоризм, что тот, у кого из инструментов есть лишь молоток, будет стремиться любую задачу считать гвоздем. Этот прин-

цип работает и в обратную сторону: если работать только с гвоздями, то любой попавший в руки инструмент будет использоваться как молоток. Многие программисты, попробовав язык функционального программирования, решают, что он не стоит затраченных усилий; они не видят в новом языке преимуществ, поскольку пытаются использовать этот новый инструмент таким же способом, как использовали бы старый.

Цель этой книги состоит не в том, чтобы научить читателя новому языку программирования; вместо этого книга призвана научить иному способу использования старого языка (а именно языка C++) – способу, настолько отличному от привычного, что у программиста *впрямь может возникнуть ощущение*, что он использует новый язык. Этот новый стиль программирования помогает создавать более продуманные программы и писать более безопасный, понятный и читаемый код и даже – не побоюсь сказать – более изящный, чем код, который обычно пишут на языке C++.

## 1.1 Что такое функциональное программирование

Функциональное программирование – это довольно старая парадигма, зародившаяся в академической среде в конце 1950-х годов и долгое время оставшаяся в этой нише. Хотя эта парадигма всегда была излюбленной темой для научных исследований, она никогда не пользовалась популярностью в «реальном мире». Вместо этого повсеместное распространение получили императивные языки: сперва процедурные, затем объектно-ориентированные.

Часто звучали предсказания, что однажды функциональные языки будут править миром, но этого до сих пор не произошло. Наиболее известные языки функционального программирования, такие как Haskell или LISP, все еще не входят в десятку наиболее популярных языков. Первые места по традиции прочно занимают императивные языки, к которым относятся языки C, Java и C++. Это предсказание, подобно большинству других, чтобы считаться сбывшимся, нуждается в известной свободе интерпретации. Вместо того чтобы популярность завоевали языки функционального программирования, произошло нечто иное: в популярные языки программирования стали добавлять все более элементов, заимствованных из языков функционального программирования.

Что собой представляет функциональное программирование (ФП)? На этот вопрос непросто ответить, так как не существует единого общепринятого определения. Согласно известному афоризму, если двух программистов, работающих в функциональном стиле, спросить, что такое ФП, они дадут по меньшей мере три различных ответа. Каждый специалист обычно старается определить ФП через другие связанные с ним понятия: чистые функции, ленивые вычисления, сопоставление с образцом и другие – и обычно при этом перечисляет характеристики своего любимого языка.

Чтобы не оттолкнуть читателя, уже проникшегося симпатией к тому или иному языку, начнем с чисто математического определения, взятого из группы Usenet, посвященной функциональному программированию:

*Функциональное программирование – это стиль программирования, в котором основную роль играет вычисление выражений, а не выполнение команд. Выражения в таких языках образованы из функций, применяемых к исходным значениям. Функциональный язык – это язык, который поддерживает функциональный стиль и способствует программированию в этом стиле.*

– FAQ из группы comp.lang.functional

В последующих главах этой книги будет разобран ряд понятий, относящихся к ФП. Оставим на усмотрение читателя выбор тех из них, которые захочется считать существенными признаками, за которые язык можно назвать *функциональным*.

Говоря шире, ФП – это стиль программирования, в котором основными блоками для построения программы являются функции в противоположность объектам или процедурам. Программа, написанная в функциональном стиле, не задает команды, которые должны быть выполнены для получения результата, а определяет, что есть этот результат.

Рассмотрим небольшой пример: вычисление суммы списка чисел. В императивном мире реализация состоит в том, чтобы перебрать один за другим все элементы списка, прибавляя значение каждого из них к переменной-накопителю. Программный текст представляет собой пошаговое описание процесса суммирования элементов списка. В функциональном же стиле, напротив, нужно лишь определить, что есть сумма чисел, и тогда компьютер сам будет знать, какие операции выполнить, когда ему понадобится вычислить сумму. Один из способов добиться этого состоит в том, что сумма списка чисел есть результат сложения первого элемента списка с суммой оставшейся части списка; кроме того, сумма равна нулю, если список пуст. Тем самым дано определение, что есть сумма, без подробного описания того, как ее вычислить.

Это различие и лежит в основе терминов «императивное программирование» и «декларативное программирование». *Императивное* программирование означает, что программист указывает компьютеру выполнить нечто, в явном виде описав каждый шаг, необходимый для вычисления результата. *Декларативное* же означает, что программист описывает требования к ожидаемому результату, а на язык программирования ложится ответственность за отыскание конкретного способа его получения. Программист определяет, что собой представляет сумма списка чисел, а язык использует это определение, чтобы вычислить сумму конкретного списка.

### 1.1.1 Соотношение функционального программирования с объектно-ориентированным

Невозможно однозначно сказать, какая из двух парадигм лучше: наиболее популярная из императивных – объектно-ориентированная (ООП) – или самая популярная из декларативных, т. е. парадигма ФП. У обеих есть свои сильные и слабые стороны.

Основная идея объектно-ориентированной парадигмы – это создание абстракций над данными. Этот механизм позволяет программисту скрывать внутреннее представление данных внутри объекта, предоставляя остальному миру лишь видимую его сторону, доступную через интерфейс.

В центре внимания ФП лежит создание абстракций над функциями. Это позволяет создавать структуры управления, более сложные, чем те, что непосредственно поддерживаются языком программирования. Когда в стандарт C++11 был добавлен цикл `for` по диапазону (часто называемый также циклом `foreach`), его поддержку нужно было реализовать во всех компиляторах, которых в мире имеется довольно много. С помощью методов ФП часто удается сделать нечто подобное без модификации компилятора. За прошедшие годы во многих сторонних библиотеках были реализованы собственные версии цикла по диапазону. Используя идиомы ФП, можно создавать новые языковые конструкции – например, цикл `for` по диапазону или иные, более сложные. Эти конструкции могут пригодиться, даже если разработка в целом ведется в императивном стиле.

В некоторых случаях удобнее оказывается одна парадигма программирования, в иных – другая. Часто самым верным решением оказывается комбинация обеих парадигм. Это видно уже из того факта, что многие старые и новые языки программирования стали поддерживать несколько парадигм, вместо того чтобы хранить верность своей первоначальной парадигме.

### 1.1.2 Сравнение императивной и декларативной парадигм на конкретном примере

Чтобы продемонстрировать различие между этими двумя стилями программирования, начнем с простой программы, написанной в императивном стиле, и преобразуем ее в эквивалентную функциональную программу. Одна из часто применяемых мер сложности программного обеспечения – это число строк кода (англ. *lines of code*, LOC). Хотя в общем случае адекватность этого показателя остается под вопросом, он превосходно подходит в качестве примера для демонстрации различий между императивным и функциональным стилями.

Предположим, нужно написать функцию, которая принимает на вход список файлов и подсчитывает число строк в каждом из них (рис. 1.1).

Чтобы сделать пример как можно проще, будем подсчитывать в каждом файле лишь число символов перевода на новую строку в предположении, что последняя строка файла непременно заканчивается этим символом.

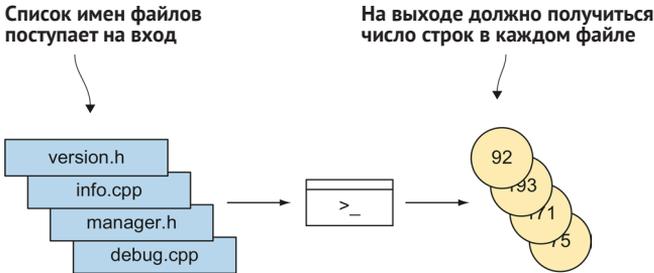


Рис. 1.1 Программа для подсчета числа строк в файлах

Рассуждая по-императивному, можно представить алгоритм решения этой задачи в виде такого списка шагов:

- 1 Открыть каждый файл.
- 2 Проинициализировать нулем счетчик строк.
- 3 Читать из файла символ за символом и наращивать счетчик всякий раз, когда попадаетея символ новой строки (`\n`).
- 4 Дойдя до конца файла, запомнить полученное значение счетчика строк.

Следующий листинг содержит реализацию описанной выше логики.

#### Листинг 1.1 Подсчет числа строк, императивное решение

```
std::vector<int>
count_lines_in_files(const std::vector<std::string>& files)
{
    std::vector<int> results;
    char c = 0;

    for (const auto& file : files) {
        int line_count = 0;

        std::ifstream in(file);

        while (in.get(c)) {
            if (c == '\n') {
                line_count++;
            }
        }

        results.push_back(line_count);
    }

    return results;
}
```



Получилось два вложенных цикла и несколько переменных, в которых хранится текущее состояние процесса. Несмотря на всю простоту этого примера, в нем есть несколько мест, где нетрудно ошибиться: например, можно оставить переменную непроинициализированной (или проинициализировать ее неправильно), неправильно изменить значение счетчика или неправильно указать условие цикла. Компилятор может выдать предупреждения о некоторых из них, но те, что пройдут компиляцию незамеченными, обычно с трудом обнаруживаются человеком, поскольку наш мозг устроен так, чтобы не замечать их, как мы не замечаем опечатки. Поэтому код лучше всего писать таким образом, чтобы свести к минимуму саму возможность подобных ошибок.

Читатели, более искушенные в языке C++, наверняка заметили, что в этом примере можно было воспользоваться стандартным алгоритмом `std::count`, вместо того чтобы подсчитывать переводы строки вручную. Стандартная библиотека языка C++ предоставляет ряд удобных абстракций, например итераторы по потокам, которые позволяют обращаться с потоками ввода-вывода таким же образом, как с обычными коллекциями: векторами и списками. Покажем пример их использования.

**Листинг 1.2** Использование алгоритма `std::count` для подсчета символов перевода строки

```
int count_lines(const std::string& filename)
{
    std::ifstream in(filename);

    return std::count(
        std::istreambuf_iterator<char>(in),
        std::istreambuf_iterator<char>(),
        '\n');
}
```



Подсчитывает символы перевода строки от текущей позиции в файловом потоке до его конца

```
std::vector<int>
count_lines_in_files(const std::vector<std::string>& files)
{
    std::vector<int> results;

    for (const auto& file : files) {
        results.push_back(count_lines(file)); ← Сохраняет результаты
    }

    return results;
}
```

С такой реализацией программист избавлен от необходимости знать детали того, как реализован подсчет. Программный код теперь прямо выражает, что нужно подсчитать количество символов перехода на новую строку в заданном входном потоке. В этом проявляется основная идея функционального стиля программирования: использовать абстракции, позволяющие описать *цель* вычисления, вместо того чтобы подробно

описывать *способ* ее достижения, – и именно этой цели служит большая часть техник, описанных в данной книге. По этой же причине парадигма функционального программирования тесно соседствует с обобщенным программированием, особенно если речь идет о языке C++: оба подхода позволяют программисту мыслить на более высоком уровне абстракции по сравнению с приземленным императивным взглядом на программу.



### Можно ли назвать язык C++ объектно-ориентированным?

Меня всегда удивляло, что большинство разработчиков называют C++ объектно-ориентированным языком. Это удивительно потому, что в стандартной библиотеке C++ (которую часто называют стандартной библиотекой шаблонов – англ. Standard Template Library, STL) почти нигде не используется полиморфизм, основанный на наследовании, – ключевой элемент парадигмы ООП.

Библиотеку STL создал Александр Степанов, известный критик ООП. Он захотел создать библиотеку для обобщенного программирования и сделал это, воспользовавшись системой шаблонов языка C++ в сочетании с методами функционального программирования.

В этом состоит одна из причин столь широкого использования библиотеки STL в данной книге: хотя она и не является библиотекой для функционального программирования в чистом виде, в ней хорошо смоделированы многие понятия ФП, что делает библиотеку STL превосходной отправной точкой для вхождения в функциональный мир.

Преимущество этого подхода в том, что программисту нужно держать в голове меньше переменных состояния и становится возможным заняться тем, чтобы на высоком уровне абстракции формулировать предназначение программы – вместо подробного описания мелких шагов, которые необходимо сделать для получения нужного результата. Не нужно больше задумываться о том, как реализован подсчет. Единственная забота функции `count_lines` состоит в том, чтобы взять поступивший на вход аргумент (имя файла) и превратить его в нечто, понятное для функции `std::count`, т. е. в пару итераторов по потоку.

Теперь продвинемся еще на шаг в том же направлении и определим весь алгоритм в функциональном стиле, описав, *что* должно быть сделано, вместо описания того, *как* это сделать. В предыдущей реализации остался цикл `for` по диапазону, который применяет функцию ко всем элементам коллекции и собирает результаты в контейнер. Это широко распространенный шаблон, и было бы естественно ожидать его поддержки стандартной библиотекой языка программирования. В языке C++ именно для этого предназначен алгоритм `std::transform` (в других языках похожая функция обычно называется `map` или `fmap`). Реализация разобранной выше логики на основе алгоритма `std::transform` показана в следующем листинге. Функция `std::transform` перебирает элементы коллекции `files` один за другим, преобразовывает каждый из них по-

средством функции `count_lines`, а полученные результаты складывает в вектор `results`.

### Листинг 1.3 Отображение имен файлов в счетчики строк функцией `std::transform`

```
std::vector<int>
count_lines_in_files(const std::vector<std::string>& files)
{
    std::vector<int> results(files.size());

    std::transform(files.cbegin(), files.cend(), ← Откуда брать исходные данные
                  results.begin(), ← Куда сохранять результаты
                  count_lines); ← Функция-преобразователь

    return results;
}
```

В этом коде описан уже не пошаговый алгоритм, а, скорее, преобразование, которому нужно подвергнуть входные данные, чтобы получить желаемый результат. Можно утверждать, что, избавив программу от переменных состояния и взяв за основу стандартную реализацию алгоритма подсчета вместо изобретения собственной, мы получим код, более устойчивый к потенциальным ошибкам.

Получившийся листинг все еще имеет недостаток: в нем слишком много клише, чтобы его можно было считать более удобочитаемым, чем первоначальный вариант. На самом деле в этом коде лишь три важных слова:

- `transform` – что код делает<sup>1</sup>;
- `files` – входные данные;
- `count_lines` – функция-преобразователь.

Все прочее – шелуха.

Эта функция могла бы стать гораздо более удобочитаемой, если бы можно было оставить только важные фрагменты и опустить все остальное. Как читатель увидит в главе 7, этого можно добиться с помощью библиотеки `ranges`. Покажем здесь, как будет выглядеть эта функция, если реализовать ее на основе библиотеки `ranges`, через диапазоны и их преобразования. Операция конвейера, обозначаемая вертикальной чертой (`|`), используется в библиотеке `ranges`, для того чтобы подать коллекцию на вход преобразователя.

### Листинг 1.4 Преобразование коллекции средствами библиотеки `ranges`

```
std::vector<int>
count_lines_in_files(const std::vector<std::string>& files)
{
    return files | transform(count_lines);
}
```

<sup>1</sup> То есть поэлементное преобразование контейнера исходных данных в контейнер результатов. – *Прим. перев.*

Этот код делает то же самое, что и код из листинга 1.3, но смысл в данном случае более очевиден. Функция берет поступивший на вход список имен файлов, пропускает каждый элемент через функцию-преобразователь и возвращает список результатов.

### Соглашения о способе записи типов функций



В языке C++ нет единого типа, представляющего понятие функции (в главе 3 речь пойдет обо всем разнообразии сущностей, которые в языке C++ могут рассматриваться как функции). Чтобы иметь возможность зафиксировать исключительно типы аргументов и возвращаемого значения функции, не задавая при этом точный тип, который она получит в языке C++, нам понадобится новая, отвлеченная от языка система обозначений.

Когда впредь мы будем писать  $f: (arg1\_t, arg2\_t, \dots, argn\_t) \rightarrow result\_t$ , это будет означать, что  $f$  есть функция, принимающая  $n$  аргументов, первый из которых имеет тип  $arg1\_t$ , второй – тип  $arg2\_t$  и т. д., причем возвращает эта функция значение типа  $result\_t$ . Если функция принимает лишь один аргумент, скобки вокруг его типа будем опускать. Кроме того, не будем пользоваться в этих обозначениях константными ссылками.

Например, если написано, что функция `repeat` имеет тип  $(char, int) \rightarrow std::string$ , это означает, что она принимает два аргумента: один символьного типа и один целочисленного – и возвращает строку. В языке C++ это можно было бы записать, например, так (второй вариант допустим начиная со стандарта C++11):

```
std::string repeat(char c, int count);
auto repeat(char c, int count) -> std::string;
```

Реализация на основе диапазонов и их преобразований также упрощает поддержку кода. Читатель мог заметить, что функция `count_lines` страдает проектным недостатком. Глядя лишь на ее имя и тип `count_lines: std::string  $\rightarrow$  int`, легко понять, что она принимает один аргумент текстового типа, но совершенно не очевидно, что этот текст должен представлять собой имя файла. Естественно было бы предположить, что функция подсчитывает число строк в поступившем на вход тексте. Чтобы избавиться от этого недочета, можно функцию разбить на две: функцию `open_file: std::string  $\rightarrow$  std::ifstream`, которая принимает имя файла и возвращает открытый файловый поток, и функцию `count_lines: std::ifstream  $\rightarrow$  int`, которая подсчитывает строки в заданном потоке символов. После такого разбиения предназначение функций становится совершенно очевидно из их имен и типов аргументов и возвращаемых значений. При этом построенная на диапазонах реализация функции `count_lines_in_files` потребует вставки лишь еще одного преобразования.

**Листинг 1.5 Преобразование коллекции средствами библиотеки `ranges`, модифицированный вариант**

```
std::vector<int>
count_lines_in_files(const std::vector<std::string>& files)
{
    return files | transform(open_file)
               | transform(count_lines);
}
```

В итоге решение получилось гораздо менее многословным, чем императивное решение из листинга 1.1, и гораздо более очевидным. На вход поступает коллекция имен файлов – не имеет значения даже, какой тип коллекции используется, для каждого ее элемента выполняются подряд два преобразования. А именно сначала из имени файла создается файловый поток, затем в потоке подсчитываются символы перевода строки. Именно это в точности выражает приведенный здесь код, без обременительных подробностей и дублирующегося кода.

## 1.2 Чистые функции

Один из главных источников ошибок при программировании – это наличие у программы изменяемого состояния. Очень сложно проследить все возможные состояния, в которых программа может находиться. Парадигма ООП дает возможность сгруппировать отдельные части состояния в объекты, тем самым помогая человеку понять состояния и управлять ими. Однако этот подход не в силах существенно уменьшить число возможных состояний.

Представим себе разработку текстового редактора. Набранный пользователем текст нужно сохранить в переменную. Предположим, что пользователь нажимает кнопку **Сохранить** и продолжает набор текста. Программа тем временем сохраняет текст, отправляя символы в файл по одному (эта картина, конечно, сильно упрощена, но автор просит читателя запастись терпением). Что произойдет, если пользователь изменит часть текста, пока программа занимается его сохранением в файл? Сохранит ли программа текст в том виде, в котором он пребывал на момент нажатия кнопки **Сохранить**, или в его нынешнем виде, или сделает что-то иное?

Проблема состоит в том, что все три случая могут произойти – это зависит от того, насколько далеко продвинулась запись текста в файл и в какую часть текста пользователь внес изменение. Так, в случае, изображенном на рис. 1.2, программа запишет в файл текст, которого никогда не было в редакторе.

В файл будут сохранены участки, взятые из текста, до того, как пользователь внесет свои изменения, другие участки попадут в файл из уже измененного текста. Таким образом, в файл вместе попадут куски двух различных состояний.

Этот блок текста сохранен до того, как пользователь начал менять текст



Затем пользователь начинает менять выделенный фрагмент

```
On a withe
red branch
A crow has
alighted:
Nightfall
in autumn.
```

Если запись текста в файл продолжится, будто ничего не случилось, получится файл, содержащий фрагменты старой версии текста вместе с фрагментами новой, т. е. в файл будет записан текст, которого никогда не существовало в редакторе



Рис. 1.2 Повреждение файла в результате модификации текста во время записи

Описанная ситуация просто не могла бы возникнуть, если бы функция сохранения текста в файл обладала собственной неизменяемой копией данных, которые ей необходимо записать (рис. 1.3). В этом состоит наиболее серьезная проблема изменяемых состояний: они создают зависимости между частями программы, которые никак не должны быть связаны между собой. Данный пример включает в себя два очевидно различных пользовательских действия: набор текста и сохранение набранного текста. Эти действия должны выполняться независимо друг от друга. Параллельное выполнение двух или более действий, относящихся к одному и тому же общему состоянию, создает нежелательную зависимость между действиями и способно привести к проблемам, подобным описанной выше.

Майкл Фезерс (Michael Feathers), автор книги «Эффективная работа с устаревшим кодом» (Working Effectively with Legacy Code – Prentice Hall, 2004), писал: «ООП делает код более понятным за счет инкапсуляции движущихся частей. ФП делает код более понятным, сокращая число движущихся частей». Даже локальные изменяемые переменные стоит считать нежелательными по той же причине. Они создают зависимости между различными частями функции, затрудняя выделение тех или иных ее участков в отдельные вспомогательные функции.

Одно из наиболее мощных понятий ФП – это понятие *чистой функции*, т. е. функции, которая использует лишь значения поступающих на вход аргументов для вычисления результата, но не изменяет аргументы. Если чистую функцию вызывают несколько раз с одними и теми же аргументами, она обязана возвращать одно и то же значение и не должна оставлять никаких следов своего вызова (т. е. не должна производить *побочные эффекты*). Все это означает, что чистые функции не могут менять состояние программы.

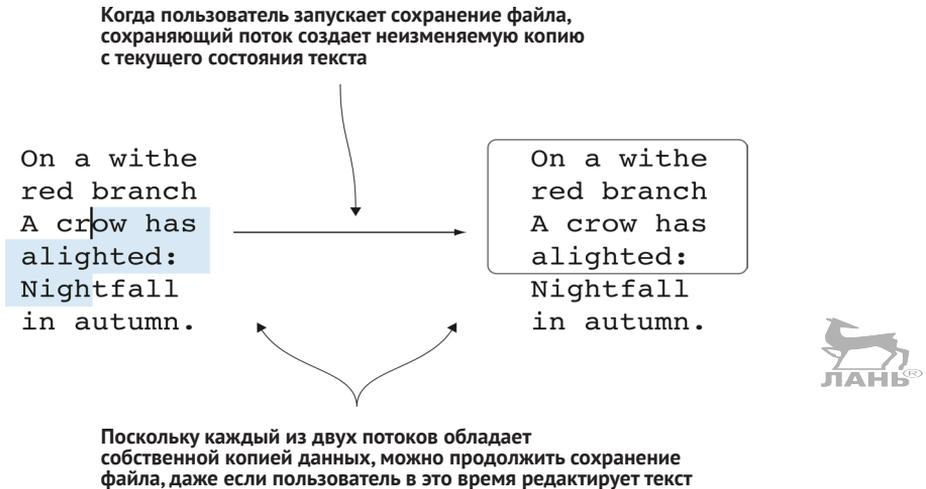


Рис. 1.3 Создание полной копии данных или использование структуры данных, запоминающей историю своих версий, позволяет разорвать зависимость между редактированием и сохранением текста

Чистые функции – это прекрасно, поскольку в этом случае программисту вообще не нужно заботиться о состоянии программы. Однако, к сожалению, чистота также означает, что функция не может читать из стандартного потока ввода, писать в стандартный поток вывода, создавать или удалять файлы, вставлять строки в базу данных и т. д. Если стремление к чистоте функций довести до предела, следовало бы запретить функциям даже изменять содержимое регистров процессора, оперативной памяти и тому подобного на аппаратном уровне.

Все это делает определение чистых функций практически непригодным для реального использования. Центральный процессор выполняет команды одна за другой, и ему необходимо следить за тем, какую команду выполнять следующей. Ничего нельзя выполнить на компьютере без того, чтобы изменить, по меньшей мере, внутреннее состояние процессора. Кроме того, невозможно создать сколько-нибудь полезную программу без взаимодействия с пользователем или другой программной системой.

Вследствие этого нам придется несколько ослабить требования и уточнить определение: *чистой* будем называть такую функцию, у которой отсутствуют наблюдаемые (на некотором высоком уровне) побочные эффекты. Клиентский код, вызывающий функцию, должен быть не в состоянии обнаружить какие-либо следы того, что функция была на самом деле выполнена, за исключением возврата ей значения. В этой книге мы не будем ограничивать себя использованием и написанием одних лишь чистых функций, однако будем всячески стараться к ограничению числа нечистых функций в наших программах.

## 1.2.1 Устранение изменяемого состояния

Разговор об ФП как стиле программирования начинался с разбора императивной реализации алгоритма, подсчитывающего символы перевода строки в коллекции файлов. Функция подсчета строк должна всегда возвращать один и тот же контейнер целых чисел, если ее многократно вызывать для одного и того же списка файлов (при условии что никакая внешняя функция не меняет сами файлы). Это означает, что нашу функцию можно реализовать чистым образом.

Глядя на первоначальную реализацию этой функции, показанную в листинге 1.1, можно увидеть лишь несколько операторов, нарушающих требование чистоты:

```
for (const auto& file : files) {
    int line_count = 0;

    std::ifstream in(file);

    while (in.get(c) {
        if (c == '\n') {
            line_count++;
        }
    }

    results.push_back(line_count);
}
```



Вызов метода `.get` для входного потока меняет как сам поток, так и значение, хранящееся в переменной `c`. Далее этот код изменяет контейнер `results`, добавляя в конец новые значения, и модифицирует переменную `line_count`, наращивая ее на единицу. На рис. 1.4 показано, как меняется состояние при обработке одного файла. Таким образом, данная функция, очевидно, реализована отнюдь не чистым образом.

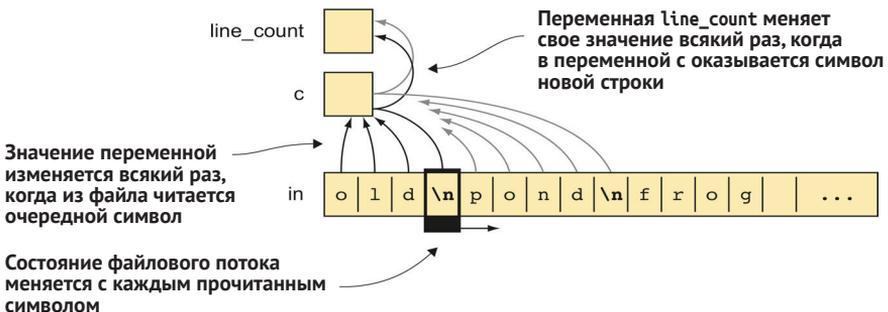


Рис. 1.4 Модификация нескольких независимых переменных при подсчете числа строк в файле

Однако наличие изменяющегося состояния – это не все, о чем следует задуматься. Еще один важный аспект – проявляется ли вовне нечистый

характер функции. В данном случае все изменяющиеся переменные локальны, через них не могут взаимодействовать даже параллельные вызовы этой функции, тем более они невидимы для вызывающего кода и каких бы то ни было внешних сущностей. Поэтому пользователи этой функции могут считать ее чистой, несмотря на то что ее внутренняя реализация не такова. Выгода для вызывающего кода очевидна, так как они могут быть уверены, что функция не изменит их состояния. С другой стороны, автор функции должен позаботиться о корректности ее внутреннего состояния. При этом нужно также удостовериться, что функция не изменяет ничего, что ей не принадлежит. Конечно, лучше всего было бы ограничить даже внутреннее состояние и сделать реализацию функции как можно более чистой. Если при реализации программы удастся использовать исключительно чистые функции, то вообще не нужно беспокоиться о возможных паразитных влияниях между состояниями, поскольку никаких изменяемых состояний в программе нет.

Во втором решении (листинг 1.2) подсчет выделен в функцию с именем `count_lines`. Эта функция снаружи выглядит чистой, хотя ее внутренняя реализация создает и модифицирует поток ввода. К сожалению, это лучшее, чего можно добиться с помощью класса `std::ifstream`:

```
int count_lines(const std::string& filename)
{
    std::ifstream in(filename);

    return std::count(
        std::istreambuf_iterator<char>(in),
        std::istreambuf_iterator<char>(),
        '\n');
}
```



Этот шаг не приводит к сколько-нибудь существенному улучшению функции `count_lines_in_files`. Некоторые элементы реализации, противоречащие чистоте функции, перемещаются из нее в другое место, однако в ней по-прежнему остается изменяемое состояние: переменная `results`. Функция `count_lines_in_files` сама не осуществляет никакого ввода-вывода и реализована исключительно на основе функции `count_lines`, которая с точки зрения вызывающего контекста выглядит чистой. Поэтому нет никаких причин, по которым функции `count_lines_in_files` следовало бы быть нечистой. Следующая версия кода, основанная на диапазонах, представляет собой реализацию функции `count_lines_in_files` без какого бы то ни было локального состояния (изменяемого или нет). Эта реализация определяет функцию через вызов другой функции на поступившем на вход контейнере:

```
std::vector<int>
count_lines_in_files(const std::vector<std::string>& files)
{
    return files | transform(count_lines);
}
```

Данное решение – превосходный пример того, как должно выглядеть программирование в функциональном стиле. Код получился кратким и содержательным, а принцип его работы – вполне очевидным. Более того, код с полной очевидностью не делает ничего иного: у него нет видимых извне побочных эффектов. Он всего лишь вычисляет желаемый результат на заданных входных данных.



## 1.3 Функциональный стиль мышления

Писать код сначала в императивном стиле, а затем преобразовывать его по кусочкам, пока он не превратится в функциональный, было бы неэффективно и контрпродуктивно. Напротив, следует с самого начала по-иному осмысливать задачу. Вместо того чтобы рассуждать о шагах, которые должен выполнить алгоритм, нужно подумать, что собой представляют входные данные и искомый результат и какие преобразования нужно выполнить, чтобы отобразить одно в другое.



В примере, показанном на рис. 1.5, даны имена файлов и нужно подсчитать число строк в каждом из них. Первое, что бросается здесь в глаза: задачу можно упростить, обрабатывая каждый раз по одному файлу. Хотя на вход поступил целый список имен файлов, каждое из них можно рассматривать независимо от остальных. Если изобрести решение задачи для единственного файла, можно легко решить также и исходную задачу (рис. 1.6).

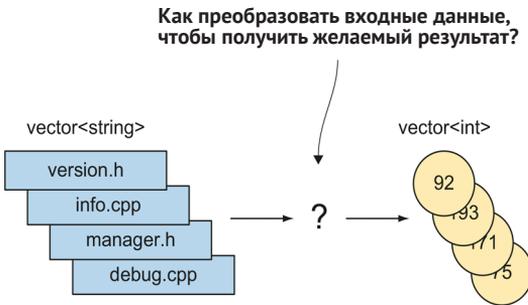


Рис. 1.5 Рассуждая о задаче в функциональном стиле, нужно думать о том, какое преобразование применить к входным данным, чтобы получить желаемый результат

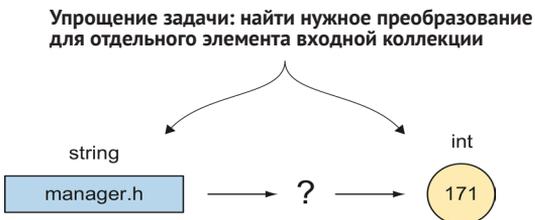


Рис. 1.6 Одно и то же преобразование применяется к каждому элементу коллекции. Это позволяет заняться более простой задачей: преобразованием единственного элемента вместо преобразования всей коллекции

Теперь важнейшей задачей становится создание функции, которая принимает на вход имя файла и подсчитывает число строк в файле с этим именем. Из этого определения вполне ясно, что на вход функции поступает что-то одно (имя файла), но для работы нужно ей что-то другое (а именно содержимое файла, чтобы было в чем подсчитывать символы перевода строки). Следовательно, нужна вспомогательная функция, которая может, получив на вход имя файла, предоставить его содержимое. Должно ли это содержимое предоставляться в виде строки, файлового потока или чего-то иного, остается всецело на усмотрение программиста. Нужна всего лишь возможность получать из этого содержимого по одному символу за раз, чтобы их можно было подать на вход функции, подсчитывающей среди них символы перевода строки.

Имея в руках функцию, которая по заданному имени файла отдает его содержимое (ее типом будет `std::string` → `std::ifstream`), к ее результату можно применить другую функцию, которая в заданном содержимом считает строки (т. е. функцию типа `std::ifstream` → `int`). Сочленение этих двух функций путем подачи файлового потока, созданного первой, на вход второй как раз и дает необходимую нам функцию, см. рис. 1.7.

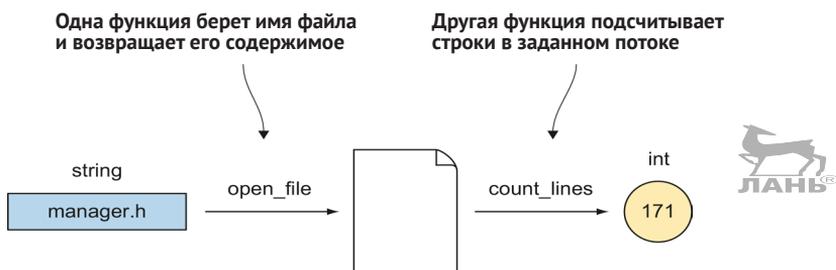


Рис. 1.7 Сложную задачу подсчета строк в файле с заданным именем можно разбить на две простые: открытие файла по заданному имени и подсчет строк в открытом файловом потоке

Тем самым задача подсчета строк в одном файле решена. Для решения же исходной задачи остается *поднять* (англ. lift) эти функции, чтобы они могли работать не над отдельными значениями, а над коллекциями значений. В сущности, это именно то, что делает функция `std::transform` (пусть и с более сложным интерфейсом): она берет функцию, которую можно применять к отдельно взятому значению, и создает на ее основе преобразование, которое можно применять к целым коллекциям значений, – см. рис. 1.8. Пока что читателю лучше всего представить себе *подъем* (англ. lifting) как обобщенный способ превратить функцию, оперирующую отдельными простыми значениями некоторого типа, в функцию, работающую с более сложными структурами данных, содержащими значения этого типа. Более подробно речь о подъеме функций пойдет в главе 4.



Ранее созданы функции, обрабатывающие одно значение за раз. Поднимая их с помощью функции `transform`, получаем функции, обрабатывающие целые коллекции значений

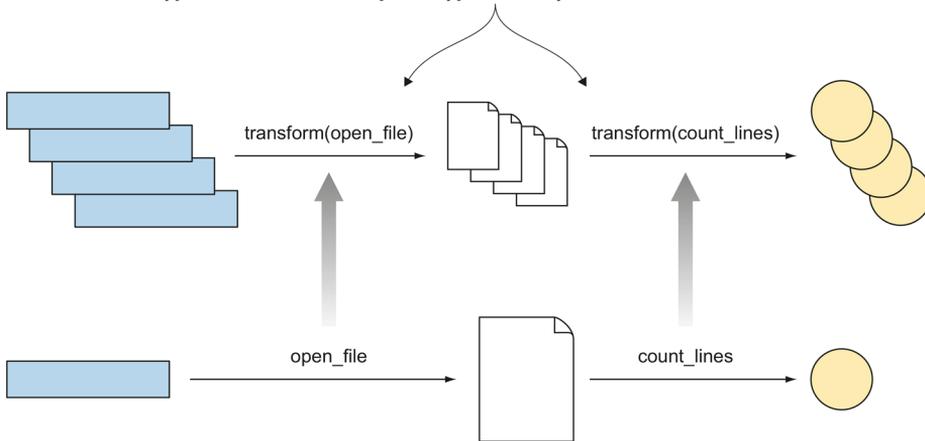


Рис. 1.8 С помощью функции `transform` можно создавать функции обработки коллекций на основе функций, обрабатывающих одно значение

Из этого простого примера хорошо виден функциональный подход к разбиению больших задач программирования на меньшие независимые подзадачи, решения которых затем легко сочленяются. Одна из полезных аналогий, помогающих представить себе композицию функций и их подъем, – это аналогия с движущейся лентой конвейера (рис. 1.9). В начале ленты находится сырье, в конце должно быть получено готовое изделие. Лента продвигается от станка к станку, каждый из которых совершает какое-то свое преобразование, пока не получится изделие в своем окончательном виде. Глядя на конвейер как целое, естественно рассуждать о цепочке преобразований, превращающей сырье в изделие, а не об элементарных действиях, совершаемых каждой машиной.

В этой аналогии сырье соответствует входным данным, а станки – функциям, применяемым последовательно к этим данным. Каждая функция узко специализирована для выполнения одной простой задачи и ничего не должна знать об остальной части конвейера. Каждой функции нужны только правильные данные на вход – но ей совершенно не важно, откуда они получены. Входные значения помещаются на конвейер одно за другим (или, возможно, на несколько параллельно работающих конвейерных лент, что позволяет обрабатывать несколько значений одновременно). Каждое значение подвергается преобразованиям, и с другого конца конвейера снимается коллекция результатов.

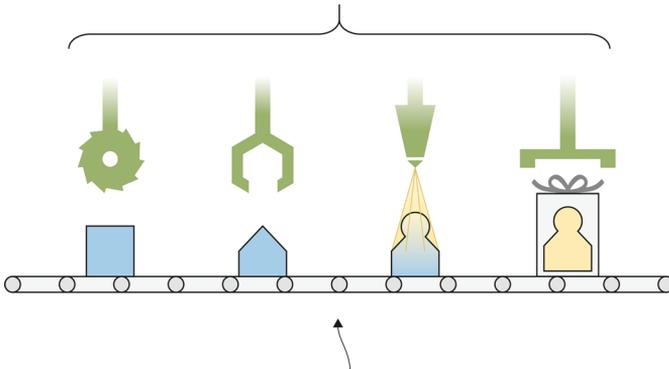
## 1.4 Преимущества функционального программирования

Каждый из многочисленных аспектов ФП обещает свои выгоды. Эти выгоды будут разобраны на протяжении всей книги, здесь же начнем с не-

скольких важнейших преимуществ, которые обеспечиваются большинством механизмов ФП.

Самое очевидное обстоятельство, которое отмечает большинство программистов, едва начав создавать программы в функциональном стиле, состоит в том, что код программ становится значительно короче. В некоторых проектах даже можно встретить разбросанные по коду комментарии наподобие «на языке Haskell это можно было бы реализовать в одну строку». Это происходит потому, что средства, предоставляемые функциональными языками, просты и в то же время очень выразительны, большую часть решения задачи они позволяют описать на высоком уровне абстракции, не вдаваясь в мелкие детали.

Различные преобразования одно за другим применяются к поступающему на вход объекту. Вместе получается композиция всех этих преобразований



Помещая на движущуюся ленту конвейера множество заготовок, получим подъем композиции преобразований от обработки отдельных объектов до обработки целых коллекций

Рис. 1.9 Композицию функций и их подъем можно сравнить с движущейся лентой конвейера. Каждое отдельное преобразование работает над одним объектом. Подъем позволяет этим преобразователям работать над коллекциями объектов. Композиция подает результат одного преобразования на вход другого. В целом получается конвейер, применяющий последовательность операций к любому числу исходных объектов

Именно это свойство, вместе со свойством чистоты, стало привлекать к функциональному программированию в последние годы все больше внимания. Чистота способствует корректности кода, а выразительность позволяет писать меньше кода, то есть оставляет меньше места для ошибок.

### 1.4.1 Краткость и удобочитаемость кода

Приверженцы функциональной парадигмы обычно утверждают, что программы, написанные в функциональном стиле, проще понимать. Это суждение субъективно, и люди, привыкшие писать и читать императив-

ный код, могут с ним не согласиться. Объективно же можно утверждать, что программы, написанные в функциональном стиле, обычно оказываются короче и лаконичнее. Это с полной очевидностью проявилось в разобранном примере: начав с двадцати строк императивного кода, мы пришли к единственной строке кода для функции `count_lines_in_files` и примерно пяти строчкам для функции `count_lines`, из которых большую часть составляет клише, навязанное языком C++ и библиотекой STL. Достичь такого впечатляющего сжатия кода удалось благодаря использованию высокоуровневых абстракций, предоставляемых той частью библиотеки STL, что ориентирована на поддержку ФП.

Одна из печальных истин состоит в том, что многие программисты, работающие на языке C++, сторонятся использования высокоуровневых абстракций наподобие алгоритмов из библиотеки STL. У них могут быть для этого различные причины, от стремления написать своими руками более эффективный код до желания избежать кода, который коллегам было бы тяжело понять. Иногда эти причины не лишены смысла, но отнюдь не в большинстве случаев. Отказ от пользования наиболее передовыми возможностями своего языка программирования снижает мощь и выразительность языка, делает код более запутанным, а его поддержку – более трудоемкой.

В 1968 г. Эдсгер Дейкстра опубликовал знаменитую статью «О вреде оператора `goto`»<sup>1</sup>. В ней он выступал за отказ от оператора перехода `goto`, который в ту эпоху использовался чрезмерно, в пользу высокоуровневых средств структурного программирования, включая процедуры, операторы ветвления и цикла.

*Разнузданное применение оператора `go to` имеет прямым следствием то, что становится ужасно трудно найти осмысленный набор координат, в которых описывается состояние процесса... Оператор `go to` сам по себе просто слишком примитивен; он создает очень сильное побуждение внести путаницу в программу»<sup>2</sup>.*

Однако во многих случаях операторы цикла и ветвления тоже оказываются слишком примитивными. Как и оператор `goto`, циклы и ветвления могут сделать программу сложной для написания и понимания, и их часто удается заменить более высокоуровневыми конструкциями в функциональном стиле. Программисты, привычные к императивному стилю, часто дублируют один и тот же, по сути, код в разных местах программы, даже не замечая, что он одинаков, так как он работает с данными разных типов или имеет мелкие различия в поведении, которые с точки зрения функционального стиля легко было бы вынести в параметр.

<sup>1</sup> *Edsger Dijkstra. Go To Statement Considered Harmful // Communications of the ACM: journal. 1968. March (vol. 11, no. 3). P. 147–148.*

<sup>2</sup> Цит. по русскому переводу: <http://hosting.vspu.ac.ru/~chul/dijkstra/goto/goto.htm>. – Прим. перев.

Используя существующие абстракции, предоставляемые библиотекой STL или какими-либо сторонними библиотеками, или создавая собственные абстракции, программист может сделать код своих программ надежнее и короче. Кроме того, проще становится обнаруживать ошибки в этих абстракциях, так как один и тот же абстрактный код будет использоваться во множестве разных мест в программе.

## 1.4.2 Параллельная обработка и синхронизация

Главную трудность при разработке параллельных систем составляет управление общим изменяемым состоянием. От программиста требуется особая внимательность и аккуратность, чтобы обеспечить согласованную работу разных частей системы и целостность состояния.

Распараллеливание программ, созданных на основе чистых функций, напротив, тривиально, поскольку эти функции не изменяют никакого состояния. Поэтому нет необходимости синхронизировать их своими руками с помощью атомарных переменных или семафоров. Код, написанный для однопоточной системы, можно почти без изменений запускать в несколько потоков. Более подробно речь об этом пойдет в главе 12.

Рассмотрим следующий фрагмент кода, который суммирует квадратные корни значений, хранящихся в векторе `xs`:

```
std::vector<double> xs = {1.0, 2.0, ...};  
auto result = sum(xs | transform(sqrt));
```

Если реализация функции `sqrt` чистая (а для иного причин нет), реализация функции `sum` может автоматически разбить вектор входных данных на сегменты и сумму каждого из них вычислить в отдельном потоке. Когда все потоки завершатся, останется лишь собрать и просуммировать их результаты.

К сожалению, в языке C++ (по крайней мере, пока) отсутствует понятие чистой функции, поэтому распараллеливание не может выполняться автоматически. Вместо этого программисту следует в явном виде вызвать параллельную версию функции `sum`. Параллельная функция `sum` может даже сама на лету определить число процессорных ядер и, исходя из этого, решить, на сколько сегментов разбить вектор `xs`. Если бы суммирование корней было реализовано в императивном стиле, через цикл `for`, его бы не удалось распараллелить столь просто. Тогда пришлось бы позаботиться о том, чтобы переменные не модифицировались одновременно разными потоками, об определении оптимального числа потоков для той системы, на которой система выполняется, – вместо того чтобы предоставить все это библиотеке, реализующей алгоритм суммирования.

**ПРИМЕЧАНИЕ** Компиляторы языка C++ иногда способны сами выполнять векторизацию и ряд других оптимизаций, если обнаруживают, что тело цикла чистое. Эти оптимизации часто затрагивают код, использующий стандартные алгоритмы, поскольку они обычно бывают реализованы на основе циклов.

### 1.4.3 Непрерывная оптимизация

Использование высокоуровневых программных абстракций из библиотеки STL или других заслуживающих доверия библиотек приносит еще одну ощутимую выгоду: программа может со временем совершенствоваться, даже если автор не меняет в ее коде ни строчки. Каждое усовершенствование в языке программирования, реализации компилятора или в реализации используемой программой библиотеки будет автоматически приводить к совершенствованию программы. Хотя это равно справедливо как для функциональных, так и для нефункциональных абстракций высокого уровня, само использование понятий ФП в программе существенно увеличивает долю кода, основанного на этих абстракциях.

Это преимущество может показаться самоочевидным, но многие программисты до сих пор предпочитают вручную писать критичные по производительности участки низкоуровневого кода, иногда даже на языке ассемблера. Этот подход может быть оправдан, но в большинстве случаев он позволяет оптимизировать код лишь для конкретной вычислительной платформы и практически лишает компилятор возможности самому оптимизировать код для всех остальных платформ.

Рассмотрим для примера функцию `sum`. Ее можно оптимизировать для системы с упреждающей выборкой машинных команд, если в теле цикла обрабатывать сразу по два (или более) элемента на каждой итерации, – вместо того чтобы прибавлять числа по одному. Это уменьшит общее количество команд перехода в машинном коде, поэтому упреждающая выборка будет срабатывать чаще. Производительность программы на данной целевой платформе, очевидно, возрастет. Но что будет, если эту же программу скомпилировать и запустить на другой платформе? На некоторых платформах оптимальным может оказаться цикл в первоначальном виде, обрабатывающий по одному элементу за итерацию, на иных же может оказаться лучше выбирать за итерацию другое количество элементов. Некоторые системы могут даже поддерживать машинную команду, которая делает все, что нужно данной функции.

Пытаясь оптимизировать код вручную, легко потерять оптимальность на всех платформах, кроме одной. Напротив, пользуясь в программе высокоуровневыми абстракциями, программист полагается на труд множества других людей по оптимизации кода для разных частных случаев. Большинство реализаций библиотеки STL содержат оптимизации, специфичные для тех или иных платформ и компиляторов.

## 1.5 Эволюция C++ как языка функционального программирования

Язык C++ возник как расширение языка C, позволяющее писать объектно-ориентированный код, и поначалу вообще назывался «C с классами». Однако даже после выхода первого стандарта языка (C++98) его

сложно было бы назвать объектно-ориентированным. С введением в язык шаблонов и с появлением библиотеки STL, в которой наследование и виртуальные методы используются лишь эпизодически, язык C++ стал в полном смысле мультипарадигмальным.

Если внимательно рассмотреть принципы библиотеки STL и ее реализацию, можно даже прийти к убеждению, что C++ – язык вовсе не объектно-ориентированного, а в первую очередь обобщенного программирования. В основе *обобщенного программирования* лежит идея о том, что код можно написать один раз, используя в нем те или иные общие понятия, а затем применять его сколько угодно раз к различным сущностям, подпадающим под эти понятия. Так, например, библиотека STL предоставляет шаблон класса `vector`, который можно использовать с разными типами данных, включая целые числа, строки или пользовательские типы, отвечающие определенным требованиям. Для каждого из этих типов компилятор генерирует хорошо оптимизированный код. Этот механизм часто называют *статическим полиморфизмом*, или *полиморфизмом времени компиляции*, – в противоположность *динамическому полиморфизму*, или *полиморфизму времени выполнения*, который поддерживается наследованием и виртуальными методами.

Что касается поддержки функционального стиля языком C++, то решающая роль шаблонов обеспечивается не столько поддержкой контейнерных классов наподобие векторов, сколько тем, что они сделали возможным появление в библиотеке STL набора универсальных общепотребительных алгоритмов – таких как сортировка или подсчет. Большинство из них позволяет программисту передавать в качестве параметров собственные функции, тем самым подстраивая их поведение без использования<sup>1</sup> типа `void*`. Таким способом, например, можно изменить порядок сортировки или определить, какие именно элементы следует пересчитать, и т. д.

Возможность передавать функции в другие функции в качестве аргументов и возможность возвращать из функций в качестве значений новые функции (точнее, некоторые сущности, *ведущие себя* подобно функциям, о чем речь пойдет в главе 3) превратили даже первую стандартизированную версию языка C++ в язык функционального про-

<sup>1</sup> Напомним, что в языке C, не обладающем средствами обобщенного программирования, единственный способ создать универсальную функцию, способную работать с данными произвольных типов, состоит в том, чтобы передать в качестве аргумента указатель на функцию, принимающую аргументы через универсальный указатель типа `void*`. Эта последняя функция отвечает за приведение указателей `void*` к фактическому типу данных и специфику работы с этим типом. Например, универсальная функция сортировки из стандартной библиотеки языка C имеет прототип `void qsort(void* base, size_t num, size_t size, int (*comp)(const void*, const void*))`, и ее последний аргумент представляет собой указатель на пользовательскую функцию сравнения элементов. Хотя использование указателей `void*` позволяет, в принципе, определять алгоритмы, работающие для произвольных типов данных, этот механизм прямо противоречит идее строгой типизации и чреват причудливыми, трудно поддающимися обнаружению ошибками. – Прим. перев.



граммирования. Стандарты C++11, C++14 и C++17 принесли с собой еще несколько нововведений, заметно упрощающих программирование в функциональном стиле. Эти дополнительные возможности представляют собой главным образом синтаксический сахар, как, например, ключевое слово `auto` и лямбда-выражения (рассматриваемые в главе 3). Кроме того, существенным усовершенствованиям подверглись алгоритмы из стандартной библиотеки. Следующая версия стандарта должна появиться в 2020 году, и в ней ожидается еще большее количество нововведений (ныне пребывающих в статусе технических спецификаций), ориентированных на поддержку функционального стиля: например, диапазонов (`range`), концептов (`concept`) и сопрограмм (`coroutine`).

### Эволюция международных стандартов языка C++

Для языка C++ существует стандарт, утвержденный международной организацией по стандартизации ISO. Каждая новая версия стандарта проходит через строго регламентированный процесс рассмотрения, прежде чем быть опубликованной. Сам по себе язык и его стандартная библиотека разрабатываются комитетом, каждое нововведение всесторонне обсуждается, ставится на голосование и лишь затем становится частью окончательного проекта изменений к новой версии стандарта. В конце, когда все изменения внесены в тело стандарта, он должен пройти через еще одно голосование – итоговое, которое проводится для каждого нового стандарта в организации ISO.

Начиная с 2012 года работа комитета разделена между рабочими группами. Каждая группа работает над отдельными элементами языка и, когда считает этот элемент вполне законченным, выпускает так называемую техническую спецификацию (ТС). ТС существуют отдельно от стандарта, в стандарт они могут войти позднее.

Цель ТС состоит в том, чтобы разработчики могли опробовать новые языковые средства и обнаружить в них скрытые недоработки и огрехи до того, как эти средства войдут в новый стандарт. От производителей компиляторов не требуется воплощать ТС, но обычно они это делают. Более подробную информацию можно найти на сайте комитета <https://isocpp.org/std/status>.

Хотя большую часть средств, о которых говорится в этой книге, можно использовать и с более ранними версиями языка C++, мы будем ориентироваться преимущественно на стандарты C++14 и C++17.

## 1.6 Что узнает читатель из этой книги

Эта книга предназначена в первую очередь для опытных разработчиков, которые используют язык C++ в своей повседневной работе и желают пополнить свой арсенал более мощными инструментами. Чтобы извлечь максимум пользы из изучения этой книги, нужно хорошее знакомство с такими основополагающими элементами языка C++, как система типов, ссылки, спецификатор `const`, шаблоны, перегрузка операций, и др.

Нет необходимости знать нововведения, появившиеся в стандартах C++14/17, поскольку они подробно разобраны в книге; эти новые элементы еще не завоевали популярность, и многие читатели о них, скорее всего, еще не осведомлены.

Для начала будут разобраны такие важнейшие понятия, как функции высших порядков, что позволит повысить выразительность языка и сделать программы заметно короче. Также будет рассказано, как писать код, не имеющий изменяемого состояния, чтобы устранить необходимость в явной синхронизации при параллельном выполнении. Затем переключимся на вторую передачу и рассмотрим более сложные темы, такие как диапазоны (альтернатива алгоритмам обработки коллекций из стандартной библиотеки, хорошо поддерживающая композицию операций) и алгебраические типы данных (с помощью которых можно уменьшить число состояний, в которых может пребывать программа). Под конец поговорим об идиоме функционального программирования, вызывающей больше всего споров, – о пресловутых *монадах* и о том, как использовать монады для создания сложных и хорошо стыкующихся между собой систем.

К концу книги читатель будет в состоянии проектировать и реализовывать надежные параллельные системы, способные масштабироваться по горизонтали с минимальными усилиями; проектировать программу таким образом, чтобы свести к минимуму или вообще устранить возможность попадания в невалидное состояние вследствие сбоя или ошибки; представлять себе программу как поток данных сквозь череду преобразователей и использовать новое замечательное средство, диапазоны, для создания таких потоков, а также многое другое. Эти умения помогут создавать более компактный и менее подверженный ошибкам код, даже продолжая работать над объектно-ориентированными системами. Те, кто решит полностью погрузиться в функциональный стиль, получат возможность разрабатывать более стройные программные системы с удобными программными интерфейсами для взаимодействия с другими системами, в чем читатель убедится в главе 13, реализовав простой веб-сервис.

**СОВЕТ** Более подробную информацию и дополнительные ресурсы по темам, затронутым в этой главе, можно найти на странице <https://forums.manning.com/posts/list/41680.page>.

## Итоги

- Важнейший принцип функционального программирования состоит в том, что программисту не нужно брать на себя заботу о том, каким именно образом работает программа, – думать следует лишь о том, что она должна сделать.
- Оба подхода к программированию, функциональный и объектно-ориентированный, сулят программисту значительные выгоды. Нужно понимать, когда лучше применить первый подход, когда второй, а когда – их комбинацию.

- С++ представляет собой мультипарадигмальный язык программирования, с его помощью можно создавать программы в различных стилях: процедурном, объектно-ориентированном, функциональном, – а также комбинировать эти стили с использованием методов обобщенного программирования.
- Функциональное программирование идет рука об руку с обобщенным, особенно в языке С++. Оба подхода побуждают программиста не фиксировать внимание на аппаратном уровне, а поднимать уровень абстракции.
- Подъем функций позволяет создавать функции, работающие с коллекциями значений, на основе функций, обрабатывающих отдельные значения. С помощью композиции функций можно подать значение на цепочку преобразователей, в которой каждая функция передает свой результат на вход следующей функции.
- Избегание изменяемого состояния способствует корректности кода и устраняет необходимость в синхронизации параллельных потоков.
- Мыслить по-функциональному означает размышлять о входных данных и преобразованиях, которые нужно выполнить, чтобы получить желаемый результат.



---

# Первые шаги в функциональном программировании

---

## О чем говорится в этой главе:

- понятие о функциях высшего порядка;
- использование функций высшего порядка из библиотеки STL;
- трудности с композицией алгоритмов из библиотеки STL;
- рекурсия и оптимизация хвостовой рекурсии;
- мощь алгоритма свертки.



В предыдущей главе было показано несколько изящных примеров того, как можно улучшить качество кода за счет использования простых конструкций функционального программирования. Там же были показаны выгоды от применения ФП, такие как лаконичность, корректность и эффективность кода. Однако те волшебные средства функциональных языков, которые позволяют писать код таким образом, рассмотрены пока не были.

Истина состоит в том, что если заглянуть за кулисы функциональных языков, то никакого волшебства там не окажется – лишь несколько простых понятий, на основе которых строится все остальное. Первое довольно простое, но богатое далеко идущими вариациями понятие из использованных в предыдущей главе – это передача функции в качестве аргумента в другую функцию, в частности в алгоритм из стандартной библиотеки шаблонов STL. Алгоритмы в библиотеке STL потому и применимы к столь широкому кругу задач, что их поведение можно настраивать с помощью аргументов-функций.

## 2.1 Функции с аргументами-функциями

Главная особенность всех языков функционального программирования (ФП) состоит в том, что с функциями можно обращаться, как с обыкновенными значениями. Их можно хранить в переменных, помещать в коллекции или структуры данных, передавать в другие функции в качестве аргументов или получать из функций в качестве возвращаемых значений.

Функции, которые принимают другие функции в качестве аргументов или возвращают функции в качестве значений, называются функциями высшего порядка. В предыдущей главе читатель видел, как сделать программу более лаконичной и эффективной, если на высоком уровне абстракции описать, что программа должна сделать, – вместо того чтобы описывать все подробности своими руками. Функции высшего порядка оказываются незаменимыми именно для этого. Они позволяют абстрактно описывать поведение и определять структуры управления, более сложные, чем те, что предоставляет сам по себе язык C++.

Проиллюстрируем сказанное примером. Предположим, дана группа людей, и нужно получить имена всех женщин из этой группы (рис. 2.1).

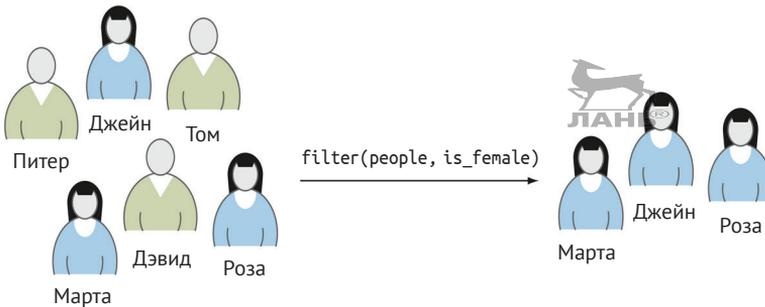


Рис. 2.1 Фильтрация коллекции людей по предикату `is_female`. Должна получиться коллекция людей, содержащая только женщин из исходной коллекции

Первая высокоуровневая конструкция, которой можно было бы воспользоваться для этой задачи, – это фильтрация коллекции. *Фильтрация* в общем случае – это простой алгоритм, который перебирает один за другим элементы и для каждого проверяет, удовлетворяет ли он определенному условию. Если условие истинно, элемент помещается в коллекцию-результат. Алгоритм фильтрации не может знать заранее, какие предикаты понадобятся пользователям для фильтрации их коллекций. Фильтрация может выполняться по какому-то одному признаку объекта (в данном примере – по определенному значению параметра «пол» объекта «человек»), или по комбинации признаков (скажем, если из коллекции людей нужно выбрать всех женщин с черными волосами), или по еще более сложно устроенному условию (например, выбрать всех женщин, которые недавно купили новый автомобиль). Поэтому алгоритм фильтрации должен предоставлять пользователю возможность задавать

собственный предикат. В данном примере пользователь подставит в алгоритм фильтрации функцию, которая принимает в качестве аргумента объект, моделирующий человека, и возвращает истинностное значение: женщина ли это. Таким образом, алгоритм фильтрации должен принимать на вход функцию-предикат и, следовательно, является функцией высшего порядка.

### Дополнение к способу записи типов функций

Чтобы обозначить произвольную коллекцию, содержащую элементы некоторого типа  $T$ , будем писать `collection<T>` или, для краткости, просто `S<T>`. Чтобы обозначить, что аргумент функции представляет собой функцию, будем писать тип функции-аргумента там, где ожидается тип аргумента.

Например, конструкция `filter` принимает в качестве аргументов коллекцию значений произвольного типа  $T$  и функцию-предикат (т. е. значение типа  $T \rightarrow \text{bool}$ ), а возвращает коллекцию значений, удовлетворяющих предикату, поэтому тип ее может быть записан следующим образом:

`filter: (collection<T>, (T → bool)) → collection<T>`

После того как фильтрация произведена, остается вторая задача: получить имена людей. Для этого нужна конструкция, которая получает совокупность людей и возвращает совокупность их имен. Как и в случае фильтрации, в общем случае эта конструкция не может знать заранее, какую информацию нужно извлекать из поступивших на вход значений. Пользователь может захотеть значение определенного атрибута (в этом примере – атрибута «имя»), комбинацию нескольких атрибутов (возможно, конкатенацию имени и фамилии) или сделать что-то более сложное (скажем, для каждого человека из исходной коллекции получить список детей). Как и предыдущая, эта конструкция должна позволять пользователю задавать свою функцию, которая принимает на вход один элемент коллекции, что-то с ним делает и возвращает значение, которое должно быть помещено в коллекцию-результат (рис. 2.2).

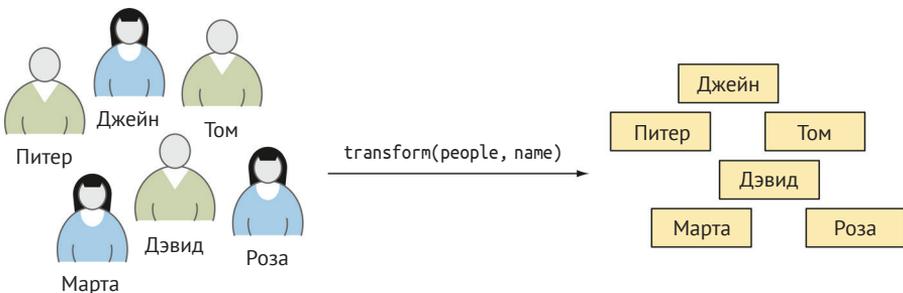


Рис. 2.2 Дана коллекция людей. Алгоритм `transform` должен применить функцию-преобразователь к каждому элементу и собрать результаты в новую коллекцию. В данном примере передается функция, которая возвращает имя человека. В итоге алгоритм `transform` строит коллекцию имен

Следует заметить, что коллекция-результат в этом случае не обязана состоять из значений того же типа, что и коллекция, поступившая на вход (в отличие от фильтрации). Данную конструкцию называют обычно `map` или `transform`, и ее тип таков:

```
transform: (collection<In>, (In → Out)) → collection<Out>
```

Если теперь построить композицию этих двух конструкций (см. рис. 2.3), подав результат первой на вход второй, получится решение исходной задачи: найти имена всех женщин из заданной группы людей.

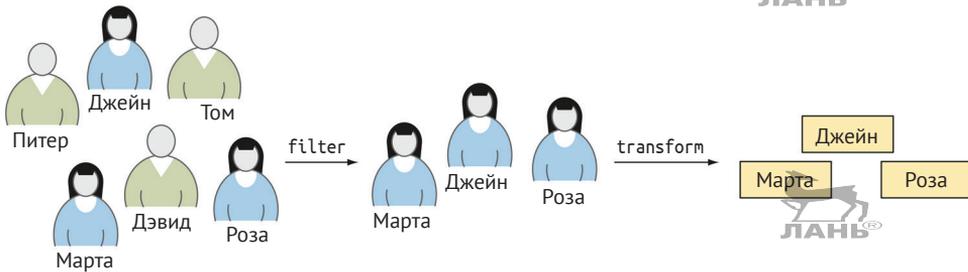


Рис. 2.3 Если дана функция, которая фильтрует коллекцию людей, отбирая из них только женщин, и вторая функция, которая у каждого человека из заданной коллекции получает имя, их можно соединить и получить функцию, которая получает коллекцию имен всех женщин из заданной группы людей

Оба этих средства, `filter` и `transform`, представляют собой широко распространенные идиомы, которые многие программисты раз за разом реализуют заново в каждом из своих проектов. Небольшие вариации в условии задачи – скажем, различные предикаты, используемые для фильтрации (по признакам пола, возраста и т. д.) – вынуждают программистов снова и снова писать один и тот же код. Функции высшего порядка позволяют вынести эти различия в параметр-функцию и оставить лишь предельно обобщенное абстрактное понятие, охватывающее все частные случаи. Помимо прочих выгод, это существенно повышает степень повторного использования кода и его покрытие тестами.

## 2.2 Примеры из библиотеки STL

Библиотека STL содержит множество функций высшего порядка, скрывающихся под наименованием *алгоритмов*. В этой библиотеке собраны эффективные реализации множества общеупотребительных идиом программирования. Использование стандартных алгоритмов вместо написания собственных реализаций на низком уровне через циклы, ветвления и рекурсии позволяет выразить логику программы меньшим объемом кода и с меньшим числом ошибок. Не будем описывать здесь все функции высшего порядка из библиотеки STL, рассмотрим лишь несколько наиболее интересных, чтобы возбудить интерес читателя.

## 2.2.1 Вычисление средних

Предположим, дан список оценок, которые посетители сайта поставили некому фильму, и требуется вычислить среднюю оценку. Императивный подход к решению этой задачи состоит в том, чтобы пройтись в цикле по коллекции оценок, суммируя элементы, и разделить полученную сумму на общее число оценок (рис. 2.4).

**Листинг 2.1** Подсчет средней оценки в императивном стиле

```
double average_score(const std::vector<int>& scores)
{
    int sum = 0; // ← Начальное значение суммы

    for (int score : scores) {
        sum += score; // ← Поэлементное суммирование
    }

    return sum / (double)scores.size(); // ← Вычисление среднего арифметического
}
```



**Рис. 2.4** Вычисление средней оценки фильма на основе списка оценок отдельных пользователей. Нужно просуммировать оценки пользователей, затем разделить сумму на число пользователей

Хотя этот подход и работает, он страдает излишней детализацией. В коде есть несколько мест, в которых легко допустить ошибку: например, можно указать не тот тип элемента в заголовке цикла, из-за чего код изменит свою семантику, но все еще будет компилироваться. Кроме того, этому коду внутренне присущ последовательный порядок выполнения, тогда как суммирование коллекции можно легко распараллелить и выполнять на нескольких ядрах или даже на специализированном процессоре.

Библиотека STL содержит функцию высшего порядка, которая умеет суммировать все элементы коллекции, – алгоритм `std::accumulate`. Она

принимает коллекцию (в виде пары итераторов) и начальное значение для суммирования, а возвращает сумму начального значения со всеми элементами коллекции. Остается лишь разделить эту сумму на общее число оценок, как в предыдущем примере. Следующий код ничего не говорит о том, каким способом должно выполняться суммирование, – в нем лишь задано, что должно быть сделано. Реализация получилась столь же обобщенной, как и представленная на рис. 2.4 постановка задачи.

### Листинг 2.2 Подсчет средней оценки в функциональном стиле

```
double average_score(const std::vector<int>& scores)
{
    return std::accumulate(
        scores.cbegin(), scores.cend(), | Суммировать все оценки в коллекции
        0 | ← Начальное значение суммы
    ) / (double)scores.size(); ← Вычислить среднее арифметическое
}
```

Хотя алгоритм `std::accumulate` сам по себе реализован последовательно, подставить вместо него параллельную версию очень просто. Для сравнения, сделать первую реализацию параллельной было бы нетривиальной задачей.



### Параллельные версии стандартных алгоритмов

Начиная с версии C++17 многие алгоритмы из библиотеки STL позволяют программисту указать, что их выполнение должно осуществляться параллельно. Алгоритмы, допускающие параллельное выполнение, принимают политику выполнения через дополнительный аргумент. Если желательно, чтобы алгоритм выполнялся параллельно, ему нужно передать политику `std::execution::par`. За более подробной информацией о политиках выполнения можно обратиться к соответствующему разделу справочника по языку C++: <http://mng.bz/EBys>.

Алгоритм `std::accumulate` стоит особняком. Он гарантирует, что элементы коллекции обрабатываются последовательно, с первого по последний, что делает распараллеливание невозможным без нарушения семантики. Если нужно просуммировать все элементы, но сделать это параллельным образом, следует воспользоваться алгоритмом `std::reduce`:

```
double average_score(const std::vector<int>& scores)
{
    return std::reduce(
        std::execution::par,
        scores.cbegin(), scores.cend(),
        0
    ) / (double) scores.length();
}
```

Если для работы используется старый компилятор и версия библиотеки STL, не имеющая полной поддержки стандарта C++17, алгоритм `reduce` можно найти в пространстве имен `std::experimental::parallel`, также можно воспользоваться сторонними библиотеками, такими как HPX или Parallel STL, см. статью автора этой книги «C++17 and Parallel Algorithms in STL – Setting Up» по адресу <http://mng.bz/8435>.

И все-таки в листинге 2.2 чего-то не хватает. Выше я написал, что алгоритм `std::accumulate` – это функция высшего порядка, но не передал другую функцию в качестве аргумента (а алгоритм не возвращает значение типа функции).

По умолчанию функция `std::accumulate` применяет к элементам коллекции операцию сложения, но если программист желает изменить ее поведение, он может передать собственную функцию. Так, например, если для каких-то целей нужно вычислить произведение всех оценок, достаточно передать в функцию `std::accumulate` дополнительным аргументом объект `std::multiplies` и число 1 в качестве начального значения, как показано в следующем листинге. Функциональные объекты `std::multiplies` будут подробно рассмотрены в следующей главе. Пока что довольно сказать, что этот объект ведет себя подобно функции, принимающей два аргумента определенного типа и возвращающей их произведение.

### Листинг 2.3 Вычисление произведения всех оценок

```
double scores_product(const std::vector<int>& scores)
{
    return std::accumulate(
        scores.cbegin(), scores.cend(), |— Все оценки в коллекции
        1, |— ← Начальное значение для произведения
        std::multiplies<int>() |— ← Перемножить, а не суммировать
    );
}
```

Возможность складывать или перемножать коллекции чисел в одно обращение к функции сама по себе выглядит привлекательно, но еще не очень впечатляет, если ограничивать себя вычислением одних лишь сумм и произведений. С другой стороны, замена сложения и умножения какими-то более интересными операциями впрямь приводит к впечатляющим результатам.

## 2.2.2 Свертки

Нередко возникает необходимость обрабатывать коллекцию элементов один за другим, вычисляя при этом некоторый результат. Этот результат может быть простым, как сумма или произведение всех элементов коллекции, как в предыдущем примере, число элементов, имеющих определенное значение или удовлетворяющих заданному предикату. Результат

может быть также чем-то более сложным: например, новой коллекцией, которая содержит лишь часть элементов исходной коллекции (это имеет место при фильтрации), или коллекцией, в которой исходные элементы расположены в другом порядке (что характерно для сортировки и разделения).

Алгоритм `std::accumulate` представляет собой реализацию чрезвычайно общего понятия, известного под названием *свертки*, или *редукции*. Это функция высшего порядка, которая абстрагирует от процесса поэлементного перебора рекурсивных структур данных (таких как векторы, списки, деревья) и позволяет постепенно накапливать требуемый результат. В разобранный выше примере свертка использовалась для суммирования рейтинга фильмов. Алгоритм `std::accumulate` сначала берет начальное значение, переданное ему в качестве аргумента, и складывает его с первым элементом коллекции. Полученный результат он складывает со вторым элементом коллекции и т. д. Этот процесс повторяется до тех пор, пока не будет достигнут конец коллекции (рис. 2.5).

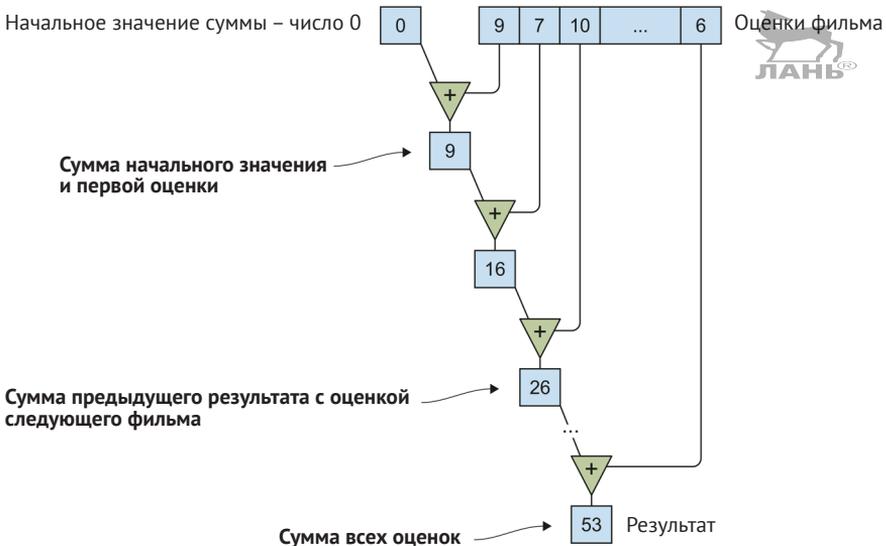


Рис. 2.5 Свертка вычисляет сумму всех оценок фильма, прибавляя первый элемент коллекции к начальному значению (нулевому), затем прибавляя второй элемент и т. д.

В общем случае не требуется, чтобы аргументы и результат бинарной операции, по которой проводится свертка, имели один и тот же тип. Свертка принимает коллекцию, состоящую из элементов типа  $T$ , начальное значение типа  $R$ , который отнюдь не обязан совпадать с типом  $T$ , и функцию  $f: (R, T) \rightarrow R$ . Свертка вызывает эту функцию, передав ей в качестве аргументов начальное значение и первый элемент коллекции. Результат этого применения передается, в свою очередь, в функцию  $f$  вместе со следующим элементом коллекции. Этот процесс повто-

руется, пока не будут обработаны все элементы. Алгоритм возвращает значение типа  $R$  – значение, которое вернул последний вызов функции  $f$  (рис. 2.6). В предыдущих примерах начальным значением было число 0 для суммирования и 1 для перемножения, а в роли бинарной функции  $f$  выступала операция сложения или, соответственно, умножения.



Рис. 2.6 В общем случае свертка берет начальное значение  $r_0$  и применяет заданную функцию  $f$  к нему и первому элементу коллекции, получая промежуточный результат  $r_1$ . Затем свертка применяет функцию  $f$  к значению  $r_1$  и второму элементу коллекции. Процесс продолжается до достижения последнего элемента коллекции

В качестве первого примера свертки по функции, у которой тип возвращаемого значения отличен от типа элементов коллекции, реализуем на основе свертки функцию, которая определяет количество строк в тексте, подсчитывая вхождения символа перевода строки. Напомним, что в главе 1 для этой цели использовалась функция `std::count`.

Из постановки задачи можно без труда вывести тип бинарной функции  $f$ , которую нужно передать в алгоритм `std::accumulate`. Объект типа `std::string` представляет собой коллекцию символов, поэтому в роли типа  $T$  должен выступать тип `char`; поскольку искомая величина есть число символов перевода строки, то типом  $R$  должен стать тип `int`. Следовательно, типом бинарной функции  $f$  будет  $(\text{int}, \text{char}) \rightarrow \text{int}$ .

Само собой разумеется, что когда функция  $f$  вызывается для первого символа строки, ее первый аргумент должен иметь значение 0, потому что в самом начале процесса еще не может быть ни одного подсчитанного перевода строки. Когда же функция  $f$  вызывается на последнем символе строки, ее результат должен составлять искомое количество. Поскольку

функция  $f$  не знает, обрабатывает она в данный момент первый, последний символ или какой-либо символ из произвольного места в строке, в ее реализации никак не должна использоваться позиция символа от начала текста. При каждом вызове функция  $f$  знает лишь значение текущего символа и результат предыдущего вызова этой же функции  $f$ . Эти рассуждения позволяют однозначно определить смысл первого аргумента функции  $f$  как *число символов перевода строки в уже просмотренной начальной части текста*. После этого дальнейшая реализация становится совершенно тривиальным делом: если текущий символ не есть перевод строки, функция  $f$  должна возвращать в неизменном виде предыдущее значение счетчика строк, а в противном случае – возвращать значение, на единицу большее. Конкретный пример такого вычисления показан на рис. 2.7, а полный текст программной реализации – в следующем листинге.

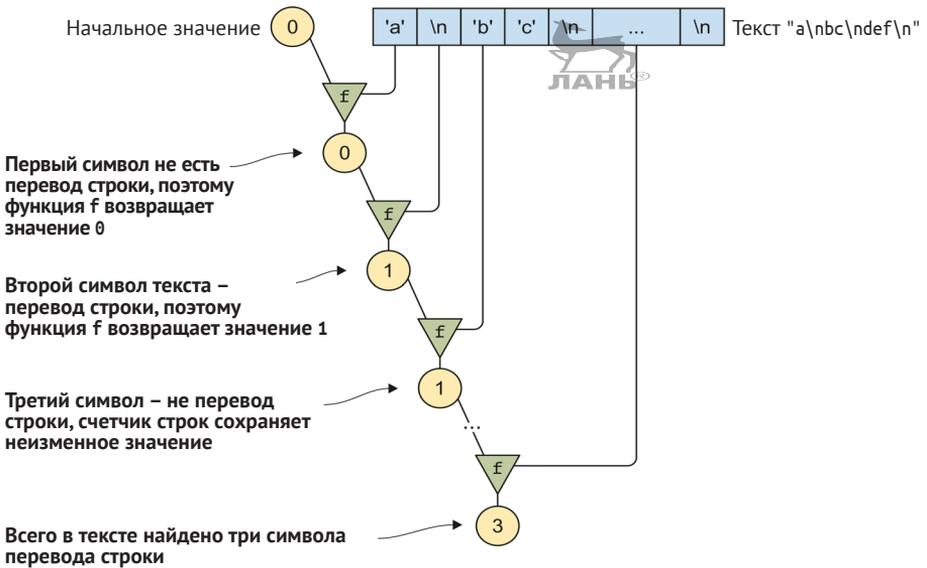


Рис. 2.7 Подсчет строк в заданном тексте посредством свертки по функции, которая наращивает значение счетчика на единицу всякий раз, когда видит символ перевода строки

#### Листинг 2.4 Подсчет символов перевода строки с помощью функции `std::accumulate`

```
int f(int previous_count, char c)
{
    return (c != '\n') ? previous_count
        : previous_count + 1;
}

int count_lines(const std::string& s)
{
```

Увеличить счетчик, если текущий символ – перевод строки

```
return std::accumulate(
    s.cbegin(), s.cend(), ← Провести свертку по всей строке
    0, ← Начать счет с нуля
    f
);
```



Можно взглянуть на свертку под иным углом, для этого нужно изменить свое понимание функции `f` от двух аргументов, передаваемой в алгоритм `std::accumulate`. Если рассмотреть ее как обычную левоассоциативную бинарную операцию наподобие операции `+`, которую можно использовать в выражении инфиксно, т. е. между операндами, то свертка оказывается эквивалентной выражению, в котором эта операция помещена между каждой парой соседних элементов коллекции. В примере с суммированием оценок фильма это означает все оценки в ряд и помещение знаков `+` между ними (рис. 2.8). Полученное выражение вычисляется слева направо, в точности как выполняет свертку алгоритм `std::accumulate`.

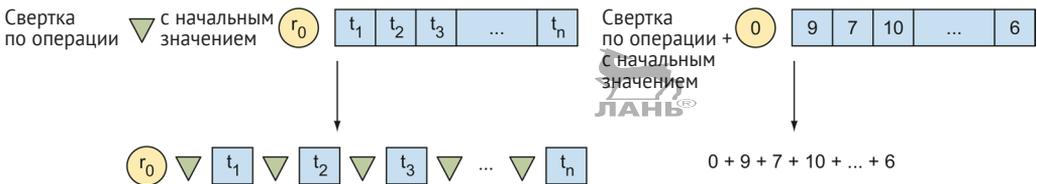


Рис. 2.8 Свертка по левоассоциативной операции эквивалентна выражению, которое получится, если выписать все элементы коллекции один за другим и вставить между каждой парой соседних элементов знак операции

Описанную выше свертку, при которой элементы коллекции обрабатываются начиная с первого, называют *левой сверткой*. Существует также *правая свертка*, при которой обработка начинается с последнего элемента коллекции и продвигается к ее началу. В инфиксной нотации правая свертка соответствует вычислению выражения, показанного на рис. 2.8, если операция обладает правой ассоциативностью, см. также рис. 2.9. В стандартной библиотеке языка C++ нет отдельного алгоритма для правой свертки, но нужного поведения легко добиться, если передать в алгоритм `std::accumulate` обратные итераторы (`crbegin` и `crend`).

На первый взгляд свертка может показаться всего лишь сложением элементов коллекции одного за другим. Но если допустить, что вместо сложения стоит произвольная бинарная операция, которая может даже давать результат другого типа, отличного от типа элементов коллекции, свертка превращается в мощный инструмент для реализации множества алгоритмов. Конкретный пример этого будет показан далее в данной главе.

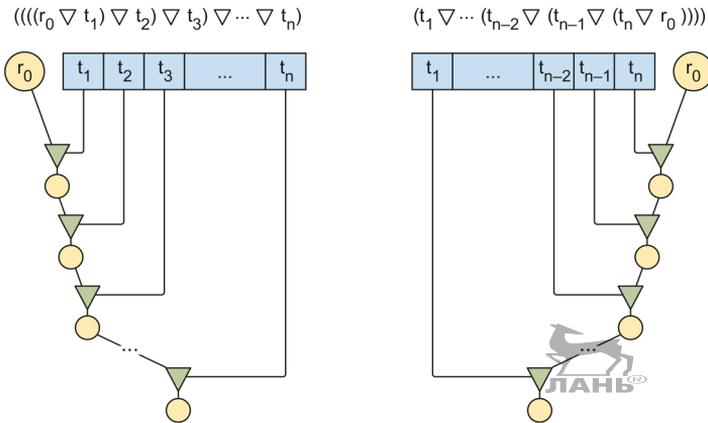


Рис. 2.9 Главное различие между левой и правой свертками состоит в том, что левая свертка начинается с первого элемента коллекции и продвигается к ее концу, тогда как правая свертка начинается с последнего элемента и движется к началу коллекции

### Несколько слов об ассоциативности

Операция  $\nabla$  называется левоассоциативной, если  $a \nabla b \nabla c = (a \nabla b) \nabla c$ , и правоассоциативной, если  $a \nabla b \nabla c = a \nabla (b \nabla c)$ . С точки зрения математики, сложение и умножение чисел одновременно лево- и правоассоциативны – не имеет значения, начинать применение этих операций к списку аргументов с правого или левого конца. Однако в языке C++ обе эти операции определены как левоассоциативные, чтобы правила языка могли гарантировать определенный порядок вычисления выражений.

Это вовсе не означает в большинстве случаев, будто язык C++ гарантирует какой-либо порядок вычисления самих аргументов; гарантирован лишь порядок, в котором операция применяется к уже вычисленным значениям. Например, если дано выражение вида  $a * b * c$  (где  $a$ ,  $b$  и  $c$  – некоторые подвыражения), нельзя сказать, какое из подвыражений будет вычислено первым<sup>1</sup>, но точно известен порядок применения операций умножения, а именно: произведение значений  $a$  и  $b$  будет умножено на  $c$ .

## 2.2.3 Удаление лишних пробелов

Предположим, дана строка, и необходимо удалить пробелы в ее начале и на конце. Например, для того, чтобы прочитать строку текста из файла и отобразить ее на экране, центрировав по ширине. Если оставить

<sup>1</sup> Стандарт C++17 все же определяет порядок вычисления аргументов в некоторых случаях (см.: *Gabriel Dos Reis, Herb Sutter, and Jonathan Caves. Refining Expression Evaluation Order for Idiomatic C++ // open-std.org. 2016, June 23. URL: <http://mng.bz/gO5J>*), но для операций наподобие сложения и умножения порядок вычисления аргументов остается неопределенным.

в строке начальные и конечные пробелы, текст может отображаться на экране со смещением.

В библиотеке STL нет функций для удаления пробелов из начала и конца строки. Зато она предоставляет вдоволь алгоритмов, на основе которых нетрудно реализовать такую операцию.

Алгоритм, нужный для данной задачи, – это алгоритм `std::find_if`. Он отыскивает в коллекции первый элемент, удовлетворяющий заданному предикату. В данном случае нужно найти в строке первый символ, не являющийся пробелом. Алгоритм возвращает итератор, указывающий на искомый элемент (или итератор, указывающий на пустое место после последнего элемента коллекции, если нужный элемент в коллекции отсутствует). Для того чтобы удалить из строки все начальные пробелы, достаточно удалить из нее элементы с первого до найденного (не включая его):

```
std::string trim_left(std::string s)
{
    s.erase(s.begin(),
            std::find_if(s.begin(), s.end(), is_not_space));
    return s;
}
```



Как обычно, алгоритм поиска принимает пару итераторов, вместе определяющих коллекцию. Если передать в него обратные итераторы, он будет искать от конца строки по направлению к ее началу. Таким способом можно удалить пробелы справа:

```
std::string trim_right(std::string s)
{
    s.erase(std::find_if(s.rbegin(), s.rend(), is_not_space).base(),
            s.end());
    return s;
}
```

### О передаче по значению

В предыдущем примере лучше передавать строку по значению, чем по константной ссылке, потому что функция изменяет свой аргумент и возвращает результат его модификации. В противном случае пришлось бы внутри функции создавать локальную копию строки. Такая реализация может работать медленнее, когда функции в качестве аргумента передают временный объект (rvalue), ведь его пришлось бы скопировать, тогда как можно было бы переместить. Единственный недостаток передачи аргумента по значению состоит в том, что если конструктор копирования бросит исключение, стек вызовов может привести автора программы в недоумение<sup>1</sup>.

<sup>1</sup> Более подробные сведения о копировании и перемещении можно найти в работе: *Alex Allain. Move Semantics and rvalue References in C++11 // Cprogramming.com. URL: <http://mng.bz/JULm>.*

Композиция этих двух функций образует нужную нам функцию `trim`:

```
std::string trim(std::string s)
{
    return trim_left(trim_right(std::move(s)));
}
```



Этот пример демонстрирует одно из возможных применений функции высшего порядка `std::find_if`. Здесь она использована для поиска первого непробельного символа – сперва с начала строки, а затем с конца. Также показано, как из композиции двух меньших функций получается функция, решающая поставленную задачу.

## 2.2.4 Разделение коллекции по предикату

Прежде чем погружаться глубже в предмет, рассмотрим еще один пример. Пусть дана коллекция людей, и пусть необходимо переупорядочить ее так, чтобы все женщины оказались в ее начале. Для этого можно воспользоваться алгоритмом `std::partition` и его вариацией – алгоритмом `std::stable_partition`.

Оба алгоритма принимают на вход коллекцию (как всегда, заданную парой итераторов) и предикат. Они меняют местами элементы коллекции так, чтобы отделить те из них, что удовлетворяют предикату, от тех, что ему не удовлетворяют. Элементы, для которых предикат истинен, перемещаются ближе к началу коллекции, а в конце остаются исключительно те элементы, для которых предикат ложен. Эти алгоритмы возвращают итератор, указывающий на первый элемент из второго сегмента, т. е. на первый элемент, не удовлетворяющий предикату. Этот итератор можно объединить в пару с итератором, отмечающим начало исходной коллекции, – получится диапазон тех и только тех элементов исходной коллекции, которые удовлетворяют предикату; если же объединить его в пару с итератором, указывающим на конец исходной коллекции, получится диапазон элементов, данный предикат нарушающих. Это справедливо и в том случае, когда любая из коллекций пуста.

Различие между этими двумя алгоритмами состоит в том, что функция `std::stable_partition` сохраняет неизменным порядок расположения элементов в пределах каждой группы. В следующем примере показан результат применения алгоритма `std::partition`: в получившемся списке Роза стоит перед Мартой, хотя в исходном списке находилась после нее (рис. 2.10).

### Листинг 2.5 Разделение коллекции людей на женщин и мужчин

```
std::partition(
    people.begin(), people.end(),
    is_female
);
```

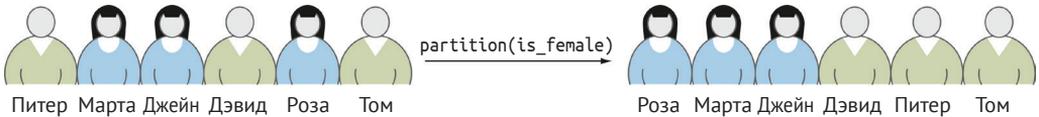


Рис. 2.10 Разделение группы людей по предикату, проверяющему, является ли данный человек женщиной. Все женщины перемещаются в начало коллекции

Хотя следующий пример может показаться несколько надуманным, представим список элементов пользовательского интерфейса. Пользователь может отметить некоторые элементы и перетащить их в другое место. Если это место есть начало списка, получаем ту же задачу, что и в предыдущем примере, только на этот раз в начало списка нужно перемещать не женщин, а выбранные пользователем визуальные элементы. Если выбранные элементы нужно переместить в конец, задача эквивалентна перемещению в начало списка тех элементов, что не выбраны.

Если же выбранные элементы нужно переместить в какое-то место в середине списка, список можно разбить на две части: перед и после точки назначения. Тогда в первом списке выбранные элементы нужно переместить в конец, а во втором – в начало.

Когда пользователь перемещает выбранные объекты, он обычно ожидает, что относительный порядок элементов внутри выбранной группы останется неизменным. То же требование справедливо и для невыбранных элементов: перемещение нескольких отмеченных пользователем элементов не должно произвольным образом тасовать остальные. Поэтому здесь нужно пользоваться алгоритмом `std::stable_partition` вместо алгоритма `std::partition`, хотя последний и более эффективен (рис. 2.11).

### Листинг 2.6 Перемещение выбранных элементов в заданную точку

```
std::stable_partition(first, destination, is_not_selected);
std::stable_partition(destination, last, is_selected);
```

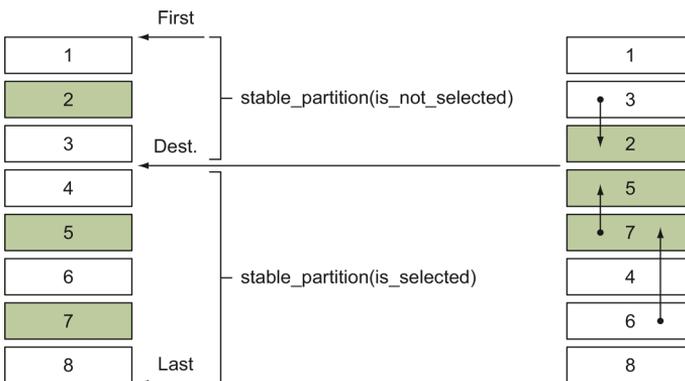


Рис. 2.11 Перемещение выбранных элементов в заданную позицию с помощью функции `std::stable_partition`. Выбранные элементы до точки назначения должны переместиться вниз, а после этой точки – вверх

## 2.2.5 Фильтрация и преобразование

Попытаемся решить задачу, описанную в начале этой главы, с помощью алгоритмов из библиотеки STL. Напомним: задача состоит в том, чтобы получить имена всех женщин из заданной группы людей (рис. 2.12).

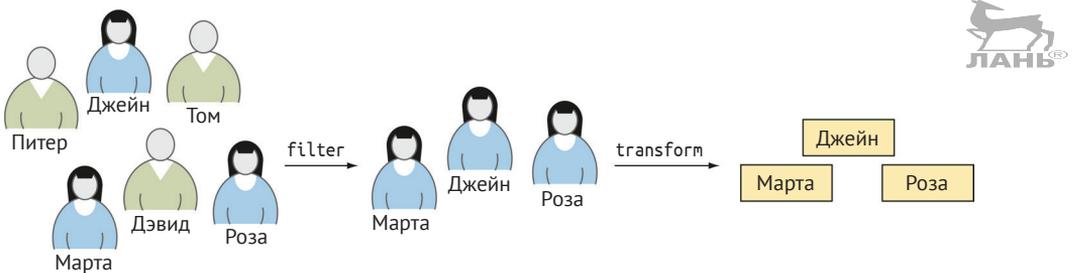


Рис. 2.12 Получение имен всех женщин из группы людей

Пусть человек представлен в программе объектом типа `person_t`, и пусть в качестве коллекции используется тип `std::vector`. Для простоты реализации предположим, что есть функции – не члены класса `is_female`, `is_not_female` и `name`:

```
bool is_female(const person_t& person);
bool is_not_female(const person_t& person);
std::string name(const person_t& person);
```



Как мы уже видели ранее, первая задача состоит в том, чтобы отфильтровать коллекцию и получить вектор, содержащий только женщин. Есть два способа сделать это с помощью стандартных алгоритмов. Если разрешается модифицировать исходную коллекцию, можно воспользоваться алгоритмом `std::remove_if` и идиомой «стереть и удалить» (англ. *erase-remove*), чтобы отбросить из исходной коллекции всех людей, кроме женщин.

### Листинг 2.7 Фильтрация списка путем удаления ненужных элементов

```
people.erase( ←————— Удалить выбранные элементы
  std::remove_if(people.begin(), people.end(),
                 is_not_female),
  people.end()); |————— Отметить элементы для удаления
```

#### Идиома «стереть и удалить»

В библиотеке STL, для того чтобы удалить из коллекции все элементы, удовлетворяющие некоторому предикату, или все элементы, имеющие определенное значение, предназначены функции `std::remove_if` и `std::remove`. К сожалению, эти алгоритмы работают с диапазоном, заданным парой итераторов, и ничего не знают о скрывающейся за ними структуре данных, и потому не могут удалить из нее элементы. Эти алгоритмы могут лишь отделить

элементы, не удовлетворяющие условию удаления, собрав их в начале диапазона.

Прочие элементы, остающиеся в конце диапазона, в общем случае оказываются в неопределенном состоянии. Алгоритм возвращает итератор, указывающий на первый из них (или на конец диапазона, если не нашлось ни одного кандидата на удаление). Этот итератор нужно передать в метод `.erase` коллекции, который удалит из нее эти элементы.

Если же исходную коллекцию менять не хочется – что следует считать правильным решением, так как мы стремимся, насколько возможно, соблюдать требование чистоты функций, – можно воспользоваться функцией `std::copy_if`, чтобы скопировать все элементы, удовлетворяющие условию фильтрации, в новую коллекцию. Этому алгоритму нужно передать пару итераторов, определяющих входную коллекцию, один итератор, указывающий на коллекцию, в которую нужно копировать элементы, и предикат, выделяющий элементы, подлежащие копированию. Поскольку число женщин в исходной коллекции заранее неизвестно (его можно приблизительно оценить, основываясь на данных статистики, но это выходит за рамки данного примера), нужно создать пустой вектор `females` и использовать итератор `std::back_inserter(females)` в качестве точки назначения.

### Листинг 2.8 Фильтрация коллекции путем копирования элементов в новую коллекцию

```
std::vector<person_t> females;
std::copy_if(people.cbegin(), people.cend(),
             std::back_inserter(females),
             is_female);
```

Создать новую коллекцию для сохранения отфильтрованных элементов

Копировать элементы, удовлетворяющие условию, в новую коллекцию

Следующий шаг – получить имена людей из отфильтрованной коллекции. Для этого можно воспользоваться функцией `std::transform`. Ей нужно передать входную коллекцию в виде пары итераторов, функцию-преобразователь и итератор, указывающий место для хранения результатов. На этот раз известно, что имен в результирующей коллекции будет ровно столько же, сколько женщин было в исходной коллекции, поэтому можно сразу создавать вектор `names` требуемого размера, а в качестве точки назначения вместо итератора `std::back_inserter` использовать итератор `names.begin()`. Это позволяет избежать повторного выделения памяти, которое происходит при увеличении размера вектора.

### Листинг 2.9 Получение имен

```
std::vector<std::string> names(females.size());
std::transform(females.cbegin(), females.cend(),
               names.begin(),
               name);
```

Исходная коллекция для преобразования

Куда записывать результаты

Функция-преобразователь

Таким образом, крупная задача разбита на две более мелкие. Вместо того чтобы создавать одну узкоспециализированную функцию, пригодную лишь для одной конкретной задачи, мы создали две отдельные, потенциально применимые для гораздо более широкого круга задач: одна фильтрует коллекцию людей по некоторому предикату, а другая получает имена всех людей из заданной коллекции. Такой код гораздо лучше подходит для многократного использования.



## 2.3 Проблема композиции алгоритмов из библиотеки STL

Решение, представленное в предыдущем разделе, корректно в том смысле, что правильно работает для любого типа входной коллекции, лишь бы к ней можно было обращаться посредством итераторов, от векторов и списков до множеств, хеш-таблиц и деревьев. Кроме того, код ясно выражает смысл программы: скопировать из исходной коллекции всех женщин в отдельную коллекцию, затем из нее получить все имена.

К сожалению, этот подход не столь эффективен и лаконичен, как следующий самодельный цикл:

```
std::vector<std::string> names;

for (const auto& person : people) {
    if (is_female(person)) {
        names.push_back(name(person));
    }
}
```



Реализация на основе библиотеки STL без необходимости копирует объекты, описывающие людей (такая операция может оказаться дорогостоящей или вообще невозможной, если конструктор копирования удален или скрыт), а также создает дополнительный вектор для промежуточных данных, на самом деле ненужный. Можно попытаться обойти эти проблемы, если вместо копирования использовать ссылки или указатели либо создать умный итератор, который пропускает людей, не являющихся женщинами, или иными способами. Однако необходимость в таких ухищрениях, требующих дополнительной работы, ясно указывает, что в данном случае библиотека STL проигрывает. Написанный своими руками узкоспециализированный цикл работает лучше и требует меньших усилий.

В чем же корень проблемы? Вспомним сигнатуры наших функций `transform` и `filter`. Они продуманы так, чтобы поддерживать композицию. Одну из них можно применять к результату другой:

```
filter : (collection<T>, (T → bool)) → collection<T>
transform : (collection<T>, (T → T2)) → collection<T2>

transform(filter(people, is_female), name)
```

Алгоритмы `std::copy_if` и `std::transform` не подходят под эти сигнатуры. Они требуют тех же данных на вход, но существенно иным способом. Входную коллекцию они принимают в виде пары итераторов, что делает невозможным применение одной из этих функций к коллекции, которую создает другая, без сохранения этой промежуточной коллекции в отдельную переменную. Вместо того чтобы возвращать коллекцию – результат фильтрации или преобразования, они требуют, чтобы им на вход в качестве аргумента передали итератор по этой коллекции. В качестве результата эти алгоритмы возвращают итератор вывода, указывающий в коллекции-приемнике на элемент, следующий после последнего вставленного:

```
OutputIt copy_if(InputIt first, InputIt last,
                OutputIt destination_begin,
                UnaryPredicate pred);

OutputIt transform(InputIt first, InputIt last,
                  OutputIt destination_begin,
                  UnaryOperation transformation);
```

Это исключает всякую возможность сочленять данные алгоритмы без создания промежуточных переменных, что и видно из предыдущего примера.

Хотя это и может показаться огрехом проектирования библиотеки, которого нужно было избежать, для такого решения, несмотря на все связанные с ним проблемы, есть практические причины. Во-первых, передача коллекции в виде пары итераторов позволяет применять алгоритмы не только к коллекции целиком, но и к ее части. Кроме того, передача итератора вывода в качестве аргумента вместо возврата коллекции позволяет делать тип коллекции-результата отличным от типа исходной коллекции. Например, если нужно получить все *различные* имена женщин, в качестве итератора вывода можно передать итератор по контейнеру типа `std::set`.

Возврат алгоритмом итератора, указывающего в коллекции-приемнике на элемент, следующий после последнего вставленного, тоже имеет свои преимущества. Скажем, если нужно создать массив из тех же людей, что и в исходной коллекции, переупорядоченных таким образом, чтобы первыми в нем находились женщины (то есть сделать примерно то же, что и алгоритм `std::stable_partition`, но с созданием новой коллекции вместо изменения исходной), достаточно просто вызвать функцию `std::copy_if` дважды: в первый раз скопировать всех женщин, а за второй вызов – всех прочих. Первый вызов вернет итератор, указывающий на позицию, с которой должен начинать вставку элементов второй вызов.

### Листинг 2.10 Разделение списка

```
std::vector<person_t> separated(people.size());

const auto last = std::copy_if( ←
    people.cbegin(), people.cend(),
    separated.begin(),
    ...)
```

Возвращает позицию после последнего вставленного элемента

```

        is_female);

std::copy_if(
    people.cbegin(), people.cend(),
    last, ← Новые элементы будут вставлены после женщин
    is_not_female);

```

Хотя стандартные алгоритмы предоставляют способ писать код в стиле, более близком к функциональному, чем если писать своими руками циклы и ветвления, лежащие в их основе принципы не позволяют состыковывать их между собой так, как это принято в других библиотеках и языках функционального программирования. В главе 7 будет показан подход, используемый в библиотеке `ranges`, более подходящий для функциональной композиции.

## 2.4 Создание своих функций высшего порядка

В библиотеке STL и свободно распространяемых сторонних библиотеках, таких как Boost, реализовано множество алгоритмов, однако часто возникает необходимость реализовать самостоятельно какие-либо алгоритмы, специфические для той или иной предметной области. Иногда бывает нужна специализированная версия стандартного алгоритма, в других случаях требуется воплотить что-то особенное полностью самостоятельно.

### 2.4.1 Передача функции в качестве аргумента

В предыдущем примере нужно было получить имена всех женщин из заданной коллекции. Предположим теперь, что имеется коллекция людей и что часто возникает потребность получить имена тех из них, кто удовлетворяет некоторому условию, но в этот раз мы не хотим ограничивать себя заранее фиксированным условием наподобие `is_female`. Вместо этого нужна возможность работать с любым предикатом, принимающим аргумент типа `person_t`. Так, пользователь может выделять людей по возрасту, цвету волос, семейному положению и т. д.

Для этого было бы полезно сделать единую функцию, которую можно вызывать для всего множества таких задач. Эта функция должна принимать вектор людей и функцию-предикат для фильтрации, а возвращать вектор строк, содержащий имена людей, удовлетворяющих предикату.

Данный пример может показаться чересчур узким. Было бы гораздо полезнее, если бы функция могла работать с любым типом коллекции на входе и на выходе, чтобы не быть ограниченной исключительно векторами людей. Также было бы лучше, если бы функция могла выделять любые данные о человеке, а не только имя. Читатель может сделать код более обобщенным, но мы стремимся показать простое решение, чтобы сфокусировать внимание на самом важном.

Очевидно, что вектор людей должен передаваться по константной ссылке, но как быть с аргументом-функцией? Можно воспользовать-

ся указателем на функцию, но это накладывало бы слишком жесткие ограничения. В языке C++ многие сущности могут вести себя подобно функциям, а использование универсального типа, охватывающего все такие сущности, неизбежно влечет за собой потерю производительности. Поэтому, вместо того чтобы гадать, какой тип лучше всего подойдет аргументу-функции, можно сделать тип функции параметром шаблона, и пускай компилятор сам определяет конкретный тип.

```
template <typename FilterFunction>
std::vector<std::string> names_for(
    const std::vector<person_t>& people,
    FilterFunction filter);
```

Такое решение позволяет пользователю передавать в функцию любую сущность, если та ведет себя как функция; в частности, ее можно вызывать таким же образом, как если бы она была обычной функцией. Все виды сущностей, которые в языке C++ могут играть роль функций, будут подробно рассмотрены в следующей главе.

## 2.4.2 Реализация на основе циклов

В предыдущих разделах было показано, как реализовать функцию `names_for` на основе алгоритмов из библиотеки STL. Хотя обычно рекомендуется использовать средства стандартной библиотеки всюду, где это возможно, это правило не следует считать абсолютным. При желании его можно нарушать, только для этого нужно иметь вескую причину и быть заранее готовым к проникновению в код большего числа ошибок. В данном конкретном случае из-за того, что использование стандартных алгоритмов требует лишней операций по выделению динамической памяти и копированию промежуточных данных, вполне оправданным будет реализовать предыдущий пример своими руками, на основе цикла, см. листинг 2.11.

### Призвать на помощь диапазоны из библиотеки `ranges`

Проблема ненужных операций выделения памяти проистекает из недостаточной пригодности алгоритмов из стандартной библиотеки к композициям. Эта проблема хорошо известна на протяжении некоторого времени, и для ее решения создан ряд библиотек.

В главе 7 понятие диапазонов (`ranges`) рассматривается подробно. В настоящее время поддержка диапазонов опубликована в виде технической спецификации и планируется к включению в стандарт C++20. До тех пор, пока диапазоны не стали частью стандартной библиотеки, они доступны в виде сторонней библиотеки, которую можно использовать с большинством компиляторов, совместимых со стандартом C++11.

С помощью диапазонов программист может добиться того, чтобы волки были сыты и овцы целы, создавая мелкие функции и сочленяя их между собой без какого бы то ни было ущерба для производительности.

## Листинг 2.11 Реализация функции своими руками на основе цикла

```

template <typename FilterFunction>
std::vector<std::string> names_for(
    const std::vector<person_t>& people,
    FilterFunction filter)
{
    std::vector<std::string> result;

    for (const person_t& person : people) {
        if (filter(person)) {
            result.push_back(name(person));
        }
    }

    return result;
}

```



В том, что столь простая функция вручную реализована на основе цикла, большой беды нет. Имени функции и ее параметров довольно, чтобы ее предназначение стало очевидным. Большая часть алгоритмов в библиотеке STL реализована в виде циклов, подобно этой функции.

Но если сами стандартные алгоритмы реализованы в виде циклов, что же плохого в том, чтобы в своих программах реализовывать все циклами? Зачем вообще утруждать себя изучением библиотеки STL?

Для этого есть несколько причин. Первая – это простота. Использование кода, который кто-то другой уже написал ранее, экономит время. Этим обусловлена вторая выгода – корректность. Если одну и ту же идею алгоритма приходится воплощать снова и снова, естественно ожидать, что когда-нибудь программист допустит ошибку. Алгоритмы из библиотеки STL тщательно протестированы и гарантированно работают корректно на любых входных данных. По этой же причине наиболее часто используемые функции в своей программе можно реализовывать самостоятельно через циклы: эти функции наверняка будут тестироваться нередко.

Хотя многие алгоритмы в библиотеке STL нельзя назвать чистыми, они оформлены в виде функций высшего порядка, что делает их более общими и расширяет сферу их применения. А чем чаще используется функция, тем меньше вероятность найти в ней незамеченную ранее ошибку.

### 2.4.3 Рекурсия и оптимизация хвостового вызова

Предыдущее решение выглядит чистым со стороны, но его реализация таковой не является. Функция модифицирует вектор `result` всякий раз, когда находит в исходной коллекции человека, удовлетворяющего критерию. В чистых функциональных языках, где нет никаких циклов, функции для линейного просмотра коллекций обычно бывают реализованы посредством рекурсии. Здесь мы не будем погружаться в данную тему слишком глубоко, поскольку программисту на языке C++ вряд ли придется пользоваться ею часто, но некоторые важные аспекты осветить нужно.

Чтобы рекурсивно обработать непустой вектор, нужно сначала обработать его *голову* (т. е. первый элемент), а дальше тем же самым алгоритмом обработать *хвост* (все остальные элементы, кроме первого), который тоже можно рассматривать как вектор. Если голова удовлетворяет предикату, ее нужно прибавить к коллекции результатов. Наконец, если на вход алгоритма приходит пустой вектор исходных данных, он должен вернуть пустой вектор результатов.

Допустим, в нашем распоряжении есть функция `tail`, которая принимает на вход вектор и возвращает его хвост. Также предположим, что дана функция `prepend`, которая принимает одиночное значение и вектор, а возвращает новый вектор, полученный из исходного вставкой нового элемента в начало.

### Листинг 2.12 Наивная рекурсивная реализация

```
template <typename FilterFunction>
std::vector<std::string> names_for(
    const std::vector<person_t>& people,
    FilterFunction filter)
{
    if (people.empty()) {
        return {};
    } else {
        const auto head = people.front();
        const auto processed_tail = names_for(
            tail(people),
            filter);
        if (filter(head)) {
            return prepend(name(head), processed_tail);
        } else {
            return processed_tail;
        }
    }
}
```

Если коллекция пуста, вернуть пустой результат

Вызвать эту же функцию рекурсивно, чтобы обработать хвост коллекции

Если первый элемент удовлетворяет предикату, добавить его к результату. Иначе пропустить элемент



Эта реализация неэффективна. Во-первых, есть причина, почему в стандартной библиотеке отсутствует функция `tail` для векторов: она должна была бы создать новый вектор и скопировать в него все элементы исходного вектора, за исключением первого. Проблему с функцией `tail` легко решить, если вместо вектора использовать в качестве аргументов пару итераторов. В этом случае получение хвоста становится тривиальным делом: достаточно увеличить на единицу итератор, отмечающий начало коллекции.

### Листинг 2.13 Рекурсивная реализация

```
template <typename FilterFunction, typename Iterator>
std::vector<std::string> names_for(
    Iterator people_begin,
```

```

    Iterator people_end,
    FilterFunction filter)
{
    ...
    const auto processed_tail = names_for(
        people_begin + 1,
        people_end,
        filter);
    ...
}

```



Второй недостаток этой реализации состоит в том, что новые элементы добавляются в начало вектора. С этим трудно что-либо поделать. Похожая ситуация имела место и в циклической реализации, хотя там элементы добавлялись в конец, что в случае вектора более эффективно.

Последняя и, пожалуй, самая важная проблема проявляется в том случае, если функция применяется к очень большой коллекции элементов. Каждый рекурсивный вызов потребляет память на стеке, поэтому в некоторый момент времени стек окажется переполнен, и программа аварийно завершится. Даже если коллекция недостаточно велика, чтобы вызвать переполнение стека, вызовы функций не бесплатны; простые машинные команды перехода, в которые компилятор превращает цикл `for`, работают гораздо быстрее.

Хотя все предыдущие проблемы можно решить относительно легко, с этой последней так просто не справиться. Программисту остается лишь надеяться, что компилятор сумеет преобразовать рекурсию в цикл. Для того чтобы это произошло, необходимо использовать особую форму рекурсии, называемую *хвостовой*. При хвостовой рекурсии вызов функции самой себя должен быть последним, что функция делает: функция не должна ничего делать с результатом рекурсивного самовывоза.

В предыдущем примере рекурсия не хвостовая, поскольку функция, получив результат самовывоза, применяет к нему еще одну операцию: если выражение `filter(head)` дает значение `true`, добавляет к нему новый элемент и лишь затем возвращает полученную коллекцию в качестве своего результата. Преобразование произвольной рекурсии в хвостовую не так тривиально, как предыдущие усовершенствования. Поскольку теперь функция должна возвращать окончательный результат, нужно найти иной способ накопления промежуточных результатов. Для этого понадобится дополнительный аргумент, который нужно передавать от вызова к вызову рекурсивной функции.

#### Листинг 2.14 Реализация на основе хвостовой рекурсии

```

template <typename FilterFunction, typename Iterator>
std::vector<std::string> names_for_helper(
    Iterator people_begin,
    Iterator people_end,
    FilterFunction filter,
    std::vector<std::string> previously_collected)
{
    if (people_begin == people_end) {

```

```

return previously_collected;

} else {
    const auto head = *people_begin;

    if (filter(head)) {
        previously_collected.push_back(name(head));
    }

    return names_for_helper(
        people_begin + 1,
        people_end,
        filter,
        std::move(previously_collected));
}
}

```



Теперь функция возвращает либо ранее вычисленное значение, либо в точности то значение, которое возвращает рекурсивный вызов. Единственное неудобство состоит в том, что для вызова этой функции нужен один дополнительный аргумент. Именно поэтому функция получила имя `names_for_helper` (англ. «вспомогательная»); ту же функцию, что нужна пользователю, можно реализовать тривиально, вызвав вспомогательную функцию и передав ей пустой вектор в качестве параметра `previously_collected`.

#### Листинг 2.15 Вызов вспомогательной функции



```

template <typename FilterFunction, typename Iterator>
std::vector<std::string> names_for(
    Iterator people_begin,
    Iterator people_end,
    FilterFunction filter)
{
    return names_for_helper(people_begin,
        people_end,
        filter,
        {});
}

```

В этом случае компиляторы, поддерживающие оптимизацию хвостовых вызовов (tail-call optimization, TCO), получают возможность преобразовать рекурсивную функцию в обычный цикл и сделать ее столь же эффективной, как и код, с которого начинался разбор этого примера (листинг 2.11).

**ПРИМЕЧАНИЕ** Стандарт языка C++ не дает никаких гарантий того, что оптимизация хвостовых вызовов будет произведена. Однако большинство современных компиляторов, включая GCC, Clang и MSVC, поддерживают ее. Иногда они даже умеют оптимизировать взаимно рекурсивные вызовы (то есть такие, когда функция `f` вызывает функцию `g`, а та, в свою очередь, вызывает `f`) и некоторые функции, в которых рекурсия, строго говоря, не хвостовая, но достаточно близка к ней.



Рекурсия предоставляет мощный механизм для реализации итеративных алгоритмов в языках, не имеющих оператора цикла. Однако рекурсия – это все же довольно низкоуровневая конструкция. С ее помощью можно добиться полной чистоты внутренней реализации функций, но во многих случаях это не так уж и нужно. Как было сказано выше, к чистоте стремятся для того, чтобы делать меньше ошибок при написании кода, избавившись от необходимости держать в голове изменяемое состояние. Однако при написании хвостовых рекурсий наподобие той, что представлена в листинге 2.14, программист имитирует изменяемое состояние и циклы, только иными средствами. Это хорошее упражнение для ума, но вряд ли что-то большее.

У рекурсий, как и у написания циклических алгоритмов своими руками, есть своя ниша. Но в большинстве случаев при рецензировании кода на языке C++ она должна служить сигналом тревоги. Нужно тщательно проверять ее корректность и убеждаться в том, что при всех возможных способах использования функции не произойдет переполнение стека.

#### 2.4.4 Реализация на основе свертки

Поскольку рекурсия относится к конструкциям сравнительно низкого уровня, реализации рекурсивных функций своими руками часто стараются избегать даже в чистых языках функционального программирования. Можно даже сказать, что конструкции высокого уровня именно потому столь популярны в функциональном программировании, что рекурсия слишком замысловата.

Выше читатель уже видел, что такое свертка, но по-настоящему глубоко это понятие пока не рассматривалось. *Свертка* каждый раз берет один элемент коллекции и применяет заданную функцию к ранее накопленному промежуточному результату и этому текущему значению, вырабатывая тем самым новое промежуточное значение. Читатели, внимательно прочитавшие предыдущий раздел, наверняка заметили, что это – в точности то, что делала хвостовая рекурсивная функция `names_for_helper` из листинга 2.14. В сущности, свертка есть не что иное, как более изящный способ создания хвостовых рекурсивных функций для обработки коллекций. Свертка позволяет абстрагироваться от общей части реализации, программисту остается задать лишь саму коллекцию, начальное значение и аккумулирующую операцию – без необходимости каждый раз своими руками писать рекурсию.

Если читателю захочется реализовать функцию `names_for` на основе свертки (в стандартной библиотеке получившей имя `std::accumulate`), это можно легко сделать теперь, когда изучена ее реализация через хвостовую рекурсию. Нужно начать с пустого вектора строк и добавлять к нему новое имя всякий раз, когда очередной элемент коллекции удовлетворяет предикату, только и всего.

**Листинг 2.16 Реализация на основе свертки**

```

std::vector<std::string> append_name_if(
    std::vector<std::string> previously_collected,
    const person_t& person)
{
    if (filter(person)) {
        previously_collected.push_back(name(person));
    }
    return previously_collected;
}

...

return std::accumulate(
    people.cbegin(),
    people.cend(),
    std::vector<std::string>{},
    append_name_if);

```

**Слишком много копий**

Если запустить функцию из листинга 2.16 на достаточно большой коллекции, можно заметить, что она работает медленно. Причина состоит в том, что функция `append_name_if` получает вектор-аргумент по значению, а это означает создание новой копии вектора всякий раз, когда функция `std::accumulate` вызывает функцию `append_name_if` и передает ей ранее накопленный результат.

Создание копии при каждом вызове оказывается в этом случае ненужной пессимизацией, которая будет исправлена в стандарте C++20. До выхода нового стандарта программист может пользоваться собственной версией алгоритма `std::accumulate`, в которой для передачи накопленного значения применяется семантика перемещения вместо копирования (именно так будет работать алгоритм `std::accumulate` в стандарте C++20). Реализацию функции `moving_accumulate` можно найти в примерах кода к этой книге.

Свертка – это мощный инструмент. Читатель имел возможность убедиться, что он достаточно выразителен, чтобы на его основе воплотить подсчет элементов, поэлементное преобразование (часто обозначаемое `transform` или `map`) и фильтрацию (в последнем примере реализована комбинация алгоритмов `transform` и `filter`). С помощью свертки можно реализовать немалую часть стандартных алгоритмов. Интересным упражнением может послужить реализация таких алгоритмов, как `std::any_of`, `std::all_of` и `std::find_if` на основе функции `std::accumulate`. При этом стоит проверить, работают ли эти реализации столь же быстро, как и исходные алгоритмы, а если они окажутся медленнее, выяснить, почему. Еще один алгоритм, который может стать превосходным упражнением на применение свертки, – это сортировка вставками.

**СОВЕТ** Более подробную информацию и ссылки по темам, затронутым в этой главе, можно найти на странице <https://forums.manning.com/posts/list/41681.page>.

## Итоги

- Передавая свои предикаты и функции в такие алгоритмы, как `transform` и `filter`, можно настраивать поведение этих обобщенных алгоритмов так, чтобы они решали конкретную прикладную задачу.
- Хотя алгоритм `std::accumulate` находится в заголовочном файле `<numeric>`, его можно применять к гораздо более широкому кругу задач, чем чисто арифметические вычисления.
- Алгоритм `std::accumulate` реализует понятие *свертки*, которое не ограничивается сложением или перемножением элементов и на основе которого можно построить множество стандартных алгоритмов.
- Если хочется сохранить неизменным порядок элементов при их перемещении по контейнеру, следует пользоваться алгоритмом `std::stable_partition` вместо более быстрого `std::partition`. По этой же причине, в особенности в задачах, касающихся визуального интерфейса пользователя, алгоритм `std::stable_sort` часто бывает предпочтительнее, чем `std::sort`.
- Хотя алгоритмы из стандартной библиотеки разрабатывались так, чтобы их можно было состыковывать между собой, они поддерживают композицию совсем не в том виде, как она выглядит в чисто функциональных языках. Этот недостаток легко устранить, если использовать диапазоны (`ranges`).
- То обстоятельство, что большая часть стандартных алгоритмов обработки коллекций реализована в виде циклов, отнюдь не означает, что программисту следует собственными руками писать циклы. Подобным образом, хотя операторы `while`, `if` и другие реализуются на основе команд перехода, программисты, как правило, не используют в своем коде операторы `goto` по хорошо известным причинам.

---



# Функциональные объекты

---



## О чем говорится в этой главе:

- различные сущности в языке C++, которые можно использовать в роли функций;
- создание обобщенных функциональных объектов;
- что такое лямбда-выражения и как они соотносятся с обычными функциональными объектами;
- создание изящных функциональных объектов с помощью библиотеки Boost.Phoenix и своими руками;
- что такое тип `std::function` и когда его стоит использовать.

В предыдущей главе читатель увидел, как определять функции, способные принимать другие функции в качестве своих аргументов. Теперь пришло время заняться другой стороной медали – изучить все виды сущностей, которые можно использовать в языке C++ в роли функций. Тема может показаться не слишком увлекательной, но ее необходимо разобрать, чтобы глубже понять суть функций в языке C++ и достичь заветной цели – научиться использовать высокоуровневые абстракции, не жертвуя при этом производительностью. Всякому, кто желает сполна воспользоваться всей мощью языка C++ в своем путешествии в мир функционального программирования, нужно хорошо знать все те «вещи», которые язык C++ позволяет рассматривать как функции, каким из них отдавать предпочтение, а каких избегать.

Если воспользоваться метафорой *утиной типизации* («если это выглядит как утка, плавает как утка и крикает как утка, то это, вероятно, и есть утка»), можно дать такую формулировку: все, что можно вызвать

подобно функции, есть функциональный объект. Более конкретно, если можно написать имя некоторой сущности, а вслед за ним указать аргументы в скобках, т. е. составить выражение вида  $f(\text{arg1}, \text{arg2}, \dots, \text{argn})$ , эта сущность является функциональным объектом. В данной главе мы рассмотрим все виды сущностей, которые в языке C++ рассматриваются как функциональные объекты.

Чтобы отличать обычные функции языка C++ от всех тех вещей, которые могут использоваться в роли функций, будем называть последние *функциональными объектами* в случаях, когда требуется явно провести такое различие.



## 3.1 Функции и функциональные объекты

*Функция* – это именованная группа операторов, которую можно вызывать из других частей программы или из самой этой функции (в этом случае функция будет рекурсивной). Язык C++ предоставляет два различных способа записать определение функции:

<code>int max(int arg1, int arg2) { ... }</code>	←	Старый синтаксис, унаследованный от языка C
<code>auto max(int arg1, int arg2) -&gt; int { ... }</code>	←	Новый синтаксис с возвращаемым типом в конце

Хотя некоторые люди предпочитают писать возвращаемый тип функции после ее аргументов, даже когда в этом нет необходимости, этот стиль не стал общепринятым. Данный синтаксис полезен главным образом при написании шаблонов функций, когда возвращаемый тип каким-то образом зависит от типов аргументов.

### Хвостовое объявление возвращаемого типа

В предыдущем фрагменте кода не имеет значения, где указывать тип возвращаемого значения: перед именем функции и перечнем аргументов или после них. Объявление возвращаемого типа после имени функции и ее аргументов бывает необходимо, когда создается шаблон функции и тип возвращаемого значения нужно вывести из типов аргументов.

Некоторые программисты предпочитают всегда использовать хвостовой синтаксис для возвращаемого типа, поскольку считают тип гораздо менее важной характеристикой функции, чем имя и аргументы, и потому его стоит писать в конце объявления. Хотя это и здравая позиция, повсеместно распространен по-прежнему традиционный подход.

### 3.1.1 Автоматический вывод возвращаемого типа

Начиная со стандарта C++14 разрешается опускать возвращаемый тип и поручать компилятору его автоматический вывод, исходя из выраже-

ния, использованного в операторе `return`. Вывод типа в этом случае осуществляется по тем же правилам, что и вывод типов-аргументов шаблона.

В следующем примере объявлена целочисленная переменная `answer` и две функции, `ask1` и `ask2`. Эти функции обладают одинаковым телом – они просто возвращают переменную `answer`. Но возвращаемый тип у них объявлен по-разному. Первая функция возвращает значение, чей тип выводится автоматически, тогда как вторая возвращает константную ссылку на нечто, чей тип выводится автоматически. В обоих случаях компилятор посмотрит на тип переменной `answer`, к которой применяется оператор `return`, обнаружит, что это тип `int`, и заменит им ключевое слово `auto`.

```
int answer = 42;
auto ask1() { return answer; } ← Возвращаемый тип - int
const auto& ask2() { return answer; } ← Возвращаемый тип - const int&
```

Хотя правила вывода типов-аргументов шаблона (и, следовательно, вывод возвращаемого типа) на самом деле сложнее, чем «просто заменить слово `auto` известным типом», в целом они работают интуитивно понятным образом, поэтому мы не будем рассматривать их здесь<sup>1</sup>.

Если в теле функции имеется несколько операторов `return`, все они должны возвращать значения одного типа. Если их типы различаются, компилятор сообщит об ошибке. В следующем фрагменте кода показана функция, которая принимает на вход логическое значение (флаг) и, в зависимости от его истинности или ложности, возвращает значение типа `int` или типа `std::string`. Компилятор отвергнет этот код, потому что из первого оператора `return` будет выведен целочисленный тип, а для второй оператор возвращает значение иного типа:

```
auto ask(bool flag)
{
    if (flag) return 42;
    else     return std::string("42"); ← Ошибка - не совпадают выведенные типы
                                           возвращаемого значения: int и std::string
}
```

После того как тип возвращаемого значения выведен, его можно использовать в оставшейся части функции. Это позволяет создавать рекурсивные функции с автоматически выведенным типом, как показано в следующем примере:

```
auto factorial(int n)
{
    if (n == 0) {
        return 1; ← Тип результата выводится как int
    } else {
        return factorial(n - 1) * n; ← Уже известно, что функция factorial
                                           возвращает значение типа int,
                                           а умножение двух чисел типа int
                                           дает результат типа int, поэтому все в порядке
    }
}
```

<sup>1</sup> Подробное изложение правил автоматического вывода типов-аргументов шаблона можно найти в справочнике по языку C++: <http://mng.bz/YXIU>.

Если переставить местами ветки `if` и `else`, получится ошибка компиляции, поскольку компилятор встретит рекурсивный вызов функции `factorial` до того, как этот тип будет выведен. При написании рекурсивных функций нужно либо в явном виде указать возвращаемый тип, либо первым разместить тот оператор `return`, который не содержит рекурсивного вызова.

Помимо ключевого слова `auto`, которое означает, что для типа возвращаемого значения будут использованы те же правила вывода, что и для типов-аргументов шаблона, можно в качестве возвращаемого типа использовать также конструкцию `decltype(auto)`. В этом случае возвращаемым типом функции будет тип `decltype` самого возвращаемого выражения. Проиллюстрируем это примером:

```
decltype(auto) ask() { return answer; }
decltype(auto) ask() { return (answer); }
decltype(auto) ask() { return 42 + answer; }
```

← Возвращаемый тип `decltype(answer)` есть `int`  
 ← Возвращаемый тип `decltype((answer))` есть ссылка на `int`, тогда как с ключевым словом `auto` был бы выведен тип `int`  
 ← Возвращаемый тип `decltype(42 + answer)` есть `int`

Это может пригодиться при написании шаблона функции высшего порядка, которая возвращает без изменений результат другой функции, переданной в качестве аргумента. В этом случае неизвестно заранее, какого типа функция будет передана, поэтому неизвестно, нужно ли ее результат отдавать наружу по значению или по ссылке. Если отдавать по ссылке результат функции, возвращающей значение, будет возвращена ссылка на временный объект, что приведет к неопределенному поведению. Если же передать наружу по значению результат функции, вернувшей ссылку, это приведет к созданию временной копии. Копирование может ухудшить быстродействие кода и иногда оказывается просто семантически неверным – ведь вызывающий код может требовать именно ссылку на объект-оригинал.

Поэтому для идеальной передачи из функции наружу результата, возвращенного другой функцией (т. е. для того, чтобы результат внутренней функции вернуть без каких бы то ни было изменений), нужно использовать конструкцию `decltype(auto)` в качестве спецификации возвращаемого типа, как показано в следующем примере:

```
template <typename Object, typename Function>
decltype(auto) call_on_object(Object&& object, Function function)
{
    return function(std::forward<Object>(object));
}
```

В этом небольшом примере рассмотрен простой шаблон функции, принимающий два аргумента: объект и функцию, которая должна быть применена к этому объекту. Объект `object` идеально передается в функцию `function` в качестве аргумента – благодаря использованию специальной функции `std::forward`; а за счет спецификации возвращаемого типа

конструкцией `decltype(auto)` результат функции `function` идеально передается в контекст, вызвавший функцию `call_on_object`.

### ИДЕАЛЬНАЯ ПЕРЕДАЧА АРГУМЕНТОВ



Иногда бывает нужно создавать функции, играющие роль оберток для других функций. Единственное, что должна делать такая функция, – это вызывать обертываемую функцию, возможно, модифицируя, добавляя или устранив некоторые из ее аргументов. В этом случае возникает вопрос: как аргументы функции-обертки передавать в ту функцию, которую она вызывает?

В ситуациях, подобных той, что описана в предыдущем примере, вызывающий контекст передает функцию, о которой обертка ничего не знает. Автор обертки, в частности, не может знать, какого способа передачи аргументов она ожидает.

Одно возможное решение состоит в том, чтобы функция-обертка `call_on_object` принимала аргумент `object` по значению и передавала его в обернутую функцию. Это, однако, приведет к проблемам в случае, если обертываемая функция ожидает ссылку на объект, для того чтобы модифицировать его. Внесенное в объект изменение не будет видно за пределами функции `call_on_object`, поскольку изменению подвергается локальная копия объекта, существующая только внутри функции `call_on_object`.

```
template <typename Object, typename Function>
decltype(auto) call_on_object(Object object,
                              Function function)
{
    return function(object);
}
```

Вторая возможность состоит в том, чтобы передавать объект по ссылке. Это сделает изменения, вносимые в объект, видимыми для кода, вызывающего функцию `call_on_object`. Но теперь проблема возникает в случае, когда обертываемая функция `function` не собирается изменять объект и принимает аргумент по константной ссылке. Вызывающий код окажется не в состоянии применить функцию `call_on_object` на константном объекте или на временном значении. Поэтому идея принимать аргумент `object` по обычной ссылке не подходит. Сделать эту ссылку константной тоже не получается, потому что некоторые обертываемые функции могут все же модифицировать объект-аргумент.

В некоторых библиотеках, разработанных до выхода стандарта C++11, с этой трудностью боролись, создавая перегрузки для обоих видов ссылок: константных и неконстантных. Это непрактично, так как число необходимых перегрузок растет экспоненциально с ростом числа аргументов, которые нужно передавать в обертываемую функцию.

В языке C++11 проблема решается с помощью *пересылающих ссылок* (forwarding references), также известных под названием *универсальных*



ссылки<sup>1</sup> (universal references). Пересылающая ссылка обозначается двумя знаками ссылки, примененными к типу-параметру шаблона. В следующем примере аргумент `fwd` представляет собой пересылающую ссылку на объект типа `T`, тогда как аргумент `value` – это обычная ссылка `rvalue`:

```
template <typename T>
void f(T&& fwd, int&& value) { ... }
```

Пересылающая ссылка позволяет принимать константные, неконстантные и временные объекты. Теперь функции-обертке остается лишь передать этот аргумент в обертываемую функцию, причем сделать это нужно, сохранив ту же категорию выражения, по которой данный аргумент был получен. Именно это делает функция `std::forward`.

### 3.1.2 Указатели на функции



Если переменная имеет тип указателя на функцию, это значит, что в ней хранится адрес, по которому функция расположена в памяти и по которому эту функцию можно позднее вызвать. Язык C++ унаследовал эту весьма низкоуровневую конструкцию от языка C для поддержки полиморфного кода. Полиморфизм времени выполнения достигается за счет того, что в переменную можно подставлять адрес то одной, то другой функции, тем самым меняя поведение кода, вызывающего функцию по указателю.

Указатели и ссылки на функции являются функциональными объектами, поскольку их можно вызывать так же, как и обычные функции. Помимо того, и все типы, допускающие неявное преобразование к типу указателя на функцию, также являются функциональными объектами, хотя пользоваться этим приемом в своих программах не рекомендуется. Вместо этого лучше пользоваться функциональными объектами в узком смысле (о них речь пойдет вскоре), у которых и возможности шире, и обращаться с ними удобнее. Объекты, поддерживающие преобразование в указатели на функции, могут быть полезны разве что в случаях, когда нужно взаимодействовать с кодом, написанным на языке C, умеющим принимать лишь указатели на функции, а приложению нужно передавать в него нечто более сложное, чем просто функцию.

Следующий пример иллюстрирует вызов через указатель на функцию, ссылку на функцию и объект, преобразующийся в указатель на функцию.

```
int ask() { return 42; }

typedef decltype(ask)* function_ptr;

class convertible_to_function_ptr {
```

<sup>1</sup> Название дано Скоттом Мейерсом в работе «Universal References in C++11» (Standard C++ Foundation, Nov. 1, 2012), доступной по ссылке <http://mng.bz/Z7nj>.

```

public:
    operator function_ptr() const
    {
        return ask;
    }
};

int main(int argc, char* argv[])
{
    auto ask_ptr = &ask;
    std::cout << ask_ptr() << '\n';

    auto& ask_ref = ask;
    std::cout << ask_ref() << '\n';

    convertible_to_function_ptr ask_wrapper;
    std::cout << ask_wrapper() << '\n';
}

```

Операция приведения типа может возвращать лишь указатель на существующую функцию. Она может возвращать указатели на разные функции в зависимости от определенных условий, но не может передавать в них никаких данных (если только не пользоваться грязными уловками)

Указатель на функцию

Ссылка на функцию

Объект, допускающий неявное преобразование в указатель на функцию

Этот пример показывает, как сделать указатель (`ask_ptr`) на определенную обычную функцию и ссылку (`ask_ref`) на ту же самую функцию; эти указатель и ссылку можно вызывать так, как если бы они сами были функциями, – используя обычный синтаксис вызова функций. В примере также показано, что можно определить объект, поддерживающий преобразование в указатель на функцию, и что его можно без всяких затруднений вызвать так, как если бы он был обычной функцией.

### 3.1.3 Перегрузка операции вызова

Помимо возможности определять типы, допускающие неявное преобразование в указатели на функции, язык C++ предоставляет и гораздо более изящный способ создавать свои типы, ведущие себя подобно функциям: перегружать в классах операцию вызова. В отличие от других перегружаемых операций, операция вызова может иметь любое число аргументов, а аргументы могут иметь любой тип, поэтому программист может создавать функциональные объекты с какой угодно сигнатурой.

Синтаксис для перегрузки операции вызова состоит всего-навсего в объявлении в классе еще одной функции-члена, только с особым именем `operator()`. При этом нужно указать возвращаемый тип и все аргументы, которых требует данная функция:

```

class function_object {
public:
    return_type operator()(arguments) const
    {
        ...
    }
};

```

Такие функциональные объекты обладают одним важным преимуществом по сравнению с обычными функциями: каждый экземпляр может обладать собственным состоянием, хоть изменяемым, хоть неизменяемым. Это состояние может влиять на поведение функции без необходимости передавать его из вызывающего кода.

Допустим, есть список людей, как и в примерах из предыдущей главы. У каждого человека есть имя, возраст и еще несколько характеристик, которые сейчас не важны. Нужно предоставить пользователю возможность подсчитать в этом списке число людей, возраст которых превышает определенную величину.

```
bool older_than_42(const person_t& person)
{
    return person.age > 42;
}

std::count_if(persons.cbegin(), persons.cend(),
              older_than_42);
```

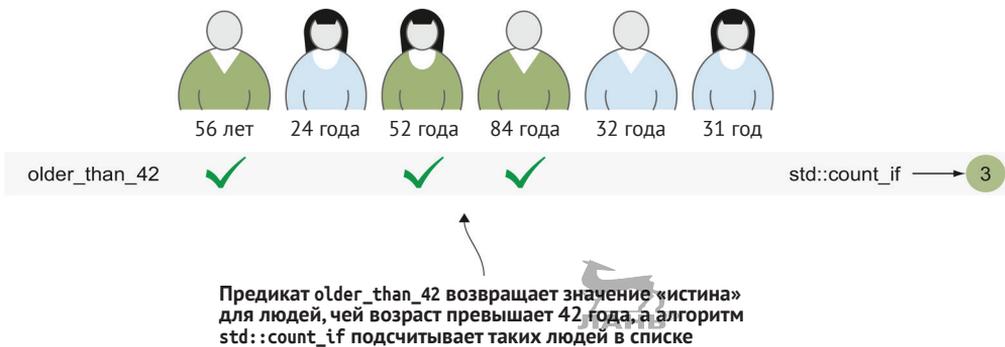


Рис. 3.1 Обычную функцию, проверяющую, старше ли человек 42 лет, можно использовать для подсчета в списке числа людей, удовлетворяющих данному условию. Однако этот подход позволяет вести подсчет только для единственного заранее определенного значения

Это решение немасштабируемо, поскольку для каждого значения возраста нужно либо создавать отдельную функцию, либо прибегать к чертовым ошибкам трюкам – например, хранить значение возраста, по которому ведется сравнение, в глобальной переменной.

Вместо этого гораздо лучше создать функциональный объект (в узком смысле, т. е. объект класса с перегруженной операцией применения), который хранит в своем внутреннем состоянии требуемый возраст. Это позволяет написать реализацию такого предиката один раз и создавать несколько экземпляров с различными значениями граничного возраста:

```
class older_than {
public:
    older_than(int limit)
        : m_limit(limit)
    {
    }
};
```

```

bool operator()(const person_t& person) const
{
    return person.age() > m_limit;
}

private:
    int m_limit;
};

```



Теперь можно определить несколько переменных этого типа и использовать их так, как если бы они были функциями:

```

older_than older_than_42(42);
older_than older_than_14(14);

if (older_than_42(person)) {
    std::cout << person.name() << " старше 42 лет\n";
} else if (older_than_14(person))
    std::cout << person.name() << " старше 14 лет\n";
} else {
    std::cout << person.name() << " 14 лет или младше\n";
}

```

**ОПРЕДЕЛЕНИЕ** Классы с перегруженной операцией вызова часто называют *функторами*. Такое использование данного термина весьма спорно, потому что за ним уже закреплено другое значение в теории категорий. Хотя мы могли бы не считаться с этим обстоятельством, теория категорий чрезвычайно важна для функционального программирования (и для программирования вообще), поэтому будем все же называть их *функциональными объектами*.

Алгоритму `std::count_if` все равно, что именно передается ему в качестве предиката, – если только эту сущность можно вызывать как обычную функцию, – а перегрузка операции вызова обеспечивает именно такую возможность:

```

std::count_if(persons.cbegin(), persons.cend(),
              older_than(42));
std::count_if(persons.cbegin(), persons.cend(),
              older_than(16));

```

Здесь важно иметь в виду, что шаблонные функции наподобие `std::count_if` не требуют, чтобы им передавали аргументы какого-то определенного типа. Они требуют лишь, чтобы аргументы обладали всеми свойствами, которые используются в реализации шаблонной функции. В данном примере алгоритм `std::count` требует, чтобы два первых аргумента вели себя как однонаправленные итераторы, а третий аргумент обладал поведением функции. Иногда говорят, что данная возможность языка C++ делает язык *слабо типизированным*, но это лишь неправильное употребление термина. Язык остается сильно типизированным – просто механизм шаблонов позволяет использовать утиную типизацию. В частности, на этапе выполнения не нужно проверять,

является ли переданный в функцию аргумент функциональным объектом, – все необходимые проверки выполняются во время компиляции.

### 3.1.4 Обобщенные функциональные объекты

В предыдущем примере показан функциональный объект, который проверяет, достиг ли человек определенного возраста. Это решение избавило нас от необходимости определять по отдельной функции для каждого значения возраста, но оно все еще остается слишком ограниченным: полученный функциональный объект принимает в качестве аргумента лишь объекты типа «человек» (`person`).

Между тем информацию о возрасте могут содержать и другие типы, представляющие как конкретные вещи (например, автомобили или домашние питомцы), так и более абстрактные явления (скажем, проекты). Если захочется подсчитать число автомобилей, которым более 5 лет (рис. 3.2), для этой задачи не удастся применить показанный выше функциональный объект, так как он определен исключительно для людей.

Снова возникает та же проблема: вместо создания отдельного функционального класса для каждого типа данных хотелось бы иметь возможность один раз определить такой класс функциональных объектов, который можно было бы использовать с любым типом данных, обладающим информацией о возрасте. Эту задачу можно решить средствами объектно-ориентированного программирования – создав базовый класс с виртуальной функцией `.age()`, но такой подход сопряжен с известной потерей производительности на этапе выполнения и, кроме того, вынуждает программиста все классы, для которых требуется поддержка функции `older_than`, делать подклассами этого базового класса. Подобный подход нарушает инкапсуляцию, и мы не будем его здесь рассматривать.

Один достаточно хороший подход состоит в том, чтобы превратить класс `older_than` в шаблон класса. Его параметром должен быть тип тех объектов, чей возраст предполагается проверять:

```
template <typename T>
class older_than {
public:
    older_than(int limit)
        : m_limit(limit)
    {
    }

    bool operator()(const T& object) const
    {
        return object.age() > m_limit;
    }

private:
    int m_limit;
};
```

Теперь шаблон `older_than` можно использовать с любым типом данных, у которого есть метод `.age()`, как показано ниже:

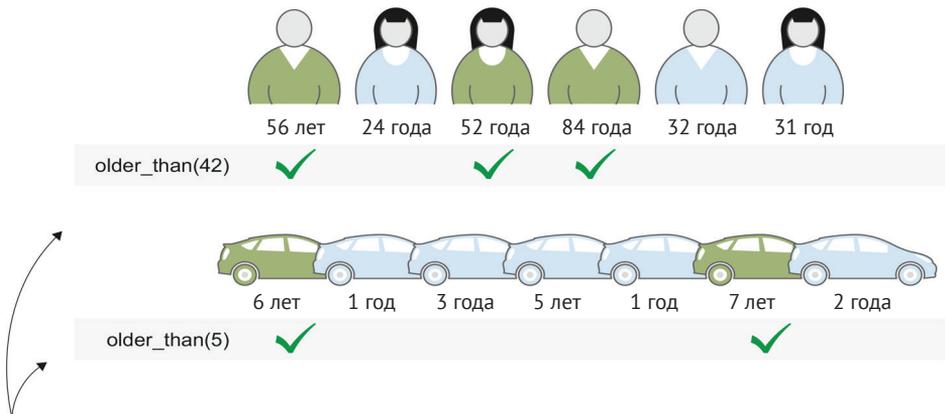
```
std::count_if(persons.cbegin(), persons.cend(),
             older_than<person_t>(42));
std::count_if(cars.cbegin(), cars.cend(),
             older_than<car_t>(5));
std::count_if(projects.cbegin(), projects.cend(),
             older_than<project_t>(2));
```



К сожалению, при создании функционального объекта программист вынужден явно указывать тип объектов-аргументов, для которых будет проверяться возраст. Хотя иногда это может оказаться полезным, в большинстве случаев слишком утомительно. Помимо того, возможны проблемы, если тип данных, фактически переданный в операцию вызова, не совпадает в точности с типом, указанным при инстанцировании шаблона.



Вместо того чтобы создавать шаблон класса, можно операцию вызова сделать шаблонным методом, как показано в следующем листинге (его можно также найти среди примеров исходного кода к этой книге: `older_than-generic/main.cpp`). В этом случае при создании функциональных объектов класса `older_than` не нужно указывать тип данных, с которым объект будет работать, – компилятор автоматически выведет тип аргумента при каждом вызове этой операции.



Один и тот же обобщенный предикат позволяет проверять достижение определенной границы возраста для объектов данных разных типов

Рис. 3.2 Полезно, когда предикат не только позволяет задавать различные границы возраста, но и поддерживает разные типы, обладающие информацией о возрасте

**Листинг 3.1 Функциональный объект с обобщенной операцией вызова**

```
class older_than {
public:
    older_than(int limit)
        : m_limit(limit)
```

```

{
}

template <typename T>
bool operator()(T&& object) const
{
    return std::forward<T>(object).age() > m_limit;
}

private:
    int m_limit;
};

```

Аргумент нужно передать идеальным образом, потому что его функция-член `age` может обладать различными перегруженными реализациями для объектов `lvalue` и `rvalue`. Благодаря совершенной передаче будет вызвана правильная реализация



Теперь функциональный объект класса `older_than` можно использовать без явного указания типа объекта, к которому он применяется. Один и тот же экземпляр этого класса можно даже использовать с аргументами различных типов (если объекты разных типов нужно проверять на достижение одного и того же граничного возраста):

### Листинг 3.2 Использование функционального объекта с обобщенной операцией вызова

```
older_than predicate(5);
```

```
std::count_if(persons.cbegin(), persons.cend(), predicate);
std::count_if(cars.cbegin(), cars.cend(), predicate);
std::count_if(projects.cbegin(), projects.cend(), predicate);
```

Единственный экземпляр класса `older_than` используется для проверки возраста переданного ему объекта: не превышает ли он 5 лет. Этот экземпляр может осуществлять проверку для объектов любого типа, лишь бы они обладали функцией-членом `.age()`, возвращающей целое число или значение иного типа, допускающего преобразование к целому. Именно этот обобщенный функциональный объект обладает наиболее подходящими свойствами.

Выше мы рассмотрели, как создавать собственные типы функциональных объектов на языке C++. Такие функциональные объекты превосходно подходят для передачи в качестве аргументов алгоритмам из стандартной библиотеки или самостоятельно написанным функциям высшего порядка. Единственная проблема с ними состоит в том, что синтаксис их объявления чересчур многословен, а объявлять их часто приходится вне того контекста, где предполагается их использование.

## 3.2 Лямбда-выражения и замыкания

Все предыдущие примеры основывались на предположении, что функция, передаваемая в алгоритм, уже существует за пределами той функции, в которой она используется совместно с каким-либо алгоритмом. Необходимость писать отдельную функцию или даже целый класс может оказаться обременительной, если их единственное предназначение –

вызов алгоритма из стандартной библиотеки или какой-либо другой функции высшего порядка. Это можно даже считать изъясном в проектировании программы, ведь программист оказывается вынужден разработать и наделить самостоятельным именем функцию, которая не будет полезна ни для кого более и используется в программе лишь в одном месте – передается в качестве аргумента при вызове алгоритма.

К счастью, в языке C++ есть *лямбда-выражения*, представляющие собой «синтаксический сахар» для создания анонимных (т. е. не наделяемых собственным именем) функциональных объектов. Лямбда-выражения позволяют создавать функциональные объекты прямо в том месте программы, где они используются, а не где-то далеко за пределами функции, которая в данный момент разрабатывается. Обратимся снова к примеру, который уже рассматривался ранее: дана группа людей, и из нее требуется отобрать только женщин (рис. 3.3).

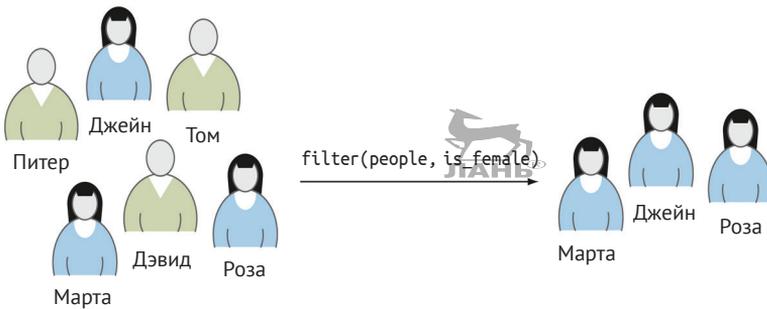


Рис. 3.3 Выборка всех женщин из заданной группы людей. Используется предикат `is_female`, представляющий собой функцию (не член класса)

В главе 2 было показано, как использовать алгоритм `copy_if`, для того чтобы отфильтровать коллекцию людей по признаку пола. При этом мы полагались на предположение, что имеется функция `is_female` – не член класса, принимающая аргумент типа `person_t` и возвращающая логическое значение «истина», если человек, переданный ей в качестве аргумента, – женщина. Поскольку тип `person_t` уже обладает функцией-членом, которая возвращает пол человека, создание еще и предикатов (оформленных в виде функций – не членов класса) для распознавания женского и мужского пола с единственной целью – использовать их при обращении к алгоритмам из стандартной библиотеки – было бы пустой тратой сил.

Вместо этого можно использовать лямбда-выражение и достичь того же результата, сохраняя при этом код компактным и не засоряя пространство имен:

```
std::copy_if(people.cbegin(), people.cend(),
            std::back_inserter(females),
            [](const person_t& person) {
```

```

        return person.gender() == person_t::female;
    }
);

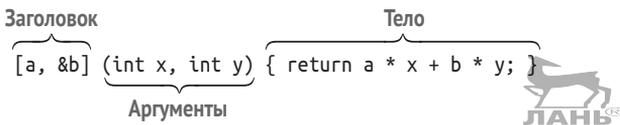
```



Функция `std::copy_if` вызывается так же, как и в предыдущем примере. Однако вместо того, чтобы передавать ей отдельно существующую функцию-предикат, мы велит компилятору создать безымянный функциональный объект, который используется только в этом месте, – функцию, которая принимает человека по константной ссылке и проверяет, женщина ли это, – и передаем этот функциональный объект в алгоритм `std::copy_if`.

### 3.2.1 Синтаксис лямбда-выражений

С синтаксической точки зрения, лямбда-выражение в языке C++ состоит из трех основных частей: заголовка, списка аргументов и тела:



Заголовок лямбда-выражения определяет, какие переменные из окружающего контекста будут в теле этого выражения – как еще говорят, захватываются лямбдой. Переменные могут захватываться по значению (такова переменная `a` в предшествующей строке кода) или по ссылке (обозначается амперсандом перед именем переменной – примером может служить переменная `b`). Если переменные захватываются по значению, их копии сохраняются в самом функциональном объекте, в который компилятор превращает лямбда-выражение; если же переменные захватываются по ссылке, сохраняется лишь адрес переменной из внешнего контекста.

Кроме того, программист может не перечислять каждую переменную, которую нужно захватить, а указать компилятору, чтобы он произвел захват из контекста всех переменных, которые упоминаются в теле лямбды. Если все переменные нужно захватить по значению, достаточно указать заголовок лямбды в виде `[=]`, а если по ссылке – то в виде `[&]`.

Рассмотрим несколько примеров того, как могут выглядеть заголовки лямбда-выражений:

- `[a, &b]` – заголовок из первого примера; переменная `a` захватывается по значению, а переменная `b` – по ссылке;
- `[]` – лямбда-выражение не захватывает никаких переменных из окружающего контекста. Такие лямбда-выражения не обладают никаким внутренним состоянием, и их можно неявно преобразовывать в обычные указатели на функции;
- `[&]` – все переменные из окружающего контекста, которые упоминаются в теле лямбды, захватить по ссылке;
- `[=]` – все переменные из окружающего контекста, которые упоминаются в теле лямбды, захватить по значению;

- [this] – используется только в нестатических функциях – членах класса и означает захватить по значению указатель this на объект, для которого вызван метод;
- [&, a] – захватить все переменные по ссылке, кроме переменной a, которая захватывается по значению;
- [=, &b] – захватить все переменные по значению, кроме переменной b, которая захватывается по ссылке.

Явное перечисление всех переменных из окружающего контекста, которые предполагается использовать в теле лямбда-выражения, может показаться утомительным по сравнению с захватом всех переменных посредством заголовков [=] и [&], однако предпочитать всегда следует именно явное перечисление, так как это предотвращает случайное использование переменной, которую программист не предполагал захватывать. Часто такая проблема возникает с указателем this, поскольку его очень легко ненамеренно втянуть в захват, просто обратившись в теле лямбда-выражения к переменной – члену класса или вызвав метод.

### 3.2.2 Что находится у лямбда-выражений «под капотом»

Хотя лямбда-выражения и предоставляют удобный синтаксис для создания функциональных объектов, в них нет ничего волшебного. С их помощью нельзя сделать ничего такого, чего нельзя было бы добиться и обычными средствами. Они суть именно то, что написано: более удобное синтаксическое средство для объявления и создания новых функциональных объектов.

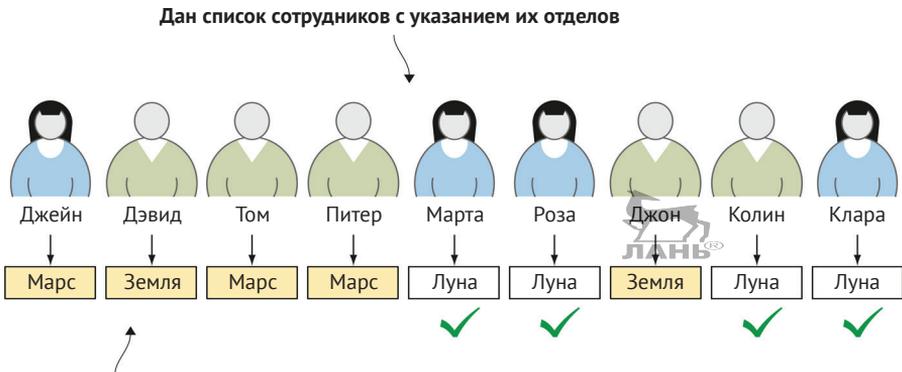
Рассмотрим это на примере. По-прежнему будем обрабатывать список людей, на этот раз – список сотрудников компании. Компания состоит из подразделений, у каждого из которых есть название. Компанию будем моделировать в программе объектами несложного класса `company_t`. Пусть у этого класса будет функция-член, позволяющая получить название подразделения, к которому относится тот или иной сотрудник. Нашей задачей будет реализовать метод, который принимает название подразделения и возвращает число сотрудников в нем. Ниже показан код, на основе которого предстоит разрабатывать свою часть:

```
class company_t {
public:
    std::string team_name_for(const person_t& employee) const;

    int count_team_members(const std::string& team_name) const;
private:
    std::vector<person_t> m_employees;
    ...
};
```

Нам нужно добавить к этому классу метод `count_team_members`. Задачу можно решить таким способом: просмотреть список всех сотрудников и для каждого из них узнать, к какому отделу он относится, при этом

подсчитать тех сотрудников, у которых название отдела совпадает с названием, переданным в функцию в качестве аргумента (рис. 3.4).



Лямбда-выражение сравнивает название отдела, к которому принадлежит сотрудник, с названием отдела, численность которого нужно определить

Рис. 3.4 В алгоритм `std::count_if` в качестве аргумента-предиката передается лямбда-выражение. Оно получает одного сотрудника и проверяет, принадлежит ли он подразделению, для которого нужно определить численность. Алгоритм `std::count_if` подсчитает все положительные ответы и вернет ответ

Как и в ранее разобранных примерах, будем использовать алгоритм `std::count_if`, но в этот раз ему в качестве предиката придется передать более сложное лямбда-выражение (в примерах кода см. файл `example:counting-team-members/main.cpp`).

### Листинг 3.3 Подсчет сотрудников в заданном подразделении

```
int company_t::count_team_members(
    const std::string& team_name) const
{
    return std::count_if(
        m_employees.cbegin(), m_employees.cend(),
        [this, &team_name]
        (const person_t& employee)
        {
            return team_name_for(employee) ==
                team_name;
        }
    );
}
```

Указатель `this` нужно захватить, чтобы иметь возможность вызывать метод `team_name_for`, а переменную `team_name` – для того, чтобы было с чем сравнивать отделы сотрудников

Как и в предыдущих примерах, у этого функционального объекта один аргумент – сотрудник. Нужно вернуть истинное значение: принадлежит ли он к заданному отделу

Что на самом деле означает этот код? В процессе компиляции лямбда-выражение будет превращено в новый класс, обладающий двумя переменными-членами: указателем на объект класса `company_t` и ссылкой на объект класса `std::string` – по одному члену данных на каждую захвачен-

ную переменную. Этот созданный компилятором класс будет обладать операцией вызова с теми же аргументами и тем же телом, что указаны в лямбда-выражении. Иными словами, получится класс, эквивалентный следующему:

#### Листинг 3.4 Превращение лямбда-выражения в класс

```
class lambda_implementation {
public:
    lambda_implementation(
        const company_t* _this,
        const std::string& team_name)
        : m_this(_this)
        , m_team_name(team_name)
        {
        }

    bool operator()(const person_t& employee) const
    {
        return m_this->team_name_for(employee) == m_team_name;
    }

private:
    const company_t* m_this;
    const std::string& m_team_name;
};
```

Помимо невидимого для программиста объявления класса, лямбда-выражение еще и создает экземпляр этого класса, называемый *замыканием*, – это объект, содержащий переменные, захваченные в текущем состоянии, в момент создания объекта.

Особенность, о которой стоит упомянуть отдельно, состоит в том, что оператор вызова у функционального объекта, получающегося из лямбда-выражения, по умолчанию константный – в отличие от остальных сущностей языка, для которых константность необходимо явно указывать. Если в лямбда-выражении предполагается изменять значения переменных, захваченных не по ссылке, а по значению, лямбда-выражение необходимо объявить с ключевым словом `mutable`. В следующем примере алгоритм `std::for_each` используется для того, чтобы пройти по списку слов и вывести на печать те из них, что начинаются с заглавной буквы. Переменная `count` при этом служит для подсчета напечатанных слов. Такой подход иногда бывает полезен для отладки, но в общем случае лямбда-выражений со словом `mutable`, способных изменять захваченные переменные, следует избегать. Для данной задачи есть, конечно, и более элегантные решения, но данный код имеет целью продемонстрировать, как работают лямбда-выражения со спецификатором `mutable`.

## Листинг 3.5 Лямбда-выражение, изменяющее захваченную переменную

```

int count = 0;
std::vector<std::string> words{"An", "ancient", "pond"};
std::for_each(words.cbegin(), words.cend(),
    [count]
    (const std::string& word)
    mutable
    {
        if (isupper(word[0])) {
            std::cout << word
                << " " << count <<std::endl;
            count++;
        }
    }
);

```

Переменная count захватывается по значению. Все ее модификации выполняются над локальной копией и видны только внутри лямбда-выражения

Спецификатор mutable ставится после списка аргументов и указывает компилятору, что операция вызова у данного функционального объекта не должна быть константной

Таким образом, лямбда-выражения не позволяют сделать ничего такого, чего невозможно было бы сделать и без них. Однако они освобождают программиста от написания большого объема кода-клише и помогают выражать логику программы более компактно, определяя вспомогательные функции именно там, где они используются, а не где-то в других местах кода. В примерах, разобранных выше, код-клише, который бы пришлось писать своими руками, не будь в языке лямбда-выражений, оказался бы более объемным, чем код, делающий полезную работу.

**ПРИМЕЧАНИЕ** Всякое лямбда-выражение компилятор превращает в класс и дает этому классу какое-то внутреннее имя. Это имя скрыто от программиста, и различные компиляторы могут по-разному генерировать такие имена. Поэтому единственный способ сохранить лямбда-выражение в переменной (если не считать такого тяжеловесного средства, как тип `std::function`, о котором речь пойдет ниже) – это объявить переменную с ключевым словом `auto`.

### 3.2.3 Создание лямбда-выражений с произвольными переменными-членами

В предыдущих примерах рассматривались лямбда-выражения, получающие доступ к окружающему контексту либо за счет ссылок на используемые переменные, либо путем сохранения копий их значений в объекте-замыкании. Однако в собственноручно созданных классах с перегруженной операцией вызова можно объявлять сколько угодно переменных-членов, никак не связанных с переменными из окружающего контекста. Их можно инициализировать как заранее заданным значением, так и результатом вызова функции. Хотя это ограничение выглядит и не очень существенным ограничением (в крайнем случае всегда можно объявить в окружающем коде локальную переменную, проинициализи-

ровать ее нужным значением и захватить ее), в некоторых случаях наличие переменных-членов может оказаться важным.

Если бы объекты можно было захватывать исключительно по ссылке и по значению, невозможно было бы хранить в лямбда-выражении объекты, поддерживающие только лишь перемещение, т. е. экземпляры таких классов, у которых определен конструктор перемещения, но отсутствует конструктор копирования. Наиболее очевидным примером такой ситуации служит передача в лямбда-выражение умного указателя типа `std::unique_ptr`, который реализует семантику исключительного владения объектом.

Допустим, требуется создать объект, представляющий сетевой запрос (англ. request), причем объект, представляющий сеанс (англ. session), уже создан и отдан во владение умному указателю `std::unique_ptr`. Создав сетевой запрос, нужно присоединить к нему функцию (заданную в виде лямбда-выражения), которая должна выполняться автоматически, как только завершится обработка запроса. Этой функции-обработчику завершения запроса нужны данные сеанса, поэтому лямбда-выражение должно захватить умный указатель на объект сеанса.

### Листинг 3.6 Ошибка при попытке захватить перемещаемый не копируемый объект

```
std::unique_ptr<session_t> session = create_session();

auto request = server.request("GET /", session->id());

request.on_completed(
    [session] ← Ошибка: у типа std::unique_ptr<session_t>
                нет конструктора копирования
    (response_t response)
    {
        std::cout << "Получен ответ: " << response
                  << " для сеанса: " << session;
    }
);
```

В этой ситуации и других подобных ей можно воспользоваться расширенным синтаксисом – обобщенным захватом. Вместо того чтобы указывать переменные, захватываемые из внешнего контекста, можно определить произвольные переменные-члены, задав их имена и начальные значения. Типы таких переменных выводятся автоматически из выражений, использованных для их инициализации.

### Листинг 3.7 Обобщенные захваты в лямбда-выражениях

```
request.on_completed(
    [ session = std::move(session), ← Передача лямбда-функции
      time = current_time()         ← исключительного владения сеансом
    ]
    (response_t response)
    {
        std::cout
```

Создается переменная-член, доступная исключительно в теле лямбда-выражения; в качестве начального значения ей присваивается текущее время



```

<< "Получен ответ: " << response
<< " для сеанса: " << session
<< " обработка заняла: "
<< (current_time() - time)
<< "миллисекунд";
    }
);

```

Этот подход позволяет перемещать объекты из окружающего кода в лямбда-функцию. Использование функции `std::move` в этом примере приводит к вызову конструктора перемещения, и переменная-член `session` функционального объекта забирает объект-сеанс в свое исключительное владение.

Помимо того, в примере показано создание новой переменной-члена с именем `time`, которая ничего не захватывает из окружающей области видимости. Это объявление совершенно новой переменной, начальным значением которой становится результат вызова функции `current_time` – оно вычисляется в момент создания функционального объекта.

### 3.2.4 Обобщенные лямбда-выражения

В предыдущих разделах было показано, что лямбда-выражения позволяют делать многое из того, на что способны обычные функциональные объекты. Ранее в этой главе был реализован обобщенный функциональный объект, у которого операция вызова была оформлена как шаблон, параметризованный типом аргумента, благодаря чему один и тот же объект стало можно использовать для подсчета в коллекции тех элементов, возраст которых превышает известный предел, – каким бы ни был тип этих элементов (листинг 3.2).

Лямбда-выражения тоже позволяют создавать обобщенные функциональные объекты – для этого вместо типов (некоторых аргументов) нужно использовать слово `auto`. Например, можно без труда создать обобщенное лямбда-выражение, способное сравнивать с заданным пределом возраст объекта произвольного типа, лишь бы он обладал методом `.age()`, – см. следующий листинг.

**Листинг 3.8** Обобщенное лямбда-выражение, принимающее аргумент любого типа, имеющего метод `age()`

```

auto predicate = [limit = 42](auto&& object) {
    return object.age() > limit;
};

std::count_if(persons.cbegin(), persons.cend(),
              predicate);
std::count_if(cars.cbegin(), cars.cend(),
              predicate);
std::count_if(projects.cbegin(), projects.cend(),
              predicate);

```

Имя типа, к которому относится аргумент, никак не задано, поэтому невозможно использовать его для идеальной передачи. Вместо этого пришлось бы писать `std::forward<decltype(object)>(object)`

Важно подчеркнуть, что обобщенное лямбда-выражение – это класс, у которого операция вызова представляет собой шаблон функции, а не шаблон класса, обладающий операцией вызова. Типы аргументов, объявленных с ключевым словом `auto` вместо типа, выводятся компилятором в каждом месте кода, где функциональный объект вызывается, а не один раз при его создании, поэтому одно и то же обобщенное лямбда-выражение можно применять к объектам разных типов.

### Еще более обобщенные лямбда-выражения в стандарте C++20

Если у лямбда-выражения несколько аргументов объявлены со словом `auto`, их типы выводятся отдельно и независимо друг от друга, исходя из конкретного вызова. Если же хочется создать такое обобщенное лямбда-выражение, у которого несколько аргументов должны относиться к одному и тому же автоматически выведенному типу, нужно прибегнуть к хитрости и воспользоваться оператором `decltype`, чтобы иметь возможность использовать фактический тип одного из таких аргументов.

Пусть, например, требуется создать обобщенное лямбда-выражение с двумя аргументами одного типа, которое проверяет, равны ли они между собой. Тип первого аргумента (пусть его имя – `first`) можно задать словом `auto`, тогда тип второго можно задать в виде `decltype(first)`:

```
[ ] (auto first, decltype(first) second) { ... }
```

В стандарте C++20 синтаксис лямбда-выражений будет расширен возможностью явно задавать параметры шаблона. Вместо того чтобы пользоваться уловкой с `decltype`, можно будет написать:

```
[ ] <typename T> (T first, T second) { ... }
```

Хотя этот новый синтаксис планируется включить лишь в версию языка C++20, его уже поддерживает компилятор GCC, и, вероятно, другие компиляторы тоже вскоре пополнятся этой возможностью.

## 3.3 Как сделать функциональные объекты еще лаконичнее

Как было показано в предыдущем разделе, лямбда-выражения выглядят изящно и избавляют от необходимости всякий раз писать своими руками крупные участки одинакового кода при создании каждого функционального объекта. С другой стороны, для них характерны свои клише, хотя и значительно меньшего размера. Рассмотрим пример, показанный на рис. 3.5.

Допустим, мы разрабатываем веб-клиент. Клиент посылает на сервер ряд запросов и получает на них ответы, представленные объектами типа `response_t`. Поскольку обработка любого запроса может завершиться неудачей, типу `response_t` нужна функция-член `.error()`, которая позволяет узнать, произошла ли ошибка. А именно, если при обработке запро-

са ошибка произошла, эта функция должна вернуть значение `true` (или объект некоторого типа, который при преобразовании к типу `bool` даст значение `true`). В противном случае она возвращает значение `false`.

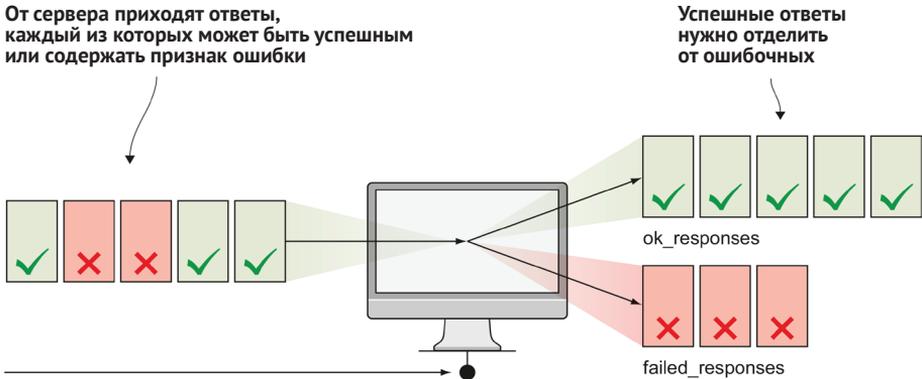


Рис. 3.5 Ответы фильтруются в зависимости от того, содержат ли они признак ошибки, затем обрабатываются отдельно

Коллекцию ответов нужно фильтровать по признаку наличия ошибки. В результате фильтрации нужно создать одну коллекцию, которая содержит только успешные ответы, и другую, в которую поместить лишь ошибочные. Эту задачу легко решить, передав лямбда-выражение в разработанную нами ранее функцию `filter`:

```
ok_responses = filter(responses,
    [](const response_t& response) {
        return !response.error();
    });
failed_responses = filter(responses,
    [](const response_t& response) {
        return response.error();
    });
```

Если это приходится делать часто, еще и для разных типов, обладающих методом `.error()`, который возвращает значение типа `bool` (или иного типа, который может быть преобразован к нему), общий объем кода-клише может оказаться существенно больше, чем если бы программист своими руками реализовал функциональный объект. Идеально было бы иметь сокращенный синтаксис лямбда-выражений, который позволял бы писать что-то наподобие:

```
ok_responses = filter(responses, _.error() == false);
failed_responses = filter(responses, _.error());
```

К сожалению, такая возможность в языке C++ отсутствует, однако имеющихся средств хватит для того, чтобы делать нечто подобное.

Наша цель – создать функциональный объект, способный работать с объектами-аргументами любых классов, лишь бы они предоставляли метод `.error()`, причем синтаксис для этого должен быть изящным и кратким. Хотя данной цели и невозможно добиться способом, показанным в предыдущих двух строчках кода, можно сделать еще лучше: дать пользователю возможность писать предикаты несколькими различными способами (разные люди предпочитают разные формы записи при работе с логическими выражениями), выглядящими как обычные выражения языка C++, вообще без кода-клише, например:

```
ok_responses = filter(responses, not_error);
// или      filter(responses, !error);
// или      filter(responses, error == false);
failed_responses = filter(responses, error);
// или      filter(responses, error == true);
// или даже filter(responses, not_error == false);
```

Чтобы добиться этого, нужно лишь написать простой класс с перегруженной операцией вызова. Он должен хранить в себе значение типа `bool`, которое определяет, какие объекты-ответы (или объекты других классов с методом `.error()`) нужно распознавать.

### Листинг 3.9 Простейшая реализация предиката, проверяющего наличие ошибки

```
class error_test_t {
public:
    error_test_t(bool error = true)
        : m_error(error)
    {
    }

    template <typename T>
    bool operator()(T&& value) const
    {
        return m_error ==
            (bool)std::forward<T>(value).error();
    }

private:
    bool m_error;
};
```

Использование функции `std::forward` здесь может показаться странным, но на самом деле это лишь идеальная передача аргумента, переданного в операцию вызова по пересылающей ссылке. Идеальная передача нужна потому, что метод `error()` может быть по-разному реализован для объектов категорий `rvalue` и `lvalue`

```
error_test_t error(true);
error_test_t not_error(false);
```

Теперь можно использовать объекты `error` и `not_error` в роли предикатов. Для поддержки двух других способов записи, показанных выше, понадобятся еще перегруженные операции сравнения `operator==` и отрицания `operator!`.



### Листинг 3.10 Вспомогательные операции для удобства пользования функциональными объектами-предикатами

```

class error_test_t {
public:
    ...

    error_test_t operator==(bool test) const
    {
        return error_test_t(
            test ? m_error : !m_error
        );
    }

    error_test_t operator!() const
    {
        return error_test_t(!m_error);
    }
    ...
};

```

Если условие истинно, вернуть предикат без изменений. В противном случае вернуть его отрицание

Вернуть отрицание имеющегося предиката

Может показаться, что для того лишь, чтобы предикаты в вызывающем коде имели более компактный вид по сравнению с лямбда-выражениями, кода написано устрашающе много. Однако усилия по его написанию окупаются, если предикат используется достаточно часто. А, как несложно предугадать, предикаты для проверки наличия ошибок могут встречаться в коде многократно.

Автор советует читателю никогда не пренебрегать упрощением важных участков кода. Реализация вспомогательного класса `error_test_t` для функциональных объектов-предикатов требует, конечно, написания достаточно объемного кода – но этот код размещается в отдельном заголовочном файле, в который (после тщательного тестирования) никто и никогда не будет смотреть, работая над содержательной частью программы. Зато применение этого класса в основном коде программы делает его более кратким и удобным для понимания, а также более простым в отладке.

### 3.3.1 Объекты-обертки над операциями в стандартной библиотеке

Как уже говорилось в предыдущей главе, поведение многих стандартных алгоритмов можно настраивать. Например, алгоритм `std::accumulate` позволяет заменить сложение на другую операцию, а алгоритм `std::sort` дает возможность программисту подставить собственную функцию сравнения элементов.

Можно написать свою функцию или лямбда-выражение и подставить их в алгоритм, но в некоторых случаях это означает лишнюю трату сил. По тем же соображениям, которыми руководствовались мы при созда-

нии функционального класса `error_test_t`, в стандартную библиотеку включены функциональные классы-обертки над всеми бинарными операциями, см. табл. 3.1. Например, если нужно перемножить коллекцию целых чисел, можно написать собственную функцию двух аргументов, которая перемножает их и возвращает результат, но удобнее воспользоваться уже существующей оберткой из библиотеки STL с именем `std::multiplies`:

```
std::vector<int> numbers{1, 2, 3, 4};
product = std::accumulate(numbers.cbegin(), numbers.cend(), 1,
    std::multiplies<int>());
// произведение равно 24
```



Таким же образом, если функцию `std::sort` хочется применить для сортировки элементов не по возрастанию, а по убыванию, можно в качестве функции сравнения использовать объект – обертку класса `std::greater`:

```
std::vector<int> numbers{5, 21, 13, 42};
std::sort(numbers.begin(), numbers.end(), std::greater<int>());
// контейнер после сортировки: {42, 21, 13, 5}
```



Таблица 3.1. Обертки над операциями в стандартной библиотеке

Группа	Имя класса-обертки	Операция
Арифметические операции	<code>std::plus</code>	<code>arg_1 + arg_2</code>
	<code>std::minus</code>	<code>arg_1 - arg_2</code>
	<code>std::multiplies</code>	<code>arg_1 * arg_2</code>
	<code>std::divides</code>	<code>arg_1 / arg_2</code>
	<code>std::modulus</code>	<code>arg_1 % arg_2</code>
	<code>std::negates</code>	<code>- arg_1</code> (унарная операция)
Операции сравнения	<code>std::equal_to</code>	<code>arg_1 == arg_2</code>
	<code>std::not_equal_to</code>	<code>arg_1 != arg_2</code>
	<code>std::greater</code>	<code>arg_1 &gt; arg_2</code>
	<code>std::less</code>	<code>arg_1 &lt; arg_2</code>
	<code>std::greater_equal</code>	<code>arg_1 &gt;= arg_2</code>
	<code>std::less_equal</code>	<code>arg_1 &lt;= arg_2</code>
Логические операции	<code>std::logical_and</code>	<code>arg_1 &amp;&amp; arg_2</code>
	<code>std::logical_or</code>	<code>arg_1    arg_2</code>
	<code>std::logical_not</code>	<code>!arg_1</code> (унарная операция)
Побитовые операции	<code>std::bit_and</code>	<code>arg_1 &amp; arg_2</code>
	<code>std::bit_or</code>	<code>arg_1   arg_2</code>
	<code>std::bit_xor</code>	<code>arg_1 ^ arg_2</code>



### Ромбическая нотация

Стандарт C++14 разрешает опускать типы-аргументы, оставляя пустой ромб из угловых скобок, при инстанцировании шаблонов классов, в частности при использовании классов оберток над операциями из стандартной библиотеки. Вместо того чтобы писать `std::greater<int>()`, можно написать просто `std::greater<>()`, что приведет к автоматическому выводу типа, подставляемого в шаблон, исходя из контекста.

Например, вызов функции `std::sort` примет следующий вид:

```
std::sort(numbers.begin(), numbers.end(),
          std::greater<>());
```

Компилятор распознает, что функция сравнения применяется к элементам коллекции, имеющим тип `int`, и подставит этот тип в шаблон.

При наличии компилятора, поддерживающего стандарт C++14, и соответствующей версии стандартной библиотеки такой стиль следует считать предпочтительным – кроме тех случаев, когда аргументы нужно привести к иному, явно заданному типу до того, как к ним будет применена операция.

## 3.3.2 Объекты-обертки над операциями в сторонних библиотеках

Хотя оберток над операциями, определенных в библиотеке STL, хватает для массовых несложных задач, они несколько громоздки, и из них неудобно составлять композиции. Несколько библиотек, помогающих в решении этой проблемы, разработаны в рамках проекта Boost (и еще ряд вне его).

### Коллекция библиотек Boost

Первоначально коллекция библиотек Boost создавалась как испытательный полигон для новшеств, которые планируется включить в будущие версии стандартной библиотеки C++. Впоследствии, однако, коллекция сильно выросла и теперь содержит библиотеки для решения множества как широко распространенных, так и узкоспециальных задач. Многие профессиональные разработчики считают коллекцию Boost неотъемлемой частью экосистемы, образовавшейся вокруг языка C++, и первым местом, куда следует заглянуть, если нужного не оказалось в стандартной библиотеке.

Ниже будет рассмотрен ряд примеров работы с функциональными объектами с использованием библиотеки Boost.Phoenix<sup>1</sup>. Нет необходимости подробно описывать здесь всю библиотеку – возможно, примеры возбудят у читателя интерес к более глубокому ее изучению, а может, и к созданию собственной реализации представленных в ней абстракций.

<sup>1</sup> Документацию по библиотеке Boost.Phoenix можно найти по адресу: <http://mng.bz/XR7E>.

Начнем с маленького примера, способного дать начальное представление о библиотеке. Пусть нужно применить алгоритм `std::partition` к коллекции чисел, чтобы отделить те из них, что меньше или равны 42, от прочих.

Как известно из предыдущего раздела, в библиотеке STL имеется функциональный класс-шаблон `std::less_equal`. Однако его нельзя использовать с функцией `std::partition`. Данный алгоритм ожидает на вход функцию одного аргумента, тогда как функциональный объект класса `std::less_equal` требует двух. Хотя стандартная библиотека и предоставляет способ связать один из этих аргументов с заранее известным значением (о том, как это сделать, речь пойдет в следующей главе), этот способ не очень изящен и обладает некоторыми существенными недостатками.

Библиотека Boost.Phoenix и другие, подобные ей, предоставляют иной способ создания функциональных объектов, в значительной степени основанный на перегрузке операций. В этой библиотеке определены *магические* заглушки для аргументов и операции, позволяющие сочленять их. С помощью этих средств решение нашей задачи становится тривиальным:

```
using namespace boost::phoenix::arg_names;

std::vector<int> numbers{21, 5, 62, 42, 53};

std::partition(numbers.begin(), numbers.end(),
               arg1 <= 42);

// измененная коллекция: {21, 5, 42, 62, 53}
//                          <= 42    > 42
```

Под именем `arg1` скрывается определенная в библиотеке Boost.Phoenix заглушка, которая связывается с первым аргументом, переданным в функциональный объект. Применение перегруженной операции `operator<=` к заглушке не сравнивает объект `arg1` со значением 42, а вместо этого строит и возвращает новый унарный функциональный объект. Лишь когда этот функциональный объект будет вызван с каким-то аргументом, он вернет результат сравнения этого аргумента с числом 42. Это поведение очень похоже на то, которым мы выше наделили операцию `operator==(bool)` из класса `error_test_t`.

Подобным же образом можно создавать и гораздо более изощренные функциональные объекты. Так, например, можно без труда подсчитать сумму половин квадратов всех чисел из заданной коллекции (если столь странное вычисление кому-либо когда-либо понадобится):

```
std::accumulate(numbers.cbegin(), numbers.cend(), 0,
                arg1 + arg2 * arg2 / 2);
```

Напоминание: у бинарной операции, по которой проводится свертка, первый аргумент – это ранее накопленное значение, а второй – очередной элемент коллекции

Результатом выражения `arg1 + arg2 * arg2 / 2` становится функциональный объект, который принимает два аргумента, возводит в квадрат вто-

рой из них, делит полученное значение на два и прибавляет частное к первому аргументу, как показано на рис. 3.6.

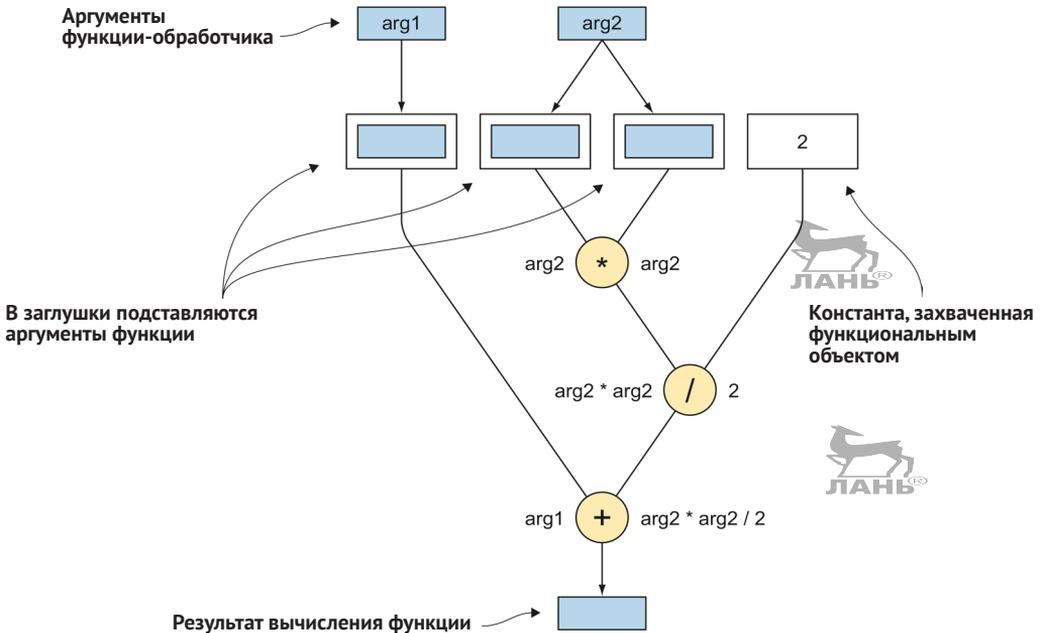


Рис. 3.6 Библиотека Phoenix позволяет составить выражение из фиксированных значений и объектов-заглушек, замещающих аргументы функции. Когда при вызове функционального объекта ему передаются значения аргументов, выражение вычисляется, и полученное значение возвращается в качестве результата функции

Этими средствами можно даже полностью заменить объекты-обертки над операциями, определенные в стандартной библиотеке, о которых шла речь выше:

```
product = std::accumulate(numbers.cbegin(), numbers.cend(), 1,
    arg1 * arg2);

std::sort(numbers.begin(), numbers.end(), arg1 > arg2);
```

Хотя эти инструменты позволяют создавать сколь угодно сложные функциональные объекты, более всего они полезны для создания небольших функций. Если нужно создать что-либо по-настоящему сложное, лучше пользоваться лямбда-выражениями. Если тело функции достаточно велико, то код-клише, присущий лямбда-выражениям, составляет незначительную часть от общего объема кода; кроме того, в случае лямбда-выражений компилятору проще оптимизировать код.

Главный недостаток библиотек, подобных Boost.Phoenix, состоит в том, что их использование способно заметно увеличить время компиляции. Если по мере разработки проекта время компиляции превращается в серьезную проблему, стоит вернуться к использованию лямбда-выра-

жений или определить собственные легковесные (и узкоспециализированные) функциональные объекты, такие как объект `errog`, реализованный ранее в этой главе.

### 3.4 Обертка над функциональными объектами – класс `std::function`

Всюду выше для работы с функциональными объектами использовался автоматический вывод типа: когда требовалось принять функциональный объект в качестве аргумента, мы делали функцию высшего порядка шаблоном, параметризованным типом функционального объекта; при необходимости сохранить лямбда-функцию в переменную ее объявляли не с явно заданным типом, а с ключевым словом `auto`. Хотя этот способ предпочтителен и оптимален, иногда им воспользоваться невозможно. Если функциональный объект нужно сохранить в переменной – члене класса, который нельзя параметризовать типом функционального объекта, тип последнего приходится указывать явно. Также необходимо указать конкретный тип, чтобы передавать функциональный объект между отдельными единицами компиляции.

Поскольку различные типы функциональных объектов не унаследованы от общего предка, который можно было бы использовать в таких случаях, в стандартной библиотеке предусмотрен класс-шаблон `std::function`, способный служить оберткой для любых типов функциональных объектов:

```
std::function<float(float, float)> test_function; ← Сначала указывается тип
                                                    возвращаемого значения функции,
                                                    затем, в скобках, – список типов
                                                    аргументов
test_function = std::fmaxf; ← Обычная функция
test_function = std::multiplies<float>(); ← Класс с операцией вызова
test_function = std::multiplies<>(); ← Класс с обобщенной операцией вызова
test_function = [x](float a, float b) { return a * x + b; }; ← Лямбда-выражение
test_function = [x](auto a, auto b) { return a * x + b; }; ← Обобщенное
                                                         лямбда-выражение
test_function = (arg1 + arg2) / 2; ← Выражение, созданное средствами
                                                         библиотеки Boost.Phoenix
test_function = [](std::string s) { return s.empty(); } // ОШИБКА! ←
                                                         Лямбда-выражение с неподходящей сигнатурой
```

Шаблон `std::function` параметризован не типом завернутого в нем функционального объекта, а лишь его сигнатурой. А именно тип-аргумент этого шаблона задает только тип результата, который должен вернуть функциональный объект, и типы аргументов, которых он требует на вход. Поэтому один и тот же тип, полученный подстановкой в шаблон

конкретной сигнатуры, можно использовать для хранения обычных функций, указателей на функции, лямбда-выражений и других сущностей, поддерживающих операцию вызова, как показано в примере кода выше, – одним словом, для хранения любых функциональных объектов, обладающих той же сигнатурой, что задана в параметре шаблона.

Можно пойти еще дальше. В эту обертку можно поместить даже некоторые сущности, к которым обычный синтаксис вызова неприменим, – такие как переменные и функции – члены класса. Например, язык C++ сам по себе запрещает попытки вызвать метод `.empty()` из класса `std::string` так, как если бы он был внешней функцией, т. е. вызовы вида `std::string::empty(str)`, и поэтому члены класса – переменные и функции – не считаются функциональными объектами. Но указатели на члены класса можно сохранить в объекте типа `std::function` и тем самым сделать возможным обращение к ним посредством привычного синтаксиса вызовов<sup>1</sup>:

```
std::string str{"A small pond"};
std::function<bool(std::string)> f;

f = &std::string::empty;

std::cout << f(str);
```

Функциональные объекты (в смысле, описанном ранее) вместе с указателями на члены класса (переменные и функции) будем называть *вызываемыми объектами*. О том, как реализовать функцию, способную вызывать любые вызываемые объекты (а не только функциональные объекты), читатель узнает из главы 11, где речь пойдет о функции `std::invoke`.

Хотя все перечисленные факторы превращают шаблон класса `std::function` в полезный инструмент, им не стоит пользоваться чрезмерно, так как он приносит заметную потерю производительности. Скрытие фактического типа завернутого объекта, как и единый интерфейс к всевозможным таким объектам, достигается за счет приема, называемого *стиранием типов*. Не будем здесь вдаваться в подробности этой технологии – для наших целей достаточно отметить, что стирание типов обычно основывается на механизме виртуальных функций. Поскольку вызовы виртуальных функций строятся во время выполнения, возможности компилятора по оптимизации кода (например, путем вставки исполняемого кода функции на место вызова) оказываются сильно ограничены.

Еще одна причина для беспокойства состоит в том, что операция вызова в классе `std::function`, хотя и объявлена со спецификатором `const`, может вызывать неконстантный вызываемый объект, что способно привести ко множеству проблем в многопоточной среде.

<sup>1</sup> При использовании компилятора Clang здесь возможна ошибка связывания по причине ошибки в библиотеке `libc++`, из-за которой символ `std::string::empty` не экспортируется.

### Оптимизация малых функциональных объектов-аргументов

Если завернутый вызываемый объект представляет собой указатель на функцию или объект-обертку над ссылкой `std::reference_wrapper` (такие объекты возвращает функция `std::ref`), гарантированно выполняется оптимизация малого объекта. Вызываемые объекты, чей размер не превышает известного предела, хранятся непосредственно внутри объекта `std::function` – без выделения какой бы то ни было динамической памяти.

Более крупные объекты могут создаваться в динамически выделенной области памяти, тогда объект `std::function` вынужден обращаться к ним через указатель. Это бьет по производительности как создания и уничтожения объектов `std::function`, так и операции вызова. Верхняя граница размера вызываемых объектов, для которых выполняется оптимизация малого объекта, различается для разных компиляторов и реализаций стандартной библиотеки.

**СОВЕТ** Более подробную информацию и ссылки по темам, затронутым в этой главе, можно найти на странице <https://forums.man-ning.com/posts/list/41682.page>.

## Итоги

- Объекты, поддерживающие неявное преобразование к типу указателя на функцию, можно использовать в программе так, как если бы они были обыкновенными функциями, однако предпочтительный способ создания функциональных объектов состоит в перегрузке операции вызова.
- Если конкретный тип возвращаемого значения не столь важен, чтобы указывать его явно, можно поручить компилятору вывод типа, исходя из типа выражения, использованного в операторе `return`, – для этого используется ключевое слово `auto`.
- Использование механизма автоматического вывода для возвращаемого типа функции предотвращает любые преобразования или сужения типов, которые могли бы произойти при явном задании типа (например, если оператор `return` возвращает значение типа `double` в теле функции, которая должна возвращать значение типа `int`).
- Если хочется создать функциональный объект, способный работать с аргументами разных типов, его операцию вызова следует сделать шаблоном.
- Лямбда-выражения – это удобный «синтаксический сахар» для определения функциональных объектов. Использование этого синтаксиса обычно предпочтительнее, чем собственноручная реализация классов с перегруженной операцией вызова.
- Лямбда-выражения начиная с версии C++14 стали полноценной заменой большинства функциональных объектов, которые можно реализовать своими руками. Добавлены новые возможности, касающиеся

захвата переменных и поддержки обобщенных функциональных объектов.

- Хотя лямбда-выражения обеспечивают компактный синтаксис для создания функциональных объектов, для некоторых задач он все еще слишком многословен. В этих случаях можно воспользоваться средствами библиотеки Boost.Phoenix (или другой подобной) или реализовать собственные вспомогательные функциональные объекты.
- Тип универсальной обертки `std::function` полезен для ряда задач, однако несет с собой потерю производительности, равную затратам на вызов виртуального метода.



---

# Средства создания новых функций из имеющихся

---

## О чем говорится в этой главе:

- понятие о частичном применении функций;
- как зафиксировать у функции значения некоторых аргументов с помощью функции `std::bind`;
- использование лямбда-выражений для частичного применения функций;
- можно ли все функции в мире считать унарными;
- создание функций, обрабатывающих коллекции элементов.



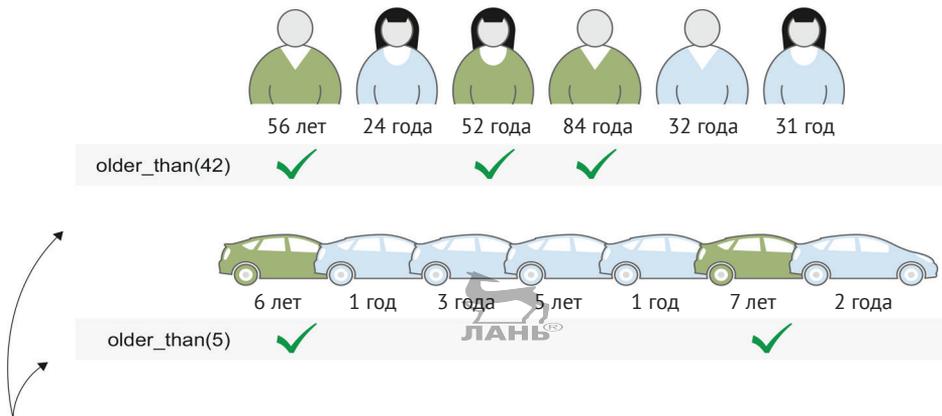
Большинство парадигм программирования стремятся к тому, чтобы обеспечить как можно более высокую степень повторного использования кода. Так, в объектно-ориентированном мире создают классы, которые можно затем применять в разнообразных ситуациях. Эти классы можно использовать как непосредственно, так и собирать из них реализацию более сложных классов. Возможность разбивать сложные системы на множество меньших компонентов, которые можно использовать и тестировать независимо друг от друга, составляет мощный инструмент разработки.

Подобно тому, как парадигма ООП предоставляет программисту средства для композиции новых типов из существующих и их расширения, парадигма функционального программирования предоставляет набор простых способов создания новых функций: путем *соединения* ранее написанных функций между собой; путем *усовершенствования* существующих функций, чтобы их можно было использовать в более широком множестве случаев; а также, напротив, путем упрощения излишне общих функций для более конкретных вариантов использования.



## 4.1 Частичное применение функций

В одном из примеров, разобранных в предыдущей главе, требовалось подсчитать в коллекции те объекты, чей возраст превышает некоторый заранее заданный предел. При внимательном рассмотрении можно заметить, что в основе примера лежит функция двух аргументов: объекта, обладающего возрастом (например, это может быть человек или автомобиль), и числа, с которым нужно сравнить возраст объекта, – см. рис. 4.1. Эта функция должна возвращать значение «истина» тогда и только тогда, когда возраст объекта больше, чем данное значение.



Один и тот же обобщенный предикат позволяет проверять достижение определенной границы возраста для объектов данных разных типов

Рис. 4.1 На высоком уровне абстракции можно сказать, что дана функция двух аргументов: объекта и целого числа, представляющего собой границу возраста. Зафиксировав граничный возраст, положив его равным некоторому заранее заданному значению, получим унарную функцию, сравнивающую возраст объекта (своего единственного аргумента) с этим фиксированным значением

Поскольку алгоритм `std::count_if` ожидает, что в качестве аргумента ему передадут унарный предикат, от программиста требуется создать функциональный объект, который внутри себя хранит граничное значение и именно с ним сравнивает возраст каждого поступающего на вход объекта-аргумента. Таким образом, вызывающий контекст передает лишь часть аргументов, которые требуются функции, так как значения остальных аргументов зафиксированы один раз наперед; именно об этом пойдет речь в настоящем разделе.

Рассмотрим упрощенную версию примера из предыдущей главы. Вместо того чтобы проверять, превышает ли возраст человека заранее заданную границу, возьмем обычную операцию сравнения «больше» (функцию двух аргументов) и зафиксируем значение второго аргумента этой функции, получив тем самым функцию одного аргумента (рис. 4.2).

Тем самым мы создали обычный функциональный объект, который в момент своего создания принимает одно значение целого типа, сохра-

няет это значение внутри себя и затем использует его для сравнения со значениями, поступающими через единственный аргумент операции вызова.

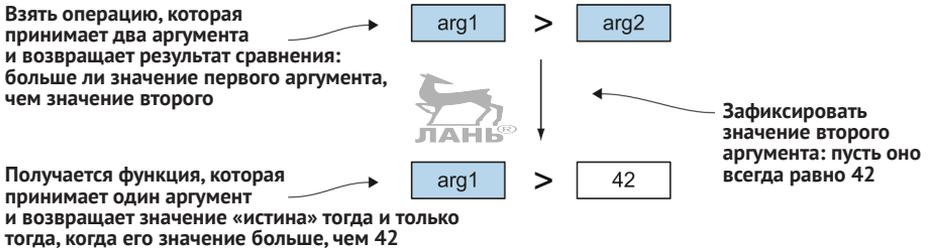


Рис. 4.2 Подстановка фиксированного значения 42 в качестве второго аргумента операции сравнения «больше» превращает ее в функцию одного аргумента, которая проверяет, превышает ли значение аргумента константу 42

Экземпляры этого класса можно передавать в любую функцию высшего порядка, которая требует предиката, – например, в функции `std::find_if`, `std::partition`, `std::remove_if`, – при условии что они будут применять этот предикат к значениям целочисленного типа (или к таким значениям, которые могут быть неявно преобразованы в целые числа).

**Листинг 4.1 Сравнение аргумента с заранее заданным значением**

```

class greater_than {
public:
    greater_than(int value)
        : m_value
    {
    }

    bool operator()(int arg) const
    {
        return arg > m_value;
    }

private:
    int m_value;
};
...
greater_than greater_than_42(42);
greater_than_42(1); // false
greater_than_42(50); // true

std::partition(xs.begin(), xs.end(), greater_than(6));
    
```

Можно создать функциональный объект – экземпляр этого класса – и использовать его сколько угодно раз, чтобы проверить, превышают ли те или иные числа заданное значение 42

Экземпляр класса можно создать непосредственно там, где он нужен для вызова функции высшего порядка, например при обращении к алгоритму `std::partition` с целью отделить числа, превышающие 6, от тех, что меньше или равны 6

В этом нет ничего сложного. Мы взяли бинарную функцию `operator > : (int, int) → bool` и создали на ее основе новый функциональный



объект, который делает то же самое, что и операция «больше», но лишь при условии, что ее второй аргумент имеет фиксированное значение (см. рис. 4.2).

Описанный здесь способ создания нового функционального объекта из существующего путем придания одному или нескольким аргументам фиксированного значения называется *частичным применением* функции. Слово «частичное» в данном контексте означает, что функция применена к части (не ко всем) аргументов, нужных для вычисления результата.

### 4.1.1 Универсальный механизм превращения бинарных функций в унарные

Попробуем немного обобщить представленное выше решение. Для этого нужно определить функциональный объект, способный хранить в себе любую бинарную функцию, переданную пользователем, и связывать один из ее аргументов фиксированным значением. Чтобы сохранить преемственность с предыдущим примером, пусть фиксированное значение придается второму аргументу, как в классе `greater_than`.

**ПРИМЕЧАНИЕ** Старые версии стандартной библиотеки содержали функцию с именем `std::bind2nd`, которая делала в точности то же, что своими руками воплощаем мы в этом разделе. Хотя эта функция исключена из стандарта с появлением лямбда-выражений и функции `std::bind`, о которой речь пойдет в следующем разделе, она может послужить хорошим примером того, как реализовать частичное применение на языке C++.

От функционального объекта требуется, чтобы он хранил в себе функцию двух аргументов и значение, которое нужно подставлять во второй из них. Поскольку типы функции и второго аргумента заранее неизвестны, наш класс должен быть шаблоном, принимающим эти два типа в качестве параметров. Конструктор должен лишь инициализировать члены-данные и ничего более. Очевидно, что аргументами конструктора должны стать функция и значение, которое далее будет подставляться в ее второй аргумент.

#### Листинг 4.2 Заготовка класса-шаблона для частичного применения функции

```
template <typename Function, typename SecondArgType>
class partial_application_on_2nd_impl {
public:
    partial_application_bind2nd_impl(Function function,
                                    SecondArgType second_arg)
        : m_function(function)
        , m_value(second_arg)
    {
    }
};
```

```

...
private:
    Function m_function;
    SecondArgType m_second_arg;
};
    
```

Поскольку тип первого аргумента функции заранее неизвестен, операцию вызова тоже нужно сделать шаблоном. Ее реализация вполне очевидна: нужно вызвать функцию, которая хранится в переменной-члене `m_function`, идеальным образом передав ей аргумент, пришедший в операцию вызова, вместе со значением, хранящимся в переменной-члене `m_second_arg`.



**Листинг 4.3 Операция вызова в классе, реализующем частичное применение функции**

```

template <typename Function, typename SecondArgType>
class partial_application_bind2nd_impl {
public:
    ...

    template <typename FirstArgType>
    auto operator()(FirstArgType&& first_arg) const
        -> decltype(m_function(
            std::forward<FirstArgType>(first_arg),
            m_second_arg))
    {
        return m_function(
            std::forward<FirstArgType>(first_arg),
            m_second_arg);
    }
    ...
};
    
```

Если компилятор не поддерживает автоматический вывод возвращаемого типа, нужно явно задавать тип с помощью громоздкой конструкции `decltype`. Если же компилятор достаточно новый, можно написать просто `decltype(auto)`



Аргумент операции вызова передается в первый аргумент обертываемой функции

Сохраненное значение передается во второй аргумент функции

**ПРИМЕЧАНИЕ** Напомним, что раз к переменной-члену `m_function` применяется обычный синтаксис вызова, класс из листинга 4.3 может работать только с функциональными объектами, но не с вызываемыми объектами в общем случае, т. е., например, не с указателями на функции – члены класса и не с указателями на переменные-члены. Если хочется расширить класс их поддержкой, следует воспользоваться механизмом `std::invoke`, о котором речь пойдет в главе 11.

Теперь, когда реализация класса готова полностью, остается доделать еще одну мелочь, чтобы сделать его использование более удобным. Для того чтобы применять этот класс в коде непосредственно, пришлось бы явно задавать тип-аргумент шаблона всякий раз при создании экземпляров. Это не только выглядит неэлегантно, но и в некоторых случаях

вообще невозможно (например, тип лямбда-выражения программисту неизвестен).

**ПРИМЕЧАНИЕ** Требование явно указывать типы-аргументы шаблона при его инстанцировании исключено из стандарта языка C++17. Однако, поскольку этот стандарт еще не стал повсеместно распространенным, мы не будем здесь полагаться на данное новшество.

Для того чтобы компилятор мог автоматически вывести типы-аргументы, нужно создать функцию-шаблон, единственное предназначение которой – создавать экземпляры данного класса-шаблона. Поскольку при вызове функций-шаблонов работает механизм автоматического вывода типов-аргументов<sup>1</sup>, пользователю нет необходимости указывать их явно при вызове такой функции. Функции остается лишь вызвать конструктор класса-шаблона, чье определение приведено выше, и передать ему свои аргументы. Код этой функции представляет собой клише, к написанию которого нас вынуждает отсутствие в языке (до стандарта C++17) способа инстанцировать шаблон класса без явного указания типов-аргументов.

#### Листинг 4.4 Функция-обертка для создания экземпляров разработанного выше класса

```
template <typename Function, typename SecondArgType>
partial_application_bind2nd_impl<Function, SecondArgType>
bind2nd(Function&& function, SecondArgType&& second_arg)
{
    return partial_application_bind2nd_impl<Function, SecondArgType>(
        std::forward<Function>(function),
        std::forward<SecondArgType>(second_arg));
}
```



Покажем теперь, как с помощью этой функции заменить использование класса `greater_than` в листинге 4.1.

#### Листинг 4.5 Использование функции `bind2nd` для создания нового функционального объекта

```
auto greater_than_42 = bind2nd(std::greater<int>(), 42);

greater_than_42(1); // false
greater_than_42(50); // true

std::partition(xs.begin(), xs.end(), bind2nd(std::greater<int>(), 6));
```

<sup>1</sup> Подробности об автоматическом выводе аргументов шаблона можно найти по адресу <http://mng.bz/YXIU>.

Таким образом, мы разработали механизм, гораздо более общий, чем класс `greater_than` из предыдущего раздела; этот механизм можно использовать в большем количестве разнообразных ситуаций; в частности, им можно без труда заменить все случаи применения класса `greater_than`. Чтобы продемонстрировать, насколько более общим является этот механизм, покажем, как использовать функцию `bind2nd` с операцией умножения вместо операции сравнения «больше».

Предположим, дана коллекция чисел, представляющих собой углы, выраженные в градусах, и пусть остальная часть кода работает с радианной мерой углов. Эта проблема очень часто возникает при разработке многочисленных графических приложений: в графических библиотеках углы поворота обычно измеряются в градусах, тогда как большинство математических библиотек, используемых вместе с ними, требует радианной меры угла. Перевод из градусов в радианы выполняется просто: градусную меру угла нужно умножить на число  $\pi/180$  (см. рис. 4.3).

**Листинг 4.6** Использование функции `bind2nd` для преобразования градусов в радианы

```
std::vector<double> degrees = {0, 30, 45, 60};
std::vector<double> radians(degrees.size());
```

```
std::transform(degrees.cbegin(), degrees.cend(),
               radians.begin(),
               bind2nd(std::multiplies<double>(),
                       PI / 180));
```

Пройти по всем элементам вектора degrees

Записать результаты преобразования в вектор radians

Передать в качестве функции-преобразователя элементов функцию, умножающую свой аргумент на  $\pi/180$

Как явствует из этого примера, с помощью функции `bind2nd` можно создавать отнюдь не только предикаты (т. е. функции, возвращающие значения типа `bool`). Она позволяет взять любую бинарную функцию и превратить ее в унарную, зафиксировав значение ее второго аргумента. Это дает возможность использовать бинарные функции там, где требуются унарные, как в предыдущем примере.

С помощью функции умножения на  $\pi/180$  можно преобразовать коллекцию углов, выраженных в градусах, в коллекцию углов, выраженных в радианах

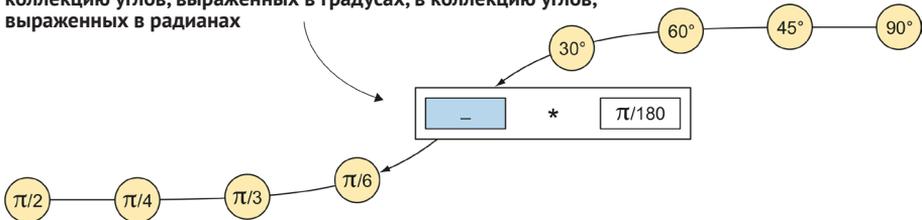


Рис. 4.3 Если значение одного из операндов умножения зафиксировать равным  $\pi/180$ , получится функция преобразования градусов в радианы

## 4.1.2 Использование функции `std::bind` для фиксации значений некоторых аргументов функции

До появления стандарта C++11 стандартная библиотека содержала две функции, похожие на те, что мы разработали в предыдущем разделе. Эти функции, `std::bind1st` и `std::bind2nd`, предоставляли программисту способ превращения бинарной функции в унарную путем придания, соответственно, первому или второму ее аргументу фиксированных значений – таким же образом, как реализованная в предыдущем разделе функция `bind2nd`.

В версии стандарта C++11 эти две функции были объявлены устаревшими, а в стандарте C++17 – вообще исключены из стандартной библиотеки в пользу более общей и мощной функции `std::bind`. Эта последняя не ограничена одними лишь бинарными функциями – ее можно применять к функциям любого числа аргументов. Кроме того, она не ограничивает пользователя в том, каким именно аргументам фиксировать значения, позволяя зафиксировать любое число аргументов в любом порядке – аргументы функции-аргумента, оставшиеся незафиксированными, становятся аргументами преобразованной функции.

Начнем с простейшего примера применения функции `std::bind`, а именно со связывания всех аргументов функции с фиксированными значениями без вызова функции. Первый аргумент функции `std::bind` – это функция (или, вообще говоря, функциональный объект), аргументы которой нужно связать со значениями, а остальные аргументы в данном простейшем случае – значения, которые будут закреплены за аргументами функции. Свяжем оба аргумента функции сравнения `std::greater` с фиксированными значениями: 6 и 42. Строго говоря, результат такого связывания нельзя назвать *частичным* применением функции, поскольку зафиксированы значения *всех* ее аргументов. Однако этот пример удобен для первого знакомства с функцией `std::bind`.

### Листинг 4.7 Связывание всех аргументов функции с фиксированными значениями

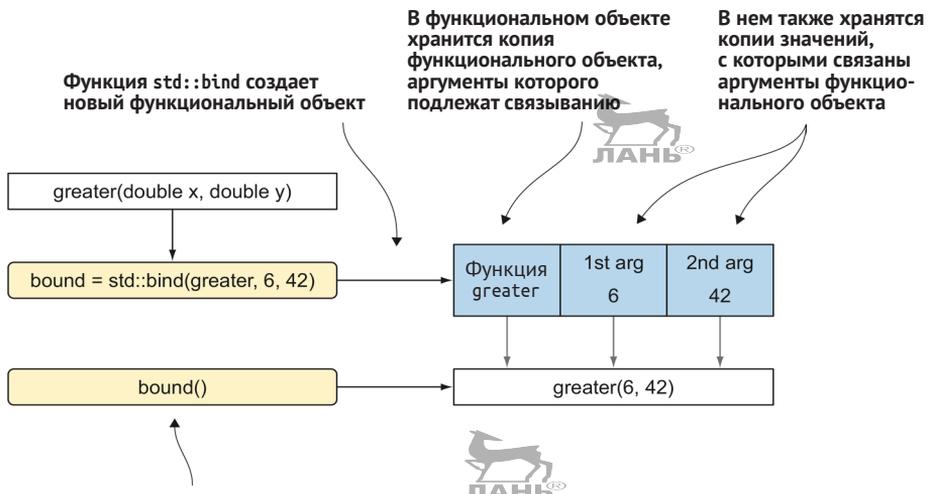
```
auto bound =
    std::bind(std::greater<double>(), 6, 42);
bool is_6_greater_than_42 = bound();
```

Лишь в момент вызова функционального объекта, полученного в результате связывания аргументов, происходит сравнение значений 6 и 42

Функциональный объект `std::greater` здесь не вызывается. Вместо этого создается новый функциональный объект, в котором хранятся объект `std::greater` и значения 6 и 42

Подставив значения для всех аргументов, мы тем самым создаем новый функциональный объект, в котором хранится и функция, аргументы которой нужно связать (в данном примере – функциональный объект `std::greater`), и значения ее аргументов (см. рис. 4.4). Связывание лишь запоминает значения аргументов, но само по себе функцию не вызывает. Вызов первоначальной функции происходит только тогда, когда кто-то вызывает функциональный объект, созданный операцией `std::bind`.

В данном случае функция сравнения `std::greater` вызывается только при вызове функционального объекта `bound`.



Вызов полученного функционального объекта приводит к вызову исходного функционального объекта и передаче ему сохраненных значений

Рис. 4.4 Функция двух аргументов проверяет, действительно ли ее первый аргумент больше второго. Связывание обоих аргументов с конкретными значениями, такими как 6 и 42, создает нуль-арный (т. е. не обладающий аргументами) функциональный объект, который, если его вызывать, вернет результат сравнения чисел 6 и 42

Рассмотрим теперь, как связать с определенным значением лишь один из аргументов, оставив второй свободным. Не получится просто указать одно значение и опустить второе, так как в этом случае функция `std::bind` не будет знать, к какому из двух аргументов относится заданное значение. Для решения этой проблемы предназначены так называемые заглушки (placeholders). Если значение какого-то аргумента нужно зафиксировать, в функцию `std::bind` нужно, как и в предыдущем примере, передать это значение. Но если какой-либо аргумент нужно оставить свободным, нужно вместо его значения поставить заглушку.

Заглушки аргументов, предназначенные для использования совместно с функцией `std::bind`, очень похожи на функциональные объекты, изученные в предыдущей главе, когда речь шла о библиотеке `Boost.Phoenix`. Правда, на этот раз у заглушек чуть другие имена: `_1` вместо `arg_1`, `_2` вместо `arg_2` и т. д.

**ПРИМЕЧАНИЕ** Объекты-заглушки определены в заголовочном файле `<functional>`, в пространстве имен `std::placeholders`. Это один из тех редких случаев, когда не стоит явно указывать пространство имен, так как можно существенно ухудшить удобочитаемость кода. Во всех приведенных ниже примерах, относящихся к функции `std::bind`, следует считать, что пространство имен `std::placeholders` используется по умолчанию.

Модифицируем предыдущий пример так, чтобы фиксировалось значение только одного из двух аргументов функции. Пусть нужно создать два предиката: один проверяет, больше ли его аргумент, чем 42, а другой – меньше ли его аргумент, чем 42. Оба предиката должны быть определены с использованием исключительно функции `std::bind` и шаблона-обертки над операцией сравнения `std::greater`. В первом случае в функцию сравнения нужно закрепить значение второго аргумента и оставить заглушку для первого, для второго предиката – наоборот.

#### Листинг 4.8 Связывание аргументов с фиксированными значениями с помощью функции `std::bind`

```
auto is_greater_than_42 =
    std::bind(std::greater<double>(), _1, 42);
auto is_less_than_42 =
    std::bind(std::greater<double>(), 42, _1);

is_less_than_42(6); // возвращает true
is_greater_than_42(6); // возвращает false
```

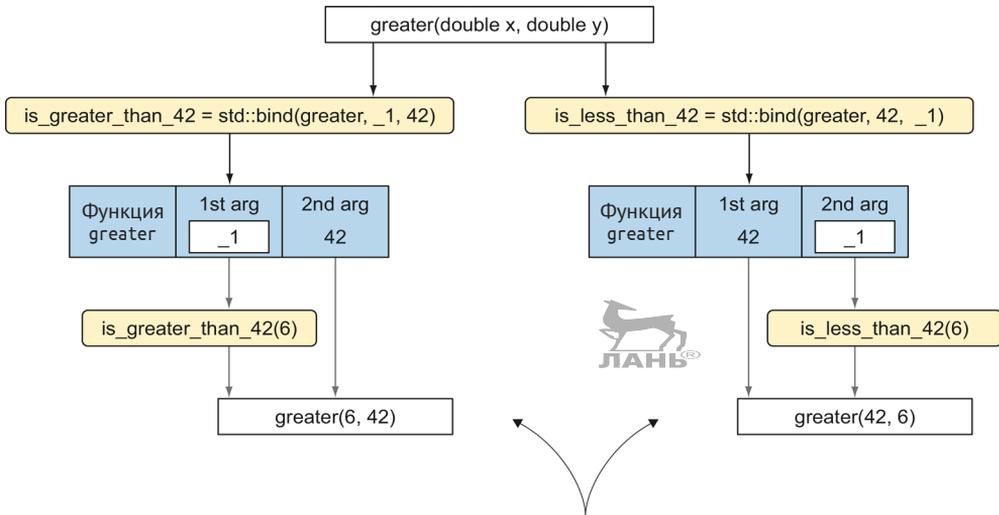
Что происходит в этом коде? В обоих случаях у функционального объекта с двумя аргументами – объекта-обертки над операцией сравнения `std::greater` – один из аргументов связывается с фиксированным значением, а другой – с заглушкой. Привязка аргумента к заглушке фактически означает, что в настоящий момент для этого аргумента нет готового значения, поэтому на месте аргумента остается пустота, которая будет заполнена позже – при вызове полученной функции.

Когда функциональный объект, построенный функцией `std::bind`, вызывается с определенным значением единственного аргумента, этим значением заполняется пустота, оставшаяся от заглушки `_1` (см. рис. 4.5). Если несколько аргументов исходной функции заполнены одной и той же заглушкой, все они получают одно и то же значение.

### 4.1.3 Перестановка аргументов бинарной функции

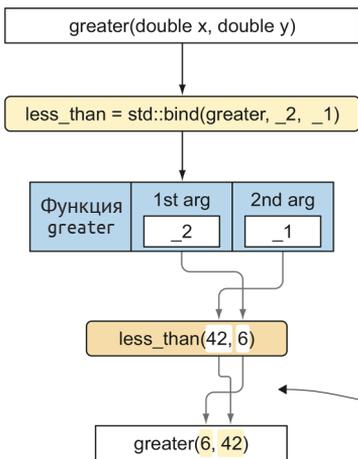
Выше было показано, как связать все аргументы функции с фиксированными значениями и как связать лишь один из двух аргументов, подставив во второй заглушку `_1`. Можно ли в одном связывании использовать сразу несколько заглушек?

Пусть дан вектор действительных чисел, который нужно отсортировать по возрастанию. Обычно для этого применяют функциональный объект `std::less` (функция `std::sort` по умолчанию ведет себя именно так), но на этот раз – исключительно в демонстрационных целях – воспользуемся функциональным объектом `std::greater` (рис. 4.6). Чтобы из отношения «больше» получить нужное нам отношение «меньше», нужно посредством функции `std::bind` создать новый функциональный объект, который меняет местами два своих аргумента, перед тем как подавать их на вход объекта `std::greater`.



Этот пример похож на предыдущий, за исключением того, что заглушка замещается значением, переданным новому функциональному объекту в качестве аргумента

Рис. 4.5 Функция двух аргументов проверяет, больше ли ее первый аргумент, чем второй. Связывание одного из этих аргументов с фиксированным значением, а другого – с заглушкой дает новый функциональный объект с одним аргументом, который при вызове замещает заглушку



Заглушка `_1` замещается первым из аргументов, передаваемых объекту `less_than`, поэтому он становится вторым аргументом функции `greater`; заглушка `_2` – аналогично

Рис. 4.6 Первая заглушка передается в функциональный объект `greater` вторым аргументом, а вторая заглушка – первым аргументом. Полученный функциональный объект ведет себя так же, как и `greater`, но с переставленными местами аргументами

Здесь создается функциональный объект, обладающий двумя аргументами (поскольку заглушка с наибольшим номером – это `_2`), который вызывает функциональный объект `std::greater`, но с обратным порядком аргументов. Таким образом, этот новый функциональный объект реали-

зует отношение «меньше» и может использоваться для сортировки по возрастанию.

#### Листинг 4.9 Сортировка оценок, выставленных фильму, по возрастанию

```
std::sort(scores.begin(), scores.end(),
          std::bind(std::greater<double>(), _2, _1));
```

← Аргументы переставлены местами:  
сначала передается заглушка \_2, затем – заглушка \_1

### 4.1.4 Использование функции `std::bind` с функциями большего числа аргументов

Теперь нам снова предстоит обрабатывать коллекцию людей. Пусть на этот раз нужно вывести данные обо всех людях в стандартный поток вывода или какой-то другой поток. Начнем с определения внешней (т. е. не являющейся членом класса) функции, выводящей данные человека в одном из заранее определенных форматов. У этой функции будет три аргумента: объект, представляющий человека, ссылка на поток вывода и желаемый формат вывода:

```
void print_person(const person_t& person,
                 std::ostream& out,
                 person_t::output_format_t format)
{
    if (format == person_t::name_only) {
        out << person.name() << '\n';
    } else if (format == person_t::full_name) {
        out << person.name() << ' '
            << person.surname() << '\n';
    }
}
```



Теперь для того, чтобы вывести в поток данные всех людей из коллекции, можно воспользоваться алгоритмом `std::for_each`, передав ему функцию `print_person`, у которой связаны аргументы `out` и `format` (см. рис. 4.7).

По умолчанию функция `std::bind` сохраняет копии подставляемых в аргументы значений в создаваемом ею функциональном объекте. Поскольку копирование объекта `std::cout` (как и других объектов-поточков) запрещено, аргумент `out` нужно связать со ссылкой на объект-поток, а не с копией этого объекта. Для этого используется вспомогательная функция `std::ref` (см. пример `printing-people/main.cpp`).

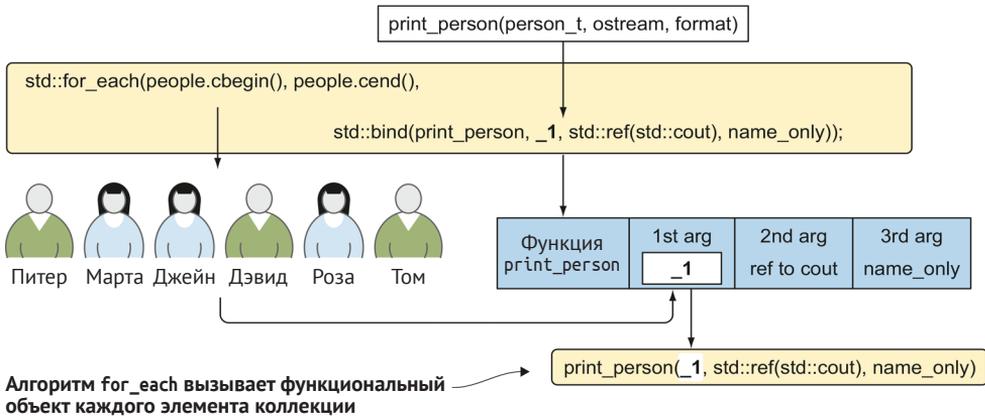


Рис. 4.7 Создание функции, которая выводит в поток данные обо всех людях из заданной коллекции. Имеется функция трех аргументов, которая выводит данные одного человека в заданный поток в заданном формате. Зафиксировав поток и формат, получим функцию одного аргумента, применением которой к каждому по очереди элементу коллекции займется алгоритм `std::for_each`

Листинг 4.10 **Связывание аргументов функции `print_person`**

```
std::for_each(people.cbegin(), people.cend(),
             std::bind(print_person,
                       _1,
                       std::ref(std::cout),
                       person_t::name_only
             ));
```

Создание унарного функционального объекта, который выводит только имя человека в стандартный поток вывода

```
std::for_each(people.cbegin(), people.cend(),
             std::bind(print_person,
                       _1,
                       std::ref(file),
                       person_t::full_name
             ));
```

Вывод имени и фамилии в заданный файл

За основу взята функция, которой требуется на вход три аргумента: объект-человек, поток вывода и формат вывода. Путем связывания аргументов из нее получены две новые функции, каждая из которых обладает лишь одним аргументом – объектом, представляющим человека. Одна выводит в стандартный поток вывода имя человека, а другая – выводит имя и фамилию в заданный файл. Все это сделано без написания кода новых функций своими руками – вместо этого из существующей функции путем связывания некоторых аргументов получены новые унарные функции, которые можно подставлять в алгоритм `std::for_each`.



Всюду ранее мы предпочитали внешние (т. е. не являющиеся членами класса) функции или статические функции-члены, так как функции-члены класса не считаются функциональными объектами и не поддерживают обычный синтаксис вызова. Это ограничение в значительной степени искусственно. Функции-члены класса в сущности своей не отличаются от обычных функций ничем, кроме одного: у них есть неявный первый аргумент `this` – указатель на тот объект, для которого вызывается эта функция.

Выше мы видели внешнюю функцию `print_person`, которая принимает три аргумента: объект типа `person_t`, поток вывода и формат. Ее можно заменить функцией – членом класса `person_t`, как показано ниже:

```
class person_t {
    ...

    void print(std::ostream& out, output_format_t format) const
    {
        ...
    }

    ...
};
```



Между функциями `print_person` и `person::print` нет существенных различий, за исключением того, что правила языка C++ не позволяют вызывать последнюю с использованием обычного синтаксиса, указывая все аргументы исключительно в скобках после имени функции. У функции `person::print` тоже три аргумента: неявный аргумент `this`, указывающий на объект типа `person_t`, и два явно передаваемых аргумента, `out` и `format`, и делает она то же самое, что и функция `print_person`.

К счастью, функция `std::bind` умеет связывать аргументы любого вызываемого (а не только функционального) объекта, поэтому для нее функции `print_person` и `person::print` одинаковы. Для того чтобы изменить предыдущий пример так, чтобы в нем использовалась функция-член класса, достаточно в первом аргументе `std::bind` заменить имя функции `print_person` указателем на функцию-член `person_t::print`.

#### Листинг 4.11 Связывание аргументов функции-члена класса

```
std::for_each(people.cbegin(), people.cend(),
    std::bind(&person_t::print, ←
        _1,
        std::ref(std::cout),
        person_t::name_only
    ));
```

Создание унарного функционального объекта путем связывания аргументов функции – члена класса

**ПРИМЕЧАНИЕ** Хорошее упражнение для читателя: повторно просмотреть все предыдущие примеры и найти, где обычные функции было бы удобно заменить функциями-членами с помощью функции `std::bind`.

Таким образом, мы показали, что функция `std::bind` позволяет выполнять частичное применение функций, связывая некоторые их аргументы с фиксированными значениями или меняя аргументы местами. Ее синтаксис может показаться необычным в мире объектно-ориентированного программирования, однако он весьма лаконичен и прост для понимания. Функция `std::bind` поддерживает любые вызываемые объекты, тем самым делая возможным использование в стандартных алгоритмах и других функциях высшего порядка не только обычных функций или функциональных объектов, но и переменных-членов и функций-членов.

### 4.1.5 Использование лямбда-выражений вместо функции `std::bind`

Хотя функция `std::bind` и обеспечивает изящный, краткий синтаксис для создания новых функциональных объектов из имеющихся путем перестановки аргументов или связывания их со значениями, у этого удобства есть своя цена: компилятору становится гораздо труднее оптимизировать код. Это не встроенное в язык средство, а библиотечная функция, и ее реализация основана на сложных техниках шаблонного метапрограммирования.

Альтернативой использованию функции `std::bind` для частичного применения могут быть лямбда-выражения. Лямбда-выражения относятся к ядру языка, поэтому компилятору гораздо проще их оптимизировать. Синтаксис при этом получается несколько более многословным, но выгода от оптимизации может того стоить.

Замена функции `std::bind` лямбда-выражением выполняется просто (рис. 4.8):

- каждую переменную или ссылку на переменную, с которой связывается аргумент, поместить в список захвата;
- все заглушки превратить в аргументы лямбда-функции;
- если аргумент связывается с фиксированным значением, поместить его прямо в теле лямбда-выражения.

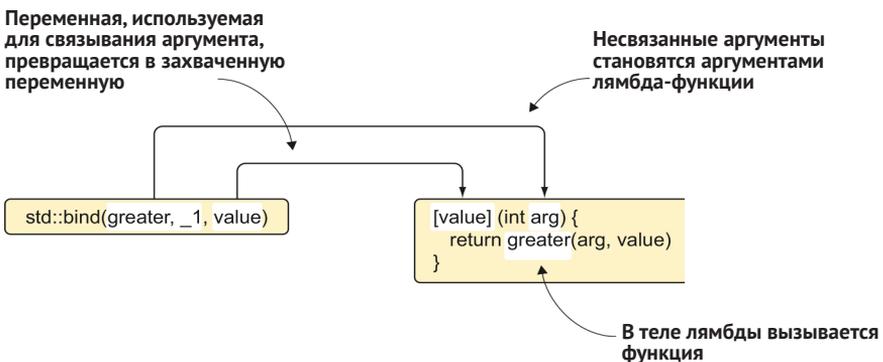


Рис. 4.8 Чтобы использовать лямбду вместо функции `std::bind`, нужно связанные аргументы превратить в захваченные переменные, а несвязанные аргументы функции – в аргументы лямбды

Посмотрим, как после такого преобразования будут выглядеть примеры из предыдущего раздела. Во-первых, мы рассматривали функцию двух аргументов и связывали оба из них с фиксированными значениями. Поскольку аргументы связаны не с переменными, не понадобится ничего захватывать. Никаких заглушек в этом примере не было, поэтому у лямбда-выражения не будет аргументов. Фиксированные значения нужно передать прямо в функцию сравнения:

```
auto bound = [] {
    return std::greater<double>()(6, 42);
};
```

Эта запись получилась бы еще короче, используйте мы вместо имени `std::greater` знак операции `>`. Однако оставим данное выражение именно в таком виде, поскольку не все функции можно заменить инфиксными операциями, а цель примера состоит в том, чтобы показать общий метод замены функции `std::bind` лямбда-выражением.

Следующий пример состоял в связывании одного из аргументов с фиксированным значением, а другого – с заглушкой. Поскольку в связывании участвует ровно одна заглушка, у лямбда-выражения будет один аргумент. Как и прежде, захватывать переменные не нужно:

```
auto is_greater_than_42 =
    [](double value) {
        return std::greater<double>()(value, 42);
    };
auto is_less_than_42 =
    [](double value) {
        return std::greater<double>()(42, value);
    };
is_less_than_42(6); // возвращает true
is_greater_than_42(6); // возвращает false
```



Как и прежде, фиксированное значение передается в функцию сравнения `std::greater` прямо в теле лямбда-выражения.

Двигаясь далее, преобразуем пример с сортировкой вектора оценок по возрастанию с использованием функционального объекта `std::greater`. Здесь при вызове функции сравнения нужно поменять местами аргументы. Функция `std::bind` использовалась с двумя заглушками, поэтому соответствующее лямбда-выражение должно иметь два аргумента.

```
std::sort(scores.begin(), scores.end(),
    [](double value1, double value2) {
        return std::greater<double>()(value2, value1);
    });
```

Как и в примере, где использовалась функция `std::bind`, алгоритм `std::sort` вызывается с функциональным объектом, который, получая при вызове два аргумента, передает их функциональному объекту `std::greater` в обратном порядке.

В последнем примере алгоритм `std::for_each` использовался для того, чтобы вывести в поток имена людей из заданной коллекции. На этот раз нужно рассмотреть несколько сущностей:

- в исходной версии примера есть лишь одна заглушка `_1`, поэтому у лямбда-выражения будет один аргумент;
- в связывании посредством функции `std::bind` участвовали фиксированные значения `person_t::name_only` и `person_t::full_name` – их можно внести непосредственно в тело лямбда-выражения, подставив при вызове функции `print_person`;
- ссылку на объект `std::cout` захватывать не нужно, поскольку этот глобальный объект и без того виден в теле лямбда-выражения;
- ссылку на поток вывода `file` нужно включить в захват лямбда-выражения.

В результате преобразования получим следующий код:

```
std::for_each(people.cbegin(), people.cend(),
    [](const person_t& person) {
        print_person(person,
            std::cout,
            person_t::name_only);
    });

std::for_each(people.cbegin(), people.cend(),
    [&file](const person_t& person) {
        print_person(person,
            file,
            person_t::full_name);
    });
```

Все полученные здесь решения в целом выглядят подобно тем, что были построены нами ранее с использованием функции `std::bind`, отличаясь лишь несколько менее компактной формой записи. Однако имеется и ряд более тонких различий. Функция `std::bind` сохраняет копии всех значений, ссылок и функций, участвующих в связывании, в функциональном объекте, который она создает. Ей нужно даже сохранить в этом объекте информацию о том, какие заглушки участвуют в связывании. Функциональные объекты, в которые компилятор превращает лямбда-выражения, напротив, содержат только те данные, которые программист явно указывает в списке захвата. Из-за этого функциональные объекты, построенные с помощью функции `std::bind`, могут работать медленнее, чем лямбда-функции, если компилятор не сумеет как следует их оптимизировать.

Таким образом, мы рассмотрели несколько способов определить частичное применение функций на языке C++. А именно в предыдущей главе был показан подход к частичному применению операций наподобие умножения или сравнения `Boost.Phoenix` – метод, обладающий изящным, компактным синтаксисом. Для частичного применения каких угодно функций к произвольным аргументам можно использовать функцию

`std::bind` или лямбда-выражения. Эти два механизма предоставляют одинаковую выразительную мощь, но различаются способом написания и эффективностью. Лямбда-выражения получаются более пространными, но обычно дают более эффективный исполняемый код; никогда излишним будет измерить производительность этих механизмов в условиях конкретного приложения.

## 4.2 Карринг – необычный взгляд на функции



Из предыдущих разделов читатель узнал, что такое частичное применение функций и как использовать его в языке C++. Теперь обратимся к явлению со странным названием «*карринг*», которое на первый взгляд часто кажется очень похожим на частичное применение. Чтобы избежать путаницы этих двух понятий, сначала дадим определение, а затем обратимся к примерам.

**ПРИМЕЧАНИЕ** *Карринг* получил свое наименование в честь Хаскелла Карри<sup>1</sup> – математика и логика, который выделил это понятие, основываясь на идеях Готтлоба Фреге и Моисея Шейнфинкеля.

Допустим, нам выпало работать с языком программирования, который не позволяет создавать функции более, чем одного аргумента. Хотя это может на первый взгляд показаться существенным ограничением, мы вскоре убедимся, что подобный язык обладает такой же выразительной силой, как и обычные языки, благодаря простому, но остроумному приему.

Вместо того чтобы создавать функцию, принимающую два аргумента и возвращающую новое значение, можно создать функцию одного аргумента, которая возвращает, в свою очередь, унарную функцию. Когда эта вторая функция вызывается, это означает, что получены оба требуемых аргумента, и она может вернуть окончательный результат. Если же нужна функция трех аргументов, ее можно преобразовать в унарную функцию, которая возвращает только что построенную каррированную версию функции двух аргументов, – и так далее, сколько бы ни было аргументов.

Рассмотрим эту идею на простом примере. Пусть дана функция под названием `greater`, которая принимает два аргумента и проверяет, больше ли первый из них, чем второй, и возвращает результат типа `bool`. Пусть также дана ее каррированная версия, которая не может возвращать значение типа `bool`, так как ей известно значение лишь первого аргумента. Вместо этого она возвращает унарную лямбда-функцию, которая удерживает захваченное значение этого аргумента и, приняв свой аргумент, сравнивает его с этим значением (см. рис. 4.9).

<sup>1</sup> В его честь также назван язык программирования Haskell – один из самых популярных на сегодняшний день языков функционального программирования. – *Прим. перев.*

**Листинг 4.12** Функция сравнения «больше» и ее каррированный вариант

```
// greater : (double, double) → bool
bool greater(double first, double second)
{
    return first > second;
}

// greater_curried : double → (double → bool)
auto greater_curried(double first)
{
    return [first](double second) {
        return first > second;
    };
}

// Вызов
greater(2, 3); ← Возвращает значение false
greater_curried(2); ← Возвращает значение false
greater_curried(2)(3); ← Возвращает значение false
```



Возвращает унарный функциональный объект, который проверяет, меньше ли его аргумент, чем 2

Если у функции больше двух аргументов, нужно по этому образцу построить цепочку вложенных друг в друга лямбда-функций, каждая из которых принимает один аргумент и захватывает его, а самая внутренняя, использовав все собранные значения, возвращает требуемый результат.



Рис. 4.9 Преобразование функции, принимающей два аргумента, в унарную функцию, которая возвращает новую унарную функцию. Это позволяет подавать аргументы по одному, а не все вместе

### 4.2.1 Простой способ создавать каррированные функции

Как читатель наверняка помнит, функция `print_person` принимала три аргумента: объект типа `person_t`, содержащий данные о человеке, поток вывода и формат вывода. Для того чтобы вызвать эту функцию, нужно либо передать все три аргумента сразу, либо произвести частичное применение, чтобы отделить те аргументы, значения которых известны

заранее, от тех, чьи значения станут известны позже. Объявление этой функции выглядит следующим образом:

```
void print_person(const person_t& person,
                 std::ostream& out,
                 person_t::output_format_t format);
```

Тогда ее вызов может иметь вид, например:

```
print_person(person, std::cout, person_t::full_name);
```

Эту функцию можно привести к каррированному виду таким же способом, каким мы ранее преобразовали функцию `greater`, – вложив друг в друга столько лямбда-выражений, сколько нужно, чтобы захватить все аргументы для вызова функции `print_person` (рис. 4.10):

```
auto print_person_cd(const person_t& person)
{
    return [&](std::ostream& out) {
        return [&](person_t::output_format_t format) {
            print_person(person, out, format);
        };
    };
}
```

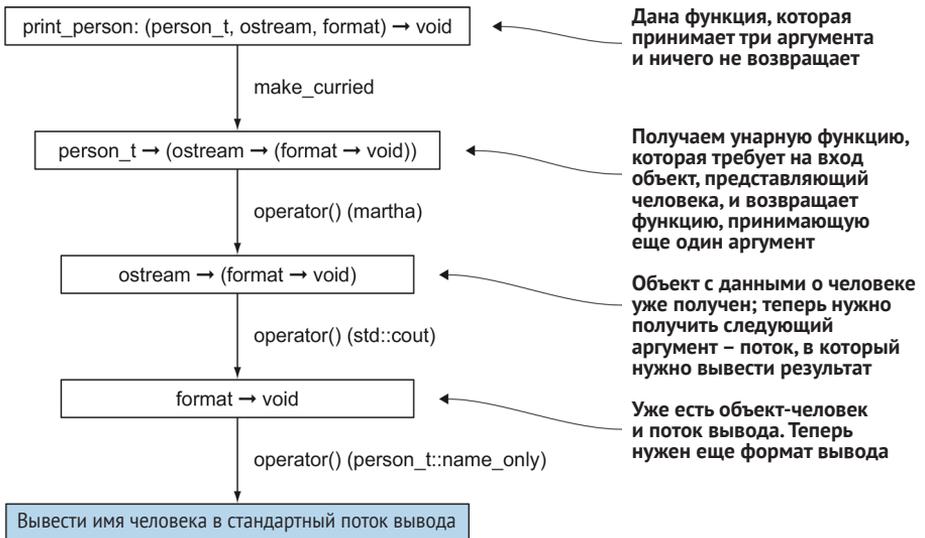


Рис. 4.10 Функция трех аргументов преобразована к каррированному виду. Каррированная функция принимает один аргумент (объект, представляющий человека) и возвращает новую функцию. Она в качестве аргумента принимает поток вывода и, в свою очередь, возвращает функцию. Эта последняя функция принимает формат вывода и выводит информацию о человеке

Поскольку написание кода в таком стиле слишком утомительно, будем впредь пользоваться вспомогательной функцией, именуемой `make_curried`. Эта функция позволяет преобразовать любую функцию в соответствующую каррированную форму. Более того, полученная в результате преобразования каррированная функция будет обладать дополнительным удобством, позволяя передавать ей за один раз более одного аргумента. Это не более, чем синтаксический сахар: каррированная функция по-прежнему остается унарной, только более удобной в использовании.

**ПРИМЕЧАНИЕ** О том, как реализовать эту функцию, читатель узнает из главы 11. Пока что реализация не важна, важен лишь факт наличия такой функции.

Прежде чем перейти к примерам использования каррированных функций в языке C++, покажем, что можно сделать с помощью функции `make_curried` и какой синтаксис применяется для вызова полученных каррированных функций.

#### Листинг 4.13 Каррированная версия функции `print_person`

```
using std::cout;

auto print_person_cd = make_curried(print_person);

print_person_cd(martha, cout, person_t::full_name);
print_person_cd(martha)(cout, person_t::full_name);
print_person_cd(martha, cout)(person_t::full_name);
print_person_cd(martha)(cout)(person_t::full_name);

auto print_martha = print_person_cd(martha);
print_martha(cout, person_t::name_only);

auto print_martha_to_cout =
    print_person_cd(martha, cout);
print_martha_to_cout(person_t::name_only);
```

Все эти вызовы делают одно и то же: посылают имя и фамилию Марты в стандартный поток вывода. Этот пример демонстрирует, что за один вызов можно передавать любое число аргументов.

Возвращает результат применения каррированной функции к первому аргументу – каррированную функцию, которая посылает данные Марты в какой-то поток в каком-то формате, которые еще предстоит задать

Возвращает функциональный объект, который посылает данные Марты в стандартный поток вывода в заданном формате

Что же представляет собой карринг: только лишь изящную идею, пришедшую в голову математику лишь потому, что об унарных функциях рассуждать удобнее, чем о функциях произвольного числа аргументов? Или же эта идея может сослужить службу в повседневной работе программиста?

## 4.2.2 Использование карринга для доступа к базе данных

Рассмотрим пример из реальной жизни. Допустим, разрабатывается приложение, которое подключается к базе данных и выполняет ряд запросов к ней. Например, приложению нужен список всех пользователей, которые поставили оценку определенному фильму.

Допустим, при создании приложения используется библиотека, которая позволяет создавать множество одновременных соединений с базой данных, открывать вложенные сеансы (например, для выполнения транзакций) и, конечно же, запрашивать данные. Предположим, что основная функция, посредством которой выполняются запросы, имеет вид:

```
result_t query(connection_t& connection,
               session_t& session,
               const std::string& table_name,
               const std::string& filter);
```



Эта функция запрашивает из заданной таблицы все строки, удовлетворяющие заданному критерию фильтрации. Все, что для этого нужно, – это ранее открытое соединение и сеанс, чтобы реализация библиотеки знала, к какой базе данных осуществляется запрос и какое ее состояние нужно использовать в данной транзакции.

Многим приложениям не нужна возможность создавать одновременно несколько соединений с базами данных – им довольно единственного соединения для всех запросов. Один из подходов, которые применяют для этого разработчики библиотек, состоит в том, чтобы сделать функцию `query` членом класса `connection_t`. Возможно и другое решение: определить для функции `query` перегруженную версию, которая не принимает соединение через аргумент, а использует некоторое глобальное соединение по умолчанию.

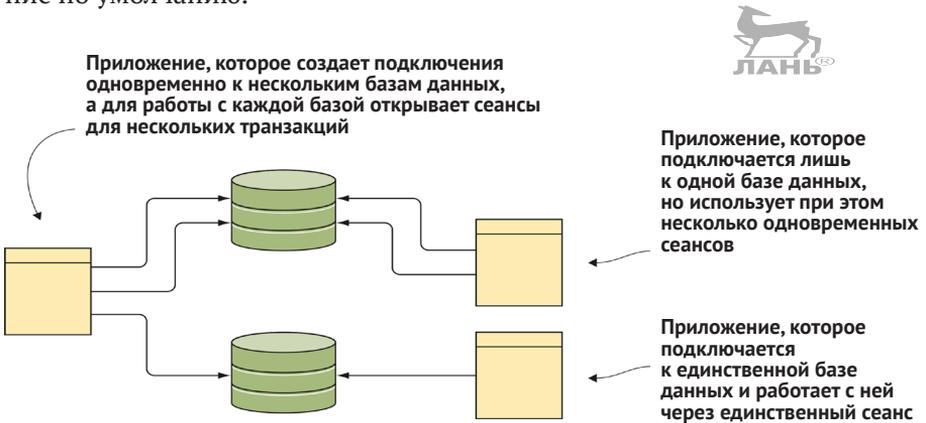


Рис. 4.11 У разных приложений – различные потребности: одним нужно подключаться к нескольким базам данных, другим нужно несколько сеансов при работе с одной базой, третьим достаточно одного соединения и одного сеанса

Ситуация становится еще более запутанной, если принять во внимание, что некоторым пользователям не нужны и множественные сеансы. Если они пользуются базой данных только для чтения, нет смысла открывать транзакции. Как и в предыдущем случае, автор библиотеки мог бы сделать метод `query` членом класса, ответственного за сеанс, и использовать во всем приложении единственный объект этого класса, или же

определить еще одну перегруженную версию данной функции – в этот раз без двух аргументов, соединения и сеанса.

Теперь легко вообразить, что некоторым частям приложения требуется доступ лишь к одной таблице (в нашем примере это таблица, в которой хранятся выставленные пользователями оценки). Тогда библиотека могла бы предоставлять класс `table_t`, обладающий собственным методом `query`, и т. д.

Трудно заранее предугадать все сценарии использования, которые могут понадобиться пользователям. Даже те, которые мы здесь описали, делают библиотеку значительно более сложную, чем необходимо. Посмотрим, может ли единственной функции `query` самого общего вида, только на этот раз каррированной, хватить для всевозможных сценариев; это освободило бы разработчика библиотеки от изнуряющей работы по написанию многочисленных перегрузок или специализированных классов для поддержки каждого отдельного сценария.

Если пользователям нужно одновременно несколько соединений с базами данных, они могут либо пользоваться функцией `query` напрямую, всякий раз явно указывая соединение и сеанс, либо определить для каждого соединения по одной вспомогательной функции, которая бы подставляла в каррированную функцию первый аргумент. Этот подход годится также и для случая, когда на все приложение нужно лишь одно соединение с базой данных.

Если в приложении используется не только единственная база данных, но и единственный сеанс, легко создать функцию, в точности подходящую для этого случая. Нужно вызвать каррированную функцию `query`, передав ей два первых аргумента, соединение и сеанс, – она вернет функцию, требующую оставшихся аргументов, которую и следует использовать в оставшейся части приложения. Наконец, если надо много раз выполнять различные запросы над одной таблицей, можно таким же образом зафиксировать значение еще одного аргумента.

#### Листинг 4.14 Борьба с разрастанием интерфейса с помощью каррированных функций

```

auto table = "Movies";
auto filter = "Name = \"Sintel\"";

results = query(local_connection, session,
               table, filter);

auto local_query = query(local_connection);
auto remote_query = query(remote_connection);

results = local_query(session, table, filter);

auto main_query = query(local_connection,
                       main_session);

results = main_query(table, filter);

```

Функция `query` используется как обычная функция, которой можно передать все нужные аргументы и получить список запрошенных строк

Создаются отдельные функции, привязанные каждая к своему соединению с базой данных. Это полезно, если в оставшейся части приложения будет многократно использоваться одно и то же соединение

Если всюду используется только одно соединение и единственный сеанс, можно определить функцию, у которой значения этих двух аргументов фиксированы, и в оставшейся части программы опускать эти два аргумента



```
auto movies_query = main_query(table);
results = movies_query(filter);
```

Если нужно часто выполнять запросы к одной и той же таблице, можно создать функцию, у которой фиксировано значение и этого аргумента

Предоставляя пользователям каррированную версию функции `query`, мы даем им возможность самостоятельно создавать в точности те функции, которые нужны им для конкретных задач, и избегаем необходимости усложнять интерфейс библиотеки. Это делает код значительно более удобным для повторного использования, поскольку нам более не нужно создавать множество специализированных классов, каждый из которых потенциально чреват своими специфическими ошибками, которые удастся обнаружить лишь тогда, когда возникает конкретный сценарий и когда используется какой-то определенный класс. Вместо всего этого появляется единая реализация, пригодная на все возможные случаи.

### 4.2.3 Карринг и частичное применение функций

На первый взгляд, между каррингом и частичным применением есть много общего. Оба механизма позволяют создавать новые функции, связывая некоторые аргументы функции с фиксированными значениями и оставляя остальные свободными. Из предыдущих разделов ясно лишь, что они различаются синтаксисом и что карринг более ограничен в том смысле, что аргументы каррированной функции можно связывать лишь по порядку: первый аргумент связывается первым, последний аргумент – последним.

Оба подхода позволяют решать сходные задачи. Можно даже реализовать один из них на основе другого. Однако между каррингом и частичным применением имеется важное различие, которое делает оба подхода в равной степени полезными.

Преимущество связывания аргументов функцией `std::bind` по сравнению с каррингом очевидно: в первом случае можно взять функцию многих аргументов, например функцию `query`, и зафиксировать значения любых ее аргументов, тогда как каррированная функция позволяет зафиксировать значение лишь первого аргумента (при многократном применении – нескольких первых аргументов).

#### Листинг 4.15 Связывание аргументов каррированной и обычной функций

```
auto local_query = query(local_connection);
auto local_query = std::bind(
    query, local_connection, _1, _2, _3);
auto session_query = std::bind(
    query, _1, main_session, _2, _3);
```

Связывание одного лишь первого аргумента `connection` со значением `local_connection`

Можно связать значение первого аргумента, но можно также связать один лишь второй аргумент

Из этого примера можно было бы заключить, что функция `std::bind` лучше карринга во всем, кроме разве что более сложного синтаксиса. Однако она обладает и одним важным недостатком, который также должен

быть очевиден из предыдущего примера. Для использования функции `std::bind` нужно заранее точно знать, сколько аргументов у функции, к которой применяется функция `std::bind`. Каждый аргумент этой функции нужно связать либо со значением (возможно, с переменной или ссылкой на переменную), либо с заглушкой. В случае же каррированной функции `query` об этом можно не беспокоиться: каждый раз фиксируется значение одного лишь первого аргумента, в результате чего получается новая функция, которая принимает все оставшиеся аргументы, сколько бы их ни было.

Это различие могло бы показаться чисто синтаксическим – в случае карринга нужно набрать на клавиатуре меньше текста, но дело не только в этом. Помимо функции `query`, запрашивающей данные, в библиотеке, скорее всего, будет и функция `update`. Эта функция должна менять значения в строках таблицы, удовлетворяющих фильтру. Она принимает те же аргументы, что и функция `query`, и еще один – правило, согласно которому должна быть изменена каждая соответствующая фильтру строка:

```
result_t update(connection_t& connection,
                session_t& session,
                const std::string& table_name,
                const std::string& filter,
                const std::string& update_rule);
```

Было бы естественно создать для функции `update` такие же специализированные версии, что были рассмотрены ранее для функции `query`. Если есть всего одно соединение и если создана специализированная функция `query`, работающая с этим соединением, естественно ожидать, что то же самое придется сделать и с функцией `update`. Если бы это делалось посредством функции `std::bind`, пришлось бы обращать внимание на количество аргументов, тогда как при использовании каррированных функций довольно было бы скопировать имеющееся определение, заменив в нем лишь имя функции:

```
auto local_query = query(local_connection);
auto local_update = update(local_connection);
```

Можно было бы по-прежнему сказать, что различие здесь чисто синтаксическое. В самом деле, программист обычно знает, сколько аргументов принимает функция, и всегда может вписать нужное количество заглушек, чтобы связать все аргументы. Но как быть, если точное число аргументов заранее неизвестно? Так, наш случай, где у нескольких функций разной арности нужно связать первый аргумент с определенным значением (в данном случае это соединение с базой данных), довольно типичен, и мы могли бы создать такую вспомогательную функцию, которая принимает в качестве аргумента произвольную функцию и связывает ее первый аргумент с объектом `local_connection`.

Если для этого пользоваться функцией `std::bind`, пришлось бы прибегнуть к изощренному метапрограммированию, для того чтобы наша

функция одинаково хорошо работала с функциями любой арности. Реализация на основе карринга, напротив, получается тривиальной:

```
template <typename Function>
auto for_local_connection(Function f) {
    return f(local_connection);
}

auto local_query = for_local_connection(query);
auto local_update = for_local_connection(update);
auto local_delete = for_local_connection(delete);
```



Таким образом, читатель мог убедиться, что, несмотря на некоторое сходство, карринг и частичное применение функций представляют собой два различных механизма, каждый из которых обладает своими преимуществами и недостатками. У каждого из этих механизмов есть свои области применения. Так, частичное применение функций лучше использовать, когда дана фиксированная функция и в разных контекстах требуется фиксировать значения то одних, до других ее аргументов. В этом случае заранее известно, сколько всего у функции аргументов, и легко выбрать те из них, которые нужно связать с заданными значениями. Карринг бывает особенно полезен в общем случае, когда единым образом нужно обрабатывать функции, обладающие различным числом аргументов. Тогда функция `std::bind` неприменима, так как нельзя заранее знать, сколько аргументов принимает преобразовываемая функция и, следовательно, сколько из них нужно связать с заглушками; фактически неизвестно даже, сколько всего нужно заглушек.

### 4.3 Композиция функций

В далеком 1986 году великий Дональд Кнут получил предложение написать программу для колонки «Жемчужины программирования» журнала «Communications of the ACM»<sup>1</sup>. Условие задачи звучало так: прочитать текстовый файл, определить в нем  $n$  наиболее часто встречающихся слов и напечатать отсортированный список этих слов вместе с их частотами. Д. Кнут представил решение на Паскале, которое занимало десять страниц. Решение сопровождалось подробными комментариями и даже включало в себя структуру данных, придуманную специально для работы со словами и их частотами.

В ответ на эту публикацию Дуглас МакИлрой написал свое решение той же задачи в виде командного сценария UNIX, которое занимает всего шесть строк:

```
tr -cs A-Za-z '\n' |
tr A-Z a-z |
sort |
```

<sup>1</sup> Ежемесячный журнал Ассоциации вычислительной техники (англ. *Association for Computing Machinery, ACM*).

```
uniq -c |  
sort -rn |  
sed ${1}q
```

Хотя мы в этой книге не занимаемся командными сценариями, этот пример превосходно иллюстрирует функциональный подход к решению задач. Во-первых, решение получилось удивительно кратким. Решение не содержит подробного перечисления шагов алгоритма, а, скорее, представляет собой перечень преобразований, которые необходимо применить к входным данным, чтобы на выходе получить желаемый результат. Такое решение не обладает состоянием, в нем нет никаких переменных. Это наиболее чистое решение поставленной задачи.

МакИлрой разбил поставленную задачу на ряд простых и мощных функций, написанных кем-то другим (в данном случае в роли функций выступают команды UNIX), и организовал передачу данных с выхода одной функции на вход другой. Именно эта возможность состыковывать функции между собой обеспечивает выразительную мощь, достаточную для решения сложной задачи в несколько строк кода.

Поскольку большинство из нас владеет языком командного интерпретатора не на том же уровне, что МакИлрой, проанализируем эту задачу с нуля, чтобы разобраться, как разбить ее на подзадачи таким образом, дабы решение было удобно построить на языке C++. Не будем заниматься здесь строчными и прописными буквами, поскольку эта часть задачи наименее интересна.

Процесс обработки данных, изображенный на рис. 4.12, состоит из следующих преобразований:

- 1 Дан файл. Его нетрудно открыть и прочесть находящийся в нем текст. Вместо того чтобы все содержимое файла представлять одной строкой, нужно получить его в виде списка слов.
- 2 Поместить все эти слова в ассоциативный массив `std::unordered_map<std::string, unsigned int>`, накапливая в нем частоты всех слов. Структура данных `std::unordered_map` используется потому, что она работает быстрее, чем `std::map`, а какая-либо сортировка слов на этом этапе не требуется.
- 3 Пройти по всем элементам этого ассоциативного массива (его элементы имеют тип `std::pair<const std::string, unsigned int>`) и поменять местами элементы каждой пары, чтобы сначала стояла частота, а затем – слово.
- 4 Полученную коллекцию пар отсортировать лексикографически (т. е. сначала по первому элементу пары, в порядке убывания, а затем по второму). Тогда наиболее часто встречающиеся слова окажутся в начале коллекции, а редко встречающиеся – в конце.
- 5 Вывести на печать слова с частотами.

Глядя на этот перечень, можно заключить, что предстоит создать пять функций. Каждая функция должна делать одну простую вещь, и работать их нужно так, чтобы из этих функций было как можно проще строить композиции. А именно так, чтобы результат каждой из функций легко было подавать на вход следующей. Файл с исходными данными

можно прочитать в виде коллекции символов – например, в виде строки `std::string`, поэтому первая в ряду функций (назовем ее `words`) должна принимать аргумент типа `std::string`. Ее возвращаемое значение – это коллекция слов, поэтому можно использовать тип `std::vector<std::string>` (конечно, можно применить и более эффективную структуру данных, позволяющую избежать ненужного копирования данных, однако рассмотрение таких структур выходит за рамки этой главы – мы вернемся к данной теме в главе 7). Таким образом, получаем следующий прототип:

```
std::vector<std::string> words(const std::string& text);
```

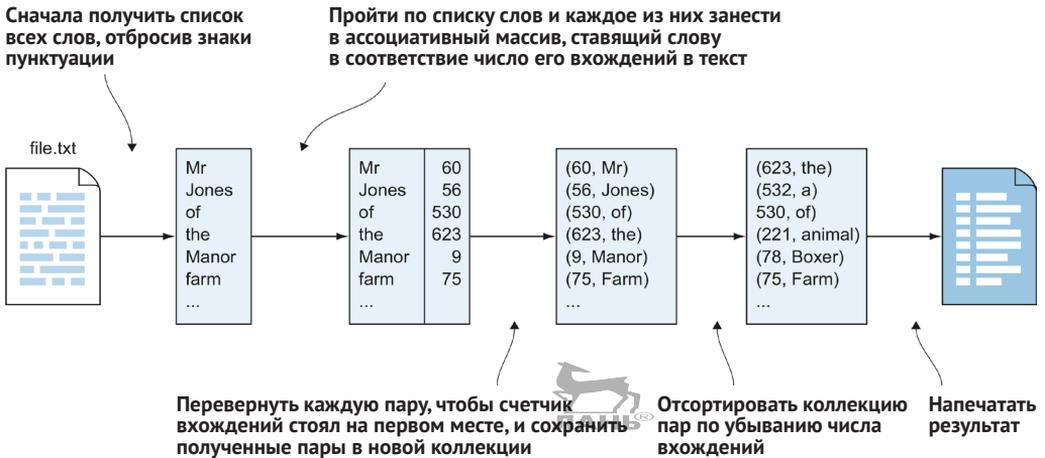


Рис. 4.12 Преобразование текста в список слов, последующий подсчет частоты встречаемости каждого слова и сортировка слов по убыванию частоты

Вторая функция, которой дадим имя `count_occurrences`, получает перечень слов, т. е. объект типа `std::vector<std::string>`. Она должна создать ассоциативный массив типа `std::unordered_map<std::string, unsigned int>`, в котором для каждого слова хранится число его вхождений в текст. Можно даже сделать эту функцию шаблоном, чтобы она могла работать не только со строками, но и с данными иных типов:

```
template <typename T>
std::unordered_map<T, unsigned int> count_occurrences(
    const std::vector<T>& items);
```

### Если добавить немного магии

Эту функцию можно параметризовать не только по типу элемента, но также и по типу коллекции:

```
template <typename C,
        typename T = typename C::value_type>
std::unordered_map<T, unsigned int> count_occurrences(
    const C& collection)
```

Теперь функция должна принимать в качестве аргумента коллекцию любого типа, из которой можно получить тип хранящихся в ней элементов (`C::value_type`). Ее можно применять для подсчета частоты букв в строке, строк в векторе строк, чисел в списке чисел и т. д.



Третья функция принимает на вход контейнер пар, по очереди берет в нем каждую пару и меняет местами ее компоненты. Функция возвращает коллекцию получившихся новых пар. Поскольку эту коллекцию предполагается в будущем сортировать и поскольку мы хотим создавать такие функции, чтобы из них легко было образовывать композиции, пусть возвращаемой коллекцией будет вектор.

```
template <typename C,  
         typename P1,  
         typename P2>  
std::vector<std::pair<P2, P1>> reverse_pairs(  
    const C& collection);
```

Теперь остается определить лишь функцию, которая сортирует вектор (назовем ее `sort_by_frequency`), и еще одну функцию, которая выводит слова вместе с их частотами в поток (`print_pairs`). Композиция этих функций дает искомое решение задачи. Обычно композиция функций записывается на языке C++ в виде подстановки вызова одной функции на место аргумента другой. В данном примере композиция принимает вид:

```
void print_common_words(const std::string& text)  
{  
    return print_pairs(  
        sort_by_frequency(  
            reverse_pairs(  
                count_occurrences(  
                    words(text)  
                )  
            )  
        )  
    );  
}
```



На этом примере видно, что задачу, решение которой в императивном стиле занимает не одну страницу, можно разбить на несколько функций, каждая из которых сравнительно мала и проста в реализации. Начав с постановки сложной задачи, мы не стали анализировать шаги, которые нужно сделать для получения результата, а начали думать о том, каким преобразованиям нужно подвергнуть исходные данные. Затем для каждого из этих преобразований мы создали простую и короткую функцию. Потом осталось объединить их в одну большую функцию, которая решает исходную задачу.

## 4.4 Повторное знакомство с подъемом функций

Тему подъема функций мы затронули еще в главе 1, теперь самое время для более развернутого пояснения. Вообще говоря, *подъем* (англ. lifting) функций – это шаблон функционального программирования, который позволяет некоторую имеющуюся функцию преобразовать в другую функцию, подобную первой, но применимую в более широком контексте. Например, если дана функция, работающая со строковым аргументом, путем подъема из нее можно сделать функцию, работающую с вектором или списком строк, указателем на строку, ассоциативным массивом, отображающим целые числа на строки, и с любыми другими структурами данных, в которых содержатся строки.

Начнем с простой функции, которая принимает строку и переводит все буквы в верхний регистр:

```
void to_upper(std::string& string);
```

Если дана функция для работы со строками, сложно ли сделать из нее функцию, работающую с указателем на строку? Она должна преобразовывать в верхний регистр строку, на которую указывает указатель, если этот указатель не равен `nullptr`. Сложно ли использовать эту функцию для преобразования всех имен, хранящихся в векторе строк или в ассоциативном массиве, отображающем числовые идентификаторы фильмов на их названия? Все эти функции создать очень просто, как показано в следующем листинге.

### Листинг 4.17 Функции для работы с коллекциями строк

<pre>void pointer_to_upper(std::string* str) {     if (str) to_upper(*str); }</pre>	<p>Указатель на строку можно рассматривать как контейнер, который может содержать либо один элемент, либо ни одного. Если указатель указывает на строку, к ней применяется преобразование</p>
<pre>void vector_to_upper(std::vector&lt;std::string&gt;&amp; strs) {     for (auto&amp; str : strs) {         to_upper(str);     } }</pre>	<p>Вектор может содержать сколь угодно много строк. Эта функция преобразовывает их все в верхний регистр</p>
<pre>void map_to_upper(std::map&lt;int, std::string&gt;&amp; strs) {     for (auto&amp; pair : strs) {         to_upper(pair.second);     } }</pre>	<p>Ассоциативный массив содержит пары типа <code>&lt;const int, std::string&gt;</code>. Первый компонент каждой пары, целое число, остается без изменений, а второй – строкового типа – переводится в верхний регистр</p>

Следует обратить внимание на то, что реализация этих трех функций совершенно не изменилась бы, будь на месте функции `to_upper` какая-

либо другая функция преобразования строк с той же сигнатурой. Иными словами, реализация функций, выполняющих поэлементное преобразование контейнеров, зависит только от типа контейнера; так, в данном примере показаны три реализации: для указателя на строку, вектора строк и ассоциативного массива (рис. 4.13).

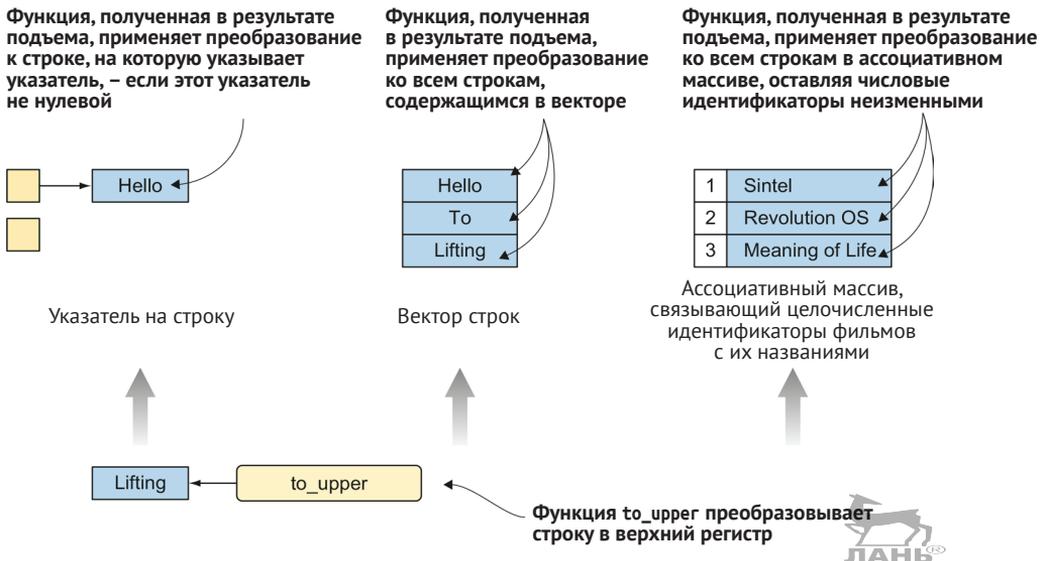


Рис. 4.13 Функция, преобразовывающая одну строку, в результате подъема превращается в несколько разных функций, применяющих преобразование к строкам, содержащимся в различных контейнерах

Следовательно, можно определить функцию высшего порядка, которая принимает в качестве аргумента произвольную функцию, работающую с одним строковым аргументом, и создает новую функцию, которая работает с указателем на строку. Таким же образом можно создать отдельную функцию, работающую с векторами строк, и еще одну – для ассоциативных массивов. Такие функции высшего порядка называют функциями *подъема*, поскольку они берут функцию, работающую с отдельно взятыми объектами некоторого типа, и превращают ее в функцию, работающую с целыми контейнерами, содержащими элементы этого типа.

Попробуем реализовать функции подъема своими руками; ради краткости кода будем пользоваться средствами стандарта C++14. Кроме того, их легко было бы реализовать в виде классов с перегруженной операцией вызова, как и некоторые примеры из предыдущей главы.

**Листинг 4.18 Функции подъема**

```
template <typename Function>
auto pointer_lift(Function f)
{
```

```

return [f](auto* item) {
    if (item) {
        f(*item);
    }
};
}
template <typename Function>
auto collection_lift(Function f)
{
    return [f](auto& items) {
        for (auto& item : items) {
            f(item);
        }
    };
}
}

```

Тип аргумента задан ключевым словом `auto`, поэтому данную функцию можно использовать не только для указателей на строки, но и для указателей на объекты данных любого типа

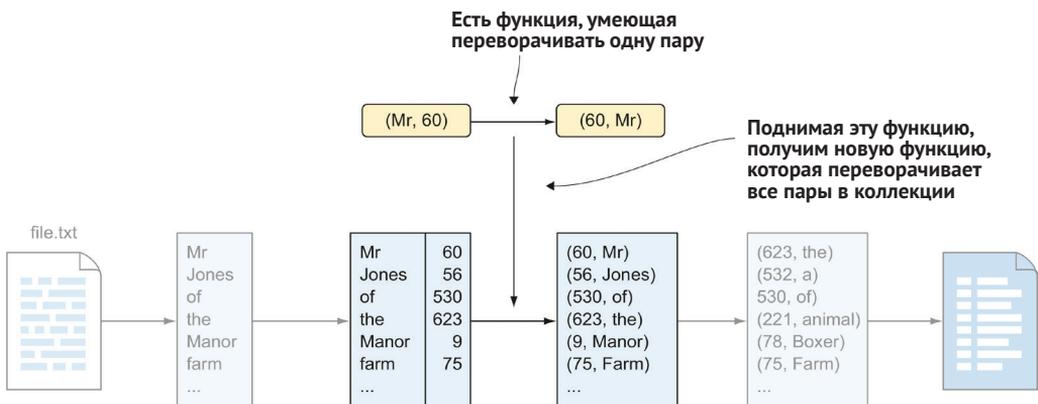


Сказанное верно и для этой функции: ее можно использовать не только с векторами строк, но и с любыми коллекциями, поддерживающими итераторы, каким бы ни был тип их элементов

Подобные функции подъема легко определить для любых других контейнерных (или похожих на контейнерные) типов: структур с полями строкового типа; умных указателей `std::unique_ptr` и `std::shared_ptr`; потоков ввода, разбивающих текст на лексемы; а также для некоторых необычных контейнерных типов, о которых речь пойдет ниже. Таким образом, программист получает возможность сначала реализовать простейшую функцию преобразования отдельного объекта, а затем, применив подходящую функцию подъема, сразу получить преобразователь нужной ему структуры данных.

#### 4.4.1 Переворачивание пар – элементов списка

Теперь, когда читатель знаком с понятием подъема функций, вернемся к задаче Д. Кнута. Одним из шагов преобразования данных была перестановка местами компонентов пар, хранящихся в коллекции, и для этого мы решили создать функцию под названием `reverse_pairs`. Рассмотрим эту функцию подробнее – см. рис. 4.14.



**Рис. 4.14** Дана коллекция пар, которые нужно перевернуть, и функция, которая умеет переворачивать одну пару. Подъем дает функцию, которая переворачивает каждую пару в коллекции



Как и в предыдущем случае, функцию `reverse_pairs` можно сделать шаблоном, чтобы она могла принимать любую коллекцию, поддерживающую итераторы (вектор, список, ассоциативный массив и т. д.) и содержащую пары значений каких угодно типов. Вместо того чтобы модифицировать элементы в самой коллекции, на этот раз реализуем функцию чистым образом. Функция должна применять одно и то же преобразование (в данном случае – перестановку компонентов пары) ко всем элементам коллекции и из полученных результатов строить новую коллекцию (см. пример `knuth-problem/main.cpp`). Читатель уже знает, что это можно сделать с помощью функции `std::transform`.

#### Листинг 4.19 Подъем лямбда-функции

```
template <
    typename C,
    typename P1 = typename std::remove_cv<
        typename C::value_type::first_type>::type,
    typename P2 = typename C::value_type::second_type
    >
std::vector<std::pair<P2, P1>> reverse_pairs(const C& items)
{
    std::vector<std::pair<P2, P1>> result(items.size());

    std::transform(
        std::begin(items), std::end(items),
        std::begin(result),
        [](const std::pair<const P1, P2>& p)
        {
            return std::make_pair(p.second, p.first);
        }
    );

    return result;
}
```

Тип `C` – это тип коллекции, `P1` – тип первого компонента пары – элемента этой коллекции (после отбрасывания спецификатора `const`), а `P2` – тип второго компонента пары

В роли преобразователя элементов выступает лямбда-функция, которая меняет местами компоненты одной пары. Передав ее в алгоритм `std::transform`, поднимаем ее до функции, которая делает то же самое со всеми парами – элементами коллекции

Функция, которая принимает пару значений и возвращает новую пару, составленную из тех же значений в обратном порядке, в результате подъема превращается в функцию, преобразующую любые коллекции пар. Сложные структуры данных и объекты строятся путем композиции более простых. Можно создавать структуры, содержащие поля других типов, а также векторы и прочие коллекции, состоящие из произвольного числа элементов одинакового типа, и т. д. Поскольку в программе нужно выполнять какие-то действия с такими объектами, программист определяет функции для работы с ними – это могут быть как функции-члены, так и обычные функции. Часто механизм подъема простых функций позволяет легко реализовывать функции по обработке сложных объектов, если требуется просто передать эстафету преобразования объектам-компонентам, из которых состоит сложный объект.

Для каждого из более сложных типов нужно определить свою функцию подъема – тогда становится возможно применять к сложному объекту



все те функции-преобразователи, которые определены для завернутых в нем более простых типов. Это одна из тех областей, в которых объектно-ориентированная и функциональная парадигмы идут рука об руку.

Большая часть подходов, рассмотренных в этой главе в контексте функционального программирования, оказывается полезной также и в мире объектно-ориентированного программирования. Так, читатель видел, что с помощью частичного применения можно связать указатель на функцию-член с конкретным экземпляром класса, в котором эта функция объявлена (листинг 4.11). В качестве самостоятельного упражнения предлагается проследить, как функции-члены с их неявным аргументом `this` ведут себя при карринге.

**СОВЕТ** Более подробную информацию и ссылки по темам, затронутым в этой главе, можно найти на странице <https://forums.manning.com/posts/list/41683.page>.

## Итоги

- Функции высшего порядка наподобие функции `std::bind` можно использовать для преобразования существующих функций в новые. Например, можно взять функцию от  $n$  аргументов и, зафиксировав значения некоторых из них с помощью функции `std::bind`, без труда превратить ее в унарную или бинарную функцию, которую удобно передавать в такие алгоритмы, как `std::partition` или `std::sort`.
- Заглушки обеспечивают высокую степень выразительности при определении того, какие аргументы функции следует оставить несвязанными. Они позволяют менять местами аргументы исходной функции, передавать один и тот же аргумент преобразованной функции в несколько аргументов исходной функции и т. д.
- Хотя функция `std::bind` предоставляет лаконичный синтаксис для частичного применения функций, она влечет некоторую потерю производительности. При написании кода с критически важным быстродействием, который вызывается достаточно часто, стоит задуматься об использовании лямбда-функций. Они обладают более пространственным синтаксисом, но компилятор способен оптимизировать их лучше, чем функциональные объекты, созданные функцией `std::bind`.
- Разработка прикладного интерфейса библиотеки – непростое дело. Часто бывает нужно покрыть множество различных сценариев использования. Здесь на помощь могут прийти каррированные функции.
- О функциональном программировании нередко говорят, что оно позволяет писать более компактный код. Если создавать функции такими, чтобы их было легко состыковывать между собой, действительно становится возможно решать сложные задачи, написав в разы меньше кода, чем потребовалось бы при традиционном императивном подходе.



# Чистота функций: как избежать изменяемого состояния

## О чем говорится в этой главе:

- трудности создания корректного кода с изменяемым состоянием;
- понятие о референциальной прозрачности и его связь с чистой функцией;
- как программировать, не изменяя значения переменных;
- как распознать ситуации, когда в изменяемом состоянии нет ничего страшного;
- использование спецификатора `const` для гарантии неизменности состояния.



С понятиями неизменяемого состояния и чистых функций читатель познакомился в главе 1. Как говорилось в ней, одна из главных причин существования ошибок в программах состоит в том, что человеку трудно удержать в голове все состояния, в которых может находиться программа. В мире объектно-ориентированного программирования с этой трудностью пытаются справиться, заключая части состояния программы в отдельные объекты. Данные, составляющие состояние, оказываются скрытыми за интерфейсом класса, и любой доступ к этим данным допускается только через интерфейс.

Это позволяет программисту держать под своим контролем то, какие изменения данных допустимы и должны быть приняты, а какие – нет. Например, устанавливая в программе дату рождения человека, можно проверить, что эта дата не находится в будущем, что человек не ока-

жется старше собственных родителей и т. д. Такие проверки уменьшают число состояний, в которых может пребывать программа, ограничивая их только такими состояниями, которые считаются корректными. Хотя этот метод и повышает шансы написать корректную программу, он по-прежнему оставляет лазейки для разнообразных проблем.



## 5.1 Проблемы изменяемого состояния

Рассмотрим следующий пример. Объект класса `movie_t` содержит название фильма и список оценок, которые этому фильму поставили пользователи. В классе определена функция-член, которая вычисляет среднюю оценку фильма:

```
class movie_t {
public:
    double average_score() const;

    ...

private:
    std::string name;
    std::list<int> scores;
};
```



Читатель уже знает из главы 2, как реализовать функцию для подсчета средней оценки фильма, и здесь можно повторно использовать ту же реализацию. Единственное различие состоит в том, что в первоначальной реализации список оценок приходил в функцию через аргумент, теперь же он представляет собой член класса `movie_t`.

### Листинг 5.1 Вычисление средней оценки фильма

```
double movie_t::average_score() const
{
    return std::accumulate(scores.begin(),
                           scores.end(), 0)
           / (double)scores.size();
}
```

Вызов методов `begin` и `end` для объекта со спецификатором `const` эквивалентен вызову методов `cbegin` и `cend`

Теперь спросим себя, корректен ли этот код – всегда ли он делает то, чего от него хочет программист. Код вычисляет сумму всех чисел в списке и делит его на количество этих чисел. Код кажется корректным.

Однако что произойдет, если кто-то добавит в список новую оценку *во время вычисления* средней оценки? Поскольку для хранения оценок используется связный список (`std::list`), итераторы гарантированно останутся валидными, и алгоритм `std::accumulate` завершится без ошибок. Он вернет сумму всех элементов списка, которые обработает. Проблема состоит в том, что вновь добавленная оценка может как войти, так и не войти в эту сумму – в зависимости от того, в какое место списка она вставлена (см. рис. 5.1). Также и метод `.size()` может вернуть как

старый, так и новый размер списка, потому что в правилах языка C++ не определено, какой из аргументов операции деления должен вычисляться первым.

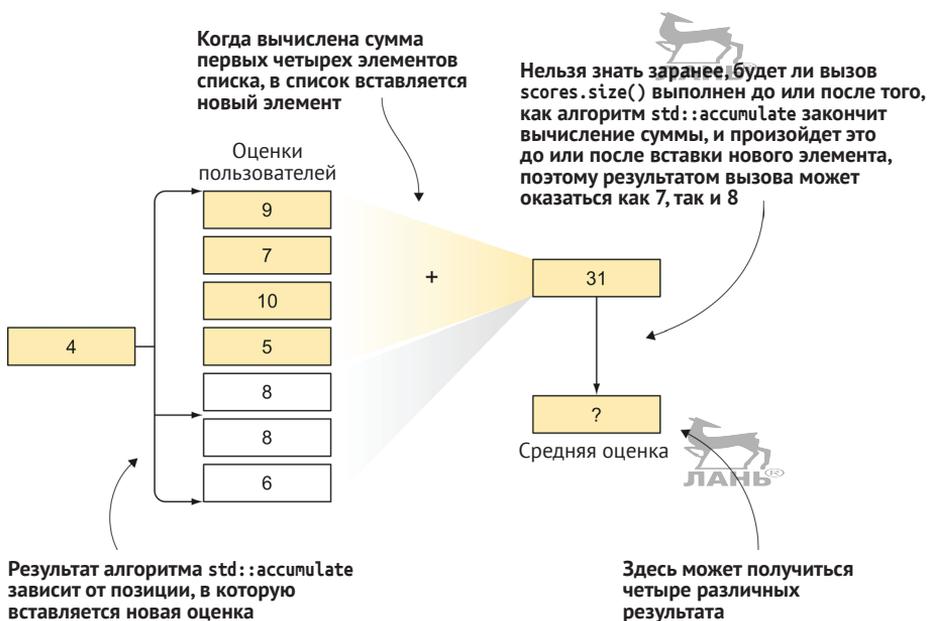


Рис. 5.1 Результат функции зависит от порядка вычисления операндов операции деления и от позиции, в которую вставляется новый элемент во время подсчета средней оценки

Таким образом, невозможно гарантировать корректность результата во всех возможных сценариях использования. Было бы даже лучше, если бы мы точно знали, что функция ведет себя некорректно во всех случаях (тогда можно было бы либо избегать ее использования, либо исправить ее), но и этого нельзя сказать наверняка. Плохо также то, что результат этой функции зависит от того, каким образом реализованы другие функции в данном классе (например, от того, в начало или в конец списка добавляются новые оценки), хотя наша функция их не вызывает.

Хотя в данном примере, для того чтобы вызвать некорректное поведение функции `average_score`, требуется наличие другой функции, выполняющейся параллельно с ней, это не единственный сценарий, при котором выполнение программы может пойти неправильно, – довольно того, что программа разрешает менять список оценок. Нередки ситуации, когда за несколько лет существования программной системы в ней появляется несколько переменных, чьи значения зависят друг от друга.

Например, до принятия стандарта C++11 от контейнера `std::list` не требовалось помнить свой размер, метод `size` вполне мог пробежать по всем элементам списка, чтобы сосчитать их. Проницательный разработчик мог бы заметить эту деталь, добавить в класс новое поле и хранить

в нем число оценок в списке – для того чтобы повысить производительность всех участков кода, которым нужно знать количество выставленных оценок.

```
class movie_t {
public:
    double average_score() const;

    ...

private:
    std::string name;
    std::list<int> scores;
    size_t scores_size;
};
```

Теперь переменные-члены `scores` и `scores_size` тесно связаны между собой, и модификация любой из них должна отражаться на другой. Это значит, что они недостаточно хорошо инкапсулированы. К ним нет прямого доступа извне. Класс `movie_t` инкапсулирован достаточно хорошо, однако внутри него с полями `scores` и `scores_size` можно делать все, что угодно, – без возражений со стороны компилятора. Нетрудно вообразить, что когда-нибудь в будущем с этим кодом будет работать новый разработчик, который в каком-то редко выполняемом участке кода изменит список и забудет обновить размер.

В описанном случае ошибку, вероятно, будет найти нетрудно, но вообще ошибки такого типа оказываются коварными и трудно обнаруживаемыми. По мере развития программной системы число тесно связанных участков кода обычно возрастает до тех пор, пока кто-то не решится на рефакторинг проекта.

Могли бы мы столкнуться со всеми этими проблемами, если бы переменные-члены `scores` и `scores_list` были неизменяемыми, т. е. объявленными со спецификатором `const`? Первая проблема возникла из-за того, что кто-то мог внести в список оценок изменение, в то время пока вычисляется средняя оценка. Если список сделать неизменяемым, никто вообще не может его изменять, поэтому данная проблема теряет смысл.

Как быть со второй проблемой? Если переменные-члены неизменяемы, значения им можно присвоить только в момент создания объекта класса `movie_t`. Конечно, можно проинициализировать переменную-член `scores_size` неправильным значением, но такая ошибка должна быть явной. Она не могла бы возникнуть просто из-за того, что кто-то забыл модифицировать значение переменной, – ошибка бы могла произойти только из-за того, что кто-то написал неверный код для вычисления этого значения. Кроме того, после присваивания ошибочного начального значения оно так и останется ошибочным – ошибка не исчезнет сама в процессе дальнейшей работы программы, прячась от отладчика.

## 5.2 Чистые функции и референциальная прозрачность

Все отмеченные выше трудности возникают из-за одного изъяна в замысле программы – из-за того, что несколько компонентов системы ответственны за одни и те же данные, но не знают о том, как и когда другие компоненты вносят в них изменения. Самый простой способ борьбы с этим – запретить любые изменения данных. Все проблемы сразу исчезают.

Однако это проще сказать, чем сделать. Любое взаимодействие программы с пользователем – это изменение состояния. Если один из компонентов программы читает строку текста из входного потока, он тем самым меняет состояние потока для всех других компонентов: они уже никогда не смогут прочесть эту строку текста. Если одна кнопка реагирует на щелчок мыши, другие кнопки (при стандартном поведении визуального интерфейса) уже никогда не узнают об этом нажатии.

Иногда побочные эффекты даже не ограничены одной программой. Если программа создает новый файл, тем самым изменяются данные, потенциально доступные всем остальным программам в системе, а именно данные на жестком диске. Простая попытка сохранить данные на диске меняет состояние как всех компонентов той же программы, так и всех вообще программ в системе. Эту проблему обычно удается решить, разграничив доступ программ к файловой системе, например выделив каждой программе собственную директорию для хранения данных. Однако и в этом случае какая-то одна программа легко может заполнить все свободное место на диске, заблокировав другим программам возможность сохранять свои данные.

Таким образом, строго следуя запрету на какие бы то ни было изменения каких угодно состояний, мы не смогли бы сделать ничего полезного. Единственным видом программ, которые при этом можно создавать, были бы программы, которые вычисляют результат, исходя из аргументов, переданных им на вход. В программах стало бы невозможно взаимодействовать с пользователем, сохранять что-либо на диск или в базу данных, пересылать данные по сети и т. д. Наши программы получились бы практически бесполезными.

Поэтому, вместо того чтобы запрещать любые изменения состояния, подумаем, как разрабатывать программы таким образом, чтобы изменения состояния и побочные эффекты свести к минимуму. Однако сперва нужно лучше понять различие между чистыми и нечистыми функциями. В главе 1 говорилось, что чистые функции – это те, которые используют значения переданных им аргументов исключительно для вычисления возвращаемого результата. У них не должно быть побочных эффектов, влияющих на работу любых других функций в данной программе и других программ в пределах системы. Кроме того, чистые функции обязаны всегда возвращать один и тот же результат, если вызывать их многократно с одинаковыми аргументами.

Сейчас мы определим понятие *чистой функции* более строго – на основе понятия *референциальной прозрачности*. Референциальная прозрачность – это характеристика выражений, а не функций. Выражением будем называть любую синтаксическую конструкцию, которая описывает вычисление, возвращающее некоторый результат. Выражение будем называть референциально прозрачным, если поведение программы не меняется от замены всего этого выражения одним лишь его значением. Если выражение референциально прозрачно, у него нет *наблюдаемых побочных эффектов* и, следовательно, все входящие в него функции – чистые.

Поясним смысл сказанного примером. Пусть нужно написать простую функцию с одним аргументом – вектором целых чисел. Эта функция должна вернуть наибольшее из находящихся в контейнере значений. Для того чтобы позднее проверять правильность вычислений, будем писать диагностические сообщения в стандартный поток сообщений об ошибках `std::cerr`. Пусть затем эта функция несколько раз вызывается из функции `main`.

### Листинг 5.2 Поиск наибольшего значения в контейнере с выводом диагностических сообщений

```
double max(const std::vector<double>& numbers)
{
    assert(!numbers.empty());
    auto result = std::max_element(numbers.cbegin(),
                                   numbers.cend());
    std::cerr << "Maximum is: " << *result << std::endl;
    return *result;
}

int main()
{
    auto sum_max =
        max({1}) +
        max({1, 2}) +
        max({1, 2, 3});

    std::cout << sum_max << std::endl; // печатает 6
}
```

Предполагаем, что вектор `numbers` не пуст, и поэтому функция `std::max_element` возвращает валидный итератор.

Можно ли функцию `max` назвать чистой? Все ли выражения, в которых она используется, референциально прозрачны? Изменится ли поведение программы, если заменить все вызовы функции `max` возвращаемыми значениями?

### Листинг 5.3 Функция `main` после замены вызовов функции `max` возвращаемыми значениями

```
int main()
{
    auto sum_max =
```

```

1 + ←———— Результат вызова max({1})
2 + ←———— Результат вызова max({1, 2})
3;  ←———— Результат вызова max({1, 2, 3})

std::cout << sum_max << std::endl;
}

```

Функция `main`, как и прежде, вычисляет и выводит на печать значение 6. Однако в целом программа ведет себя не так, как прежде. Оригинальная версия программы делала больше этого: она не только выводила число 6 в поток `std::cout`, но перед этим еще и выводила числа 1, 2 и 3 (не обязательно в этом порядке) в поток `std::cerr`. Функция `max` не является ни референциально прозрачной, ни, следовательно, чистой.

Выше говорилось, что чистая функция использует для вычисления результата лишь свои аргументы. Функция `max`, конечно, использует свои аргументы в выражении, вычисляющем значение `sum_max`. Однако, помимо этого, она использует еще объект `std::cerr`, который не передается ей в качестве аргумента. Более того, функция не только использует поток `std::cerr`, но и меняет его состояние тем, что пишет в него данные.

Если же из функции `max` убрать строки, осуществляющие вывод в поток `std::cerr`, функция становится чистой:

```

double max(const std::vector<double>& numbers)
{
    auto result = std::max_element(numbers.cbegin(),
                                   numbers.cend());
    return *result;
}

```

Теперь функция `max` использует свой аргумент исключительно для того, чтобы вычислять свой результат, и делает это с помощью чистой функции `std::max_element`.

У нас уже есть уточненное определение чистой функции. Кроме того, появился точный смысл у фразы «отсутствие наблюдаемых побочных эффектов»: если вызов функции нельзя полностью заменить возвращаемым значением, не изменяя этим поведения программы, эта функция обладает *наблюдаемыми побочными эффектами*.

Из практических соображений, чтобы не приносить пользу в жертву теории, впредь будем пользоваться несколько ослабленной версией этого определения. Вряд ли было бы полезно отказываться от вывода диагностических сообщений только лишь для того, чтобы функцию можно было назвать *чистой*. Если все, что выводится в поток `std::cerr`, считать не слишком важным для функционирования программы, если пользователю нет дела то того, что программа пишет в этот поток, или если пользователь вообще этого потока не видит, функцию `max` можно считать чистой даже несмотря на то, что она, строго говоря, не удовлетворяет определению. Если вывод функции `max` заменить ее возвращаемым значением, единственное изменение в поведении программы будет состоять в том, что программа станет писать меньше сообщений

в поток `std::cerr`, который, однако, нам не важен; поэтому можно сказать, что поведение программы не изменилось.

### 5.3 Программирование без побочных эффектов

Мало заявить: «Перестаньте использовать изменяемое состояние – и обретете счастье навсегда», ведь отнюдь не очевидно, как этого добиться. Мы привыкли смотреть на программную систему (впрочем, и на мир вообще!) как на автомат, постоянно меняющий свое состояние. Хорошим примером может служить автомобиль. Допустим, дан автомобиль, который стоит с выключенным мотором. Если повернуть в замке ключ (или нажать кнопку запуска), автомобиль перейдет в новое состояние: теперь он *запускается*. Когда запуск будет выполнен до конца, автомобиль переходит в состояние «*работает*».

В чистом функциональном программировании, вместо того чтобы изменять значение, создают новое. Вместо того чтобы изменять некоторое свойство объекта, создается копия объекта, отличающаяся тем, что в ней изменено значение этого свойства. Если вся программная система разработана таким образом, проблемы, подобные тем, что были показаны в предыдущем примере, когда кто-то изменяет список оценок одновременно с обработкой этого списка, становятся невозможны. На этот раз никто не может изменить уже существующий список оценок; можно лишь создать новый список на основе имеющегося – возможно, добавив новые оценки.

Данный подход, на первый взгляд, противоречит нашей интуиции. Кажется, что он противоположен нашим представлениям о том, как работает мир. Может показаться удивительным, но идея, лежащая в основе такого подхода, десятилетиями занимала видное место на страницах научной фантастики и даже обсуждалась некоторыми древнегреческими философами. Довольно популярно представление о том, что мы живем в одном из множества параллельных миров и что всякий раз, когда мы делаем какой-то выбор, мы тем самым создаем новый мир, в котором сделали такой-то определенный выбор, а вместе с ним – и несколько параллельных миров, в которых сделан иной выбор.

Разрабатывая программу, мы обычно не задумываемся обо всех возможных мирах – нам бывает нужен только один из них. Именно поэтому мы склонны считать, что меняем единственный мир, а не постоянно создаем новые миры и отбрасываем старые. Это интересная модель бытия, и независимо от того, как мир устроен на самом деле, она может сослужить хорошую службу при разработке программ.

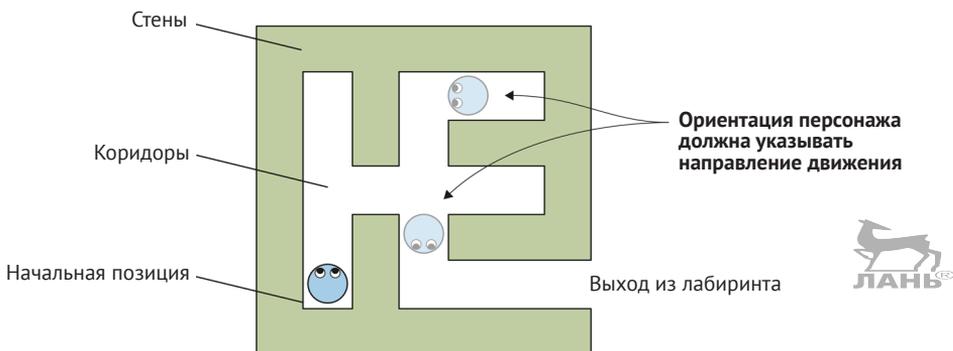
Покажем это на небольшом примере. Допустим, мы разрабатываем небольшую игру, в которой пользователю нужно выбраться из лабиринта, как показано на рис. 5.2. Лабиринт определяется квадратной матрицей, в каждой клетке которой может быть значение `Hallway` (коридор), `Wall` (стена), `Start` (начальное положение персонажа) или `Exit` (выход из лабиринта). Мы хотим, чтобы игра выглядела красиво, поэтому изобра-

жения персонажа должны различаться в зависимости от направления его движения.

Если бы мы собирались строить реализацию в императивном стиле, логика программы выглядела бы следующим образом (за исключением красивых графических эффектов: смены изображений персонажа и плавного перехода из клетки в клетку).

**Листинг 5.4 Перемещение по лабиринту**

```
while (1) {
    - отобразить лабиринт и игрока
    - считать пользовательский ввод
    - если персонаж должен перемещаться (если пользователь нажал клавишу со стрелкой), проверить, находится ли в следующей клетке стена, если нет, переместить персонажа в нее
    - проверить, достигнут ли выход, и если достигнут, показать сообщение и выйти из цикла
}
```



**Рис. 5.2** Лабиринт определяется матрицей, каждый элемент которой может быть стеной или коридором. Персонаж может двигаться по коридорам, но не сквозь стены

Это прекрасный пример задачи, для решения которой может пригодиться изменяемое состояние. Есть лабиринт и персонаж, и персонажа нужно перемещать по лабиринту, тем самым меняя состояние игры. Посмотрим, как моделировать это в отсутствие изменяемых состояний.

Во-первых, заметим, что лабиринт никогда не изменяется, и единственный элемент игры, состояние которого изменяется, – это игровой персонаж. Поэтому нужно сфокусировать внимание на нем, а лабиринт легко сделать неизменяемым объектом.

На каждом шаге игры нужно перемещать персонажа в новую позицию. Какие данные нужны для вычисления новой позиции? Нетрудно догадаться, что перечень таков:

- направление движения;
- предыдущая позиция;



- данные о том, можно ли перемещаться из текущей позиции в заданном направлении.

Нужно создать функцию, которая принимает эти три аргумента и возвращает новую позицию.

### Листинг 5.5 Функция для вычисления новой позиции в игре

```

position_t next_position(
    direction_t direction,
    const position_t& previous_position,
    const maze_t& maze
)
{
    const position_t desired_position{previous_position, direction};

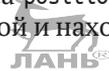
    return maze.is_wall(desired_position) ? previous_position
        : desired_position;
}

```

Вычислить желаемую позицию – даже если там может оказаться стена

Если в желаемой позиции нет стены, вернуть ее. Иначе вернуть старую позицию

Таким образом, мы реализовали логику перемещения персонажа без каких бы то ни было изменяемых состояний. Чтобы пример стал полным, нужно определить конструктор класса `position_t`, который вычисляет координаты клетки, соседней с заданной и находящейся от нее в заданном направлении.



### Листинг 5.6 Вычисление координат соседней клетки

```

position_t::position_t(const position_t& original,
    direction_t direction)
    : x { direction == Left ? original.x - 1 :
        direction == Right ? original.x + 1 :
        original.x }
    , y { direction == Up ? original.y + 1 :
        direction == Down ? original.y - 1 :
        original.y }
{
}

```

Тернарная операция использована для разбора возможных направлений движения для присваивания правильных начальных значений координатам `x` и `y`. Можно было бы также использовать оператор `switch` в теле конструктора

Теперь, когда мы разобрались с логикой, нужно разобраться с тем, как показывать направление движения персонажа. Нужно сделать свой выбор между двумя вариантами: либо поворачивать изображение персонажа после того, как он успешно переместился в другую ячейку, либо менять ориентацию изображения даже в том случае, если персонажу не удалось переместиться, – тем самым показывая пользователю, что персонаж понял команду пользователя, но не может ее выполнить, так как уткнулся носом в стену. Этот вопрос относится к уровню отображения и не составляет неотъемлемую часть логики функционирования персонажа. Оба указанных выше варианта имеют право на существование.

Слой программы, отвечающий за отображение, должен рисовать на экране лабиринт и игрока в нем. Поскольку лабиринт неизменен, его

отрисовка не меняет состояния игры. Функция отображения персонажа также не меняет его состояния. Функции нужно знать лишь позицию игрока и направление, куда он повернут лицом:

```
void draw_player(const position_t& position,
                direction_t direction)
```



Как говорилось выше, есть два способа показывать ориентацию персонажа в пространстве. При одном способе ориентация персонажа зависит от того, удалось ли ему совершить перемещение. В этом случае ориентацию легко вычислить, потому что при каждом ходе создается новая текущая позиция (экземпляр типа `position_t`), а предыдущая позиция остается неизменной. Второй вариант, когда ориентация персонажа в пространстве меняется даже тогда, когда ему не удастся совершить перемещение, тоже легко воплотить, каждый раз поворачивая персонажа в том же направлении, которое передается в функцию `next_position`. Можно пойти дальше, например использовать для отображения персонажа различные картинки в зависимости от текущего времени или от времени, прошедшего после предыдущего хода, и т. д.

Теперь нужно связать воедино все эти элементы. В демонстрационных целях цикл обработки событий реализуем рекурсивно. В главе 2 говорилось, что программисту не следует злоупотреблять рекурсией, поскольку нет гарантий, что компилятор сумеет оптимизировать программу. В этом примере рекурсия используется исключительно для демонстрации того, что реализация программы в целом (а не только отдельных ее частей) может быть чистой – не допускающей изменения каких-либо объектов после того, как они созданы.

**Листинг 5.7 Рекурсивная реализация цикла обработки событий**

```
void process_events(const maze_t& maze,
                  const position_t& current_position)
{
    if (maze.is_exit(current_position)) {
        // show message and exit
        return;
    }

    const direction_t direction = ...;

    draw_maze();
    draw_player(current_position, direction);

    const auto new_position = next_position(
        direction,
        current_position,
        maze);

    process_events(maze, new_position);
}
```

Вычислить направление, исходя из пользовательского ввода

Отобразить лабиринт и персонажа

Вычислить новую позицию

Продолжить обработку событий – однако теперь уже с новой позицией персонажа

```
int main()
{
    const maze_t maze("maze.data");
    process_events(
        maze,
        maze.start_position());
}
```

Функция main должна всего лишь прочитать лабиринт из файла и вызвать функцию process\_events, передав ей начальную позицию персонажа

На этом примере показан общий способ моделирования изменяемых состояний при отсутствии в программе модифицируемых объектов и наличии только лишь чистых функций. Реальные программы обычно бывают устроены гораздо сложнее. В данном примере единственный меняющийся параметр – это позиция игрока на карте. Все остальное, включая способ отображения персонажа на экране, можно вычислить, исходя из этой позиции.

В больших системах много «движущихся частей». Программист мог бы создать громадную, всеохватывающую структуру данных под названием «мир» и создавать новый ее экземпляр всякий раз, когда в ней нужно что-то изменить. Это повлекло бы за собой большие накладные расходы (даже если воспользоваться специальными структурами данных, оптимизированными для функционального программирования, о которых речь пойдет в главе 8) и сильно усложнило бы устройство системы.

Вместо этого при создании программных систем в функциональном стиле обычно делают какое-то изменяемое состояние, которое, скорее всего, было бы накладно постоянно копировать и передавать в каждую функцию; функции же в этом случае возвращают описания того, что должно быть изменено в этом состоянии, – вместо того чтобы менять само состояние или возвращать его новую копию. Главная выгода, которую приносит этот подход, – возможность четко отделить чистые части системы от тех, что обладают изменяемым состоянием.

## 5.4 Изменяемые и неизменяемые состояния в параллельных системах

Большинство современных программных систем в той или иной степени параллельны – это может быть выполнение сложных вычислений наподобие обработки графического изображения в несколько параллельных потоков с целью ускорить решение задачи или одновременное выполнение нескольких пользовательских задач, как в случае веб-браузера, способного загружать файлы, пока пользователь просматривает страницу.

Наличие в программе изменяемого состояния может привести к проблемам, потому что когда каждый из нескольких потоков отвечает за корректность состояния, на самом деле за нее не отвечает никто. Подобное размытие ответственности противоречит даже принципам ООП.

Простейший пример для демонстрации этой проблемы – это два параллельных потока, пытающихся одновременно менять значение одной

и той же целочисленной переменной. Предположим, цель состоит в том, чтобы определять число клиентов, одновременно подключенных к серверу, – чтобы при переходе в энергосберегающий режим проверять, все ли клиенты отсоединились. Пусть в начальный момент времени есть два подключенных клиента, и пусть оба отключаются одновременно. За обработку отключения клиента отвечает следующая функция:

```
void client_disconnected(const client_t& client)
{
    // Освободить ресурсы, которые использовал клиент
    ...

    // Уменьшить счетчик подключенных клиентов
    --connected_clients;

    if (connected_clients == 0) {
        // go to the power-save mode
    }
}
```

Обратим внимание на строку, которая уменьшает на единицу счетчик подключенных клиентов. Она выглядит вполне невинно. В начальный момент времени значение переменной `connected_clients` должно равняться двум, так как к серверу подключены два клиента. Когда один из них отсоединится, значение переменной `connected_clients` должно стать равным единице, а когда отключится и второй, счетчик достигнет нуля, и сервер перейдет в энергосберегающий режим. Проблема заключается в том, что этот оператор, выглядящий столь просто для человека, отнюдь не прост для машины. В этой строчке кода компьютеру необходимо выполнить следующее:

- 1 Получить значение счетчика из памяти и поместить его в регистр процессора.
- 2 Уменьшить значение в регистре.
- 3 Отправить новое значение из регистра в оперативную память.

С точки зрения процессора это три отдельных действия, ему невдомек, что в коде они обозначаются одной строкой и с точки зрения человека составляют единую операцию. Если функция `client_disconnected` вызывается в один и тот же момент времени из двух различных потоков, выполнение этих трех действий одним потоком может в каком угодно порядке чередоваться с выполнением тех же действий другим потоком. В частности, один вызов может получить значение счетчика из оперативной памяти в регистр до того, как другой поток закончит менять это значение. В этом случае каждый поток уменьшит значение счетчика с 2 до 1, и поэтому счетчик никогда не достигнет нуля, см. рис. 5.3.

Если проблемы могут возникнуть при одновременном доступе потоков всего лишь к одной целочисленной переменной, легко представить, насколько серьезнее обстоят дела с более сложными структурами данных. К счастью, универсальное решение для всех подобных проблем хо-

рошо известно. Программисты давно поняли, что возможность вносить в данные несколько одновременных изменений ведет к ошибкам. Решение очевидно: программист должен запретить одновременный доступ к данным посредством *семафоров* (англ. mutex).

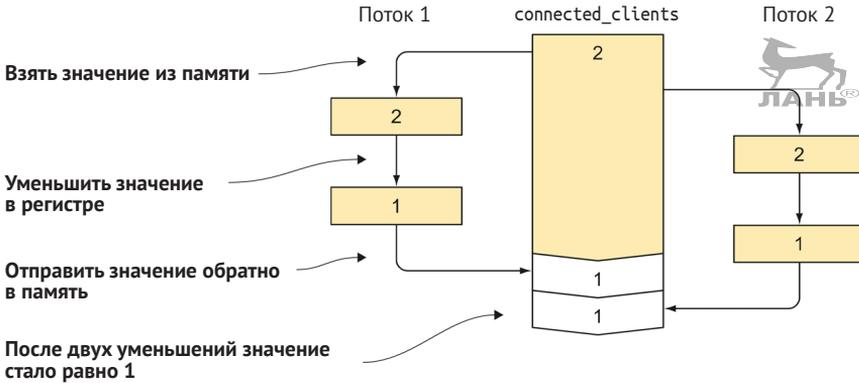


Рис. 5.3 Два параллельных потока хотят уменьшить одну и ту же переменную на единицу. Каждому потоку нужно получить текущее значение переменной из памяти, уменьшить его и записать новое значение в память. Поскольку начальное значение счетчика равно двум, можно ожидать, что в конце оно станет равным нулю. Однако если второй процесс прочитает из памяти текущее значение счетчика до того, как первый запишет туда новое значение, результат может оказаться неожиданным

Проблема этого подхода, в свою очередь, заключается в том, что на самом деле мы хотим параллельного выполнения нескольких потоков – ради эффективности или для каких-то иных целей. А семафоры решают проблему одновременного доступа, устраняя параллельность.

*Я часто говорил в шутку, что, вместо того чтобы подхватывать изобретенное Дейкстрой остроумное сокращение (mutex означает «mutual exclusion» – взаимное исключение), нам следовало бы назвать основной объект синхронизации «бутылочным горлышком». Бутылочные горлышки синхронизации бывают нужны, а иногда и просто незаменимы – но они никогда не бывают хорошим решением. В лучшем случае они – неизбежное зло. Все, абсолютно все, что побуждает кого бы то ни было чрезмерно использовать их или удерживать их слишком долго, идет во вред. И дело здесь не только в очевидном ударе по производительности из-за лишней операции захвата и освобождения семафора – дело в гораздо более глубоком и всеохватывающем, хоть и трудноуловимом, ударе по общей степени параллельности приложения.*

Дэвид Батенхоф,  
из сообщения в группе comp.programming.threads



Семафоры иногда бывают необходимы<sup>1</sup>, но их следует использовать как можно реже, и никогда – в качестве оправдания для плохо продуманной архитектуры программы. Семафоры, подобно циклам `for` и рекурсиям, представляют собой примитивы низкого уровня и полезны главным образом для реализации более высокоуровневых абстракций, но никак не для прямого использования в коде приложения. Может показаться странным, но в соответствии с законом Амдала если лишь 5 % кода требуют синхронизированного выполнения, а 95 % допускают полностью параллельное выполнение, то теоретический максимум выигрыша в скорости, которого можно ожидать по сравнению с выполнением всего кода в один поток, – это ускорение в 20 раз, сколько бы тысяч процессоров ни задействовать для выполнения этого кода (рис. 5.4).

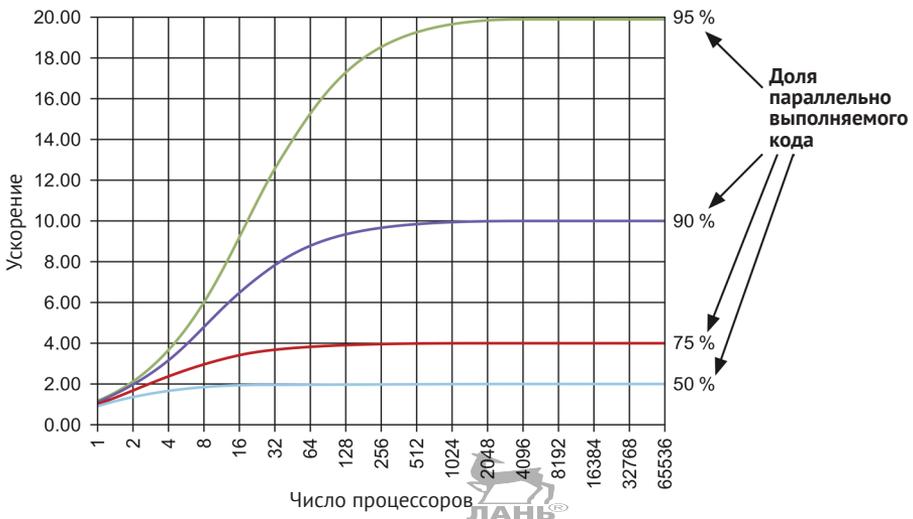


Рис. 5.4 Было бы идеально, если бы с увеличением числа процессоров во столько же раз возрастала бы общая скорость вычислений. К сожалению, скорость с ростом числа процессоров растет нелинейно. Согласно закону Амдала, если лишь 5 % кода требуют последовательного выполнения (посредством семафоров или иных примитивов синхронизации), максимально возможный выигрыш в скорости, по сравнению с выполнением всего кода в один поток, составляет 20 раз – даже если задействовать более 60 тысяч процессоров

Если мы не хотим допустить снижения степени параллельности, доступной нашему коду, нужно искать иные, помимо семафоров, решения для проблемы параллельного доступа к разделяемому состоянию. Выше говорилось, что проблема возникает только тогда, когда несколько па-

<sup>1</sup> Читателю стоит порекомендовать книгу Энтони Вильямса «Параллельное программирование на языке C++ в действии» (*C++ Concurrency in Action* by Anthony Williams), где можно найти подробности о параллельном программировании, семафорах и о традиционных подходах к написанию многопоточного кода. Методы создания параллельных и асинхронных программ, характерные для функционального программирования, будут рассмотрены нами в главах 10 и 12.

параллельных потоков совместно владеют одним изменяемым объектом данных. Поэтому если одно возможное решение состоит в том, чтобы отказаться от параллельного доступа к этим данным, то другое – отказаться от изменяемости данных. Впрочем, есть и третий вариант: оставить данные изменяемыми, но отказаться от совместного доступа к ним из различных потоков. Если поток ни с кем не делится доступом к данным, он никогда не столкнется с тем, что какой-то другой поток изменит эти данные без его ведома.

Итак, всего есть четыре варианта: неизменяемые данные с исключительным доступом; изменяемые данные с исключительным доступом; неизменяемые данные с совместным доступом; изменяемые данные с совместным доступом. Из всех четырех лишь последний случай чреват проблемами (рис. 5.5). Тем не менее похоже, что около 99 % всех программ попадают именно в эту категорию.



**Рис. 5.5** Данные могут быть изменяемыми и неизменяемыми и находиться в совместном или исключительном доступе. Единственный случай, приводящий к проблемам в многопоточной среде, – это изменяемые данные с совместным доступом. Остальные три случая вполне безопасны

Главы 10 и 12 посвящены дальнейшему развитию этой темы. В них рассматриваются специфические для функциональной парадигмы абстракции, призванные сделать параллельное и многопоточное программирование – как с изменяемым состоянием, так и без него – почти столь же простым, как и обычное функциональное программирование.

## 5.5 О важности констант

Теперь, когда читатель знаком с некоторыми из тех проблем, что возникают при наличии изменяемого состояния, рассмотрим встроенные в язык C++ средства, помогающие разрабатывать код, не обладающий изменяемым состоянием. В языке C++ есть два способа ограничить изменение данных: это спецификатор `const` и появившийся в стандарте C++11 спецификатор `constexpr`.

Смысл спецификатора `const` в том, чтобы предотвратить изменение данных. В языке C++ это достигается с помощью системы типов. Если объект некоторого типа `T` хочется сделать неизменяемым, его тип нужно объявить как `const T`. Это простое объявление влечет далеко идущие следствия. Допустим, мы объявили переменную `name` типа `std::string` неизменяемой и хотим из нее присвоить значения другим переменным:

```
const std::string name{"John Smith"};

std::string name_copy          = name;
std::string& name_ref          = name; // ошибка
const std::string& name_constref = name;
std::string* name_ptr         = &name; // ошибка
const std::string* name_constptr = &name;
```



Первый пример присваивает значение константной строки `name` в неконстантную переменную `name_copy`. Это разрешено: создается новый объект строкового типа, и в него копируется содержимое объекта `name`. Если в дальнейшем будет меняться состояние объекта `name_copy`, на объект-оригинал это никак не повлияет, поэтому его константность не нарушена.

Во втором примере делается попытка создать переменную `name_ref` типа ссылки на строку, так чтобы она ссылалась на переменную `name`. Компилятор выдаст ошибку, поскольку объявлена ссылка на объект типа `std::string`, тогда как справа от знака равенства стоит объект другого типа: `const std::string` (и эти типы не связаны между собой наследованием, которое могло бы сделать допустимым приведение типа).

Подобную ошибку получим и при попытке присвоить адрес переменной `name` в переменную `name_ptr`, которая, согласно объявлению, должна быть указателем на неконстантную строку. Такое поведение компилятора совершенно правильно, так как в противном случае последующий код мог бы менять состояние объекта, на который ссылается переменная `name_ref` или указывает переменная `name_ptr`, то есть объекта `name`, который, согласно объявлению, должен оставаться неизменным. Компилятор остановил нас еще до того, как мы попытались изменить объект `name`.

В следующем примере объявляется ссылка на объект типа `const std::string`, а в правой части стоит переменная в точности этого типа. Эту строчку компилятор примет без ошибки. Также допустимо присваивание адреса переменной `name` переменной, чей тип объявлен как указатель на строку-константу.

### Особенность константных ссылок

То, что компилятор разрешает создавать ссылки типа `const T`, инициализируя их переменными того же самого типа, вполне ожидаемо. Помимо этого, константные ссылки (т. е. ссылки на объекты константных типов) обладают еще одним свойством. Такие ссылки можно связывать не только с переменными типа `const T`, но и с временными значениями, продлевая тем самым время их жизни. Более подробные сведения о времени жизни временных объектов можно найти на странице <http://mng.bz/o19v>.

Что произойдет, если попытаться эту строку-константу передать в функцию в качестве аргумента? Пусть функция `print_name` принимает параметр типа строки, функция `print_name_ref` – типа ссылки на строку, а функция `print_name_constref` принимает константную ссылку на строку:

```
void print_name(std::string name_copy);
void print_name_ref(std::string& name_ref); // вызов невозможен
void print_name_constref(const std::string& name_constref);
```

Как и в предыдущих примерах, совершенно нормально значение строковой константы присвоить параметру строкового типа (при этом будет создана копия строки-оригинала, сама по себе неконстантная), а также инициализировать параметр типа константной ссылки на строку так, чтобы он ссылался на константную строковую переменную, поскольку константность объекта `name` при этом сохраняется. Единственное, чего сделать нельзя, – это передать константный объект по неконстантной ссылке или указателю.

Таким образом, мы разобрали, как спецификатор `const` работает для присваиваний и вызовов функций. Теперь посмотрим, что он означает в применении к функциям – членам класса. Возьмем наш обычный пример – класс `person_t` – и проверим, как правила языка C++ гарантируют неизменность объекта. Пусть объявлены следующие функции-члены:

```
class person_t {
public:
    std::string name_nonconst();
    std::string name_const() const;

    void print_nonconst(std::ostream& out);
    void print_const(std::ostream& out) const;

    ...
};
```

За объявлениями функций-членов компилятор языка C++ видит обычные функции, обладающие дополнительным неявным аргументом – указателем `this`. Квалификатор `const` в применении к функции-члену означает не что иное, как константность типа объекта, на который указывает указатель `this`. Поэтому объявленные выше функции-члены изнутри выглядят следующим образом (в действительности `this` – это указатель, но ради ясности представим себе, будто это ссылка):

```
std::string person_t_name_nonconst(
    person_t& this
);
std::string person_t_name_const(
    const person_t& this
);
void person_t_print_nonconst(
    person_t& this,
    std::ostream& out
);
```

```
void person_t_print_const(  
    const person_t& this,  
    std::ostream& out  
);
```

Если рассматривать функции-члены как обычные функции с дополнительным неявным параметром `this`, смысл вызова таких функций на константных и неконстантных объектах становится очевидным – в точности таким, как показано в предыдущем примере, где мы пытались передавать в обычные функции в качестве аргументов константные и неконстантные объекты. Попытка вызвать неконстантную функцию-член на константном объекте приводит к ошибке компиляции потому, что при этом фактически делается попытка передать константный объект по неконстантной ссылке.

### Почему `this` – именно указатель?

Ключевое слово `this` означает указатель-константу на (возможно, неконстантный) экземпляр класса, в котором эта функция объявлена. В соответствии с правилами языка этот указатель никогда не должен быть нулевым (хотя в действительности некоторые компиляторы допускают вызов не виртуальных методов на нулевых указателях).

Вопрос состоит в следующем: если указатель `this` нельзя изменить так, чтобы он указывал на другой объект, и если он не может быть нулевым, почему он сделан именно указателем, а не ссылкой? Ответ прост: когда в языке C++ появился указатель `this`, в нем еще отсутствовало понятие ссылки. С появлением ссылок менять семантику слова `this` было уже поздно, так как это нарушило бы обратную совместимость с предыдущими версиями языка и сделало бы непригодным к использованию ранее созданный код.

Читатель мог убедиться, что механизм, лежащий в основе квалификатора `const`, прост, но работает на удивление хорошо. Используя слово `const`, программист сообщает компилятору, что значение той или иной переменной не должно изменяться; компилятор же отвечает ошибкой на каждую попытку программиста изменить такую переменную.

## 5.5.1 Логическая и внутренняя константность

Как говорилось выше, неизменяемые типы данных нужны для того, чтобы избежать проблем, связанных с изменяемым состоянием. Возможно, самый простой способ добиться этого на языке C++ состоит в том, чтобы при создании классов объявлять все их члены константными. Например, класс `person_t` мог бы быть реализован следующим образом:

```
class person_t {  
public:  
    const std::string name;  
    const std::string surname;  
    ...  
};
```

Не нужно даже определять методы для доступа к членам-данным, их можно сделать открытыми. У этого подхода есть, однако, свои недостатки: компилятор не может применять некоторые оптимизации. Едва в классе появляются константные данные, становится невозможным перемещающий конструктор и перемещающая операция присваивания (более подробные сведения о семантике перемещения можно найти на странице <http://mng.bz/JULm>).

Поэтому воспользуемся иным подходом. Вместо того чтобы объявлять константами все переменные-члены, сделаем константными все открытые функции-члены:

```
class person_t {
public:
    std::string name() const;
    std::string surname() const;

private:
    std::string m_name;
    std::string m_surname;

    ...
};
```



Теперь, хотя члены-данные не объявлены константами, пользователи этого класса не могут изменить их, так как указатель `this` во всех доступных ему методах указывает на объект типа `const person_t`. Этим обеспечивается как *логическая константность* (т. е. неизменность данных, видимых пользователю), так и *внутренняя константность* (т. е. невозможность изменения внутренних данных объекта). В то же время не теряется возможность оптимизации кода компилятором: всюду, где это нужно, компилятор сможет сгенерировать операции перемещения.

Последняя оставшаяся трудность состоит в том, что иногда бывает нужно изменить внутренние данные объекта, но таким образом, чтобы изменения остались невидимыми для пользователя. Это может понадобиться, например, для того, чтобы сохранить для последующего быстрого доступа результат долгой операции.

Допустим, нужно реализовать метод `employment_history`, который возвращает список всех предыдущих мест работы данного человека. Этот метод должен выполнить запрос к базе данных. Поскольку все функции-члены объявлены с квалификатором `const`, единственным возможным местом для инициализации переменных-членов остается конструктор класса `person_t`. Запросы к базе данных не относятся к числу быстрых операций, поэтому не хотелось бы заново выполнять один и тот же запрос каждый раз, когда вызывается метод `employment_history`. Это нежелательно не только по причинам, связанным с производительностью, но и потому, что функция могла бы возвращать каждый раз разные результаты (если меняется состояние базы данных), а это нарушило бы наше обещание, что объект типа `person_t` должен оставаться неизменным.



С этой трудностью легко справиться, если объявить в классе переменную-член с квалификатором `mutable` (англ. «изменяемый») – такие переменные разрешается менять даже из константных методов. Если мы хотим гарантировать неизменность объектов нашего класса с точки зрения пользователя, нужно сделать так, чтобы никакие два параллельных вызова функции `employment_history` не могли вернуть различные значения. Иными словами, второй вызов не должен выполняться до тех пор, пока первый не получит данные из базы и не сохранит их в объекте.

### Листинг 5.8 Использование изменяемых членов для буферизации данных

```
class person_t {
public:
    employment_history_t employment_history() const
    {
        std::unique_lock<std::mutex>
            lock{m_employment_history_mutex};

        if (!m_employment_history.loaded()) {
            load_employment_history();
        }

        return m_employment_history;
    }
private:
    mutable std::mutex m_employment_history_mutex;
    mutable employment_history_t m_employment_history;
    ...
};
```

Закрывание семафора гарантирует, что остальные параллельные вызовы функции `employment_history` не будут выполняться до тех пор, пока не закончена выборка данных из базы

Получить данные, если они не были получены ранее

Данные получены, можно вернуть их

При выходе из этого блока автоматически уничтожается объект `lock`, что приводит к открытию семафора

Чтобы запереть семафор из константной функции, его нужно объявить изменяемым; то же верно и для переменной, в которой сохраняются полученные из базы данные

Прием, когда классы выглядят неизменными со стороны пользователя, но иногда меняют внутреннее состояние своих объектов, довольно распространен. От константных функций-членов требуется, чтобы они либо сохраняли данные неизменными, либо выполняли все изменения с синхронизацией, дабы эти изменения не могли стать видимыми даже из параллельных вызовов.

## 5.5.2 Оптимизированные функции-члены для временных объектов

Если класс разрабатывается неизменяемым, всякий раз, когда в нем нужно определить модифицирующий метод, приходится оформлять его в виде функции, которая возвращает копию объекта, изменив в ней значение того или иного поля. Создание копий с целью получить модифицированную версию объекта неэффективно, хотя в некоторых случаях компилятор и может оптимизировать этот процесс. Это особенно важно в тех случаях, когда объект-оригинал после модификации более не нужен.



Допустим, дан экземпляр типа `person_t`, и нужно построить его обновленную версию с измененным именем и фамилией. Для этого нужно было бы написать что-то наподобие

```
person_t new_person {
    old_person.with_name("Joanne")
                .with_surname("Jones")
};
```

Функция `.with_name` возвращает новый экземпляр типа `person_t`, во всем совпадающий со старым, за исключением нового имени `Joanne`. Поскольку этот объект не присваивается какой-либо переменной, он будет уничтожен сразу же, как только завершится вычисление всего выражения. Он используется только методом `.with_surname`, который создает еще один экземпляр типа `person_t`, отличающийся от оригинала как именем `Joanne`, так и фамилией `Jones`.

Таким образом, здесь создаются два экземпляра типа `person_t`, а все содержащиеся в объекте-оригинале данные копируются дважды, несмотря на то что пользователю нужно создание лишь одного экземпляра – того, который будет присвоен переменной `new_person`. Хорошо было бы избежать лишнего копирования, распознав, что метод `.with_surname` вызывается на временном объекте и поэтому нет нужды создавать копию – вместо этого данные можно переместить в объект-результат. К счастью, этого можно добиться, создав по две реализации для каждого из методов `.with_name` и `.with_surname`: одну для работы с «настоящими» объектами и другую – для временных объектов.



### Листинг 5.9 Раздельные методы для работы с долгоживущими и временными объектами

```
class person_t {
public:
    person_t with_name(const std::string& name) const & ← Эта функция-член будет вызываться
    {                                                    для обычных объектов и ссылок lvalue
        person_t result(*this); ← Создать копию объекта
        result.m_name = name; ← Изменить в этой копии имя. Это корректно, поскольку
        return result; ← Вернуть вновь созданный экземпляр типа person_t. ← данный экземпляр типа person_t пока не виден извне
    }                                                    Отныне он становится неизменяемым

    person_t with_name(const std::string& name) && ← Этот метод будет вызываться
    {                                                    только для временных
        person_t result(std::move(*this)); ← Вместо конструктора копирования
        result.m_name = name; ← Изменить имя в только что созданном объекте ← вызывается конструктор перемещения
        return result; ← Вернуть вновь созданный объект
    }
};
```

Для функции `.with_name` объявлены две перегруженные реализации. Как и ранее в случае квалификатора `const`, квалификатор вида ссылки влияет только на тип указателя `this`. Вторая перегруженная версия будет вызываться лишь для таких объектов, из которых разрешается забирать данные: временных объектов и других ссылок `rvalue`.

Первая перегруженная версия создает новую копию объекта типа `person_t`, на который указывает указатель `this`, затем устанавливает в объекте-копии новое значение имени и возвращает этот объект. Во второй перегруженной версии новый экземпляр типа `person_t` создается не копированием, а перемещением данных из объекта, на который указывает указатель `this`.

Обратим внимание, что вторая версия функции объявлена без квалификатора `const`. В противном случае было бы невозможно забрать данные из старого экземпляра в новый, тогда пришлось бы данные копировать, как и в предыдущем случае.

Таким образом, читатель познакомился с одним из возможных способов оптимизировать код, предотвращая копирование временных объектов. Как и другие способы оптимизации, этот стоит применять только тогда, когда он приводит к заметному улучшению. Следует избегать преждевременной оптимизации, а перед тем как браться за нее, измерять производительность кода.



### 5.5.3 Недостатки константных объектов

Ключевое слово `const` – одно из самых полезных в языке C++, и при разработке своих классов им стоит пользоваться как можно больше, даже если разработка ведется не в функциональном стиле. Как и вся система типов, оно позволяет обнаруживать на этапе компиляции множество распространенных ошибок.

Однако использование констант – не всегда веселая забава. Иногда при этом можно столкнуться с неудобствами.

#### КОНСТАНТНОСТЬ ДЕЛАЕТ НЕВОЗМОЖНЫМ ПЕРЕМЕЩЕНИЕ ОБЪЕКТА

При написании функции, возвращающей объект определенного типа (не ссылку или указатель на объект), часто объявляют локальную переменную этого типа, что-то с ней делают и возвращают ее значение. Именно так были реализованы обе перегруженные версии функции `.with_name`. Выделим суть этой идиомы:

```
person_t some_function()
{
    person_t result;

    // do something before returning the result

    return result;
}

...

person_t person = some_function();
```

Если бы компилятор не выполнял никаких оптимизаций при обработке этого кода, функция создала бы новый локальный экземпляр класса `person_t` и вернула бы его вызывающему коду. Далее этот экземпляр был бы передан конструктору копирования или перемещения, чтобы инициализировать переменную `person`. После копирования (или перемещения) тот экземпляр, который вернула функция `some_function`, удаляется. Как и в предыдущем примере с расположенными подряд вызовами функций `.with_name` и `.with_surname`, здесь создаются два экземпляра класса `person_t`, один из которых временный – он удаляется сразу после использования.

К счастью, современные компиляторы умеют оптимизировать этот процесс так, чтобы функция `some_function` конструировала свой локальный объект `result` непосредственно в адресном пространстве, зарезервированном вызывающей стороной для переменной `person`, тем самым устраняя необходимость в дополнительных операциях копирования и перемещения. Эта оптимизация известна под названием «оптимизация именованного возвращаемого значения» (*named return value optimization*, NRVO).

Это один из тех случаев, когда использование квалификатора `const` может принести вред. Когда компилятор не в состоянии выполнить оптимизацию NRVO, возврат константного объекта приводит к необходимости копирования, так как применить к нему операцию перемещения невозможно.

### ПОВЕРХНОСТНАЯ КОНСТАНТНОСТЬ

Еще одна проблема состоит в том, что спецификатор `const` бывает легко обойти. Рассмотрим следующий пример: пусть есть класс `company_t`, моделирующий компанию, в котором содержится вектор указателей на всех сотрудников. Пусть в этом классе определена константная функция-член, возвращающая вектор имен всех сотрудников:

```
class company_t {
public:
    std::vector<std::string> employees_names() const;

private:
    std::vector<person_t*> m_employees;
};
```

Компилятор не позволит вызывать любые неконстантные функции-члены данного класса или применять неконстантные функции к члену `m_employees` в теле функции `employees_names`, поскольку она объявлена константной. Таким образом, этой функции-члену запрещено менять любые члены-данные в текущем экземпляре класса `company_t`. Но являются ли экземпляры типа `person_t` данными-членами для экземпляра типа `company_t`? Нет: таковыми являются только хранящиеся в векторе указатели на них. В функции `employees_names` не разрешается менять сами указатели, но объекты, на которые они указывают, изменять можно.

Положение получается запутанным, и оно может привести к неприятностям, если класс `person_t` не сделать неизменяемым. Если где-то в программе есть функции, изменяющие состояние объектов типа `person_t`, их разрешается вызывать из функции `employees_names`. Было бы хорошо, если бы компилятор мог защитить программиста от попытки их вызвать, потому что квалификатор `const` должен служить обещанием, что в объекте ничего не изменяется.

Ключевое слово `const` дает программисту возможность выразить свое намерение, что объект не должен подвергаться изменениям, и поручает компилятору следить за соблюдением этого ограничения. Писать это слово в каждом объявлении утомительно, но если дать себе такой труд, то всякий раз, увидев переменную без такого квалификатора, можно быть *уверенным*, что ее значение *должно* изменяться. В этом случае, перед тем как использовать переменную, нужно проверить в коде все места, где она могла бы быть изменена. Если же переменная объявлена со словом `const`, достаточно посмотреть лишь на ее объявление, чтобы при использовании знать ее точное значение.

### Обертка `propagate_const`

Решить описанную выше проблему можно с помощью обертки над указателями и подобными им объектами, называемой `propagate_const`. В настоящее время она опубликована в виде технической спецификации «Library Fundamentals Technical Specification» и должна войти в будущий стандарт языка C++.

Если компилятор и стандартная библиотека поддерживают экспериментальные возможности, обертку `propagate_const` можно найти в заголовочном файле `<experimental/propagate_const>`. Как и у всего, что объявлено в пространстве имен `experimental::`, интерфейс и поведение этой обертки могут в будущем измениться, поэтому пользоваться такими средствами следует с осторожностью.

Применять класс `propagate_const` довольно просто: у всех полей класса, чьи типы – это указатели или иные типы, имитирующие поведение указателей (скажем, умные указатели), нужно заменить тип следующим образом: `std::experimental::propagate_const<T*>`. Теперь всякий раз, когда такое поле используется из константного метода, оно с точки зрения компилятора будет вести себя как указатель на объект типа `const T`.

**СОВЕТ** Более подробную информацию и ссылки по темам, затронутым в этой главе, можно найти на странице <https://forums.man-ning.com/posts/list/41684.page>.

## Итоги

- Современные компьютеры обладают, как правило, несколькими вычислительными ядрами. Разрабатывать программы нужно так, чтобы они наверняка корректно работали в многопоточной среде.



- Злоупотребляя примитивами синхронизации, программист ограничивает наибольшую степень параллельности, достижимую для программы. Из-за накладных расходов на синхронизацию совокупная скорость программы не может расти линейно с ростом числа процессоров.
- Изменяемое состояние – это не всегда плохо. Держать в программе такое состояние безопасно – если только не допускать совместного доступа к этому состоянию из различных компонентов системы.
- Если функция-член класса объявляется с квалификатором `const`, это означает обещание, что эта функция не изменяет никаких членов-данных объекта или же (если некоторые данные-члены объявлены с ключевым словом `mutable`) что все изменения выглядят для пользователя атомарными.
- Копирование структуры данных целиком, для того чтобы изменить в ней лишь одно поле, неэффективно. Для этого можно пользоваться специальными неизменяемыми структурами данных, о которых будет подробнее рассказано в главе 8.



---

# Ленивые вычисления

---



## О чем говорится в этой главе:

- вычисление значений по мере необходимости;
- запоминание значений чистых функций для последующего использования;
- модифицированный алгоритм быстрой сортировки, обрабатывающий лишь фрагменты коллекции;
- использование шаблонов выражений для ленивого вычисления;
- работа с бесконечными или почти бесконечными структурами данных.



Вычисления требуют времени. Допустим, даны две матрицы,  $A$  и  $B$ , и известно, что когда-то в будущем может понадобиться их произведение. Одно из возможных решений состоит в том, чтобы сразу же вычислить это произведение:

```
auto P = A * B;
```

Недостаток здесь состоит в том, что произведение матриц может так и не понадобиться – и драгоценные такты центрального процессора окажутся потраченными зря.

Другой подход состоит в том, чтобы запомнить: если когда-либо понадобится, матрицу  $P$  можно вычислить как  $A * B$ . Вместо того чтобы выполнять вычисление, его можно просто определить. Лишь когда в каком-то месте программы понадобится значение  $P$ , оно будет вычислено, но не ранее.

Вычисления обычно определяют, создавая функции. Вместо того чтобы хранить в переменной *P* произведение матриц *A* и *B*, можно превратить ее в лямбда-функцию, которая захватывает значения *A* и *B* и, будучи вызванной, возвращает их произведение:

```
auto P = [A, B] {
    return A * B;
};
```



Теперь, если кому-либо понадобится значение, достаточно вызвать этот объект как функцию: *P()*.

Таким образом, мы оптимизировали код, содержащий сложное вычисление, результат которого может не понадобиться. Однако тем самым мы создали и новую проблему: что, если вычисляемое функцией значение понадобится более одного раза? При текущем решении приходится вычислять результат каждый раз, когда программа его использует. Вместо этого было бы лучше запоминать результат, когда он вычисляется в первый раз.

Все это вместе взятое и составляет суть ленивых вычислений: вместо того чтобы делать какие-либо вычисления впрок, их откладывают, пока возможно. С другой стороны, быть ленивым – значит избегать повторения однажды сделанной работы, поэтому, получив результат вычислений, его сохраняют для дальнейшего использования.



## 6.1 Ленивые вычисления в языке C++

К сожалению, язык C++, в отличие от многих других языков программирования, не обладает встроенной поддержкой ленивых вычислений, зато в нем есть средства, с помощью которых программист может имитировать в своих программах такое поведение. Создадим шаблон класса под названием `lazy_val`, способный хранить функцию и запоминать ее результат, когда он вычислен (этот прием часто называют мемоизацией). Таким образом, тип должен содержать следующие данные:

- функцию для вычисления значения;
- признак того, вычислен ли уже результат;
- вычисленный результат.

В главе 2 говорилось, что наиболее эффективный способ принять произвольный функциональный объект в качестве аргумента – сделать его тип параметром шаблона. Воспользуемся этим советом и сейчас: шаблон `lazy_val` будет параметризован типом функционального объекта, ответственного за вычисление требуемого результата. Нет нужды задавать тип результата отдельным параметром шаблона, поскольку его легко вывести из типа функции.

## Листинг 6.1 Переменные-члены шаблона класса lazy\_val

```

template <typename F>
class lazy_val {
private:
    F m_computation;
    mutable bool m_cache_initialized;
    mutable decltype(m_computation()) m_cache;
    mutable std::mutex m_cache_mutex;

public:
    ...
};

```

Объект должен знать, был ли результат уже получен

Функциональный объект, используемый для вычисления результата

Хранилище для вычисленного результата. Тип этой переменной – это тип значения, возвращаемого функциональным объектом

Семафор нужен, чтобы предотвратить попытку нескольких потоков одновременно вычислить и сохранить результат

Шаблон класса `lazy_val` можно сделать неизменяемым – по крайней мере, он выглядит таким для использующего его кода. Все функции-члены объявим константными. Внутренней реализации, однако, нужна возможность присваивать новое значение переменной, в которой хранится результат: а именно когда он впервые получен. Поэтому все переменные-члены, относящиеся к сохраненному результату, должны быть объявлены с ключевым словом `mutable`.

Выбор типа функционального объекта в качестве параметра шаблона имеет свой недостаток: автоматический вывод параметров для шаблонов типов поддерживается лишь начиная с C++17. Но в большинстве случаев указывать тип этого параметра в явном виде невозможно, так как для вычисления значений значений наверняка будут использоваться лямбда-функции, а их тип записать невозможно. Поэтому понадобится еще функция `make_lazy_val`, ведь автоматический вывод параметров работает для шаблонов функций, что дает возможность использовать класс `lazy_val` с компиляторами, не поддерживающими стандарт C++17.

## Листинг 6.2 Конструктор и фабричная функция

```

template <typename F>
class lazy_val {
private:
    ...

public:
    lazy_val(F computation)
        : m_computation(computation)
        , m_cache_initialized(false)
    {
    }
};

template <typename F>
inline lazy_val<F> make_lazy_val(F&& computation)
{
    return lazy_val<F>(std::forward<F>(computation));
}

```

Инициализация вычисляющей функции. Сохраненное значение еще не проинициализировано

Вспомогательная функция, которая автоматически подставляет типы в шаблон и создает экземпляр инстанцированного класса

Конструктор не делает ничего особенного, помимо того что сохраняет в объекте функцию для вычисления значения и сбрасывает флаг готовности результата.

**ЗАМЕЧАНИЕ** Эта реализация требует, чтобы тип возвращаемого функцией результата обладал конструктором по умолчанию. В классе имеется поле `m_cache`, которому не присваивается начальное значение, поэтому конструктор класса `lazy_val` должен неявно вызвать для этого поля конструктор по умолчанию. Данное ограничение присуще лишь этой конкретной реализации, но не понятию ленивого вычисления как таковому. Полную реализацию шаблона класса, избавленную от указанной проблемы<sup>1</sup>, можно найти среди примеров к данной главе в директории `lazy_val`.

Последний шаг – создание метода, который бы возвращал «ленивое» значение. Если этот метод вызывается в первый раз, он должен вычислить значение и сохранить его. В противном случае он возвращает ранее сохраненное значение. Можно определить в классе `lazy_val` операцию вызова, тем самым сделав его функциональным, или же определить операцию приведения типа, что позволит объектам класса `lazy_val` выглядеть как обычные переменные типа `T`. Каждый из этих двух подходов обладает своими преимуществами, и выбор какого-либо из них – дело вкуса. Примем здесь второй вариант.

Реализация этого метода вполне очевидна. Нужно запереть семафор, чтобы никакой другой поток не имел доступа к сохраненному значению, пока оно не будет вычислено; если же значение еще не сохранено, нужно вызвать содержащуюся в объекте функцию и сохранить полученное значение.

### Листинг 6.3 Операция приведения типа для шаблона класса `lazy_val`

```
template <typename F>
class lazy_val {
private:
    ...
public:
    ...
    operator const decltype(m_computation())& () const ←
    {
        std::unique_lock<std::mutex>
            lock{m_cache_mutex};
    }
};
```

Этот метод делает возможным неявное преобразование объектов класса `lazy_val` к типу константной ссылки на результат вычисления

Предотвратить одновременный доступ к сохраненному значению

<sup>1</sup> Наиболее очевидное решение – использовать шаблон `std::optional<T>`, введенный в стандарте C++17, или его прямой аналог из библиотеки Boost. Он представляет собой контейнер, содержащий не более одного значения типа `T`, – т. е. либо пустой, либо содержащий один объект. Наличие конструктора по умолчанию у типа-параметра не требуется. Если C++17 и библиотека Boost по каким-либо причинам недоступны, в качестве обертки для объекта типа `T` можно использовать умный указатель `std::unique_ptr<T>`. – *Прим. перев.*

```

    if (!m_cache_initialized) {
        m_cache = m_computation();
        m_cache_initialized = true;
    }

    return m_cache;
}
};

```

Сохранить результат вычисления  
для последующего использования



Читатель, поднаторевший в многопоточном программировании, может заметить, что показанная выше реализация не вполне оптимальна. Всякий раз, когда программе нужно получить из объекта сохраненное значение, объект запирает и затем открывает семафор. Однако на самом деле семафор необходимо запирает только на время первого вызова функции – пока значение вычисляется.

Вместо семафора – универсального низкоуровневого примитива синхронизации – можно использовать средство, в точности подходящее для данного случая. Операция приведения типов делает две вещи: инициализирует поле `m_cache` и возвращает значение, сохраненное в этой переменной; при этом инициализация должна произойти лишь один раз за время жизни объекта. Таким образом, есть часть функции, которую нужно выполнить только при первом ее вызове. В стандартной библиотеке есть для этого решение – функция `std::call_once`. С ней реализация ленивого вычисления принимает следующий вид:

```

template <typename F>
class lazy_val {
private:
    F m_computation;
    mutable decltype(m_computation()) m_cache;
    mutable std::once_flag m_value_flag;

public:
    ...

    operator const decltype(m_computation())& () const
    {
        std::call_once(m_value_flag, [this] {
            m_cache = m_computation();
        });

        return m_cache;
    }
};

```



Таким образом, семафор и поле логического типа `m_cache_initialized` из предыдущей реализации заменены объектом типа `std::once_flag`. Функция `std::call_once` в первую очередь проверяет, установлен ли флаг, и если не установлен, выполняет функцию, переданную во втором аргументе – в данном случае это лямбда-функция, которая инициализирует поле `m_value`.

Это решение гарантирует, что поле `m_value` будет проинициализировано безопасным образом (параллельный доступ к полю из других потоков блокируется на то время, пока длится его инициализация) и что инициализация выполнится лишь один раз. Новая реализация проще первоначальной: из ее текста ясно видно, что должно быть выполнено. Кроме того, она эффективнее, ведь после того, как значение вычислено, блокировка потоков уже не требуется.

## 6.2 Ленивые вычисления как средство оптимизации программ

Теперь, когда читатель познакомился с простейшим случаем ленивого вычисления – вычислением единственного значения, когда оно понадобится в первый раз, с сохранением для последующего использования, – можно переходить к более сложным примерам. Часто на практике можно встретить задачи, которые не удастся решить простой заменой типа какой-либо переменной на его ленивый аналог, как в предыдущем примере. Иногда приходится разрабатывать альтернативные, ленивые версии хорошо известных алгоритмов. Всякий раз, когда алгоритму, работающему с обширной структурой данных, требуется лишь небольшая ее часть, представляется возможность оптимизировать код за счет ленивых вычислений.

### 6.2.1 Ленивая сортировка коллекций

Допустим, дан вектор с личными данными сотен сотрудников. Приложение позволяет просматривать список в окне, в котором помещается десять записей за один раз. У пользователя есть возможность сортировать список сотрудников по различным критериям: имени, возрасту, стажу работы в компании и т. д. Когда пользователь выбирает, скажем, сортировку по возрасту, программа должна показать 10 старших сотрудников и позволить пользователю перейти к следующей странице, где показаны следующие по возрасту.

Этого легко добиться, если отсортировать всю коллекцию и затем отображать ее по 10 элементов на страницу. Хотя этот подход достаточно хорош для случаев, когда пользователю нужен весь отсортированный список, он может приводить к пустой трате усилий, если это не так. Пользователю может быть нужна только первая десятка отсортированного списка, а алгоритм вынужден при каждой смене критерия заново сортировать весь список.

Чтобы добиться как можно более высокой эффективности, нужно придумать, как сортировать коллекцию ленивым способом. При этом можно взять за основу алгоритм быстрой сортировки `quicksort` – самый популярный алгоритм сортировки данных в памяти.

Принцип работы обычного алгоритма quicksort прост: выбрав некоторый элемент коллекции, нужно переместить все элементы, большие, чем выбранный, в начало коллекции, а меньшие или равные – в конец (для этого можно использовать даже функцию `std::partition`). Затем та же операция повторяется отдельно для первого и второго полученного сегмента.

Что можно изменять, чтобы алгоритм стал ленивым? Можно сортировать лишь ту часть коллекции, которую предполагается показывать пользователю, оставляя прочие элементы неотсортированными. Разделения коллекции на два сегмента избежать невозможно, однако можно отложить рекурсивный вызов алгоритма сортировки для тех частей коллекции, которые еще не понадобились (рис. 6.1).

Таким образом, ленивый алгоритм сортирует элементы, только когда (и если) в этом возникает необходимость. Это позволяет избежать большинства рекурсивных вызовов алгоритма на тех участках массива, элементов которых пользователь пока не запрашивает. Простую реализацию такого алгоритма можно найти среди примеров к этой главе в директории `lazy-sorting`.

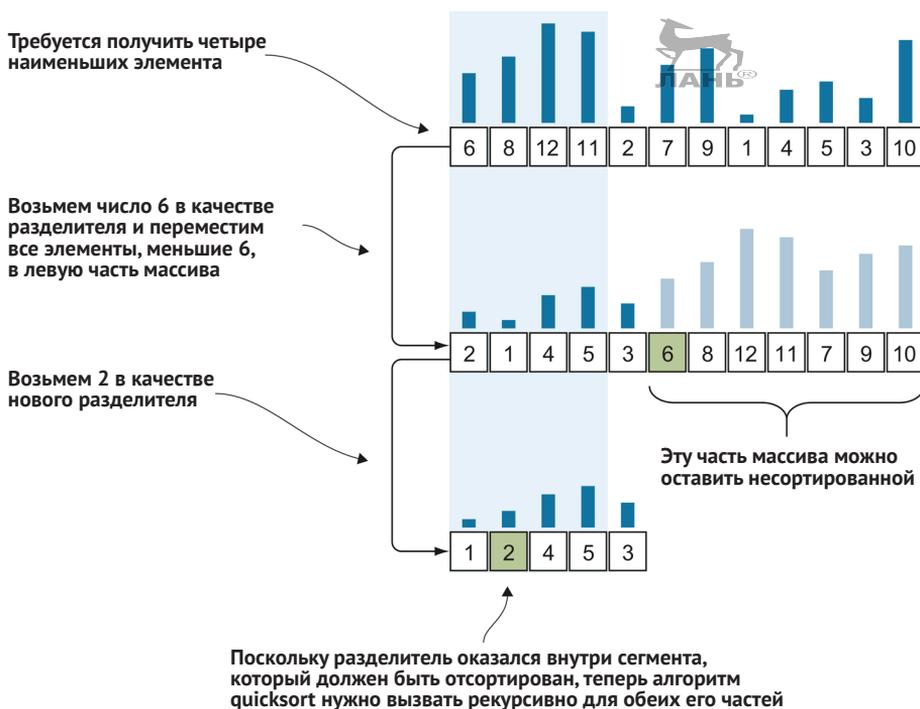


Рис. 6.1 Алгоритм quicksort можно оптимизировать так, чтобы он не вызывал себя рекурсивно для тех участков массива, которые еще не понадобились.

### Временная сложность ленивой версии алгоритма quicksort

Поскольку в стандарте языка C++ зафиксированы требования к временной сложности всех определенных в нем алгоритмов, было бы небезынтересно выяснить сложность ленивого алгоритма quicksort, описанного выше. Положим, коллекция содержит  $n$  элементов, и пусть из них требуется получить  $k$  первых в порядке сортировки.

Для первого разделения коллекции на два сегмента нужно в среднем  $O(n)$  операций, для второго уже  $O(n/2)$  и т. д., пока не окажется, что очередной сегмент весь нуждается в сортировке. Следовательно, в общей сложности на разделение сегментов нужно затратить порядка  $O(2n)$  операций, что эквивалентно  $O(n)$ .

Далее, для сортировки сегмента длиной  $k$  потребуется  $O(k \log k)$  операций, поскольку к нему применяется обычный алгоритм quicksort. Таким образом, общая временная сложность алгоритма составляет  $O(n + k \log k)$ , что весьма изящно, так как если применить этот алгоритм для поиска одного наибольшего элемента коллекции, временная сложность получится такой же, как и у алгоритма `std::max_element`, а именно  $O(n)$ . Противоположным образом, если с помощью ленивого алгоритма отсортировать всю коллекцию, сложность обратится в  $O(n \log n)$ , как у обычного алгоритма quicksort.

## 6.2.2 Отображение элементов в пользовательском интерфейсе

В предыдущем примере было показано, как один конкретный алгоритм преобразовать в ленивый, однако те же аргументы в пользу ленивой стратегии вычислений справедливы и во многих других случаях. Например, всякий раз, когда имеется обширный набор данных и ограниченное экранное пространство для их отображения, это может быть хорошим поводом оптимизировать алгоритм, сделав его ленивым. Весьма типична ситуация, когда данные хранятся где-то в базе данных и их нужно тем или иным способом показать пользователю.

Обратимся снова к идее предыдущего примера: пусть есть база данных, в которой хранятся данные о сотрудниках. В пользовательском интерфейсе хотелось бы отображать их имена вместе с фотографиями. В предыдущем примере потребность в ленивом алгоритме была продиктована тем, что сортировка всей последовательности ради показа лишь десяти ее элементов приводила бы к пустой трате усилий.

Теперь у нас есть база данных, которая может сама выполнять сортировку. Означает ли это, что программе следует загружать все данные сразу? Конечно же, нет. Пусть наша программа не занимается сортировкой сама, но эту работу делает база данных. Подобно показанному выше ленивому алгоритму, базы данных стараются сортировать данные лишь тогда, когда клиент их запрашивает. Если программа попытается получить данные всех сотрудников за один раз, базе данных придется отсортировать все записи. Но затраты времени на сортировку – это не



единственная проблема. Ведь программе нужно отобразить фотографию каждого сотрудника, а загрузка фотографий тоже требует времени и памяти. Если загружать все данные заранее, программа станет работать с черепашной скоростью и потреблять слишком много памяти.

Чаще всего прибегают к ленивой загрузке данных, т. е. загружают данные лишь тогда, когда их действительно нужно показать пользователю (см. рис. 6.2). Это позволяет потреблять меньше процессорного времени и оперативной памяти. Единственная трудность состоит в том, что не удастся полностью следовать ленивому принципу. Невозможно хранить в памяти все ранее загруженные фотографии сотрудников, так как для этого потребовалось бы столь же много памяти. Чтобы избежать этого, нужно со временем забывать ранее загруженные данные и подгружать их повторно, когда они понадобятся снова.

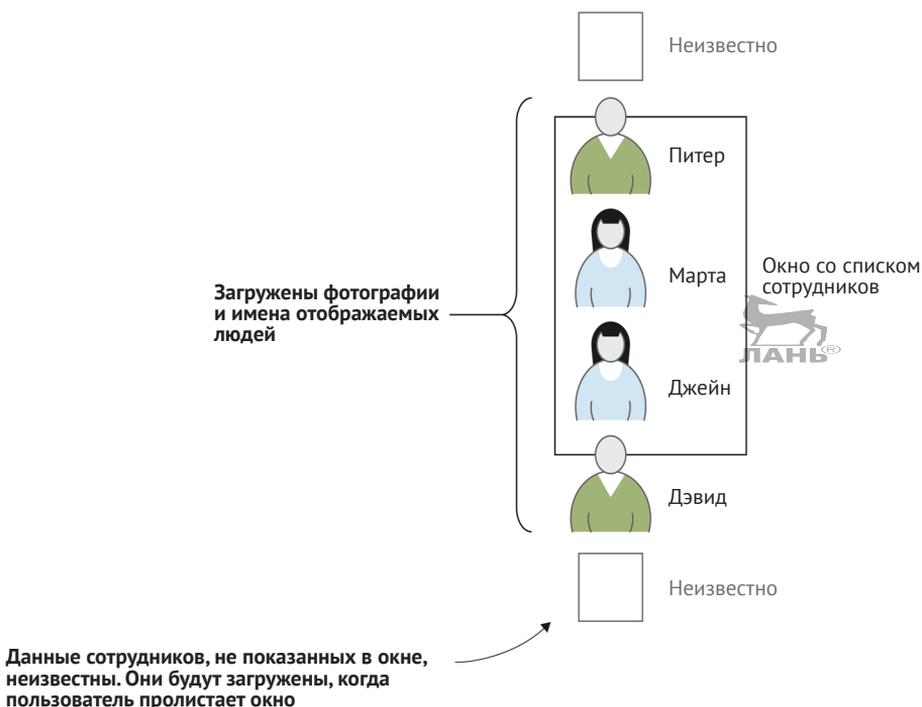


Рис. 6.2 Не нужно загружать данные, которые не видны пользователю. Информацию можно запрашивать тогда, когда она понадобится.

### 6.2.3 Подрезка дерева рекурсивных вызовов за счет запоминания результатов функции

В отсутствие прямой поддержки ленивых вычислений языком C++ хорошо то, что можно своими руками реализовать их желаемым образом. Только от программиста зависит, насколько ленивыми должны быть вычисления в каждом конкретном случае. Обратимся к широко известному



примеру – вычислению чисел Фибоначчи. Исходя из определения, алгоритм можно было бы реализовать следующим образом:

```
unsigned int fib(unsigned int n)
{
    return n == 0 ? 0 :
           n == 1 ? 1 :
           fib(n - 1) + fib(n - 2);
}
```

Эта реализация неэффективна. В общем случае функция должна сделать два рекурсивных вызова, каждый из которых вынужден проделать одну и ту же работу дважды, так как значение  $\text{fib}(n - 2)$  требуется для вычисления как  $\text{fib}(n)$ , так и  $\text{fib}(n - 1)$ . В свою очередь, для вычисления  $\text{fib}(n - 1)$  и  $\text{fib}(n - 2)$  нужно сперва найти  $\text{fib}(n - 3)$  и т. д. Число рекурсивных вызовов зависит от  $n$  экспоненциально (рис. 6.3).

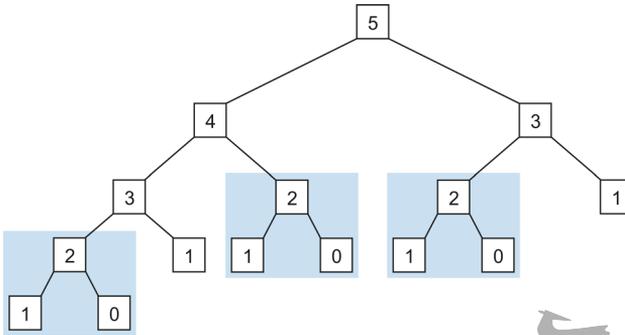


Рис. 6.3 При рекурсивной реализации функции  $\text{fib}$  одно и то же значение вычисляется многократно. Так, дерево рекурсивного вычисления  $\text{fib}(5)$  содержит три одинаковых поддерева для вычисления  $\text{fib}(3)$ . Дублирование работы растет экспоненциально как функция от  $n$

Эта функция – чистая: она всегда возвращает одинаковый результат, если ей на вход подавать одно и то же значение. Поэтому, вычислив значение  $\text{fib}(n)$ , можно поместить его в какое-нибудь хранилище и извлекать всякий раз, когда оно снова понадобится. Если таким образом запоминать все ранее вычисленные значения, можно устранить все повторяющиеся поддеревья. В этом случае дерево процесса вычисления будет выглядеть так, как показано на рис. 6.4. Точный порядок вычисления может отличаться от показанного, так как правила языка C++ не гарантируют, что  $\text{fib}(n - 1)$  будет вычислено раньше, чем  $\text{fib}(n - 2)$ , но в одном все возможные деревья окажутся похожи: в них не будет совпадающих поддеревьев.

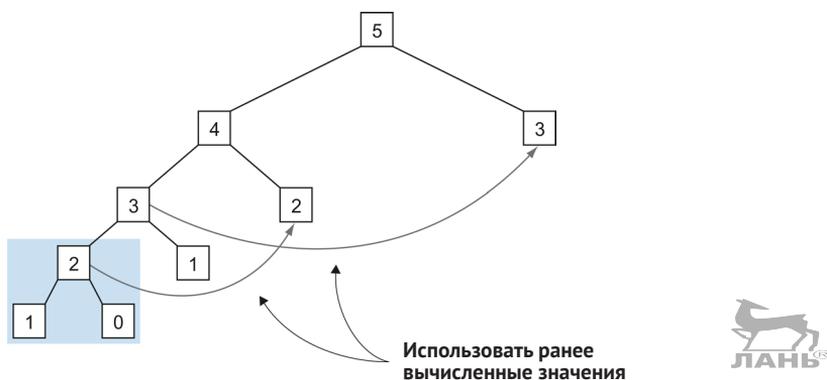


Рис. 6.4 Ленивая реализация функции `fib` позволяет избежать многократного вычисления одних и тех же значений. Если одна ветка дерева вызовов вычислит значение `fib(2)`, остальные ветки используют его. Теперь число вершин дерева зависит от  $n$  линейно

#### Листинг 6.4 Вычисление чисел Фибоначчи с мемоизацией результатов

```
std::vector<unsigned int> cache{0, 1};
unsigned int fib(unsigned int n)
{
    if (cache.size() > n) {
        return cache[n];
    } else {
        const auto result = fib(n - 1) + fib(n - 2);
        cache.push_back(result);
        return result;
    }
}
```

← В качестве хранилища результатов можно использовать вектор, ведь для того, чтобы вычислить значение `fib(n)`, нужно знать значения `fib(k)` при всех  $k < n$

Вернуть значение, если оно уже вычислено и находится в хранилище

Вычислить результат и запомнить его. Элемент можно помещать в конец контейнера, так как гарантируется, что все предыдущие значения уже находятся в хранилище

В этом подходе хорошо то, что не нужно изобретать новый алгоритм вычисления чисел Фибоначчи. Не нужно даже понимать, как алгоритм работает. Достаточно распознать, что алгоритм многократно обрабатывает одно и то же значение аргумента и что всякий раз при этом получаются одинаковые результаты, так как функция чистая.

Единственный недостаток данного решения состоит в том, что для хранения ранее вычисленных значений требуется дополнительная память. Если хочется оптимизировать также и этот аспект, нужно исследовать, как алгоритм работает и когда используются сохраненные результаты.

Если внимательно проанализировать код, легко увидеть, что алгоритму не нужны никакие сохраненные значения, кроме двух, добавленных последними. Поэтому вектор произвольной длины можно заменить хранилищем на всего лишь два значения. Чтобы новое хранилище обладало тем же интерфейсом, что и контейнер `std::vector`, создадим класс, для клиента выглядящий подобно вектору, но хранящий лишь два последних значения (см. пример `fibonacci/main.cpp`).

### Листинг 6.5 Эффективное хранилище ранее вычисленных чисел Фибоначчи

```

class fib_cache {
public:
    fib_cache()
        : m_previous{0}
        , m_last{1}
        , m_size{2}
    {
    }

    size_t size() const
    {
        return m_size;
    }

    unsigned int operator[] (unsigned int n) const
    {
        return n == m_size - 1 ? m_last :
            n == m_size - 2 ? m_previous :
                0;
    }

    void push_back(unsigned int value)
    {
        m_size++;
        m_previous = m_last;
        m_last = value;
    }
};

```

Начало последовательности Фибоначчи: {0, 1}

Количество ранее сохраненных значений (включая уже забытые)

Вернуть запрошенное число, если оно все еще находится в хранилище. В противном случае вернуть 0. Вместо этого можно было бы генерировать исключение

Добавить новое значение в хранилище, забыть старое значение и нарастить счетчик

## 6.2.4 Метод динамического программирования как разновидность ленивого вычисления

Динамическое программирование – это метод решения сложных задач путем разбиения их на ряд более простых. Идея метода состоит в том, чтобы решения мелких подзадач сохранять для последующего использования. Этот прием используется для решения многих практических задач, включая поиск кратчайшего пути в графе или нахождение расстояния между строками.

Оптимизация, примененная выше для вычисления чисел Фибоначчи, также основывается на принципе динамического программирования. Получив задание вычислить значение  $\text{fib}(n)$ , мы, опираясь на определение числа Фибоначчи, разбили его на два более простых: вычислить  $\text{fib}(n - 1)$  и  $\text{fib}(n - 2)$ . Запоминая результаты всех таких подзадач, удалось существенно оптимизировать первоначальный алгоритм.

Для функции  $\text{fib}$  подобная оптимизация была вполне очевидной, однако в общем случае это не всегда так. Рассмотрим задачу нахождения численной меры сходства двух строк. Одна из таких метрик – расстояние

Левенштейна<sup>1</sup>, или дистанция редактирования, определяемая как наименьшее количество удалений, вставок и замен, необходимых для преобразования одной строки в другую. Например:

- расстояние между строками «пример» и «пример» составляет 0, поскольку строки совпадают;
- между строками «примерный» и «пример» расстояние 3, так как в первой строке нужно удалить три символа, чтобы получить вторую;
- расстояние между строками «пример» и «промер» составляет 1, ведь достаточно заменить одну букву «и» на букву «о».

Найти расстояние между двумя произвольными строками нетрудно. В самом деле, если уже известно расстояние между строками *a* и *b*, легко найти следующие расстояния:

- между строкой *a* и строкой, полученной добавлением одной буквы в конец строки *b* (соответствует операции добавления символа к исходной строке);
- между строкой, полученной из строки *a* добавлением в конец одного символа, и строкой *b* (соответствует операции удаления символа из исходной строки);
- между строками, полученными из *a* и *b* добавлением символов в конец. Если к этим строкам добавляется один и тот же символ, расстояние остается неизменным, в противном случае имеем операцию замены одной буквы.

В общем случае исходную строку *a* можно преобразовать в требуемую строку *b* множеством способов, как показано на рис. 6.5. Из них нужно выбрать тот, который требует наименьшего числа операций.

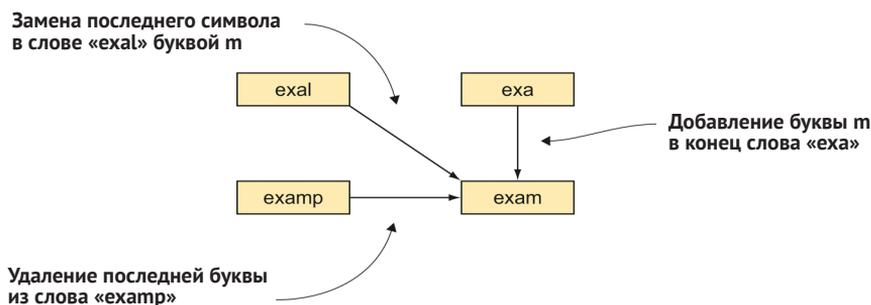


Рис. 6.5 Каждый вызов функции  $lev(m, n)$  вычисляет наименьшее расстояние по всем путям, у которых последняя операция есть дописывание буквы в конец; по всем путям, заканчивающимся операцией удаления последней буквы; по всем путям, последняя операция которых есть замена символа

Расстояние между первыми *m* символами строки *a* и первыми *n* символами строки *b* будем обозначать через  $lev(m, n)$ . Эту функцию можно реализовать рекурсивно.

<sup>1</sup> Более подробные сведения можно найти в статье по адресу [https://ru.wikipedia.org/wiki/Расстояние\\_Левенштейна](https://ru.wikipedia.org/wiki/Расстояние_Левенштейна).



### Листинг 6.6 Рекурсивное вычисление расстояния Левенштейна

```

unsigned int lev(unsigned int m, unsigned int n)
{
return m == 0 ? n | Если одна из строк пуста,
    : n == 0 ? m | расстояние равно длине второй строки
    : std::min({
        lev(m - 1, n) + 1, ← Операция добавления символа
        lev(m, n - 1) + 1, ← Операция удаления символа
        lev(m - 1, n - 1) + (a[m - 1] != b[n - 1]) ← Операция замены символа –
    });                                     только если символы различны
}

```

Передачу строк a и b через аргументы опускаем для упрощения кода

Данная реализация функции `lev` перебирает все возможные преобразования, которые превращают строку `a` в строку `b` (за исключением последовательностей с заведомо лишними операциями, такими как добавление и последующее удаление символа). Как и в случае с функцией `fib`, число рекурсивных вызовов здесь экспоненциально зависит от `m` и `n`.

Всем известно, как реализовать функцию `fib` обыкновенным циклом, и никто не стал бы на самом деле писать реализацию, с которой мы здесь начали, однако в случае расстояния между строками оптимизация отнюдь не очевидна. Единственное, что пока что очевидно, – это то, что функция `lev(m, n)` является чистой, подобно функции `fib(n)`: результат функции зависит единственно от ее аргументов, и его вычисление не приводит к побочным эффектам. При этом может понадобиться не более, чем  $m * n$  промежуточных результатов. Поэтому экспоненциальная сложность вычисления может объясняться только тем, что одни и те же промежуточные результаты вычисляются по несколько раз.

Какое решение первым приходит в голову? Запоминать все ранее полученные результаты. Поскольку аргументы функции – это два целых числа без знака, естественно в качестве хранилища результатов использовать матрицу.

## 6.3 Универсальная мемоизирующая обертка

Хотя обычно бывает лучше определить специальное хранилище значений для каждой отдельной задачи, чтобы собственноручно управлять длительностью хранения промежуточных значений (как в случае забывающего хранилища для функции `fib(n)`) или выбрать наиболее подходящую структуру данных (например, использование матрицы для функции `lev(m, n)`), иногда бывает полезно заключить функцию в универсальную обертку и автоматически получить функцию, запоминающую ранее вычисленные результаты.

Наиболее общая структура данных для сохранения результатов, когда невозможно заранее сказать, с какими аргументами будет вызываться, –

это ассоциативный массив (англ. map), т. е. структура данных, некоторым ключам ставящая в соответствие результат. Любая чистая функция – это отображение аргументов в значения, поэтому ассоциативный массив позволяет моделировать любую чистую функцию<sup>1</sup>.

**Листинг 6.7 Создание мемоизирующей обертки из указателя на функцию**

Создается таблица соответствия аргументов ранее вычисленным результатам. Если предполагается использовать эту функцию в многопоточной среде, доступ к хранилищу нужно синхронизировать, как в листинге 6.1

```

template <typename Result, typename... Args>
auto make_memoized(Result (*f)(Args...))
{
    std::map<std::tuple<Args...>, Result> cache;

    return [f, cache](Args... args) mutable -> Result
    {
        const auto args_tuple =
            std::make_tuple(args...);
        const auto cached = cache.find(args_tuple);

        if (cached == cache.end()) {
            auto result = f(args...);
            cache[args_tuple] = result;
            return result;
        } else {
            return cached->second;
        }
    };
}
    
```

Собрать аргументы  
лямбда-функции  
в кортеж и проверить,  
есть ли в таблице ранее  
вычисленный результат

Если нужное значение в хранилище  
не найдено, вызвать функцию  
и сохранить полученное значение

Если значение найдено в хранилище  
ранее вычисленных, вернуть его

Теперь любую функцию можно преобразовать так, чтобы она запоминала (и затем быстро вспоминала) результаты своих предыдущих вызовов<sup>2</sup>. Попробуем проделать это с функцией, генерирующей числа Фибоначчи.

**Листинг 6.8 Использование функции make\_memoized**

```

auto fibmemo = make_memoized(fib);

std::cout << "fib(15) = " << fibmemo(15)
    << std::endl;
std::cout << "fib(15) = " << fibmemo(15)
    << std::endl;
    
```

Вычислить значение fibdemo(15)  
и запомнить результат

Повторный вызов fibdemo(15)  
извлечет ранее сохраненное  
значение, вместо того чтобы  
вычислять его заново

<sup>1</sup> При условии, что для типов всех ее аргументов определено отношение сравнения «меньше». – *Прим. перев.*

<sup>2</sup> Здесь уместно вспомнить о классическом шаблоне проектирования «декоратор». – *Прим. перев.*

Если измерить производительность этой программы, легко убедиться, что второй вызов `fibdemo(15)` работает быстрее. Однако первый вызов работает медленно, и с увеличением аргумента функции время его работы растет экспоненциально. Проблема здесь в том, что наша обертка не может полностью оптимизировать функцию `fib`. Функция `fibdemo` сохраняет полученный от функции `fib` результат для повторного использования, но реализация самой функции `fib` не пользуется этим хранилищем, а вызывает саму себя. Поэтому обертка `make_memoized` хорошо работает лишь для обычных функций, но не подходит для оптимизации рекурсивных.

Если хочется оптимизировать также и рекурсивные функции, нужно изменить функцию `fib` так, чтобы она вызывала не саму себя непосредственно, а некоторую функцию, переданную ей в качестве аргумента<sup>1</sup>:

```
template <typename F>
unsigned int fib(F&& fibmemo, unsigned int n)
{
    return n == 0 ? 0
           : n == 1 ? 1
           : fibmemo(n - 1) + fibmemo(n - 2);
}
```

Теперь, вызывая функцию `fib`, можно в первом аргументе передавать ей мемоизированную версию этой же функции. К сожалению, мемоизирующую обертку придется для этого сделать более сложной, поскольку теперь она должна передавать в рекурсивную функцию результат мемоизации. В противном случае мемоизация рекурсивной функции запомнила бы лишь результат последнего вызова, но не результаты промежуточных самовывозов, ведь рекурсивно вызывалась бы первоначальная функция, а не ее мемоизирующая версия.

Таким образом, нужно создать функциональный объект, в котором бы хранилась обертыаемая функция и таблица результатов ее предыдущих вызовов (см. пример `recursive-memoization/main.cpp`).

### Листинг 6.9 Мемоизирующая обертка для рекурсивных функций

```
class null_param {};

template <class Sig, class F>
class memoize_helper;

template <class Result, class... Args, class F>
class memoize_helper<Result(Args...), F> {
private:
    using function_type = F;
    using args_tuple_type
```

← Пустой класс используется в конструкторе, чтобы исключить конфликт перегрузки с конструктором копирования

<sup>1</sup> Таким же приемом пользуются для преобразования рекурсивной функции в неподвижную точку функции высшего порядка, и это совпадение неслучайно. — *Прим. перев.*

```

= std::tuple<std::decay_t<Args>...>;

function_type f;
mutable std::map<args_tuple_type, Result> m_cache;
mutable std::recursive_mutex m_cache_mutex;

public:
    template <typename Function>
    memoize_helper(Function&& f, null_param)
        : f(f)
    {
    }

    memoize_helper(const memoize_helper& other)
        : f(other.f)
    {
    }

    template <class... InnerArgs>
    Result operator()(InnerArgs&&... args) const
    {
        std::unique_lock<std::recursive_mutex>
            lock{m_cache_mutex};

        const auto args_tuple =
            std::make_tuple(args...);
        const auto cached = m_cache.find(args_tuple);

        if (cached != m_cache.end()) {
            return cached->second;
        } else {
            auto&& result = f(
                *this,
                std::forward<InnerArgs>(args)...);
            m_cache[args_tuple] = result;
            return result;
        }
    }
};

template <class Sig, class F>
memoize_helper<Sig, std::decay_t<F>>
make_memoized_r(F&& f)
{
    return {std::forward<F>(f), detail::null_param()};
}

```

Таблица ранее вычисленных значений и семафор для синхронизации вносимых в нее изменений



Конструкторы должны проинициализировать лишь обертываемую функцию. Конструктор копирования мог бы также скопировать и таблицу значений, но необходимости в этом нет



Поиск в таблице ранее полученных результатов

Если соответствие найдено, вернуть результат, не вызывая функцию f

Если такой вызов в хранилище не найден, вызвать функцию f и запомнить результат. В первый аргумент функции f передать \*this, чтобы для рекурсивных самовывозов функция f пользовалась мемоизирующей версией самой себя

Теперь мемоизирующую версию функции fib можно создать следующим образом:

```

auto fibmemo = make_memoized_r<
    unsigned int(unsigned int)>(

```

```

[](auto& fib, unsigned int n) {
    std::cout << "Вычисляется " << n << "!\\n";
    return n == 0 ? 0
           : n == 1 ? 1
           : fib(n - 1) + fib(n - 2);
});

```



Результаты всех промежуточных вызовов будут запоминаться для повторного использования. Лямбда-функция вызывает себя не напрямую, а косвенно, через посредство объекта `memoize_helper`, который и будет передаваться в ее первый аргумент `fib`. Еще одно преимущество этой реализации по сравнению с предыдущей, по имени `make_memoized`, состоит в том, что она может работать с функциональными объектами любого типа, тогда как предыдущей функции можно было передавать только указатель на функцию.

## 6.4 Шаблоны выражений и ленивая конкатенация строк



Предыдущие примеры были посвящены главным образом оптимизации на этапе выполнения – т. е. случаям, когда выполнение программы может пойти различными путями и оптимизировать хочется их все. Однако ленивые вычисления могут принести пользу даже тогда, когда заранее известен каждый шаг будущего вычисления.

Рассмотрим одну из самых часто встречающихся операций – конкатенацию строк:

```
std::string fullname = title + " " + surname + ", " + name;
```

Здесь нужно соединить между собой несколько строк. Эта реализация безусловно корректна и делает в точности то, что нужно. Однако она работает не так быстро, как хотелось бы. Посмотрим на этот код глазами компилятора. Перегруженная бинарная операция `operator+` левоассоциативна, поэтому выражение эквивалентно следующему:

```
std::string fullname = (((title + " ") + surname) + ", ") + name;
```

При вычислении первого подвыражения `(title + " ")` создается новый временный объект строкового типа. Для каждой из последующих конкатенаций для этой временной строки вызывается метод `append` (рис. 6.6). При этом каждый раз длина строки возрастает. Иногда буфер, выделенный для временного объекта, будет оказываться недостаточно большим, чтобы в нем поместились новые данные. В этом случае объекту нужно выделить новый буфер большего размера и скопировать туда все данные из старого.

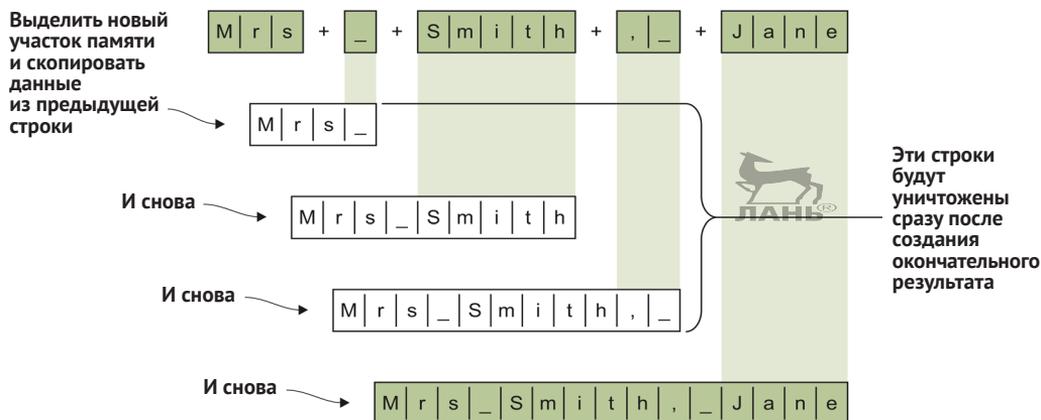


Рис. 6.6 При «жадном» способе конкатенации строк выделение памяти может произойти в любой из промежуточных операций, если предыдущий буфер окажется недостаточно большим для результата конкатенации

Этот метод неэффективен. Он требует создания (и последующего уничтожения) ненужных временных объектов. Если соединить нужно несколько строк, было бы гораздо более эффективно отложить создание результата до тех пор, пока не станут известны все они. Тогда можно будет один раз выделить буфер такого размера, чтобы в нем поместился окончательный результат, и скопировать в него данные из исходных строк.

Здесь и приходят на помощь шаблоны выражений. Этот прием позволяет строить определения значений, вместо того чтобы вычислять значения выражений. Перегруженный метод `operator+` можно реализовать так, чтобы вместо строки (результата конкатенации) он возвращал объект, представляющий способ получения результата, а сам результат пусть вычисляется позже. В данном примере основная трудность состоит в том, что операция `operator+` бинарна, а нам нужно соединять между собой несколько строк. Поэтому надо определить структуру данных, представляющую выражение, составленное из произвольного числа строк-слагаемых (см. пример `string-concatenation/main.cpp`). Поскольку она должна хранить заранее неизвестное число строк, ее можно реализовать в виде рекурсивного шаблона структуры. В каждом узле должна храниться одна строка (поле `data`) и узел, представляющий оставшуюся часть выражения (поле `tail`).

#### Листинг 6.10 Структура для хранения произвольного числа строк

```
template <typename... Strings>
class lazy_string_concat_helper;

template <typename LastString, typename... Strings>
class lazy_string_concat_helper<LastString,
                                Strings...> {
```

```

private:
    LastString data; ← Копия исходной строки
    lazy_string_concat_helper<Strings...> tail; ← Структура с остальными
                                                    строками
public:
    lazy_string_concat_helper(
        LastString data,
        lazy_string_concat_helper<Strings...> tail)
        : data(data)
        , tail(tail)
    {
    }

    int size() const
    {
        return data.size() + tail.size();
    }

    template <typename It>
    void save(It end) const
    {
        const auto begin = end - data.size();
        std::copy(data.cbegin(), data.cend(),
                  begin);
        tail.save(begin);
    }

    operator std::string() const
    {
        std::string result(size(), '\0');
        save(result.end());
        return result;
    }

    lazy_string_concat_helper<std::string,
                              LastString,
                              Strings...>
    operator+(const std::string& other) const
    {
        return lazy_string_concat_helper
            <std::string, LastString, Strings...>(
            other,
            *this
        );
    }
};

```

Вычисление суммарной длины всех строк

Строки хранятся в обратном порядке: поле data содержит строку, добавленную последней, поэтому ее нужно дописать в конец буфера

Чтобы определение выражения преобразовать в настоящую строку, выделить буфер памяти достаточного размера и скопировать в него строки-слагаемые

Создать новый объект, представляющий структуру выражения с одной добавленной строкой

Поскольку этот шаблон структуры рекурсивен, нужно дать определение базового случая, чтобы не уйти в бесконечную рекурсию:

```

template <>
class lazy_string_concat_helper<> {
public:

```

```

lazy_string_concat_helper()
{
}

int size() const
{
    return 0;
}

template <typename It>
void save(It) const
{
}

lazy_string_concat_helper<std::string>
operator+(const std::string& other) const
{
    return lazy_string_concat_helper<std::string>(
        other,
        *this
    );
}
};
    
```

В этой структуре может храниться сколь угодно много строк. Она не выполняет конкатенации до тех пор, пока пользователь не запросит окончательный результат путем преобразования объекта к типу `std::string`. Операция приведения типа создаст новую строку в точности нужной длины и скопирует в нее данные из хранящихся в структуре строк-слагаемых (рис. 6.7).

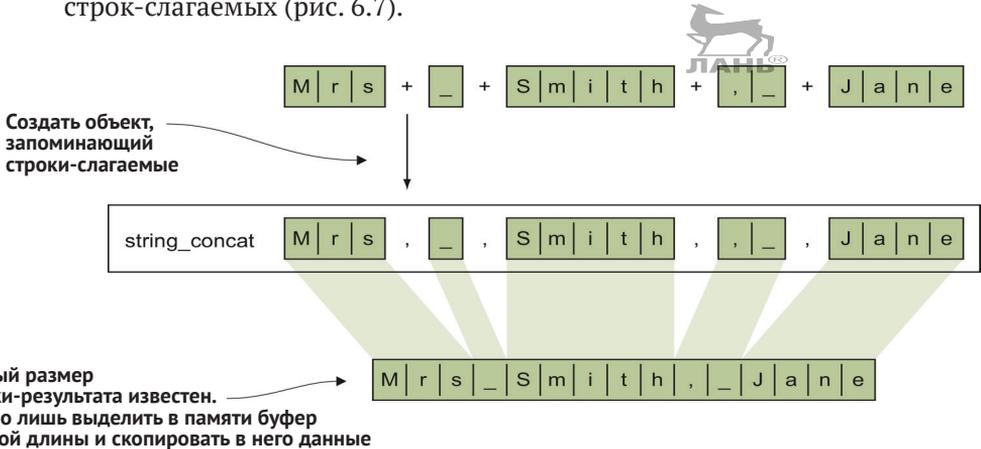


Рис. 6.7 Вместо того чтобы возвращать строку, операция `operator+` создает объект, хранящий в себе строки, которые предстоит соединить. Когда придет время создавать строку-результат, точный размер буфера для формирования результата будет известен

### Листинг 6.11 Использование шаблона выражения для эффективной конкатенации строк

```

lazy_string_concat_helper<> lazy_concat;
int main(int argc, char* argv[])
{
    std::string name = "Jane";
    std::string surname = "Smith";

    const std::string fullname =
        lazy_concat + surname + ", " + name;

    std::cout << fullname << std::endl;
}

```

Операция оператор+ для строк уже определена, и перегрузить ее по-своему нельзя, поэтому прибегнем здесь к небольшой хитрости: поставив первым слагаемое нужного типа, заставим всю сумму стать структурой выражения

Это решение работает примерно так же, как и структура `lazy_val`, рассмотренная в начале главы. Однако на этот раз задача состоит не в том, чтобы произвольную лямбда-функцию преобразовать в ленивое вычисление, а в том, чтобы ленивое вычисление построить из заданного пользователем выражения.

## 6.4.1 Чистота функций и шаблоны выражений

Внимательный читатель мог заметить, что в объектах `lazy_string_concat_helper` хранятся копии строк-слагаемых. Шаблоны выражения работали бы гораздо эффективнее, если бы в них использовались ссылки на значения-оригиналы. Ведь мы начали рассмотрение именно с идеи, как можно сильнее оптимизировать конкатенацию строк. Попробуем оптимизировать и этот аспект:

```

template <typename LastString, typename... Strings>
class lazy_string_concat_helper<LastString,
                                Strings...> {
private:
    const LastString& data;
    lazy_string_concat_helper<Strings...> tail;

public:
    lazy_string_concat_helper(
        const LastString& data,
        lazy_string_concat_helper<Strings...> tail)
        : data(data)
        , tail(tail)
    {
    }
    ...
};

```

Однако здесь проявляются два недостатка. Во-первых, такой шаблон выражения невозможно использовать за пределами области видимости,

в которой определены строки-слагаемые. С выходом из области видимости строки разрушаются, и ссылки на них, хранящиеся в объекте `lazy_string_concat_helper`, становятся невалидными.

Во-вторых, пользователь ожидает, что результат конкатенации строк будет строкой, а не объектом какого-либо вспомогательного типа. Если оптимизированная конкатенация используется вместе с механизмом автоматического вывода типов, можно ожидать неожиданностей. Рассмотрим следующий пример:

```
std::string name = "Jane";
std::string surname = "Smith";
```



```
const auto fullname = lazy_concat + surname + ", " + name;
```

Может показаться, что здесь создана строка "Smith, Jane", но...

```
name = "John";
```

```
std::cout << fullname << std::endl; ← ...напечатано будет "Smith, John"
```

Проблема в том, что теперь в шаблоне выражения хранятся ссылки на строки, которые предстоит соединить. Если эти строки изменяются между моментом, когда шаблон выражения создается (в данном примере это инициализация переменной `fullname`), и моментом, когда по шаблону выражения вычисляется его значение (в данном примере – запись переменной `fullname` в поток стандартного вывода), результат будет отличаться от ожидаемого. В окончательный результат войдут изменения, внесенные в строки *после* того, как создана их конкатенация.

Это важный аспект, о котором всегда следует помнить: для корректной работы ленивых вычислений требуется чистота функций. Чистые функции дают один и тот же результат, сколько бы ни вызывать их с одинаковыми аргументами. Именно поэтому их вычисление можно откладывать без нежелательных последствий. Едва лишь в программе появляются побочные эффекты (наподобие переменной, изменение которой влияет на результат операции), попытка выполнить ее ленивым способом искажает результат.

Шаблоны выражений позволяют создавать структуры, описывающие способ вычисления значения, вместо того чтобы сразу вычислять значение. С их помощью можно выбирать, в какой момент времени должно происходить настоящее вычисление; можно даже переопределить порядок, в котором выражения вычисляются на языке C++, и преобразовывать выражения каким угодно образом<sup>1</sup>.

Более подробную информацию и ссылки по темам, затронутым в этой главе, можно найти на странице <https://forums.manning.com/posts/list/41685.page>.

<sup>1</sup> Для построения и преобразования более сложных деревьев выражений может пригодиться библиотека Boost.Proto, которую можно найти по адресу <http://mng.bz/pEqP>.

## Итоги

- Ленивое выполнение кода и запоминание полученного результата для последующего использования может существенно повысить скорость работы программ.
- Часто бывает нелегко (а иногда и невозможно) построить ленивую версию алгоритмов, нередко используемых в программах. Если это удастся, можно заметно улучшить отзывчивость программ.
- Есть обширный класс алгоритмов с экспоненциальной временной сложностью, которые удастся оптимизировать до линейной или квадратичной сложности просто за счет сохранения промежуточных результатов.
- Расстояние Левенштейна находит множество применений (например, обработка звука, анализ ДНК, проверка орфографии) и может, в частности, пригодиться при разработке обычных пользовательских приложений. Когда приложение оповещает графический интерфейс об изменениях в модели данных, желательно свести к минимуму число операций, которые должен выполнить графический интерфейс.
- Хотя механизм запоминания ранее полученных значений чаще всего бывает нужно разрабатывать отдельно для каждой задачи, может оказаться полезной и универсальная обертка наподобие `memoized`, хотя бы для измерения выигрыша в скорости, который можно получить, храня таблицу результатов функции.
- Шаблоны выражений – это мощный прием для организации отложенных вычислений. Он часто используется в библиотеках, оперирующих с матрицами, и в иных местах, где выражения нужно оптимизировать перед тем, как передать их на исполнение.



# 7

## Диапазоны



### О чем говорится в этой главе:

- проблемы передачи пар итераторов в алгоритмы;
- что такое диапазоны и как их использовать;
- реализация цепочек преобразований диапазонов с использованием синтаксиса конвейеров;
- представления диапазонов и операции с ними;
- выполнение итераций без циклов `for`.

В главе 2 вы узнали, почему следует избегать простых циклов `for` и вместо них использовать обобщенные алгоритмы, предлагаемые библиотекой STL. Этот подход дает большие преимущества, но так же, как вы могли видеть, имеет некоторые недостатки. Алгоритмы из стандартной библиотеки трудно компоновать друг с другом, потому что главной целью их создателей было дать возможность реализовать более продвинутую версию алгоритма, применяя один алгоритм несколько раз.

Прекрасным примером может служить алгоритм `std::partition`, который перемещает все элементы коллекции, соответствующие предикату, в ее начало и возвращает итератор, ссылающийся на первый элемент в коллекции-результате, который не соответствует предикату. Используя этот алгоритм, можно написать функцию, которая разбивает коллекцию на несколько групп – не ограничиваясь предикатами, возвращающими `true` или `false`, – многократно вызывая `std::partition`.

Для примера реализуем функцию, которая группирует людей в коллекции по их принадлежности к той или иной команде. Она будет получать коллекцию людей, функцию, возвращающую название команды для

человека, и список команд. В этой функции можно вызвать `std::partition` несколько раз – по одному для каждой команды – и получить список людей, сгруппированных по командам.

### Листинг 7.1 Группировка людей по командам

```
template <typename Persons, typename F>
void group_by_team(Persons& persons,
                  F team_for_person,
                  const std::vector<std::string>& teams)
{
    auto begin = std::begin(persons);
    const auto end = std::end(persons);

    for (const auto& team : teams) {
        begin = std::partition(begin, end,
                               [&](const auto& person) {
                                   return team == team_for_person(person);
                               });
    }
}
```




Такой способ компоновки алгоритмов имеет право на существование, но иногда удобнее иметь возможность последовательно передавать коллекцию-результат от одной операции к другой. Вспомним пример из главы 2: у нас имелась коллекция людей, и мы должны были извлечь из нее имена женщин. Цикл `for`, решающий эту задачу, выглядит тривиально просто:

```
std::vector<std::string> names;

for (const auto& person : people) {
    if (is_female(person)) {
        names.push_back(name(person));
    }
}
```

Чтобы решить эту задачу с помощью алгоритмов STL, нужно создать промежуточную коллекцию для сохранения имен, соответствующих предикату (`is_female`), а затем использовать `std::transform`, чтобы извлечь имена женщин из коллекции. Это неоптимально с точки зрения производительности и памяти.

Главная проблема в том, что алгоритмы STL вместо самой коллекции принимают итераторы ее начала и конца. Это имеет несколько следствий:

- алгоритмы не могут вернуть коллекцию как результат;
- даже при наличии функции, возвращающей коллекцию, ее нельзя передать непосредственно в алгоритм: вам придется создать временную переменную, чтобы вызвать ее методы `begin` и `end`;
- по причинам, перечисленным выше, большинство алгоритмов изменяют свои аргументы и возвращают измененную коллекцию.

Эти факторы осложняют реализацию логики программ без использования хотя бы локальных изменяемых переменных.

## 7.1 Введение в диапазоны

Было несколько попыток решить эти проблемы, но наиболее жизнеспособной оказалась идея диапазонов. Давайте пока будем рассматривать диапазоны как простую структуру с двумя итераторами: один ссылается на первый элемент в коллекции, а другой – на мнимый элемент, следующий за последним.

**ПРИМЕЧАНИЕ** Диапазоны все еще не являются частью стандартной библиотеки, но предполагается, что они войдут в стандарт C++20. В качестве основы для добавления диапазонов в стандарт C++ предложена библиотека `range-v3`, созданная Эриком Ниблером (Eric Niebler), поэтому мы используем ее для примеров кода в этой главе. Существует также более старая, но проверенная временем библиотека – `Boost.Range`. Она содержит меньше функций, чем `range-v3`, зато поддерживает старые компиляторы. Обе библиотеки предлагают почти идентичный синтаксис и понятия, которые мы рассмотрим далее.

Какие преимущества дает хранение двух итераторов в одной структуре вместо использования их как двух отдельных значений? Главное преимущество: возможность вернуть полный диапазон из одной функции и передать в другую, не создавая локальных переменных для хранения промежуточных результатов.

Передача пар итераторов также имеет склонность к ошибкам. Например, можно передать итераторы, принадлежащие двум отдельным коллекциям, в алгоритм, который работает с одной коллекцией, или в неправильном порядке, когда первый итератор ссылается на элемент, следующий за вторым итератором. В обоих случаях алгоритм попытается обойти все элементы от начального итератора до конечного, что может привести к неопределенному поведению.

При использовании диапазонов предыдущий пример превращается в простую композицию функций `filter` и `transform`:

```
std::vector<std::string> names =
    transform(
        filter(people, is_female),
        name
    );
```

Функция `filter` возвращает диапазон элементов из коллекции `people`, соответствующих предикату `is_female`, а функция `transform` выполняет обход этого диапазона и возвращает имена людей в нем.

При желании можно вложить сколько угодно преобразований диапазонов, таких как `filter` и `transform`. Но возникает еще одна проблема: при

наличии более нескольких составных преобразований код становится очень громоздким.

По этой причине библиотеки диапазонов обычно предлагают специальный синтаксис *конвейера* (*pipe*), основанный на перегрузке оператора `|` и использовании его в роли аналогичного оператора командной оболочки UNIX. То есть вместо вложения вызовов функций друг в друга можно пропустить исходную коллекцию через серию преобразований:

```
std::vector<std::string> names = people | filter(is_female)
                                     | transform(name);
```



Так же как в предыдущем примере, здесь коллекция людей фильтруется предикатом `is_female` и из результата извлекаются имена. Основное отличие заключается в том, что здесь оператор `|` имеет смысл *пропустить через* вместо привычного *порядное ИЛИ*, благодаря чему код легче писать и рассуждать о нем, чем в предыдущем примере.

## 7.2 Создание представлений данных, доступных только для чтения

При взгляде на подобный код, как в предыдущем разделе, возникает вопрос: насколько он эффективен, по сравнению с простым циклом `for`, делающим то же самое. В главе 2 вы видели, что использование алгоритмов STL влечет снижение производительности из-за необходимости создавать новый вектор для хранения копий всех персон женского пола из исходной коллекции, чтобы получить возможность вызвать `std::transform`. Прочитав решение с диапазонами, вы можете подумать, что ничего не изменилось, кроме синтаксиса. В этом разделе я объясню, почему это не так.

### 7.2.1 Функция `filter` для диапазонов

Преобразование `filter` по-прежнему должно возвращать коллекцию людей, чтобы для нее можно было вызвать `transform`. Именно здесь в игру вступает магия диапазонов. Диапазон – это абстракция, представляющая коллекцию элементов, но никто не сказал, что это коллекция – он просто должен выглядеть и действовать как коллекция. Он должен иметь начало и конец и позволять добраться до каждого его элемента.

Вместо коллекции, такой как `std::vector`, функция `filter` будет возвращать структуру, описывающую диапазон, с итератором `begin` – «умным» прокси-итератором, который указывает на первый элемент в исходной коллекции, соответствующий заданному предикату, – и итератором `end` – прокси-итератором для итератора `end` исходной коллекции. Единственное, чем прокси-итератор отличается от итератора исходной коллекции, – он должен указывать только на элементы, соответствующие предикату фильтрации (см. рис. 7.1).

Проще говоря, каждый раз, когда выполняется наращивание прокси-итератора, он должен найти следующий элемент в исходной коллекции, соответствующий предикату.

### Листинг 7.2 Оператор инкремента для фильтрующего прокси-итератора

```
auto& operator++()
{
    ++m_current_position;
    m_current_position =
        std::find_if(m_current_position,
                    m_end,
                    m_predicate);
    return *this;
}
```

Итератор для фильтруемой коллекции. Когда выполняется инкремент прокси-итератора, нужно отыскать в коллекции первый элемент, удовлетворяющий предикату, который следует за текущим

Начать поиск со следующего элемента

Если элемент, удовлетворяющий предикату, не найден, вернуть итератор, ссылающийся на конец исходной коллекции, который также является концом отфильтрованного диапазона

При наличии фильтрующего прокси-итератора не требуется создавать временную коллекцию с копиями значений из исходной коллекции, соответствующих предикату. В результате мы получили новое *представление* существующих данных.

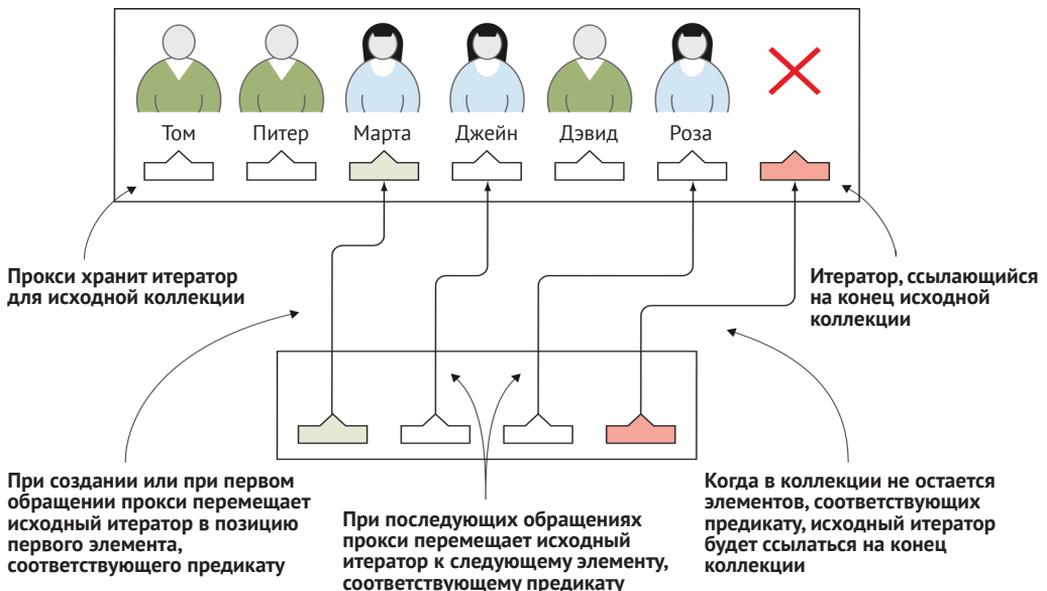


Рис. 7.1 Представление, созданное функцией `filter`, хранит итератор для исходной коллекции. Итератор может ссылаться только на элементы, соответствующие предикату. Это представление можно использовать как обычную коллекцию, содержащую только три элемента: Марта, Джейн и Роза

Это представление можно передать в алгоритм `transform`, и он обработает его как самую обычную коллекцию. Когда ему понадобится новое значение, он потребует от прокси-итератора переместиться на одну по-

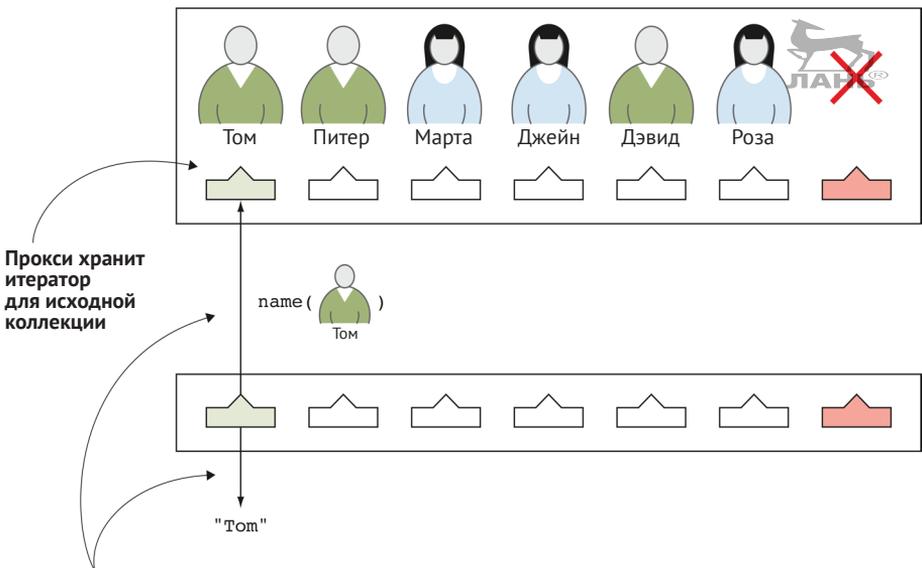
зицию вправо, который, в свою очередь, отыщет в исходной коллекции следующий элемент, соответствующий предикату. Алгоритм `transform` будет перебирать элементы в исходной коллекции, но *не будет видеть* персон неженского пола.



## 7.2.2 Функция `transform` для диапазонов

По аналогии с `filter` функция `transform` не обязана возвращать новую коллекцию. Она тоже может вернуть представление, основанное на существующих данных. В отличие от `filter` (которая возвращает новое представление, содержащее элементы из исходной коллекции, только не все из них), `transform` должна вернуть то же число элементов, что имеется в исходной коллекции, причем для каждого исходного элемента она должна вернуть его преобразованную версию.

От оператора инкремента не требуется ничего особенного; он должен просто наращивать итератор исходной коллекции. Но оператор разыменования итератора теперь будет реализован иначе. Он должен получить значение из исходной коллекции, применить к нему заданную функцию преобразования и вернуть результат (см. рис. 7.2).



При попытке разыменовать прокси-итератор он возвращает результат применения функции преобразования к элементу из исходной коллекции

Рис. 7.2 Представление, созданное алгоритмом `transform`, хранит итератор для исходной коллекции. Итератор перебирает все элементы в исходной коллекции, но представление возвращает не сами элементы, а результат применения функции преобразования к ним

### Листинг 7.3 Оператор разыменования прокси-итератора преобразования



```
auto operator*() const
{
    return m_function(
        *m_current_position
    );
}
```

Получить значение из исходной коллекции, применить функцию преобразования и вернуть результат как значение, на которое ссылается прокси-итератор

Это решение, как и в случае с `filter`, избавляет от необходимости создавать новую коллекцию для хранения преобразованных элементов. Оно создает представление, которое возвращает уже преобразованные элементы. Теперь получившийся диапазон можно передать в другое преобразование или скопировать в соответствующую коллекцию.

## 7.2.3 Ленивые вычисления с диапазоном значений

Даже если будет иметься два диапазона для преобразования – один в `filter` и один в `transform`, – для вычисления общего результата обход элементов исходной коллекции будет выполнен только один раз, как в цикле `for`, написанном вручную. Представления диапазонов вычисляются «лениво»: вызовы `filter` и `transform` просто определяют представления; они не выполняют никаких вычислений с участием элементов коллекции.

Давайте изменим пример и получим имена трех первых женщин в коллекции. Для этого можно использовать преобразование диапазона `take(n)`, которое создаст новое представление исходного диапазона, включающего только первые  $n$  элементов (или меньше, если число элементов в исходном диапазоне меньше  $n$ ):

```
std::vector<std::string> names = people | filter(is_female)
                                     | transform(name)
                                     | take(3);
```

Проанализируем работу этого фрагмента шаг за шагом.

- 1 Фрагмент `people | filter(is_female)` фактически ничего не делает, он просто создает новое представление. Он не обращается ни к каким элементам в коллекции `people`, а только инициализирует итератор, ссылающийся на первый элемент в исходной коллекции, соответствующий предикату `is_female`.
- 2 Созданное представление передается в `| transform(name)`. Единственное, что делает этот фрагмент, – создает новое представление. Этот код также не обращается к элементам коллекции и не вызывает их функцию `name`.
- 3 К полученному результату применяется код `| take(3)`. И снова он всего лишь создает новое представление.
- 4 Чтобы получить результат преобразования, из представления, возвращаемого фрагментом `| take(3)`, нужно сконструировать вектор строк.

Чтобы создать вектор, необходимы значения, которые будут помещаться в него. На этом шаге как раз и происходит обход всех элементов представления.

При попытке сконструировать вектор имен (`names`) на основе диапазона вычисляются все значения, попадающие в диапазон. Для каждого элемента, добавляемого в вектор, выполняются следующие действия (см. рис. 7.3):

- 1 вызывается оператор разыменования прокси-итератора, принадлежащего представлению диапазона, которое возвращается функцией `take`;
- 2 прокси-итератор диапазона, созданного функцией `take`, обращается к прокси-итератору диапазона, созданного функцией `transform`. Этот итератор, в свою очередь, обращается к прокси-итератору диапазона, созданного функцией `filter`;
- 3 выполняется попытка разыменовать прокси-итератор, созданный преобразованием `filter`. Он выполняет обход исходной коллекции, находит и возвращает первый элемент, соответствующий предикату `is_female`. Это самый первый раз, когда происходит обращение к содержимому коллекции, и первый раз, когда вызывается функция `is_female`;

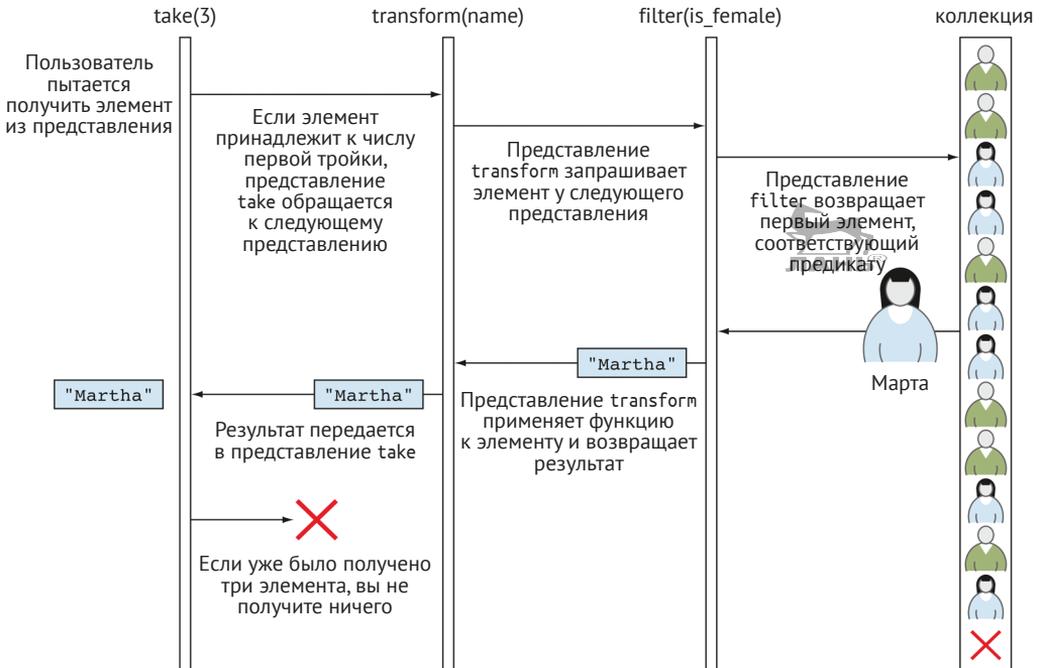


Рис. 7.3 При попытке получить элемент из представления прокси-итератор этого представления обращается к следующему представлению в комбинации преобразований или к самой коллекции. В зависимости от его типа представление может преобразовать результат, пропустить элементы, выполнить обход в каком-то ином порядке и т. д.

- 4 элемент, полученный обращением к прокси-итератору `filter`, передается в функцию `take`, и ее результат возвращается в прокси-итератор `take`, который передает его для вставки в вектор `names`.

После вставки элемента происходит переход к следующему, потом к следующему – и так до тех пор, пока не будет достигнут конец коллекции. Однако в данном случае, так как мы ограничили размер представления тремя элементами, нет необходимости продолжать перебирать элементы в коллекции `people` после извлечения третьего элемента.

Это был пример ленивых вычислений в действии. Даже притом что код получился более коротким и универсальным, чем эквивалентный ему цикл `for`, он делает все то же самое, и без ущерба для производительности.

## 7.3 Изменение значений с помощью диапазонов

Многие полезные преобразования можно реализовать как простые представления, но иногда требуется изменить содержимое самой коллекции. Такие преобразования принято называть не *представлениями*, а *действиями*, или *операциями*.

Типичным примером операции преобразования может служить сортировка. Чтобы отсортировать коллекцию, нужно обратиться ко всем ее элементам и переупорядочить их. При этом можно изменить саму исходную коллекцию или создать и сохранить ее отсортированную копию. Второй подход приобретает особую важность, когда исходная коллекция (например, связанный список) не поддерживает произвольного доступа к ее элементам и не может быть отсортирована достаточно эффективно; в такой ситуации обычно элементы копируются в новую коллекцию другого типа, поддерживающего произвольный доступ, которая затем сортируется.

### Представления и операции в библиотеке `range-v3`

Как отмечалось выше, в качестве основы для внедрения поддержки диапазонов в STL была предложена библиотека `range-v3`, поэтому в своих примерах мы будем использовать ее и ее номенклатуру. Преобразования диапазонов, создающие представления, такие как `filter`, `transform` и `take`, определены в пространстве имен `ranges::v3::view`, тогда как операции – в пространстве имен `ranges::v3::action`. Важно не путать эти два вида преобразований, поэтому начиная с этого момента мы будем указывать пространства имен `view` и `action`.

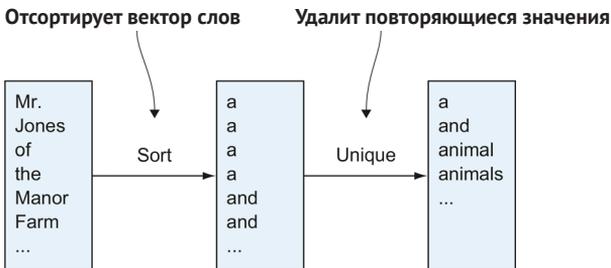
Представьте, что у вас есть функция `read_text`, возвращающая текст в виде вектора слов, и вам нужно отобрать все уникальные слова. Самый простой способ реализовать это – отсортировать слова и затем удалить дубликаты, следующие друг за другом. (Для простоты будем считать, что все слова состоят только из символов нижнего регистра.)

Получить список уникальных слов, появляющихся в заданном тексте, можно, применив к результату функции `read_text` операции `sort` и `unique`, как показано на рис. 7.4 и в следующем фрагменте кода:

```
std::vector<std::string> words =
    read_text() | action::sort
                | action::unique;
```



Так как результат функции `read_text` нигде больше не используется, можно сразу передать вектор в операцию `sort`, не создавая его копии для обработки. То же относится к `unique` – эту операцию можно применить непосредственно к результату операции `sort`. Если вы решите сохранить промежуточный результат, используйте преобразование `view::unique`, которое не изменяет исходной коллекции, а создает представление, пропускающее все повторяющиеся последовательные вхождения значения.



**Рис. 7.4** Чтобы получить список уникальных слов в тексте, достаточно отсортировать их и удалить повторяющиеся последовательные значения

Важно не путать представления и операции. Преобразования-представления создают ленивые представления исходных данных, тогда как операции выполняют действия с исходной коллекцией немедленно.

Операции могут применяться не только к временным значениям, но также к левосторонним (`lvalue`) значениям при помощи оператора `|=`, например:

```
std::vector<std::string> words = read_text();
words |= action::sort | action::unique;
```

Такое деление на представления и операции дает возможность выбирать, когда что-то должно выполняться лениво, а когда – немедленно. Ленивые вычисления особенно эффективны, когда преобразование должно применяться не ко всем элементам исходной коллекции и только один раз; немедленные вычисления предпочтительнее, когда требуется обработать все элементы и коллекция-результат будет использоваться неоднократно.

## 7.4 Ограниченные и бесконечные диапазоны

Эту главу мы начали с предположения, что диапазон – это структура, хранящая пару итераторов, один из которых ссылается на начало, а другой – на конец, как того требуют алгоритмы STL, но только в виде единой структуры. Итератор `end` самый необычный в этой паре. Его не требуется разыменовывать, потому что он ссылается на мнимый элемент, следующий за последним элементом. Кроме того, его крайне редко требуется перемещать, поскольку основное его назначение – служить индикатором конца коллекции:

```
auto i = std::begin(collection);
const auto end = std::end(collection);
for (; i != end; i++) {
    // ...
}
```

По большому счету он даже не должен быть итератором – это должно быть нечто, что можно использовать для проверки достижения конца диапазона. Это специальное значение называется *ограничителем* (sentinel) и дает больше свободы при реализации проверки достижения конца диапазона. Эта свобода почти не ощущается при работе с обычными коллекциями, и все ее преимущества проявляются только при создании ограниченных и бесконечных диапазонов.

### 7.4.1 Использование ограниченных диапазонов для оптимизации обработки входных диапазонов

*Ограниченный диапазон* – это диапазон, конец которого неизвестен заранее, но имеется функция-предикат, которая может сообщить, когда будет достигнут конец. Примерами таких ограниченных диапазонов являются строки с нулевым символом в конце или потоки ввода: вы продолжаете обход символов в строке, пока не встретится символ `'\0'`, или продолжаете выбирать лексемы по одной из потока ввода, пока поток не станет недействительным, то есть не вернет признак ошибки при попытке извлечь новую лексему. В обоих случаях вы знаете начало диапазона, но чтобы узнать, где находится конец, вы должны перебрать все элементы в диапазоне, пока проверка достижения конца не вернет `true`.

Давайте проанализируем код, который вычисляет сумму чисел, прочитанных из потока стандартного ввода:

```
std::accumulate(std::istream_iterator<double>(std::cin),
                std::istream_iterator<double>(),
                0);
```

Здесь создаются два итератора: первый – обычный итератор, представляющий начало коллекции чисел типа `double`, извлекаемых из `std::cin`, а второй – особый итератор, не принадлежащий никакому потоку ввода. Этот итератор представляет специальное значение, которое алгоритм

`std::accumulate` использует для проверки достижения конца коллекции; этот итератор играет роль ограничителя.

Алгоритм `std::accumulate` будет читать значения, пока итератор, управляющий обходом, не станет равным конечному итератору. Для итератора `std::istream_iterator` нужно реализовать `operator==` и `operator!=`. Оператор определения равенства должен уметь работать с обоими итераторами, обычными и особыми, играющими роль ограничителя. Вот как может выглядеть такая реализация:

```
template <typename T>
bool operator==(const std::istream_iterator<T>& left,
               const std::istream_iterator<T>& right)
{
    if (left.is_sentinel() && right.is_sentinel()) {
        return true;
    } else if (left.is_sentinel()) {
        // Проверить, вернет ли предикат ограничителя
        // true для итератора right
    } else if (right.is_sentinel()) {
        // Проверить, вернет ли предикат ограничителя
        // true для итератора left
    } else {
        // Оба итератора являются обычными итераторами,
        // проверить, ссылаются ли они оба
        // на один и тот же элемент коллекции
    }
}
```

Вы должны охватить все возможные варианты – когда роль ограничителя играет итератор `left`, то же самое для итератора `right`. Эти проверки выполняются на каждом шаге алгоритма.

Но это неэффективное решение. Было бы намного проще, если бы компилятор мог отличить ограничитель на этапе компиляции. Это можно сделать, если отменить требование, что ссылка на конец коллекции должна быть итератором, то есть разрешить, чтобы такая ссылка могла быть чем угодно, что можно сравнить с обычным итератором. При таком подходе четыре варианта в предыдущем коде превращаются в отдельные функции, и компилятор будет знать, какую следует вызвать в том или ином случае. Если в алгоритме используются два обычных итератора, он вызовет `operator==` для двух итераторов; если используются обычный итератор и ограничитель, он вызовет `operator==` для итератора и ограничителя; и т. д.

### Циклы `for` для диапазонов и ограничители

Цикл `for` для диапазона в C++11 и C++14 требует, чтобы `begin` и `end` принадлежали одному типу; они оба должны быть итераторами. Цикл `for` для диапазона нельзя использовать с ограниченными диапазонами в C++11 и C++14. Это требование было убрано в C++17. Теперь вы можете использовать `begin` и `end` разных типов, а это значит, что `end` может быть ограничителем.



## 7.4.2 Создание бесконечного диапазона с помощью ограничителя

Возможность использовать ограничитель позволяет оптимизировать работу с ограниченными диапазонами. Но это не предел возможностей: эта возможность позволяет также создавать бесконечные диапазоны. Бесконечные диапазоны не имеют конца, как диапазон всех целых положительных чисел. Он имеет начало – число 0, но не имеет конца.

Бесконечные диапазоны редко используются на практике, но иногда они очень удобны. Один из наиболее распространенных примеров – использование диапазона целых чисел для перечисления элементов в другом диапазоне. Представьте, что у вас есть коллекция фильмов, отсортированных по рейтингу, и вы хотите вывести первую десятку в стандартный вывод вместе с их порядковыми номерами, как показано в листинге 7.4 (см. `example:top-movies`).

Для этого можно использовать функцию `view::zip`. Она принимает два диапазона<sup>1</sup> и объединяет в пары элементы из этих диапазонов. Первый элемент в получившемся диапазоне будет представлен парой, содержащей первый элемент из первого диапазона и первый элемент из второго диапазона. Второй элемент будет представлен парой, содержащей второй элемент из первого диапазона и второй элемент из второго диапазона, и т. д. Объединение завершится, как только будет достигнут конец одного из исходных диапазонов (см. рис. 7.5).

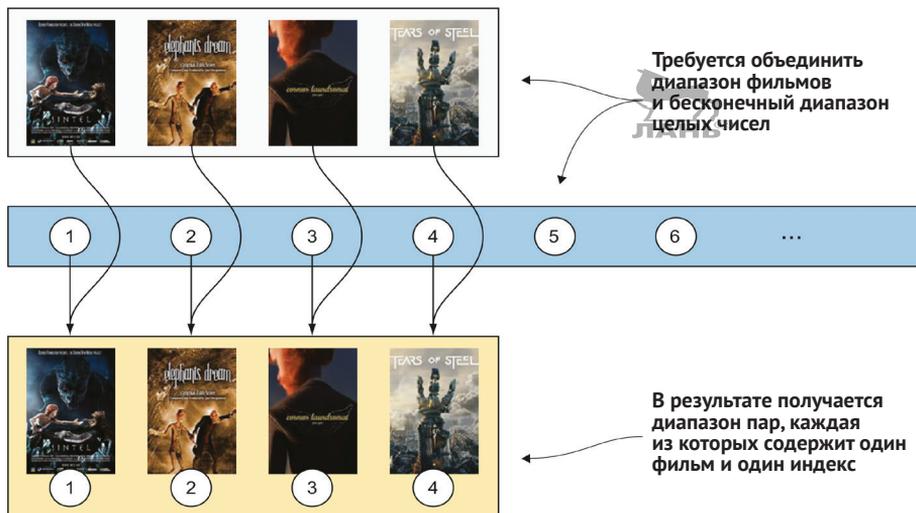


Рис. 7.5 Диапазон не поддерживает понятия «индекс элемента». Чтобы добавить индексы к элементам в диапазоне, можно объединить исходный диапазон с диапазоном целых чисел. В результате получится диапазон пар, каждая из которых содержит элемент из исходного диапазона и его индекс

<sup>1</sup> Функция `view::zip` может объединить большее число диапазонов. В результате вместо диапазона пар получится диапазон кортежей, каждый с размером  $n$ .

### Листинг 7.4 Вывод первой десятки фильмов с их порядковыми номерами

```

template <typename Range>
void write_top_10(const Range& xs)
{
    auto items =
        view::zip(xs, view::ints(1))
        | view::transform([](const auto& pair) {
            return std::to_string(pair.second) +
                " " + pair.first;
        })
        | view::take(10);
    for (const auto& item : items) {
        std::cout << item << std::endl;
    }
}

```

Объединит диапазон фильмов с диапазоном целых чисел, начиная с 1. В результате получится диапазон пар: название фильма и его индекс

Функция transform принимает пару и генерирует строку с индексом и названием фильма

Требуется получить только первые 10 фильмов

Вместо бесконечного диапазона целых чисел можно использовать числа из диапазона от 1 до `xs.length()`. Но это не универсальное решение. На практике нередко случаются ситуации, когда диапазон не знает своего размера, и вы не сможете определить его, не выполнив обход всех элементов. В таком случае обход придется выполнить дважды: один раз, чтобы узнать размер, и второй раз, чтобы вызвать `view::zip` и `view::transform`. Это не только неэффективно, но иногда и невозможно. Некоторые диапазоны, например представляющие потоки ввода, нельзя обойти дважды; прочитав значение из потока, вы не сможете прочитать его снова.

Еще одно преимущество бесконечных диапазонов не в том, чтобы использовать, а в том, чтобы писать код, способный их обрабатывать. Такой код получается более универсальным. Если вы написали алгоритм, способный обрабатывать бесконечные диапазоны, он сможет обрабатывать диапазоны любого размера, в том числе и диапазоны, размер которых заранее неизвестен.

## 7.5 Использование диапазонов для вычисления частоты слов

Давайте разберем более сложный пример, чтобы посмотреть, насколько более элегантный вид приобретает программа, если вместо кода в старом стиле использовать диапазоны. Мы повторно реализуем пример из главы 4, определяющий, как часто встречаются слова в текстовом файле. Напомним условия задачи: нам дается текст, и мы должны сохранить *n* наиболее часто встречающихся в нем слов. Разобьем задачу на несколько простых преобразований, как уже делали выше, но кое-что изменим, чтобы нагляднее продемонстрировать взаимодействия между представлениями и операциями с диапазонами.

Первым делом получим список слов в нижнем регистре и удалим любые специальные символы. Источником данных для нас послужит поток ввода, в данном примере – `std::cin`.

Библиотека `range-v3` предлагает шаблонный класс `istream_range`, который создает поток лексем из указанного потока ввода:

```
std::vector<std::string> words =
    istream_range<std::string>(std::cin);
```



В данном случае, так как необходимые нам лексемы имеют тип `std::string`, диапазон будет читать слова из стандартного потока ввода. Но этого недостаточно, потому что все слова должны быть в нижнем регистре и не содержать знаков препинания. То есть мы должны преобразовать каждое слово в нижний регистр и удалить любые не алфавитно-цифровые символы (см. рис. 7.6).

### Листинг 7.5 Получение списка слов в нижнем регистре, состоящих только из букв и цифр

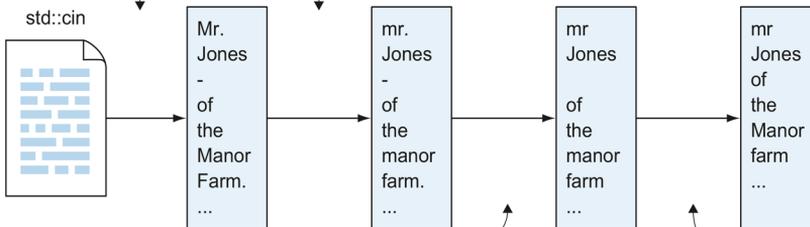
```
std::vector<std::string> words =
    istream_range<std::string>(std::cin)
    | view::transform(string_to_lower) ← Преобразовать слова в нижний регистр
    | view::transform(string_only_alnum) ← Оставить только буквы и цифры
    | view::remove_if(&std::string::empty); ←
```



Если лексема не содержит ни одной буквы или цифры, в результате может получиться пустая строка. Такую строку нужно пропустить

Создать диапазон для потока ввода, возвращающий диапазон слов

Преобразовать каждое слово в нижний регистр – `view::transform(to_lower)`



Игнорировать символы, не являющиеся ни буквами, ни цифрами – `view::filter(isalnum)`

Если лексема не содержит ни одной буквы или цифры, может получиться пустая строка. Такие строки должны пропускаться

В итоге получается очищенный список слов, пригодный для передачи остальной части алгоритма

Рис. 7.6 Слова читаются из потока ввода. Прежде чем подсчитывать частоту слов, нужно слова преобразовать в нижний регистр и удалить из них все знаки пунктуации

Исключительно ради полноты примера реализуем также функции `string_to_lower` и `string_only_alnum`. Первая представляет преобразование в нижний регистр каждого символа в строке, а вторая – фильтр, пропускающий не алфавитно-цифровые символы. Тип `std::string` – это коллекция символов, поэтому строками можно манипулировать, как любыми другими диапазонами:

```
std::string string_to_lower(const std::string& s)
{
    return s | view::transform(tolower);
}

std::string string_only_alnum(const std::string& s)
{
    return s | view::filter(isalnum);
}
```

Теперь у нас есть все слова для обработки, и следующим шагом мы должны отсортировать их (см. рис. 7.7). Преобразование `action::sort` требует, чтобы коллекция поддерживала произвольный доступ к элементам, так что нам повезло, что мы объявили `words` как вектор `std::vector` строк. Мы легко можем его отсортировать:

```
words |= action::sort;
```

Сгруппировать все повторяющиеся слова с помощью `view::group_by(std::equal_to<>())`. В результате получится диапазон диапазонов

Нас интересуют только сами слова и число их вхождений в текст; мы легко можем преобразовать группу в пару `(count, word)`

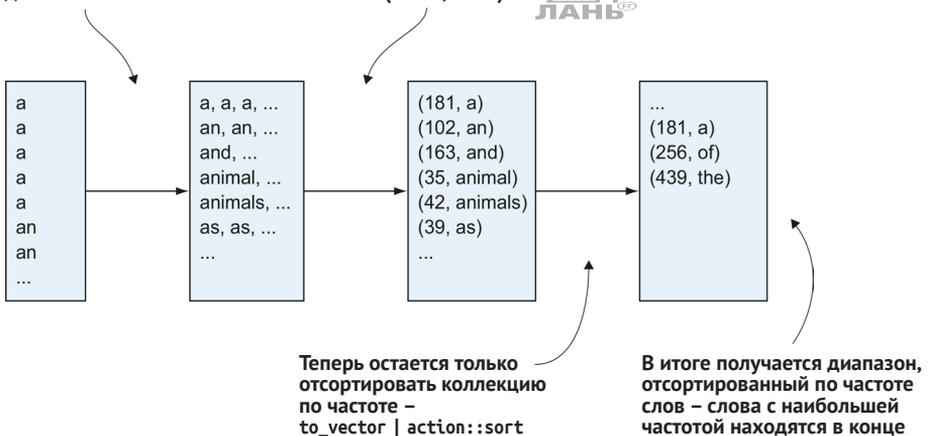


Рис. 7.7 Мы имеем отсортированный диапазон слов. Теперь нужно сгруппировать одинаковые слова, подсчитать их число в каждой группе и затем отсортировать группы по числу элементов в них

После сортировки списка мы легко можем сгруппировать одинаковые слова с помощью `view::group_by`. Эта функция создаст диапазон групп слов (группы – тоже, кстати, диапазоны). Каждая группа будет содержать

несколько экземпляров одного и того же слова – по числу вхождений этого слова в исходном тексте.

Диапазон можно преобразовать в пары; первый элемент в паре – число элементов в группе, а второй – слово. В результате получится диапазон, содержащий все слова из исходного текста и число их вхождений.

Так как частота является первым элементом в паре, мы можем передать этот диапазон в `action::sort`. Для этого можно использовать оператор `|=`, как в предыдущем фрагменте, или добавить преобразование в конвейер, предварительно преобразовав диапазон в вектор, как показано ниже (см. `example:word-frequency`). Это позволит объявить переменную `results` как `const`.

### Листинг 7.6 Создание сортированного списка пар (частота, слово) из сортированного списка слов

```
const auto results =
  words | view::group_by(std::equal_to<>())
        | view::transform([](const auto& group) {
            const auto begin = std::begin(group);
            const auto end = std::end(group);
            const auto count = distance(begin, end);
            const auto word = *begin;

            return std::make_pair(count, word);
        })
        | to_vector | action::sort;
```

Сгруппировать одинаковые слова в диапазоне words

Взять размер каждой группы и вернуть пару с количеством вхождений и словом

Чтобы отсортировать слова по частоте, диапазон нужно сначала преобразовать в вектор

Последний шаг – вывести  $n$  самых часто встречающихся слов в стандартный вывод. Так как результат был отсортирован в порядке возрастания, а нам нужны самые часто встречающиеся слова, нужно прежде перевернуть диапазон и затем взять  $n$  первых элементов:

```
for (auto value: results | view::reverse
      | view::take(n)) {
    std::cout << value.first << " " << value.second << std::endl;
}
```

Вот и все. Нам понадобилось написать меньше 30 строк кода, чтобы реализовать программу, прежде занимавшую десяток страниц. Мы создали набор легко комбинируемых компонентов и, не считая вывода, обошлись без циклов `for`.

### Циклы `for` для диапазонов и диапазоны

Как уже упоминалось, циклы `for` для диапазонов начали поддерживать ограничители в C++17. Предыдущий код не будет компилироваться старыми компиляторами. Если вы используете старый компилятор, применяйте макрос `RANGES_FOR`, предлагаемый библиотекой `range-v3` взамен цикла `for` для диапазонов:

```
RANGES_FOR (auto value, results | view::reverse
            | view::take(n)) {
    std::cout << value.first << " " << value.second << std::endl;
}
```

Кроме того, если диапазон слов сортировать так же, как список результатов (без operator |=), в программе не останется изменяемых переменных.

**СОВЕТ** Дополнительную информацию по теме, обсуждавшейся в этой главе, можно найти по адресу: <https://forums.manning.com/posts/list/43776.page>.

## Итоги

- Чаще всего ошибки, возникающие при использовании алгоритмов из STL, обусловлены передачей неправильных итераторов – иногда даже итераторов, принадлежащих разным коллекциям.
- Некоторые структуры, похожие на коллекции, не знают, где они заканчиваются. Для таких структур следует предоставлять итераторы-ограничители; они работают, но страдают неэффективностью.
- Идея диапазонов – это абстракция итеративных данных любого типа. Они могут моделировать обычные коллекции, потоки ввода и вывода, результаты запросов к базам данных и многое другое.
- Диапазоны планируются включить в C++20, но в настоящее время уже есть библиотеки, предлагающие такую же функциональность.
- Представления диапазонов не владеют данными и не могут их изменить. Если потребуется изменять существующие данные, используйте вместо представлений операции (действия).
- Бесконечные диапазоны – хорошая мера обобщенности алгоритма. Если алгоритм справляется с бесконечными диапазонами, он справится и с конечными.
- Используя диапазоны и исследуя логику выполнения с точки зрения преобразований диапазонов, программу можно разложить на компоненты многократного использования.



# Функциональные структуры данных

## **О чем говорится в этой главе:**

- причины повсеместного использования связанных списков в функциональных языках;
- общие данные в функциональных структурах данных;
- приемы работы с префиксными деревьями;
- сравнение стандартного вектора с его неизменяемым аналогом.



До сих пор я рассказывал в основном о высокоуровневых понятиях функционального программирования, и мы потратили довольно много времени на изучение преимуществ программирования без изменяемого состояния. Проблема в том, что программы имеют много движущихся частей. Обсуждая чистоту функций в главе 5, я сказал, что один из вариантов добиться ее – поддерживать изменяемое состояние только в главном компоненте. Все остальные компоненты – чистые, они лишь вычисляют изменения для применения главным компонентом, но сами ничего не изменяют. При подобном подходе все изменения производит главный компонент.

Такая организация обеспечивает ясное разделение между чистыми частями программы и частями, обслуживающими изменяемое состояние. Проблема в том, что спроектировать подобную архитектуру программного обеспечения часто очень непросто, потому что приходится учитывать порядок, в котором должны применяться вычисленные изменения. Стоит немного ошибиться, и можно столкнуться с состоянием



гонки за данные, подобным тому, что нередко наблюдается при работе с изменяемыми данными в многопоточной среде.

Поэтому иногда лучше вообще избегать любых изменений и даже отказываться от центрального изменяемого состояния. При использовании стандартных структур нужно каждый раз копировать данные, когда потребуется получить новую их версию. Всякий раз, когда понадобится изменить элемент в коллекции, нужно создавать новую коллекцию, содержащую все то же самое, что и прежняя коллекция, кроме одного изменившегося элемента. Но это очень неэффективно, если использовать структуры данных из стандартной библиотеки. В этой главе мы познакомимся со структурами, поддерживающими эффективные операции копирования и создания измененных копий.

## 8.1 *Неизменяемые связанные списки*

Одной из старейших структур, оптимизированных для подобного способа использования, является односвязный список. Он был положен в основу старейшего языка функционального программирования Lisp (сокращенно от List Processing – обработка списков). Связанные списки имеют низкую эффективность в большинстве задач, потому что имеют плохую локальность в памяти, которая плохо сочетается с механизмами кэширования современных процессоров. Зато они служат простейшим примером, как можно реализовать неизменяемую структуру, и являются идеальным образцом для знакомства с этой темой.

*Список* – это коллекция узлов; каждый узел хранит единственное значение и указатель на следующий узел в списке (nullptr, если это последний элемент в списке), как показано на рис. 8.1. Основные операции, которые поддерживают списки, – это добавление и удаление элемента в начале или в конце списка, вставка или удаление элементов в середине списка и изменение значения в конкретном узле.

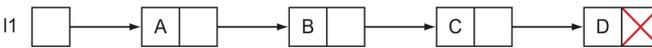


Рис. 8.1 Односвязный список. Каждый узел хранит значение и указатель на следующий элемент

Сосредоточимся на операциях добавления и удаления элемента в начале или в конце списка, потому что все остальные операции можно выразить через них. Имейте в виду, что когда я говорю об *изменении*, то подразумеваю *создание нового измененного списка на основе старого, при этом старый список остается неизменным*.

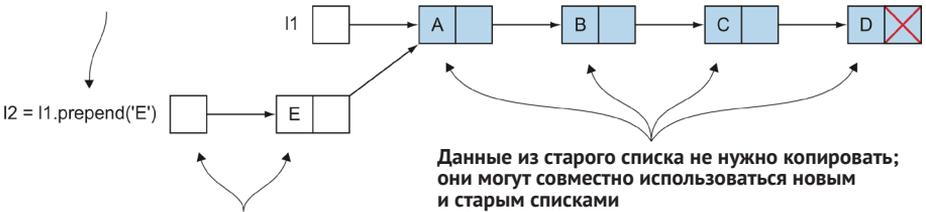
### 8.1.1 *Добавление и удаление элемента в начале списка*

Сначала поговорим об изменении начала списка. Если существующий список никогда не будет изменяться, тогда создание нового списка с заданным значением в начале превращается в тривиальную операцию:



достаточно создать новый узел со значением и указателем на голову ранее созданного списка (см. рис. 8.2). В результате получится два списка: старый, неизменный, и новый, содержащий все элементы старого списка, плюс новый элемент в начале.

Так как l1 – неизменяемый список, операция добавления нового элемента в начало вернет новый список

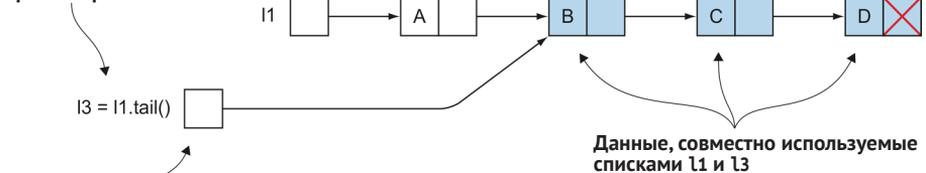


Для создания нового списка достаточно выделить память только для одного узла и сохранить в ней добавляемое значение

Рис. 8.2 Операция добавления нового элемента в начало неизменяемого списка возвращает новый список. Копировать данные из старого списка не требуется; они могут использоваться повторно. Это решение не подходит для случая изменяемых списков, потому что изменение общих данных в одном списке затронет другой

Удаление элемента из начала списка выполняется аналогично. Создастся новый указатель на начало списка, указывающий на второй элемент в исходном списке. И снова в результате получается два списка: старый и новый (см. рис. 8.3).

Хвост списка l1 содержит все элементы, кроме первого



Память выделять не требуется; достаточно записать указатель на второй узел списка l1 в l3

Рис. 8.3 Операция получения хвоста списка создает новый список из исходного, включающий те же элементы, кроме первого. Новый список может повторно использовать узлы старого списка, и ничего не нужно копировать

Эти две операции очень эффективны  $O(1)$  в отношении как памяти, так и времени выполнения. Они не требуют копировать данные, потому что оба списка вместе используют одни и те же узлы.

Важно отметить, что при наличии гарантий неизменности старого списка и в отсутствие функций, изменяющих вновь добавленные элементы, можно также гарантировать неизменность нового списка.

## 8.1.2 Добавление и удаление элемента в конце списка

В отличие от изменения элемента в начале списка, добавить или удалить элемент в конце списка намного сложнее. Обычно, чтобы добавить новый элемент в конец списка, нужно найти последний элемент и записать в него указатель на новый узел. Проблема в том, что такой подход не годится для неизменяемых списков (см. рис. 8.4).



Рис. 8.4 Попытавшись проделать тот же трюк, чтобы избежать копирования данных при добавлении нового элемента в конец списка, и просто записав в старый последний элемент указатель на новый узел, вы измените все списки, использующие общие данные с изменяемым списком. Обратите внимание, что если списки имеют общие узлы, они также имеют общий конец

Создав новый узел и записав указатель на него в последний узел старого списка, вы измените старый список. И не только старый список, но и все другие списки, имеющие общие данные с ним – в конце всех этих списков появится новый элемент.

Не существует эффективного способа добавить новый элемент в конец неизменяемого списка. Придется скопировать исходный список и добавить новый элемент в конец этой копии (см. рис. 8.5).

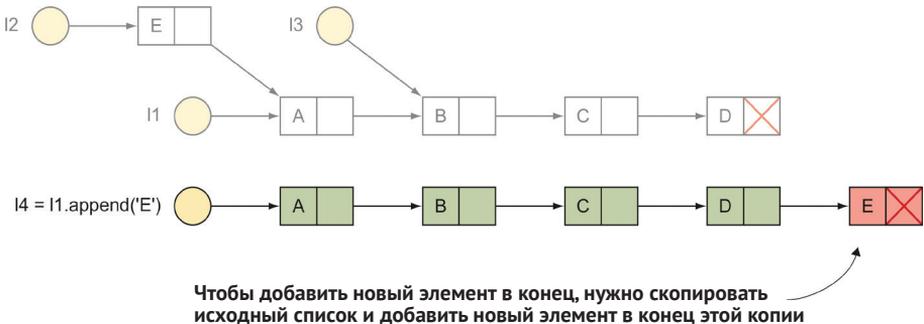


Рис. 8.5 Чтобы создать список, содержащий все то же, что и исходный список, плюс дополнительный элемент в конце, нужно скопировать все узлы из исходного списка и записать указатель на новый узел в последний элемент этой копии

То же относится и к операции удаления элемента в конце. Если выполнить удаление на месте, это приведет к изменению всех списков, имеющих общие узлы с текущим.

### 8.1.3 Добавление и удаление элемента в середине списка

Чтобы добавить или удалить элемент в середине списка, нужно скопировать все предшествующие элементы из исходного списка – в точности как при изменении конца списка (см. рис. 8.6), то есть все элементы, предшествующие точке изменения. В результате вы получите первую часть исходного списка, которую можно использовать повторно в новом списке. Затем вставить новый элемент и связать его с последующей частью старого списка.

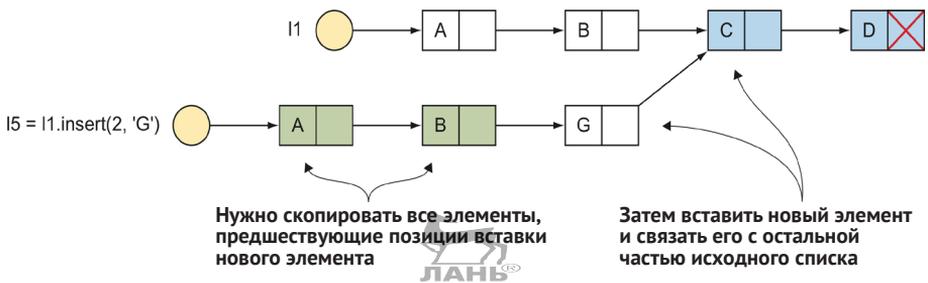


Рис. 8.6 Чтобы вставить элемент в середину списка, нужно скопировать из исходного списка все элементы, предшествующие позиции вставки, добавить в конец копии новый элемент и связать его с оставшейся частью исходного списка. Новый и старый списки будут совместно использовать только элементы, следующие за позицией вставки

Такая неэффективность односвязных списков позволяет эффективно использовать их только в роли стека, когда элементы добавляются и удаляются исключительно в начале списка (см. табл. 8.1).

Таблица 8.1 Сложность функций для работы с односвязными списками

$O(1)$	$O(n)$
Чтение первого элемента	Добавление в конец
Добавление в начало	Объединение (конкатенация)
Удаление первого элемента	Вставка в любую позицию, кроме первой
	Чтение последнего элемента
	Чтение $n$ -го элемента

### 8.1.4 Управление памятью

Одно из требований к реализациям списков – узлы должны размещаться в памяти динамически. Если определить список как структуру, хранящую только значение и указатель на другой список, можно столкнуться с проблемой, когда один из списков выйдет из области видимости и бу-

дет удален. Все списки, зависящие от этого списка, станут недействительными.

Кроме того, так как узлы размещаются динамически, нужен некоторый механизм, который удалял бы их из памяти, когда они становятся ненужными. Когда список выходит из области видимости, нужно освободить все узлы, принадлежащие только этому списку, и оставить узлы, совместно используемые другими списками. Например, когда l5 (на рис. 8.7) выйдет из области видимости, нужно удалить все узлы, предшествующие С, и оставить остальные в неприкосновенности.

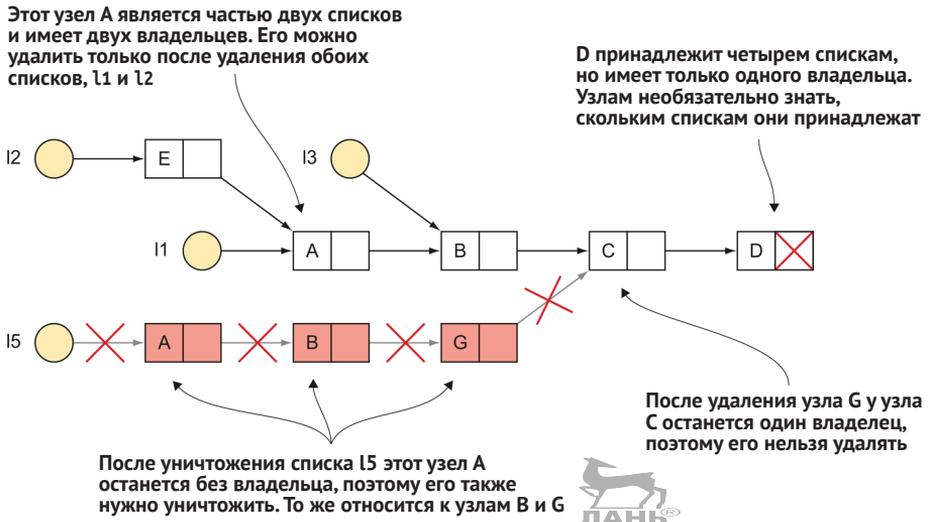


Рис. 8.7 При удалении списка нужно удалить все узлы, не используемые другими списками. Узлу необязательно знать, скольким спискам он принадлежит, но он должен знать, сколько прямых владельцев у него имеется. После удаления всех владельцев узла можно удалить и сам узел, потому что не останется ничего, что ссылалось бы на него

Подобный механизм реализуют «умные» указатели (smart pointers), что обеспечивает автоматическое освобождение памяти. Для простых связанных списков можно было бы использовать `std::unique_ptr`, потому что голова списка владеет его хвостом и хвост должен удаляться при удалении головы.

Но этот вариант не подходит в данном случае. Как показано на рис. 8.7, каждый узел может иметь несколько владельцев, если принадлежит нескольким спискам. По этой причине нужно использовать `std::shared_ptr`. Он хранит счетчик ссылок и автоматически уничтожает экземпляр объекта, на который указывает, когда счетчик уменьшается до нуля. В данном случае это подразумевает автоматическое удаление узла, если окажется, что он не принадлежит ни одному списку.

## Сборка мусора

Большинство функциональных языков полагаются на механизм сборки мусора, который автоматически освобождает память, ставшую ненужной. В C++ подобное поведение можно получить, если использовать «умные» указатели.

Для большинства структур данных наиболее очевидной является модель владения. Например, если говорить о списках, голова списка владеет остальной его частью (при этом она может быть не единственным владельцем, потому что одни и те же данные могут использоваться несколькими списками).

Вот как могла бы выглядеть простейшая структура списка с внутренним классом узла `node`:

```
template <typename T>
class list {
public:
    ...
private:
    struct node {
        T value;
        std::shared_ptr<node> tail;
    };

    std::shared_ptr<node> m_head;
};
```



Эта структура обеспечит своевременное удаление всех узлов, ставших ненужными после удаления экземпляра `list`. Экземпляр `list` проверит, является ли единственным владельцем головного узла, и удалит его, если это так. Затем деструктор узла проверит, является ли единственным владельцем узла, представляющего хвост, и удалит его, если это так. Так будет продолжаться до первого узла, у которого обнаружится еще один владелец, или до конца списка.

Проблема со всеми этими деструкторами в том, что они вызываются рекурсивно, и если удаляемый список окажется достаточно большим, может произойти исчерпание стека. Чтобы избежать этого, нужно реализовать нестандартный деструктор, преобразующий рекурсию в плоский цикл.

### Листинг 8.1 Преобразование рекурсивных вызовов деструкторов в плоский цикл

```
~node()
{
    auto next_node = std::move(tail);
    while (next_node) {
        if (!next_node.unique()) break;
    }
}
```

Вызов `std::move` обязателен. Иначе `next_node.unique()` никогда не вернет `true`

Если данный узел не является единственным владельцем следующего узла, ничего не делать

```

std::shared_ptr<node> tail;
swap(tail, next_node->tail);
next_node.reset();
next_node = std::move(tail);
}
}

```

Удалить узел; рекурсивный вызов здесь не произойдет, потому что мы записали nullptr в указатель на его хвост

Спрятать хвост обрабатываемого узла, чтобы предотвратить рекурсивный вызов деструктора

std::move здесь можно не вызывать, но этот вызов повышает производительность

Если этот код потребуется использовать в многопоточной среде, его нужно доработать. Несмотря на то что подсчет ссылок в `std::shared_ptr` ведется потокобезопасным способом, некоторые вызовы в этой реализации все же придется изменить.

## 8.2 Неизменяемые векторы

Из-за своей низкой эффективности списки подходят лишь для узкого круга задач. В большинстве же случаев нужна какая-то другая структура, поддерживающая эффективные операции и быстрый поиск.

`std::vector` является отличным кандидатом на эту роль как обладающий следующими достоинствами, в основном потому, что хранит все элементы в одном блоке памяти:

- быстрый доступ к элементам по индексам, поскольку для заданного индекса легко вычислить местоположение соответствующего элемента в памяти – достаточно просто прибавить индекс к адресу первого элемента в векторе;
- при извлечении значения из памяти многие современные процессоры извлекают не только само значение, но целый фрагмент памяти, окружающей его, и сохраняют его в своем кеше, чтобы увеличить скорость доступа к соседним элементам. При выполнении итераций по значениям в `std::vector` это оказывается весьма кстати: при обращении к первому элементу процессор извлечет сразу несколько элементов, что позволит ему прочесть последующие элементы намного быстрее;
- даже в самых тяжелых случаях, когда возникает необходимость перераспределить память для вектора и скопировать или переместить имеющиеся данные в новое местоположение, вектор не утрачивает преимуществ, которые дает хранение всех данных в одном непрерывном блоке памяти, потому что современные системы оптимизированы для перемещения блоков памяти.

Проблема с `std::vector` в том, что если использовать его в качестве неизменяемой структуры данных, при каждом изменении придется копировать все данные. Нам нужна какая-то другая альтернатива, обладающая максимальным количеством преимуществ `std::vector`.

Одной из популярных альтернатив являются *префиксные деревья*<sup>1</sup> (bit-mapped vector trie). Эта структура данных была предложена Риком Хикки (Rick Hickey) для языка программирования Clojure (который, в свою очередь, многое заимствовал из статьи<sup>2</sup> Фила Багвелла (Phil Bagwell)).

### Копирование при записи

Мы часто передаем объекты функциям, которые никак не изменяют их. Копирование объекта при передаче в функцию, которая его не изменяет, – непростительное расточительство.

Копирование при записи (Copy-On-Write, COW), или ленивое копирование, – это оптимизация, откладывающая создание копии данных до момента, пока кто-то не попытается их изменить. Когда вызывается конструктор копирования или оператор присваивания, объект просто сохраняет ссылку на исходные данные.

Когда пользователь вызывает любую функцию-член, изменяющую объект, и мы не можем позволить изменить исходные данные, мы должны создать копию и изменить ее. Изменить сами исходные данные можно, только если никакой другой объект не ссылается на них.

Более подробное описание механизма копирования при записи можно найти в книге Герба Саттера (Herb Sutter) «More Exceptional C++» (Addison-Wesley Professional, 2001)<sup>3</sup>.

Первоначально структура имеет вид COW-вектора, максимальный размер которого ограничен предопределенным числом  $m$ . Для максимальной эффективности число  $m$  должно быть степенью 2 (причину вы узнаете далее). Большинство реализаций используют число  $m = 32$ , но, чтобы упростить обсуждение, установим предел, равный 4.

До достижения максимального размера структура действует как самый заурядный COW-вектор (см. рис. 8.8). После заполнения вектора, при попытке вставить следующий элемент, создается новый вектор с тем же ограничением максимального размера, и новый элемент становится его первым элементом. Теперь в структуре имеется два вектора: один полный и один, содержащий единственный элемент. При добавлении новых элементов они будут помещаться во второй вектор, пока тот не заполнится, после чего будет создан новый вектор.



Рис. 8.8 Если число элементов меньше или равно  $m$ , префиксное дерево содержит только один COW-вектор

<sup>1</sup> Иногда их называют растровыми векторными деревьями. – Прим. перев.

<sup>2</sup> Фил Багвелл (Phil Bagwell). Ideal Hash Trees. 2001. <https://lampwww.epfl.ch/papers/idealhashtrees.pdf>.

<sup>3</sup> Саттер Герб, Красиков Игорь. Решение сложных задач на C++. Вильямс, 2008. ISBN: 978-5-8459-1971-7. – Прим. перев.



Для объединения векторов (потому что логически они являются частью одной структуры) создается вектор векторов: вектор, хранящий указатели на, самое большее,  $t$  подвекторов, хранящих сами данные (см. рис. 8.9). Когда вектор верхнего уровня заполняется, добавляется новый уровень. Если в структуре хранить не более  $t$  элементов, она будет иметь только один уровень. Если в структуре хранить не более  $t \times t$  элементов, она будет иметь два уровня. Если в структуре хранить не более  $t \times t \times t$  элементов, она будет иметь три уровня (см. рис. 8.10) и т. д.

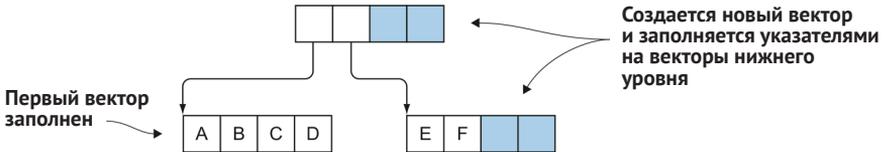


Рис. 8.9 Когда начальный вектор с емкостью  $t$  оказывается заполненным, создается еще один вектор с емкостью  $t$ , в который записывается новый элемент. Для хранения указателей на векторы с данными создается индекс: еще один вектор с емкостью  $t$ . В результате получается двухуровневое дерево

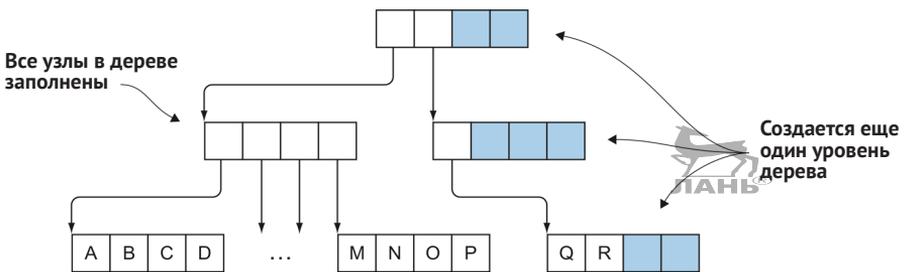


Рис. 8.10 После заполнения вектора индекса, при попытке добавить еще один элемент, создается еще один индекс и вектор для хранения данных. А чтобы не потерять ссылки на индексы, создается новый корневой вектор, куда сохраняются указатели на индексы

## 8.2.1 Поиск элементов в префиксном дереве

Эта структура представляет собой дерево особого вида, в котором значения хранятся в листьях. Узлы дерева, не являющиеся листьями, не хранят данных; они хранят указатели на векторы уровнем ниже. Все листья сортируются по индексам. Самый левый лист всегда содержит значения с индексами от 0 до  $t - 1$ ; следующий лист содержит значения с индексами от  $t$  до  $2t - 1$  и т. д.

Я уже отмечал, что эта структура обеспечивает быстрый поиск по индексам. Так как выполняется поиск элемента с индексом  $i$ , например?

Здесь все просто. Если имеется только один уровень – единственный вектор, – тогда  $i$  представляет индекс элемента в векторе. Если имеется два уровня, каждому индексу в векторе верхнего уровня соответствует  $t$  элементов (каждый элемент этого вектора хранит указатель на вектор

с  $m$  элементами). Первый вектор-лист хранит элементы с индексами до  $m - 1$ ; второй – элементы с индексами от  $m$  до  $2m - 1$ ; третий – элементы с индексами от  $2m$  до  $3m - 1$  и т. д. То есть легко можно определить, в каком листе находится  $i$ -й элемент: в листе с индексом  $i/m$  (аналогично определяются индексы для более высоких уровней).

Индекс элемента можно рассматривать не как число, а как массив битов. Эту маску можно разбить на фрагменты. В результате получится путь от корневого узла к искомому элементу: каждый фрагмент – это индекс следующего дочернего узла, а последний фрагмент – это индекс элемента в листовом векторе (см. рис. 8.11).

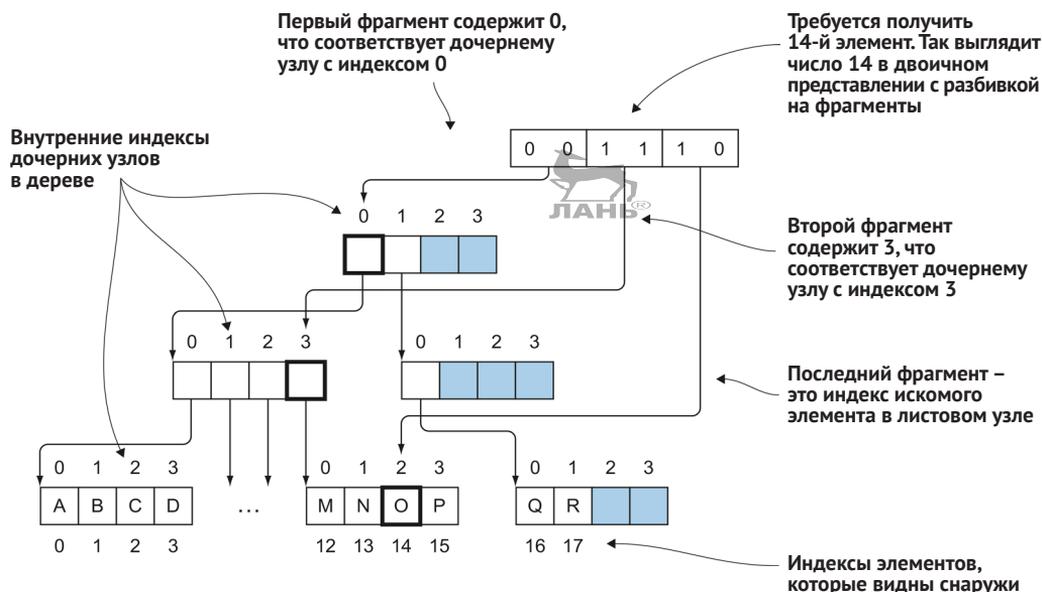


Рис. 8.11 Поиск элемента с заданным индексом осуществляется просто. Каждый уровень имеет индексы от 0 до  $m$ , где  $m$  должно быть степенью 2 (на этом рисунке  $m = 4$ , однако на практике чаще выбирается число 32). Внутренние индексы, если рассматривать индекс как массив битов, изменяются от 00 до 11. Индекс элемента можно рассматривать как массив битов и разбить его на фрагменты по два бита в каждом. Каждый фрагмент определяет внутренний индекс в соответствующем узле дерева

Данный подход обеспечивает логарифмическую сложность поиска, где  $m$  – основание логарифма. Обычно для  $m$  выбирается число 32, благодаря чему в большинстве случаев дерево получается достаточно мелким. Например, дерево с глубиной 5 может хранить 33 млн элементов. Вследствие небольшой глубины и ограниченности системной памяти на практике поиск в деревьях с  $m = 32$  или больше выполняется практически за постоянное время ( $O(1)$ ). Даже если заполнить всю адресуемую память одной коллекцией, глубина дерева не превысит 13 уровней. (Точное число не так важно, важен сам факт, что это число фиксированное.)

## 8.2.2 Добавление элементов в конец префиксного дерева

Теперь, посмотрев, как происходит поиск элементов в префиксном дереве, перейдем к следующему вопросу: как изменять его. В случае изменяемой структуры не возникает никаких сложностей; нужно отыскать первое свободное место и записать в него новый элемент. Если все листья заполнены, следует создать новый лист.

Однако нам нужно обеспечить сохранность прежних версий. Для этого можно использовать тот же подход, что применялся при работе со связанными списками: создать новое дерево, использующее как можно больше данных совместно с исходным деревом.

Если сравнить три дерева на рис. 8.12, можно заметить, что они не сильно отличаются друг от друга. Основные отличия сосредоточены в листьях. Поэтому копировать нужно только листья, затронутые изменениями. Затем для измененного листа нужно создать новое дерево, которое станет его владельцем, нового владельца для этого владельца и т. д.

То есть чтобы добавить элемент, нужно скопировать не только листовую узел, но также весь путь от верхнего уровня до добавленного элемента (см. рис. 8.13).

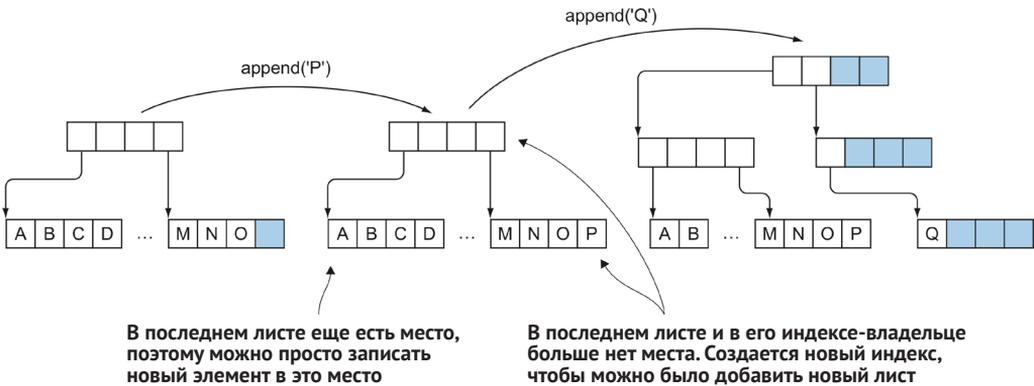


Рис. 8.12 В изменяемом префиксном дереве добавление нового элемента осуществляется просто. Если в последнем листовом узле есть свободное место, новое значение сохраняется в него. Если все листья заполнены, создается новый лист. Указатель на новый лист нужно сохранить в узле уровнем выше. Следующие действия выполняются рекурсивно: если в родительском узле есть место, указатель записывается туда, иначе создается новый узел и указатель на него сохраняется в узле уровнем выше. Так повторяется до достижения корневого узла. Если в корневом узле нет места, создается новый корневой узел, который становится родителем предыдущего корневого узла

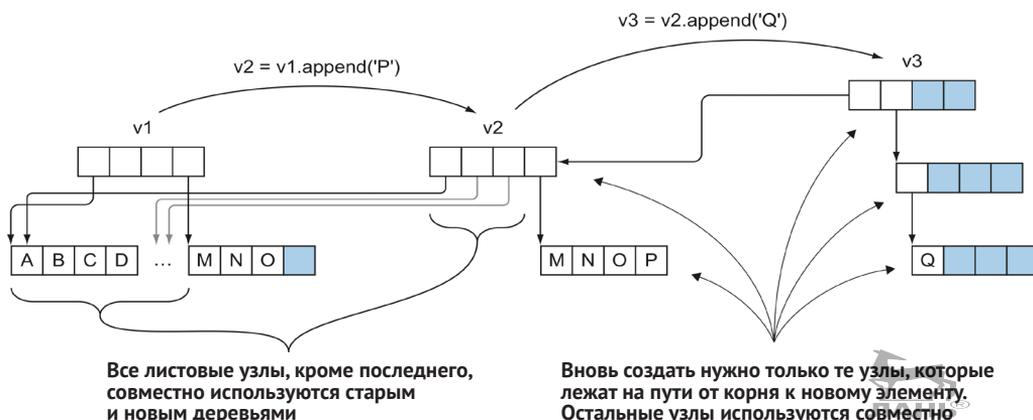


Рис. 8.13 В неизменяемом префиксном дереве используется та же механика добавления нового элемента. Разница лишь в том, что для новых данных нужно создать новое дерево, потому что исходная коллекция должна оставаться неизменной. К счастью, как и в случае со связанными списками, старое и новое деревья могут совместно использовать большую часть данных. Создать требуется лишь узлы, которые лежат на пути от корня к новому элементу. Все остальные узлы остаются общими

Нужно предусмотреть действия в следующих случаях:

- в последнем листе имеется место для нового элемента;
- в любых нелистовых узлах есть место для нового элемента;
- все узлы заполнены.

Первый случай самый простой. Достаточно скопировать путь до листового узла и сам листовой узел и вставить новый элемент непосредственно во вновь созданный листовой узел (см. рис. 8.14).

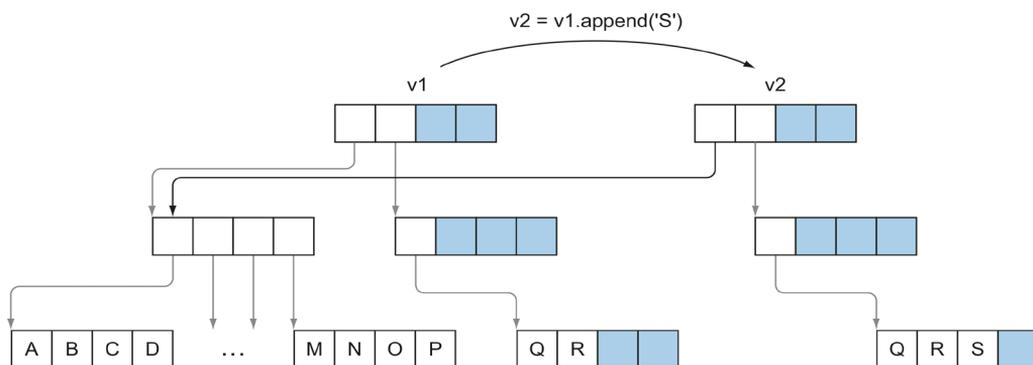


Рис. 8.14 Когда элемент добавляется в дерево, где имеется место в листовом узле, нужно скопировать этот узел и всех его родителей и добавить новый элемент во вновь скопированный листовой узел

Во втором случае нужно найти самый нижний уровень с узлом, где имеется свободное место. Создать все уровни ниже, вплоть до нового листового узла, куда будет добавлен новый элемент (см. рис. 8.15).

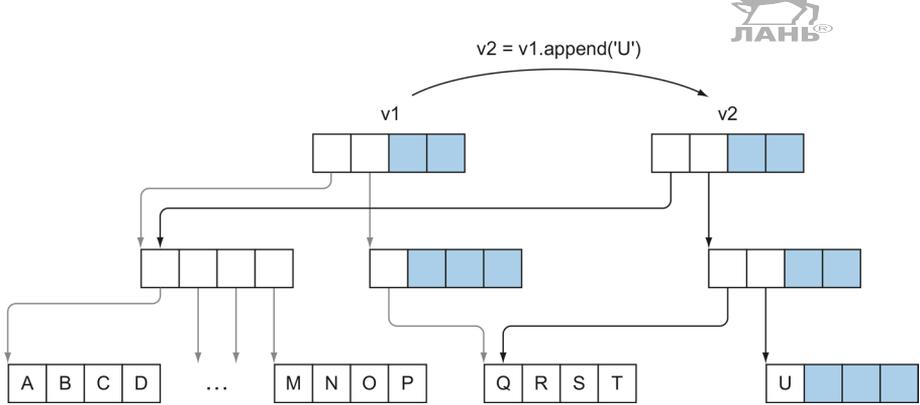


Рис. 8.15 Когда в последнем листовом узле нет свободного места, нужно создать новый листовый узел, содержащий только что добавленный элемент. Указатель на вновь созданный листовый узел следует вставить в узел уровнем выше

Третий случай самый сложный. Если в дереве нет ни одного узла со свободным местом, сохранить новый элемент под текущим корневым узлом не получится. Нужно создать новый корневой узел и сделать предыдущий корень его первым элементом. После этого возникает ситуация, соответствующая второму случаю: появляется нелистовой узел, в котором имеется свободное место (см. рис. 8.16).

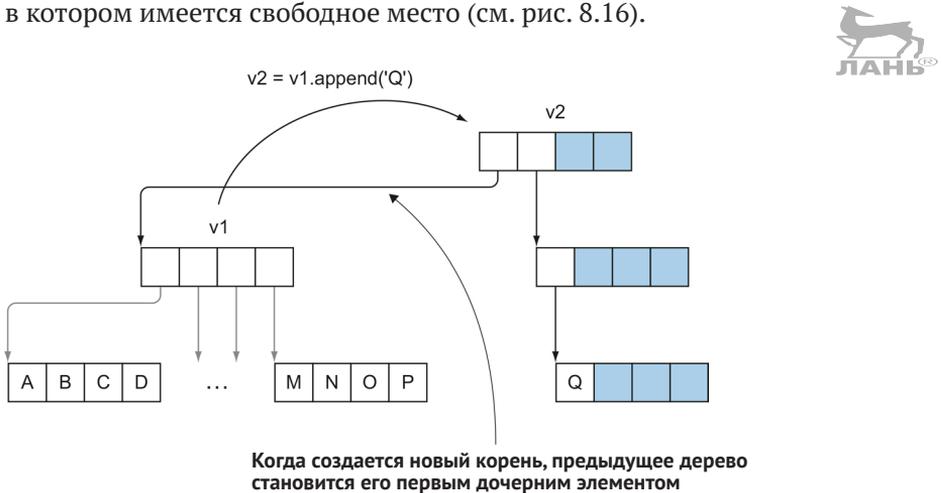


Рис. 8.16 Это особый случай, когда ни в одном из листьев и ни в одном из других узлов, включая корень, нет свободного места. В этом случае нельзя создать новый путь внутри дерева и нужно создать новый корневой узел, который будет ссылаться на предыдущее и на новое дерево с той же глубиной, содержащее только что добавленный элемент

Насколько эффективна операция добавления нового элемента? В ходе выполнения она либо копирует узлы, встречающиеся в пути, либо выделяет новые, либо, иногда, создает новый корневой узел. Все эти действия сами по себе имеют постоянное время выполнения (каждый узел имеет практически постоянное число элементов). Если рассуждать так же, как мы рассуждали, когда исследовали поиск, нетрудно прийти к выводу, что копировать или создавать всегда приходится практически постоянное число узлов. То есть добавление нового элемента в структуру тоже имеет сложность  $O(1)$ .



### 8.2.3 *Изменение элементов в префиксном дереве*

Теперь, узнав, как добавлять новые элементы, давайте посмотрим, как их изменять. И снова поразмышляем, что изменится в структуре, если сделать ее изменяемой и изменить элемент в определенной позиции.

Чтобы изменить элемент, сначала нужно найти лист, в котором он находится. В этом случае ничего создавать не нужно; структура дерева остается той же самой, просто в ней изменяется единственное значение.

Изменение элемента похоже на первый случай добавления нового элемента: когда в последнем листе имеется свободное место. Новое дерево может использовать все данные вместе со старым деревом, кроме пути к измененному элементу.

### 8.2.4 *Удаление элемента из конца префиксного дерева*

Удаление элемента из конца напоминает добавление в конец, только с точностью до наоборот. Нужно предусмотреть действия в следующих случаях:

- последний лист содержит более одного элемента;
- последний лист содержит только один элемент;
- в корне останется единственный элемент после удаления элемента с данными.

В первом случае нужно скопировать путь к листу, содержащему удаляемый элемент, и удалить элемент из вновь созданного листа.

Во втором случае в листе имеется только один элемент. Во вновь созданном дереве этот лист должен отсутствовать. И снова нужно скопировать полный путь и удалить пустые узлы с конца. Удалить нужно сам лист, его родительский узел, если в нем не осталось ни одного дочернего узла, а также его родителя и т. д.

Если после удаления пустых узлов с конца в корневом узле останется единственный элемент, его нужно сделать новым корневым узлом, чтобы уменьшить глубину дерева.

### 8.2.5 *Другие операции и общая эффективность префиксных деревьев*

Вы увидели, как изменять элементы в дереве, добавлять их в конец и удалять из конца. Вы также увидели, что все эти операции имеют высо-

кую эффективность. А что можно сказать об операциях вставки элемента в начало или в произвольное место в дереве? А о конкатенации?

К сожалению, эти операции не имеют эффективной реализации (см. табл. 8.2). Добавление в начало и вставка требуют сместить все остальные элементы на одну позицию вправо. А конкатенация требует создать достаточное количество листовых узлов и скопировать их из объединяемой коллекции.

**Таблица 8.2 Сложность функций для работы с префиксными деревьями**

O(1)	O(n)
Чтение элемента по индексу	Добавление в начало
Добавление в конец	Объединение (конкатенация)
	Вставка в любую позицию



Те же самые сложности, что мы видели, когда обсуждали `std::vector`. Изменение существующего элемента, а также добавление и удаление элементов в конце коллекции выполняются за постоянное время; но вставка и удаление элементов в любой другой позиции выполняются за время, линейно зависящее от размера коллекции.

Однако даже при одинаковой сложности алгоритмов префиксные деревья все равно будут немного уступать в скорости вектору `std::vector`. Чтобы прочитать элемент, требуется пересечь несколько уровней косвенности, и промахи кеша будут случаться чаще, потому что элементы хранятся не в одном непрерывном блоке памяти.

Единственное, когда префиксные деревья оказываются производительнее, – копирование, что для нас весьма немаловажно.

Итак, у нас есть структура, имеющая производительность, сопоставимую с производительностью `std::vector`, и при этом оптимизированная для использования в роли неизменяемой структуры; вместо изменения существующих значений мы всегда можем достаточно быстро создать немного модифицированную копию оригинала.

В предыдущих главах я уже говорил о неизменяемости данных и о том, как использовать характерные особенности C++, такие как `const`, для реализации более безопасного и компактного кода. Стандартные классы коллекции в STL не оптимизированы для такого использования. Они предназначены для применения в изменчивой манере и не поддерживают безопасного использования разными компонентами, если не копировать их, что очень неэффективно.

Структуры, представленные в этой главе, особенно префиксное дерево, лежат на другом конце спектра. Они спроектированы так, чтобы обеспечить максимальную эффективность при использовании в чисто функциональном стиле: когда исходные данные никогда не изменяются после инициализации. Да, их производительность чуть ниже, зато они позволяют копировать себя невероятно быстро.

**СОВЕТ** Дополнительную информацию по теме, обсуждаемой в этой главе, можно найти по адресу: <https://forums.manning.com/posts/list/43777.page>. Кроме того, отличное введение в неизменяемые структуры данных вы найдете в книге Криса Окасаки (Chris Okasaki) «Purely Functional Data Structures»<sup>1</sup> (Cambridge University Press, 1999).

## Итоги

- Основной оптимизацией для всех неизменяемых структур данных, которые также называются *постоянными* (persistent), является совместное использование общих данных. Он позволяет хранить несколько версий одних и тех же данных, немного отличающихся друг от друга, без лишних затрат памяти, как при использовании классов из стандартной библиотеки.
- Наличие эффективного способа хранения прошлых значений переменной позволяет путешествовать по истории их изменений. Если сохранять все предыдущие состояния программы, всегда можно будет вернуться к любому из них.
- Подобные версии существуют и для других структур. Например, в роли неизменяемого ассоциативного контейнера можно использовать красно-черные деревья, приспособленные для совместного использования общих данных разными экземплярами, по аналогии с префиксными деревьями.
- Неизменяемые структуры данных всегда были предметом активных исследований. Многие находки в этой области были сделаны в научных кругах (как это часто бывает с функциональным программированием), а также самими разработчиками.
- Всегда проверяйте, какие структуры лучше подходят для каждого конкретного случая. В отличие от лучшей коллекции в C++ (`std::vector`), все неизменяемые структуры данных имеют свои недостатки (даже если они столь же великолепны, как префиксные деревья). Некоторые из них могут не подходить в какой-либо конкретной ситуации.

---

<sup>1</sup> Окасаки Крис. Чисто функциональные структуры данных. М.: ДМК Пресс, 2016. ISBN: 978-5-97060-233-1, 978-0-5216635-0-2. – Прим. перев.



# Алгебраические типы данных и сопоставление с образцом

## О чем говорится в этой главе:

- удаление недопустимых состояний из программ;
- использование алгебраических типов данных;
- обработка ошибок с использованием необязательных значений и вариантов;
- создание перегруженных объектов функций;
- обработка алгебраических типов данных сопоставлением с шаблонами.



Выше в этой книге мы рассмотрели несколько проблем, связанных с состоянием программы. Вы видели, как проектировать программное обеспечение, не имеющее изменяемого состояния, и как реализовать структуры данных, обеспечивающие эффективное копирование. Но мы рассмотрели далеко не все проблемы, имеющие отношение к состоянию программы, например неожиданные или недействительные состояния.

Представьте следующую ситуацию: у вас есть приложение, которое подсчитывает общее количество слов в веб-странице. Программа может начать отсчет, как только получит первый фрагмент веб-страницы. Она имеет три основных состояния:

- *начальное состояние* – процесс подсчета еще не начался;
- *состояние счета* – получена часть веб-страницы, и счет начался;
- *конечное состояние* – веб-страница получена полностью, и все слова подсчитаны.

При реализации чего-то подобного обычно создается класс, хранящий все необходимые данные. В итоге получается класс, включающий дескриптор, с помощью которого осуществляется доступ к веб-странице (сокет или поток данных), счетчик, содержащий количество слов, и флаги, указывающие состояние процесса (начало, выполнение, завершение).

Вот как могла бы выглядеть такая структура:

```
struct state_t {
    bool started = false;
    bool finished = false;
    unsigned count = 0;
    socket_t web_page;
};
```

Первое, что приходит на ум, – `started` и `finished` не обязательно должны быть отдельными флагами типа `bool`, их можно заменить перечислением, содержащим три значения: `init`, `running` и `done`. Если сохранять их как отдельные значения, открывается возможность получить недопустимое состояние, например когда `started` имеет значение `false`, а `finished` – `true`.

Те же проблемы могут возникнуть с другими переменными. Например, значение `count` никогда не должно быть больше нуля, если `started` имеет значение `false`, и никогда не должно изменяться после того, как `finished` получит значение `true`; сокет `web_page` должен быть открыт, только если `started` имеет значение `true`, а `finished` – значение `false`, и т. д.

При замене значений `bool` перечислением с тремя возможными значениями уменьшается количество состояний, в которых может находиться программа, и устраняются некоторые недопустимые состояния. То же полезно проделать для других переменных.

## 9.1 Алгебраические типы данных

В функциональном мире создание новых типов на основе старых обычно выполняется с помощью двух операций: суммы и произведения (поэтому такие новые типы называются *алгебраическими*). Произведение двух типов `A` и `B` – это новый тип, содержащий экземпляр `A` и экземпляр `B` (то есть декартово произведение множества всех значений типа `A` и множества всех значений типа `B`). В примере подсчета слов в веб-странице тип `state_t` – это произведение двух типов `bool`, одного типа `unsigned` и одного типа `socket_t`. Аналогично, произведение более двух типов – это новый тип, тип-произведение, содержащий экземпляр каждого типа, участвующего в произведении.

Тип-произведение – привычная ситуация в C++. Каждый раз, когда требуется объединить несколько типов в один, мы создаем новый класс либо используем `std::pair` или `std::tuple`, если имена членов не имеют значения.

## Пары и кортежи

Типы `std::pair` и `std::tuple` – это удобные универсальные типы для создания производений типов на скорую руку. Тип `std::pair` является производением двух типов, тогда как `std::tuple` может включать столько типов, сколько вы пожелаете.

Одной из особенно полезных особенностей пар и кортежей является автоматическая поддержка операторов сравнения. Многие часто реализуют операторы сравнения для своих классов через сравнение кортежей. Например, если имеется класс, содержащий имя и фамилию человека, и требуется реализовать для него оператор «меньше, чем», сравнивающий сначала фамилию, а затем имя, это можно сделать так:

```
bool operator<(const person_t& left, const person_t& right)
{
    return std::tie(left.m_surname, left.m_name) <
           std::tie(right.m_surname, right.m_name);
}
```

Функция `std::tie` создает кортеж ссылок на переданные ей значения. При создании кортежа данные не копируются и сравниваются оригинальные строки.

Единственная проблема пар и кортежей: значения, хранящиеся в них, не имеют имен. Увидев вызов функции, возвращающей `std::pair<std::string, int>`, вы не сможете сказать, что означают эти значения. Если бы возвращаемое значение было структурой с переменными-членами `full_name` и `age`, такой проблемы не возникло бы.

По этой причине желательно реже использовать пары и кортежи, и только для локальных вычислений. Делать их частью публичного API (даже притом что это не редкость в STL) считается дурным тоном.

Суммы типов, или типы-суммы, не так заметны в C++, как произведения. Сумма типов A и B – это тип, который может содержать экземпляр A или экземпляр B, но не оба одновременно.

## Перечисления как суммы типов

Перечисления – это особый вид суммы типов. Перечисление определяется как список возможных значений. Экземпляр типа перечисления может содержать только одно из этих значений. Если рассматривать эти значения как множества с одним элементом, перечисление является суммой типов этих множеств.

Суммы типов можно рассматривать как обобщение перечислений, когда при определении суммы типов вместо множеств с одним элементом указываются множества с произвольным числом элементов.

Итак, в нашем примере имеется три основных состояния: начальное состояние, состояние счета и конечное состояние. В начальном состоянии структура не должна содержать никакой дополнительной информации, в состоянии счета должны поддерживаться счетчик и дескриптор для доступа к веб-странице, а в конечном состоянии должно иметься только окончательное количество слов. Поскольку все состояния являются взаимоисключающими, общее состояние программы можно смоделировать, определив тип – сумму из трех типов, по одному типу для каждого состояния.

### 9.1.1 Определение типов-сумм через наследование

C++ предлагает несколько способов реализации типов-сумм. Один из них – создание иерархии классов; в этом случае определяется суперкласс, представляющий тип-сумму, и производные классы, представляющие типы-слагаемые.

Чтобы представить три состояния в нашем примере программы, можно создать суперкласс `state_t` и три подкласса – по одному для каждого из основных состояний: `init_t`, `running_t` и `finished_t`. Класс `state_t` можно оставить пустым, потому что он нужен только как основа; в программе будет иметься переменная с указателем на тип `state_t`, ссылающаяся на экземпляры подклассов. Для проверки текущего состояния можно использовать `dynamic_cast`. Как вариант, поскольку динамическое приведение работает медленно, можно добавить в суперкласс целочисленный тег, или индикатор, который поможет различать подклассы.

#### Листинг 9.1 Суперкласс с тегом, определяющий тип-сумму через наследование

```
class state_t {
protected:
    state_t(int type)
        : type(type)
    {
    }
public:
    virtual ~state_t() {};
    int type;
};
```

Чтобы экземпляр этого класса нельзя было создать непосредственно, его конструктор объявлен защищенным (`protected`). Его смогут вызвать только классы, наследующие `state_t`

При создании экземпляра каждого подкласса передается свое значение аргумента `type`. Его можно использовать как более эффективную замену для `dynamic_cast`

В листинге 9.1 определен класс, экземпляр которого нельзя создать непосредственно. Его можно использовать только как описатель экземпляров его подклассов. Теперь добавим подклассы, по одному для каждого состояния (листинг 9.2).

## Листинг 9.2 Типы, обозначающие разные состояния

```

class init_t : public state_t {
public:
    enum { id = 0 };
    init_t()
        : state_t(id)
    {
    }
};

class running_t : public state_t {
public:
    enum { id = 1 };
    running_t()
        : state_t(id)
    {
    }

    unsigned count() const
    {
        return m_count;
    }

    ...

private:
    unsigned m_count = 0;
    socket_t m_web_page;
};

class finished_t : public state_t {
public:
    enum { id = 2 };
    finished_t(unsigned count)
        : state_t(id)
        , m_count(count)
    {
    }

    unsigned count() const
    {
        return m_count;
    }

private:
    unsigned m_count;
};

```

Класс, представляющий начальное состояние, не хранит никаких данных; в нем нет ни обработчика веб-страницы, ни счетчика слов. Он должен лишь записать в поле `type` свой идентификатор `id` (ноль)

В состоянии «счет» должен иметься счетчик слов и обработчик веб-страницы, откуда будут извлекаться слова

Когда счет завершится, обработчик веб-страницы станет ненужным. Но счетчик слов должен остаться



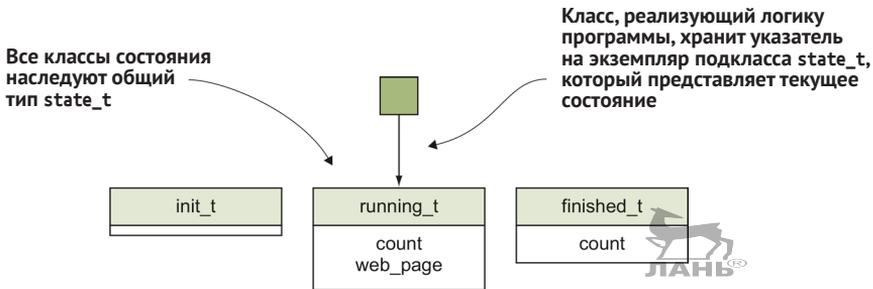


Рис. 9.1 Реализовать тип-сумму с использованием наследования очень просто. Достаточно создать несколько классов, наследующих общий класс state\_t, а в программе для хранения текущего состояния предусмотреть указатель, ссылающийся на экземпляр одного из этих классов

### Листинг 9.3 Основная программа

```
class program_t {
public:
    program_t()
        : m_state(std::make_unique<init_t>())
    {
    }
    ...
    void counting_finished()
    {
        assert(m_state->type == running_t::id);

        auto state = static_cast<running_t*>(
            m_state.get());

        m_state = std::make_unique<finished_t>(
            state->count());
    }
private:
    std::unique_ptr<state_t> m_state;
};
```

Начальное состояние представляет экземпляр init\_t. Переменная m\_state никогда не должна содержать NULL

В момент завершения счета программа должна находиться в состоянии running\_t. Если вы не можете гарантировать обязательное выполнение этого условия, используйте if-else вместо assert

В настоящий момент точный тип экземпляра, на который указывает m\_state, известен, поэтому можно использовать статическое приведение типа

Переключение в новое состояние, хранящее конечный результат. Экземпляр, представлявший предыдущее состояние, уничтожается

При таком подходе программа не сможет оказаться в недопустимом состоянии. Счетчик не сможет быть больше нуля, если счет еще не начался (в данном случае счетчика просто не существует). Счетчик не может случайно измениться после завершения процесса подсчета, и мы в любой момент точно знаем, в каком состоянии находится программа.

Более того, нет необходимости обращать внимание на срок использования ресурсов, приобретаемых для конкретных состояний. Возьмем, к примеру, сокет web\_page. В первоначальной версии, когда все необходимые переменные находились в структуре state\_t, мы могли забыть

закрывать сокет, закончив читать содержимое веб-страницы из него. Экземпляр сокета мог бы продолжать существовать до тех пор, пока существует экземпляр `state_t`. При использовании типа-суммы все ресурсы, необходимые для определенного состояния, будут автоматически освобождаться после перехода в другое состояние. В данном случае сокет `web_page` будет закрыт деструктором `gunning_t`.

При реализации типов-сумм с использованием наследования получают открытые типы. То есть состояние может быть экземпляром любого класса, наследующего `state_t`, что упрощает возможность расширения спектра возможных состояний. Иногда это полезно, но чаще точно известно, какие состояния может иметь программа, и желательно запретить другим компонентам динамически расширять набор состояний.

Подход на основе наследования также имеет несколько недостатков. Если понадобится сохранить его открытость, придется использовать виртуальные функции и динамическую диспетчеризацию (по крайней мере, для деструкторов), теги типов, чтобы не полагаться на медленное динамическое приведение, и динамически распределять память для объектов, представляющих состояния, из кучи. Также придется позаботиться о том, чтобы указатель `m_state` никогда не мог получить недействительное значение (`nullptr`).

## 9.1.2 Определение типов-сумм с использованием объединений и `std::variant`



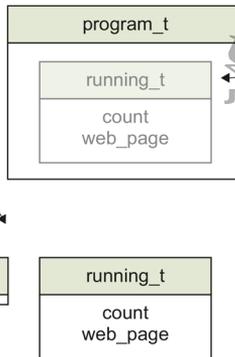
Альтернативный подход к определению типов-сумм основан на использовании типа `std::variant`, обеспечивающем безопасную реализацию объединений. С помощью `std::variant` можно определить закрытый тип-сумму – тип, экземпляр которого может содержать только те типы, которые вы указали.

При использовании наследования для реализации типов-сумм тип переменной-члена `m_state` является (умным) указателем на `state_t`. Сам тип ничего не сообщает о возможных состояниях; `m_state` может указывать на любой объект, класс которого наследует `state_t`.

При использовании `std::variant` все типы, которые могут применяться для представления состояния программы, должны быть явно указаны при определении переменной `m_state`; они будут закодированы в ее типе. Чтобы расширить такой тип-сумму, необходимо изменить определение типа, что было необязательно в решении на основе наследования (см. рис. 9.2).

**СОВЕТ** Тип `std::variant` был введен в C++17. Если вы пользуетесь более старой версией компилятора и более старой реализацией STL, не поддерживающей возможности C++17, вместо этого типа можно использовать тип `boost::variant`.

Классы состояний – это произвольные типы; в данном случае не требуется ни наследование, ни динамическое размещение в памяти



Тип-сумма `variant<...>` достаточно большой, чтобы вместить экземпляр любого из типов-слагаемых

**Рис. 9.2** Для определения требуемых типов-сумм можно использовать варианты. В результате получается тип значения, способный вместить любой из типов-слагаемых. Экземпляр варианта будет занимать места в памяти ровно столько, сколько занимает наибольший из типов-слагаемых, независимо от конкретного типа значения, хранимого в данный момент

Для реализации состояния программы с помощью `std::variant` можно повторно использовать определения классов `init_t`, `running_t` и `finish_t`, но теперь не требуется, чтобы они наследовали общий класс, и не требуется хранить в них целочисленный тег, определяющий конкретный тип:

```
class init_t {
};

class running_t {
public:
    unsigned count() const
    {
        return m_count;
    }

    ...

private:
    unsigned m_count = 0;
    socket_t m_web_page;
};

class finished_t {
public:
    finished_t(unsigned count)
        : m_count(count)
    {
    }

    unsigned count() const
    {
        return m_count;
    }
};
```





```
private:
    unsigned m_count;
};
```

Здесь мы избавились от шаблонного кода, требовавшегося для правильной работы наследования (точнее, динамического полиморфизма). Класс `init_t` теперь пустой, потому что в этом состоянии программа не хранит ничего. Классы `running_t` и `finished_t` определяют только свои данные и ничего больше. Теперь основная программа может иметь значение `std::variant`, способное хранить экземпляр любого из этих трех типов (см. рис. 9.3).



Рис. 9.3 Программа подсчета количества слов в веб-странице имеет три основных состояния. Пока нет веб-страницы, программа ничего не может сделать. Получив веб-страницу, она может начать подсчет. В процессе подсчета программа должна хранить текущий счетчик и дескриптор веб-страницы. По завершении подсчета дескриптор веб-страницы становится ненужным, но счетчик с окончательным результатом нужно сохранить

#### Листинг 9.4 Основная программа, использующая `std::variant`

```
class program_t {
public:
    program_t()
        : m_state(init_t())
    {
    }
    ...
    void counting_finished()
    {
        auto* state = std::get_if<running_t>(&m_state);
        assert(state != nullptr);
        m_state = finished_t(state->count());
    }
};
```

Начальное состояние представлено экземпляром `init_t`

Использовать `std::get_if` для проверки типа экземпляра, хранящегося в `std::variant`. Возвращает `nullptr`, если вариант хранит экземпляр другого типа

Чтобы изменить состояние, достаточно присвоить новое значение переменной

```
private:  
    std::variant<init_t, running_t, finished_t> m_state;  
};
```

Здесь (листинг 9.4) мы инициализировали `m_state` значением типа `init_t`. Обратите внимание, что вместо указателя (`new init_t()`), как в реализации на основе наследования, мы присваиваем фактическое значение. Тип `std::variant` не использует динамический полиморфизм; он не хранит указатели на объекты в куче. Он хранит фактический объект непосредственно в себе, как самое обычное объединение. Разница лишь в том, что он автоматически создает и уничтожает объекты, хранящиеся в нем, и точно знает тип объекта, который хранит в каждый конкретный момент времени.

При обращении к значению, хранящемуся в `std::variant`, можно использовать функции `std::get` и `std::get_if`. Обе позволяют обращаться к элементам варианта по типу или по индексу типа в объявлении:

```
std::get<running_t>(m_state);  
std::get_if<1>(&m_state);
```

Разница лишь в том, что `std::get` возвращает значение или возбуждает исключение, если вариант не содержит значения указанного типа, тогда как `std::get_if` возвращает указатель на значение или пустой указатель как признак ошибки.

Реализация на основе `std::variant` имеет множество преимуществ перед предыдущим решением. В ней почти отсутствует шаблонный код, потому что тип состояния определяется самим экземпляром `std::variant`. Здесь также не нужно создавать иерархию наследования, в которой все типы-слагаемые должны наследовать супертип; вместо этого можно использовать существующие типы, такие как строки и векторы. Кроме того, `std::variant` не использует динамическую память. Экземпляр варианта, как и обычное объединение, имеет размер, соответствующий размеру наибольшего типа, который он должен хранить (плюс несколько байтов для внутренних нужд).

Единственный недостаток – сложность расширения `std::variant`. Чтобы добавить в сумму новый тип, придется изменить определение типа варианта. В тех редких случаях, когда нужна расширяемость, используйте типы-суммы на основе наследования.

**ПРИМЕЧАНИЕ** Вместо открытых типов-сумм на основе наследования можно использовать тип `std::any`. Это типобезопасный контейнер, способный хранить значения любого типа. Его удобно использовать в некоторых обстоятельствах, но он не так эффективен, как `std::variant`, поэтому не следует применять этот тип как простую замену типа `std::variant`.

### 9.1.3 Реализация конкретных состояний

Теперь, когда у нас есть класс `program_t` и классы, представляющие возможные состояния, можно приступить к реализации логики. Возникает

вопрос: где должна находиться логика? Можно пойти самым очевидным путем и реализовать всю логику в `program_t`, но тогда в каждой функции вам придется проверить, в правильном ли состоянии находится программа. Например, функция в классе `program_t`, запускающая процесс подсчета, сначала должна проверить, находится ли программа в начальном состоянии `init_t`, и только потом изменять состояние на `running_t`.

Для этого можно вызвать функцию-член `index` класса `std::variant`. Она вернет индекс текущего типа, экземпляр которого хранится в варианте. Поскольку `init_t` – это первый тип, указанный в определении типа переменной-члена `m_state`, ему соответствует индекс 0:

```
void count_words(const std::string& web_page)
{
    assert(m_state.index() == 0);

    m_state = running_t(web_page);

    ... // подсчитать число слов

    counting_finished();
}
```



Проверив текущее состояние, функция переключает состояние на `running_t` и запускает процесс подсчета. Подсчет выполняется синхронно (приемы параллельного и асинхронного выполнения мы обсудим в главах 10 и 12), и когда он закончится, мы сможем обработать результат.

Как можно заметить в предыдущем фрагменте, для перевода программы в состояние `finished_t` вызывается функция `counting_finished`. Она также должна проверить текущее состояние. Функция `counting_finished` должна вызываться, только когда программа находится в состоянии `running_t`. Нам нужен результат подсчета, поэтому вместо `index` используем `std::get_if` и проверим результат:

```
void counting_finished()
{
    const auto* state = std::get_if<running_t>(&m_state);

    assert(state != nullptr);

    m_state = finished_t(state->count());
}
```

В состоянии `finished_t` нужно запомнить только подсчитанное количество слов.

Обе эти функции изменяют состояние программы, поэтому им самое место в `program_t`, даже притом, что им приходится выполнять проверку состояния. Но что, если функция должна выполнять логику, которая относится только к определенному состоянию?

Например, после перехода в состояние `running_t` программа должна открыть поток для извлечения содержимого веб-страницы, прочитать этот поток, слово за словом, и подсчитать количество слов. Эта логика

не меняет состояния программы. Она вообще не должна ничего знать о других состояниях; она просто выполняет свою работу, обрабатывая имеющиеся данные.

По этой причине нет никакого смысла помещать такую логику в класс `program_t`. Наиболее подходящее для нее место – класс `running_t`:

```
class running_t {
public:
    running_t(const std::string& url)
        : m_web_page(url)
    {
    }

    void count_words()
    {
        m_count = std::distance(
            std::istream_iterator<std::string>(m_web_page),
            std::istream_iterator<std::string>());
    }

    unsigned count() const
    {
        return m_count;
    }

private:
    unsigned m_count = 0;
    std::istream m_web_page;
};
```



Этот подход рекомендуется использовать при разработке программ с состоянием на основе типов-сумм: логику, связанную с одним состоянием, поместите внутрь объекта, определяющего это состояние, а логику, которая изменяет состояние, поместите в основную программу. Полная реализация этой программы, которая работает с файлами вместо веб-страниц, доступна в примере `word-counting-states`.

Альтернативный подход – поместить всю логику в классы состояний. Этот подход избавляет от необходимости проверять состояние с помощью `get_if` и `index`, потому что функция-член класса, представляющего определенное состояние, будет вызываться только в том случае, если программа находится в данном состоянии. Недостаток такого решения – все классы состояний должны изменять состояние программы, то есть они должны знать о существовании друг друга, что является нарушением принципа инкапсуляции.

### 9.1.4 Особый тип-сумма: необязательные значения

Я уже упоминал функцию `std::get_if`, которая возвращает указатель на значение, если оно существует, или пустой указатель в противном случае. Указатель используется, чтобы обозначить особый случай, когда значение отсутствует.

Указатели могут иметь много разных значений в разных ситуациях и ничего не сообщают о том, для чего они используются. Столкнувшись с функцией, которая возвращает указатель, возможны несколько вариантов:

- если это фабричная функция, возвращающая указатель на созданный ею объект, вы становитесь владельцем этого объекта;
- функция может вернуть указатель на существующий объект, которым вы не владеете;
- функция может потерпеть неудачу и сообщить об этом, вернув пустой указатель.

Чтобы узнать, какой из этих случаев имеет место с вызываемой функцией, такой как `std::get_if`, нужно заглянуть в документацию с ее описанием.

Вместо этого часто лучше использовать типы, которые четко сообщают результат функции. В первом случае лучше заменить указатель на `std::unique_ptr`. Во втором случае подойдет `std::shared_ptr` (или `std::weak_ptr`).

Третий случай – особенный. Нет никаких веских причин, по которым результат функции должен быть указателем. Единственная причина, почему он – указатель, а не обычное значение, заключается в *расширении* исходного типа состоянием «нет значения», то есть, чтобы обозначить, что значение является необязательным – оно может существовать, но может быть неопределенным.

Можно сказать, что необязательное значение некоторого типа `T` является либо значением типа `T`, либо *пустым* значением. То есть значение представляет сумму всех значений типа `T` плюс еще одно значение, обозначающее отсутствие значения. Этот тип-сумму можно легко реализовать с помощью `std::variant`:

```
struct nothing_t {};
```

```
template <typename T>
using optional = std::variant<nothing_t, T>;
```

Теперь, увидев функцию, которая возвращает `optional<T>`, вы будете знать, что можете вернуть значение типа `T` или *ничего*. Вам не придется помнить о времени жизни результата или о необходимости его уничтожить. Вы не сможете забыть проверить, является ли он пустым, как в случае с указателями.

Необязательные значения оказались настолько полезными, что в стандартную библиотеку был включен тип `std::optional` – более удобный, чем созданный нами псевдоним `optional` для `std::variant`.

**СОВЕТ** Тип `std::optional`, как и `std::variant`, был введен в версии C++17. Если у вас более старый компилятор, используйте тип `boost::optional`.

Поскольку `std::optional` является более конкретным типом-суммой, чем `std::variant`, он также имеет более удобный API и функции-члены,



помогающие проверить наличие значения, оператор `->` для доступа к значению, если оно существует, и т. д.

Ниже показано, как реализовать свою функцию `get_if`, использующую `std::get_if` и проверяющую действительность результата. Она проверяет указатель и возвращает само значение, заключенное в экземпляр `std::optional`, если указатель действительный, и пустое необязательное значение в противном случае:

```
template <typename T, template Variant>
std::optional<T> get_if(const Variant& variant)
{
    T* ptr = std::get_if<T>(&variant);

    if (ptr) {
        return *ptr;
    } else {
        return std::optional<T>();
    }
}
```

Тот же подход можно применить к любой функции, возвращающей пустой указатель, чтобы сообщить об отсутствии значения. Теперь, написав свой вариант `get_if`, возвращающий необязательное значение, мы можем переопределить функцию `counting_finished`:

```
void counting_finished()
{
    auto state = get_if<running_t>(m_state);

    assert(state.has_value());

    m_state = finished_t(state->count());
}
```

Код выглядит почти так же, как в версии с использованием указателей. Наиболее очевидное отличие: в первой версии утверждение `assert` проверяло условие `state != nullptr`, а теперь у нас есть `state.has_value()`. Мы могли бы в обоих случаях проверять `state`, потому что и указатели, и необязательные значения преобразуются в `bool::true`, если имеют допустимое значение, или `false` в противном случае.

Но главное отличие не так заметно. Экземпляр `std::optional` является полноценным значением и владеет своими данными. В примере с указателем мы столкнулись бы с неопределенным поведением, если бы экземпляр `m_state` был уничтожен или его заменил другой тип состояния между вызовами `std::get_if` и `state->count()`. В этом примере объект `optional` содержит копию значения, поэтому здесь данная проблема отсутствует. В этом примере не нужно думать о времени жизни какой-либо переменной; можно положиться на поведение по умолчанию, которое обеспечивает язык C++.

### 9.1.5 Типы-суммы для обработки ошибок

Необязательные значения могут пригодиться для сообщения вызывающему коду информации об ошибке. Например, мы реализовали функцию `get_if`, которая возвращает действительное значение, если все в порядке, или пустое значение, если была сделана попытка получить экземпляр типа, который в данный момент отсутствует в `variant`. Проблема заключается в том, что тип `optional` позволяет передать фактическое значение, если оно имеется, но не позволяет передать информацию об ошибке, если значение отсутствует.

Чтобы получить возможность сообщать об ошибках, можно создать тип-сумму, способный содержать либо значение, либо ошибку. Ошибку можно представить целым числом или более сложной структурой – даже указателем на исключение (`std::exception_ptr`).

Нам нужна сумма типов `T` и `E`, где `T` – это тип возвращаемого значения, а `E` – тип ошибки. Как и в случае с типом `optional`, тип-сумму можно определить как `std::variant<T, E>`, но такое решение не дает удобного API. Нам придется использовать такие функции, как `index`, `std::get` и `std::get_if`, чтобы получить значение или ошибку, что очень неудобно. Было бы лучше иметь специализированный класс, предлагающий функции-члены с более говорящими именами, такие как `value` и `error`.

Поэтому напишем свою реализацию (листинг 9.5) с именем `expected<T, E>`. Возвращая значение этого типа, функция четко сообщает, что ожидаемым (`expected`) ее результатом является значение типа `T`, но она также может вернуть ошибку типа `E`.

Внутренне этот класс можно реализовать как простое объединение с тегами. Отдельный флаг обозначает, что хранит экземпляр – фактическое значение или ошибку, а объединение позволит передать экземпляр типа `T` или `E`.

#### Листинг 9.5 Внутренняя структура `expected<T, E>`

```
template<typename T, typename E>
class expected {
private:
    union {
        T m_value;
        E m_error;
    };

    bool m_valid;
};
```

Самая простая часть реализации – функции чтения. При попытке получить значение, когда экземпляр хранит ошибку, можно возбудить исключение, и наоборот (листинг 9.6).

#### Листинг 9.6 Функции чтения в `expected<T, E>`

```
template<typename T, typename E>
class expected {
```

```

...
T& get()
{
    if (!m_valid) {
        throw std::logic_error("Missing a value");
    }

    return m_value;
}

E& error()
{
    if (m_valid) {
        throw std::logic_error("There is no error");
    }

    return m_error;
}
};

```

Константные варианты (const) функций чтения можно реализовать точно так же и возвращать из них фактические значения или константные ссылки вместо обычных ссылок.

Обработка значений внутри объединения выглядит сложнее. Поскольку объединение может состоять из сложных типов, нужно вручную вызвать конструкторы и деструкторы для создания и уничтожения их экземпляров.

Значение `expected<T, E>` можно сконструировать из значения типа `T` или из ошибки типа `E`. Поскольку эти два типа могут быть разными, нужно определить отдельные функции для каждого из них и использовать их вместо конструкторов. Это не обязательно, но при таком подходе код получается чище (листинг 9.7).

### Листинг 9.7 Конструирование значений в `expected<T, E>`

```

template<typename T, typename E>
class expected {
...
    template <typename... Args>
    static expected success(Args&&... params)
    {
        expected result;
        result.m_valid = true;
        new (&result.m_value)
            T(std::forward<Args>(params)...);
        return result;
    }

    template <typename... Args>
    static expected error(Args&&... params)

```

Конструктор по умолчанию, создающий неинициализированное объединение

Инициализировать тег объединения. Объединение хранит действительное значение

Вызвать размещающую версию оператора new для инициализации значения типа T и размещения его в памяти m\_value

```

    {
        expected_result;
        result.m_valid = false;
        new (&result.m_error)
            E(std::forward<Args>(params)...);
        return result;
    }
};

```

Создать экземпляр ошибки таким же образом, но вместо конструктора типа T вызвать конструктор типа E

Теперь, если в программе имеется функция, возвращающая `expected<T, E>`, вы сможете вызвать `success` или `error`, чтобы обработать результат.

### Размещающая версия `new`

В отличие от обычного оператора `new`, который выделяет память и инициализирует значение (вызывает конструктор), *размещающая версия* `new` позволяет использовать уже выделенную память и вставить объект в нее. В случае `expected<T, E>` память уже выделена для переменной – члена объединения. На практике этот прием используется нечасто, но иногда он оказывается весьма кстати, особенно в реализациях типов-сумм, которые не используют динамическую память.

Для правильной обработки жизненного цикла экземпляра `expected<T, E>` и хранящихся в нем значений необходимо создать деструктор и конструкторы копирования и перемещения вместе с оператором присваивания. Деструктор класса `expected<T, E>` должен вызвать деструктор `m_value` или `m_error` – в зависимости от того, какой тип имеет хранимый в данный момент экземпляр:

```

~expected() {
    if (m_valid) {
        m_value.~T();
    } else {
        m_error.~E();
    }
}

```

Конструкторы копирования и перемещения похожи друг на друга. Они оба должны проверить действительность копируемого или перемещаемого экземпляра, а затем инициализировать правильный член объединения (листинг 9.8).

### Листинг 9.8 Конструкторы копирования и перемещения для `expected<T, E>`

```

expected(const expected& other) |————| Конструктор копирования инициализирует флаг,
    : m_valid(other.m_valid)     |————| отличающий фактическое значение от ошибки
{

```

```

        if (m_valid) {
            new (&m_value) T(other.m_value);
        } else {
            new (&m_error) E(other.m_error);
        }
    }
}

```

Если копируется экземпляр, содержащий фактическое значение, вызывается конструктор копирования этого значения для инициализации члена-переменной other.m\_value с использованием размещающей версии new

```

    expected(expected&& other)
    : m_valid(other.m_valid)
    {
        if (m_valid) {
            new (&m_value) T(std::move(other.m_value));
        } else {
            new (&m_error) E(std::move(other.m_error));
        }
    }
}

```

Если исходный экземпляр содержит ошибку, выполняется копирование этой ошибки

Конструктор перемещения действует подобно конструктору копирования, но может просто перенести данные из оригинального экземпляра

Вместо конструктора копирования для m\_value или m\_error можно вызвать конструктор перемещения

С конструкторами все просто, но с оператором присваивания ситуация несколько сложнее. Необходимо предусмотреть четыре случая:

- оба экземпляра, который копируется и в который производится копирование, содержат допустимые значения;
- оба экземпляра содержат ошибки;
- this содержит ошибку, а other – действительное значение;
- this содержит действительное значение, а other – ошибку.

Как обычно, реализуем оператор присваивания с использованием идиомы копирования и замены (copy-and-swap), что означает, что мы должны добавить в класс `expected<T, E>` функцию `swap` (листинг 9.9).

### Идиома копирования и замены

Чтобы предотвратить утечки ресурсов при появлении исключений, а также гарантировать, что операция будет либо успешно завершена, либо не выполнена вовсе, для реализации оператора присваивания обычно используется идиома копирования и замены (copy-and-swap). Суть ее заключается в следующем: создается временная копия объекта-источника, а затем выполняется обмен данными между объектом-приемником и этой копией. Если операция выполнится успешно, временный объект будет уничтожен, и с ним уничтожатся предыдущие данные.

Если возникнет исключение, данные из временного объекта останутся в нем, и экземпляр-приемник не изменится. Дополнительные сведения можно найти в статье Герба Саттера (Herb Sutter) «Exception-Safe Class Design, Part 1: Copy Assignment» ([www.gotw.ca/gotw/059.htm](http://www.gotw.ca/gotw/059.htm)) и в теме «What Is the Copy-and-Swap Idiom?» на Stack Overflow (<http://mng.bz/ayHD>).



## Листинг 9.9 Функция swap для expected&lt;T, E&gt;

```

void swap(expected& other)
{
    using std::swap;
    if (m_valid) {
        if (other.m_valid) {
            swap(m_value, other.m_value);
        } else {
            auto temp = std::move(other.m_error);
            other.m_error.~E();
            new (&other.m_value) T(std::move(m_value));
            m_value.~T();
            new (&m_error) E(std::move(temp));
            std::swap(m_valid, other.m_valid);
        }
    } else {
        if (other.m_valid) {
            other.swap(*this);
        } else {
            swap(m_error, other.m_error);
        }
    }
}

expected& operator=(expected other)
{
    swap(other);
    return *this;
}

```

Если «this» и «other» содержат действительные значения, поменять их местами

Если «this» содержит действительное значение, а «other» – ошибку, переместить ее во временную переменную, чтобы подготовить «other» для приема действительного значения. Затем можно безопасно скопировать ошибку в «this»

Если «this» содержит ошибку, а «other» – действительное значение, можно использовать базовую реализацию из предыдущего случая

Если оба экземпляра содержат ошибку, поменять ошибки местами

Оператор присваивания имеет тривиальную реализацию. Аргумент «other» содержит копию значения, которое нужно присвоить экземпляру «this». (Оператор получает экземпляр «other» по значению, а не по константной ссылке.) Мы просто выполняем обмен с текущим экземпляром

Все остальное реализуется легко и просто. Можно добавить оператор приведения к типу `bool`, который будет возвращать `true`, если экземпляр хранит допустимое значение, и `false` в противном случае. Также можно добавить оператор приведения для преобразования экземпляра в `std::optional`, чтобы использовать тип `expected<T, E>` с кодом, который использует `std::optional`:

```

operator bool() const
{
    return m_valid;
}

operator std::optional<T>() const
{
    if (m_valid) {
        return m_value;
    } else {
        return std::optional<T>();
    }
}

```

Теперь можно использовать тип `expected` взамен `std::optional`. Давайте еще раз переопределим функцию `get_if` и для простоты используем `std::string` в качестве типа ошибки:



```
template <typename T, template Variant,
        template Expected = expected<T, std::string>>
Expected get_if(const Variant& variant)
{
    T* ptr = std::get_if<T>(variant);

    if (ptr) {
        return Expected::success(*ptr);
    } else {
        return Expected::error("Variant doesn't contain the desired type");
    }
}
```

Мы получили функцию, которая возвращает либо действительное значение, либо сообщение об ошибке. Это может пригодиться для отладки или когда понадобится показать текст ошибки пользователю. Встраивание ошибок в типы также можно использовать в асинхронных системах, это упростит передачу ошибок между разными асинхронными процессами.

## 9.2 Моделирование предметной области с алгебраическими типами

При проектировании типов данных особое внимание следует уделять невозможности появления недопустимых состояний. Именно поэтому функция-член `size` типа `std::vector` возвращает целое без знака (даже притом что многие не любят беззнаковые типы<sup>1</sup>) – этот тип подсказывает, что размер не может быть отрицательным, – и такие алгоритмы, как `std::reduce` (о котором рассказывалось в главе 2), принимают специальные типы, обозначающие политики выполнения вместо обычных целочисленных флагов.

Вы должны поступать со своими типами и функциями точно так же. Нужно думать не о том, какие данные использовать, чтобы охватить все возможные состояния программы, и затем помещать их в класс, а о том, как определить данные, охватывающие *только допустимые* состояния, в которых может находиться ваша программа.

Для демонстрации подобного подхода я использую упражнение «Tennis kata» (<http://codingdojo.org/kata/Tennis>). Цель этого упражнения – реализовать простую игру в теннис. В теннисе два игрока играют друг против друга. Всякий раз, когда у игрока не получается вернуть мяч на площадку противника, он теряет мяч, и счет обновляется.

<sup>1</sup> См. статью Дэвида Крокера (David Crocker) «Danger – Unsigned Types Used Here» в его блоге *David Crocker's Verification Blog*, 7 апреля 2010, <http://mng.bz/sq4z>.

В теннисе используется уникальная, но простая система подсчета очков:

- игрок может иметь 0, 15, 30 или 40 очков;
- если игрок имеет 40 очков и выигрывает подачу, он выигрывает гейм;
- если оба игрока имеют по 40 очков, действуют несколько другие правила: судья объявляет счет «равно»;
- игрок, выигравший мяч после объявления счета «равно», получает счет «больше», а проигравший – «меньше»;
- если игрок со счетом «больше» выигрывает мяч, он выигрывает гейм, а если проиграет, счет опять становится «равно».

В этом разделе мы рассмотрим несколько реализаций состояния программы для игры в теннис. Мы обсудим проблемы каждой из них и в конечном итоге придем к решению, когда недопустимые состояния окажутся невозможными.

### 9.2.1 Простейшее решение

Простейшее решение заключается в создании двух целочисленных переменных, хранящих очки, заработанные игроками. Для обозначения счета «больше» можно использовать специальное значение:

```
class tennis_t {
private:
    int player_1_points;
    int player_2_points;
};
```

Этот подход охватывает все возможные состояния, но проблема в том, что он позволяет присвоить игроку недопустимое количество очков. Обычно данная проблема решается проверкой данных в методе записи, но было бы проще и удобнее, если бы ничего проверять не требовалось – если бы типы вынуждали писать только правильный код.

Следующий шаг: замена числа очков перечислением, которое позволяет использовать только правильное количество очков:

```
class tennis_t {
    enum class points {
        love, // ноль очков
        fifteen,
        thirty,
        forty
    };

    points player_1_points;
    points player_2_points;
};
```

Это значительно снижает количество состояний, в которых может находиться программа, но и у данного решения есть проблемы. Во-первых, оно позволяет обоим игрокам набрать 40 очков (что технически запре-

щено – для этого состояния есть специальное имя), и у нас нет возможности назначить счет «больше». Можно, конечно, добавить в перечисление элементы `deuce` («равно») и `advantage` («больше»), но это создаст новые недопустимые состояния (один игрок может иметь счет «равно», а другой – ноль очков).

Как видите, это решение плохо подходит для данной задачи. Давайте попробуем пойти другим путем и разобьем оригинальную игру на отдельные состояния, а затем определим их в программе.

## 9.2.2 Более сложное решение: проектирование сверху вниз

Из правил видно, что в игре есть два основных состояния: когда используется числовой счет и когда используется счет «равно» или «больше». В состоянии с нормальным числовым счетом результаты обоих игроков сохраняются одновременно. Но, к сожалению, все не так просто. Если использовать перечисление `points`, объявленное в предыдущем разделе, оба игрока могут получить по 40 очков, что недопустимо; это состояние представлено как состояние «равно».

Можно попробовать решить проблему, удалив элемент `forty` из перечисления, но тогда потеряется возможность набрать 40 очков, когда у другого игрока их меньше. Вернемся к началу задачи. *Нормальное состояние подсчета очков* – это не одно состояние: два игрока могут набрать до 30 очков, или один игрок может набрать 40 очков, а другой – до 30:

```
class tennis_t {
    enum class points {
        love,
        fifteen,
        thirty
    };

    enum class player {
        player_1,
        player_2
    };

    struct normal_scoring {
        points player_1_points;
        points player_2_points;
    };

    struct forty_scoring {
        player leading_player;
        points other_player_scores;
    };
};
```

Все это представляет *обычные* состояния счета очков. Нам осталось описать состояния «равно» и «больше». В состоянии «равно» не требуется хранить какие-либо значения, тогда как состояние «больше» должно указывать, какой игрок имеет преимущество:



```
class tennis_t {
    ...
    struct deuce {};
    struct advantage {
        player player_with_advantage;
    };
};
```

Теперь определим тип-сумму, объединяющую все эти состояния:

```
class tennis_t {
    ...
    std::variant
        < normal_scoring
        , forty_scoring
        , deuce
        , advantage
        > m_state;
};
```

Мы смогли охватить все возможные состояния игры в теннис и исключили возможность появления любых недопустимых состояний.

**ПРИМЕЧАНИЕ** Здесь отсутствует одно важное состояние: состояние завершения игры, которое должно обозначать игрока-победителя. Чтобы просто вывести имя победителя и завершить программу, это состояние не нужно. Но если вы хотите, чтобы программа продолжала работать, это состояние придется реализовать. Сделать это несложно; нужно просто создать еще одну структуру с одной переменной-членом для сохранения победителя и дополнить вариант `m_state`.

Как обычно, не всегда имеет смысл заходить так далеко, стремясь удалить недопустимые состояния (со случаем счета 40:40 вполне можно разобраться вручную). Основная задача этого примера – показать процесс проектирования алгебраических типов данных, соответствующих моделируемой предметной области. Сначала исходное пространство состояний делится на более мелкие независимые части, которые затем описываются по отдельности.

## 9.3 Алгебраические типы и сопоставление с образцом

Главная проблема реализации программ с необязательными значениями, вариантами и другими алгебраическими типами данных в том, что каждый раз, когда требуется получить значение, приходится проверять

его наличие и извлекать из типа-обертки. Эквивалентные проверки не-обходимы, даже когда класс состояния не является типом-суммой, правда, только перед записью значений, а не при каждом обращении.

Такие проверки быстро начинают утомлять, поэтому многие функциональные языки предлагают специальный синтаксис, упрощающий эту задачу. Обычно этот синтаксис реализован в виде инструкции `switch-case`, способной сопоставлять не только с конкретными значениями, но также с типами и более сложными шаблонами.

Применительно к игре в теннис можно было бы определить перечисление для хранения состояния программы и использовать следующую конструкцию для проверки:

```
switch (state) {
    case normal_score_state:
        ...
        break;
    case forty_scoring_state:
        ...
        break;
    ...
};
```



В зависимости от значения переменной `state` код будет выполнять ту или иную ветвь `case`. Но, к сожалению, этот код работает только с целочисленными типами.

Теперь представьте себе мир, в котором этот код может также проверять строки, работать с вариантами и выполнять разные ветви `case`, в зависимости от типа значения, хранящегося в варианте. А что, если каждая ветвь `case` может быть комбинацией из проверок типа, значения и пользовательского предиката? Именно так выглядит сопоставление с образцом в большинстве функциональных языков программирования.

C++ поддерживает возможность сопоставления с образцом для шаблонов метапрограммирования (как будет показано в главе 11), но он не подходит для обычных программ, а значит, нужен какой-то другой подход. Стандартная библиотека включает функцию `std::visit`, принимающую экземпляр `std::variant`, и функцию, которая должна вызываться для значения, хранящегося внутри. Например, вот как можно вывести текущее состояние игры в теннис (учитывая, что реализован оператор `<<` для вывода типов состояний в стандартный вывод):

```
std::visit([] (const auto& value) {
    std::cout << value << std::endl;
},
m_state);
```

Здесь используется обобщенное лямбда-выражение (с типом аргумента, указанным как `auto`), чтобы оно могло работать со значениями разных типов, потому что вариант может иметь четыре совершенно разных типа, и при этом код остается статически типизированным.

Использование обобщенного лямбда-выражения с `std::visit` часто полезно, но в большинстве случаев этого недостаточно. Нам нужна возможность выполнять разный код в зависимости от фактического типа значения, хранящегося в варианте, подобно тому, как это делает конструкция `switch-case`.

Для этого можно создать перегруженный объект-функцию, который будет определять реализации для разных типов; выбор реализации будет зависеть от типа значения, хранящегося в экземпляре варианта. Чтобы сделать определение объекта максимально коротким, используем возможности языка, доступные в C++17. Реализацию, совместимую со старыми компиляторами, вы найдете в примерах кода, сопровождающих книгу:

```
template <typename... Ts>
struct overloaded : Ts... { using Ts::operator()...; };

template <typename... Ts> overloaded(Ts...) -> overloaded<Ts...>;
```

Шаблон `overloaded` принимает список объектов-функций и создает новый объект, который представляет операторы вызова всех объектов-функций как свои собственные (используя `Ts::operator()...`).

**ПРИМЕЧАНИЕ** Фрагмент кода, реализующий перегруженную структуру, использует механизм определения типа аргумента шаблона для классов, появившихся в C++17. Определение типа аргумента шаблона опирается на конструктор класса. Вы можете либо реализовать свой конструктор, либо предоставить подсказку для определения типа, как в предыдущем примере.

Теперь попробуем использовать это определение в примере с теннисом. Каждый раз, когда игрок выигрывает мяч, вызывается функция-член `point_for`, обновляющая состояние игры:

```
void point_for(player which_player)
{
    std::visit(
        overloaded {
            [&](const normal_scoring& state) {
                // Увеличить счет или переключить состояние
            },
            [&](const forty_scoring& state) {
                // Игрок выиграл или перейти в состояние "равно"
            },
            [&](const deuce& state) {
                // Перейти в состояние "больше"
            },
            [&](const advantage& state) {
                // Игрок выиграл или вернуться в состояние "равно"
            }
        },
        m_state);
}
```

`std::visit` вызовет перегруженный объект-функцию, который сопоставит тип аргумента со всеми перегруженными версиями и выполнит ту, которая является наилучшей (в смысле типа). Это не самый удачный синтаксис, но такой код представляет собой эффективный эквивалент оператора `switch`, выбирающего реализацию по типу объекта, хранящегося в варианте.

Вы можете легко создать функцию `visit` для `std::optional`, для класса `expected` и даже для типов-сумм на основе наследования и получить унифицированный синтаксис для обработки всех типов-сумм, созданных вами.

## 9.4 Сопоставление с образцом с помощью библиотеки Mach7

До сих пор мы рассматривали простое сопоставление с образцом для типов. Однако точно так же можно реализовать сопоставление для определенных значений, скрыв цепочки `if-else` за структурой, напоминающей `overloaded`.

Но было бы намного удобнее иметь возможность сопоставлять с более сложными образцами. Например, нам необходимы отдельные обработчики `normal_scoring` для случаев, когда у игрока меньше 30 очков и когда у него 30 очков, потому что в последнем случае состояние игры нужно изменить на `forty_scoring`.

К сожалению, в C++ нет синтаксиса, поддерживающего такую возможность. Зато есть библиотека Mach7, позволяющая определять более сложные образцы, хотя синтаксис оставляет желать лучшего.

### Библиотека Mach7 для эффективного сопоставления с образцом

Библиотека Mach7 (<https://github.com/solodon4/Mach7>) была создана Юрием Солодким (Yuriy Solodkyu), Габриэлем Дос Рейсом (Gabriel Dos Reis) и Бьярном Страуструпом (Bjarne Stroustrup) и является основой для добавления поддержки сопоставления с образцом в C++ в будущем. Первоначально библиотека создавалась для экспериментов, но достигла достаточно высокого уровня стабильности для общего пользования. Как правило, она эффективнее шаблона «Посетитель» (не путайте с `std::visit` для вариантов). Основным недостатком Mach7 является неудобный синтаксис.

Используя библиотеку Mach7, можно указать объект для сопоставления и перечислить все образцы и действия, которые должны выполняться при совпадении с образцами. В игре в теннис реализация функции-члена `point_for` могла бы выглядеть так:

```
void point_for(player which_player)
{
    Match(m_state)
    {
```

```

    Case(C<normal_scoring>()) ← Увеличить счет или изменить состояние
    Case(C<forty_scoring>()) ← Игрок выиграл или перейти в состояние «равно»
    Case(C<deuce>()) ← Перейти в состояние «больше»
    Case(C<advantage>()) ← Игрок выиграл или вернуться в состояние «равно»
  }
  EndMatch
}

```

Второй образец можно разбить на несколько отдельных образцов. Если игрок, выигравший мяч, имел перед этим 40 очков, значит, он выиграл гейм. Иначе нужно посмотреть, можно ли увеличить очки второго игрока или следует перейти в состояние `deuce` («равно»).

Если в текущий момент программа находится в состоянии `forty_scoring` и игрок, имеющий 40 очков, выиграл мяч, он выигрывает гейм независимо от числа очков у другого игрока. Обозначить это можно с помощью образца `C<forty_scoring>(which_player, _)`. Подчеркивание означает, что фактическое значение игнорируется при сопоставлении – в этом случае нас не интересует количество очков у второго игрока.

Если игрок, имеющий 40 очков, не выиграл мяч, нужно проверить, имел ли перед этим второй игрок 30 очков, чтобы перейти в состояние `deuce`. Проверить эту ситуацию можно с помощью шаблона `C<forty_scoring>(_, 30)`. Здесь не требуется проверять количество очков у первого игрока, потому что мы знаем, что если игрок, у которого ранее было 40 очков, выиграл мяч, он совпадет с предыдущим образцом.

Если проверяемый объект не совпадет ни с одним из этих двух образцов, значит, нужно увеличить количество очков у второго игрока. При этом можно проверить переход в состояние `forty_scoring`.

### Листинг 9.10 Раздельное сопоставление с разными вариантами одной ситуации

```

void point_for(player which_player)
{
  Match(m_state)
  {
    ...
    Case(C<forty_scoring>(which_player, _)) ←
    Case(C<forty_scoring>(_, 30)) ←
    Case(C<forty_scoring>()) ←
    ...
  }
  EndMatch
}

```

Если игрок, имевший 40 очков, выиграл мяч, выигрывает гейм; количество очков у второго не имеет значения

Если игрок, выигравший мяч, имел меньше 40 очков (нет совпадения с предыдущим образцом) и другой игрок к данному моменту имел 30 очков, игра переходит в состояние «равно»

Если нет совпадения ни с одним из предыдущих образцов, увеличить количество очков у игрока

Независимо от того, какой путь вы выберете при работе с алгебраическими типами – `std::visit` с перегруженными объектами-функциями или полноценное сопоставление с образцом, с использованием такой библиотеки, как Mach7, у вас есть все возможности, чтобы писать правильные программы. Компилятор заставит вас определять исчерпывающие шаблоны, иначе программа не будет компилироваться, а пространство возможных состояний будет минимальным.

**СОВЕТ** За дополнительной информацией по теме, рассмотренной в этой главе, обращайтесь по адресу: <https://forums.manning.com/posts/list/43778.page>.

## Итоги

- Применение алгебраических типов данных для реализации состояния программы требует времени на размышления и большего объема кода, но позволяет минимизировать количество возможных состояний и избавляет от появления недопустимых состояний.
- Для реализации типов-сумм в объектно-ориентированных языках программирования часто используются наследование, динамическая диспетчеризация и шаблон проектирования «Посетитель». Основным недостатком наследования в данном случае заключается в снижении производительности во время выполнения.
- Если точно известно, какие типы войдут в состав типа-суммы, для его реализации лучше использовать варианты, а не наследование. Основным недостатком подхода на основе `std::variant` заключается в большом объеме типового кода из-за функции `std::visit`.
- В отличие от исключений, которые, как следует из их названия, должны обозначать исключительные ситуации, необязательные значения и класс `expected` идеально подходят для случаев, когда требуется явно указать возможность появления ошибки. Их также проще передавать между потоками выполнения и процессами.
- Идея наличия типа, который содержит либо значение, либо ошибку, давно завоевала популярность в мире функционального программирования. В сообществе C++ она тоже стала пользоваться популярностью, после того как Андрей Александреску (Andrei Alexandrescu) выступил с докладом «Systematic Error Handling in C++» на конференции «C++ and Beyond» в 2012 году (<http://mng.bz/q5XF>), в котором представил свою версию типа `expected`. Эта версия похожа на нашу, но для обозначения ошибки поддерживает только тип `std::exception_ptr`.



# 10

## Монады

### О чем говорится в этой главе:

- основы функций;
- расширение возможностей transform с использованием монад;
- композиция функций, возвращающих типы-обертки;
- выполнение асинхронных операций в функциональном стиле.



В мире функционального программирования не так много шаблонов проектирования, но имеется масса обобщенных абстракций, которые используются повсеместно. Эти абстракции предлагают одинаковые решения различных задач из совершенно разных областей.

В C++ уже есть одна такая абстракция: итераторы. Для перемещения по элементам обычных массивов можно использовать указатели. Для перехода к следующему и предыдущему элементам можно использовать операторы ++ и -- соответственно, а для обращения к ним можно использовать оператор разыменования \*. Проблема в том, что этот прием пригоден только при работе с массивами и структурами, хранящими данные в непрерывной области памяти. Он не подходит для таких структур, как связанные списки, множества и ассоциативные массивы, реализованные с использованием деревьев.

Для решения этой проблемы были созданы итераторы. Они используют перегрузку операторов, чтобы создать абстракцию, своим поведением напоминающую указатели и способную работать не только с массивами, но и со множеством других структур данных. Итераторы могут

также работать с потоками ввода и вывода, которые традиционно не считаются *структурами данных*.

В главе 7 была представлена еще одна абстракция, основанная на итераторах: диапазоны. Диапазоны расширяют возможности итераторов, абстрагируя разные структуры данных, вместо абстрагирования доступа к данным в этих структурах. В этой главе вы познакомитесь с еще одной разновидностью диапазонов – всплывающими диапазонами.

## 10.1 Функторы

Я немного отступлю от привычного стиля изложения, принятого в этой книге: я начну не с примера, а с определения понятия, и только потом покажу примеры. Начнем с понятия *функтора*. Как я упоминал в главе 3, многие разработчики на C++ используют этот термин для обозначения *класса с оператором вызова*, но это неправильно. В функциональном программировании под словом «функтор» подразумевается нечто совершенно иное.

Функтор – это понятие из абстрактного раздела математики, называемого *теорией категорий*, и его формальное определение звучит так же абстрактно, как и теория, из которой он исходит. Мы немного перефразируем это определение, чтобы сделать его понятнее для разработчиков на C++.

Шаблон класса  $F$  является функтором, если в нем определена функция `transform` (или `map`), как показано на рис. 10.1. Функция `transform` принимает экземпляр  $f$  типа  $F<T1>$  и функцию  $t: T1 \rightarrow T2$  и возвращает значение типа  $F<T2>$ . Эта функция может иметь несколько форм, поэтому для ясности используем обозначение конвейера из главы 7.

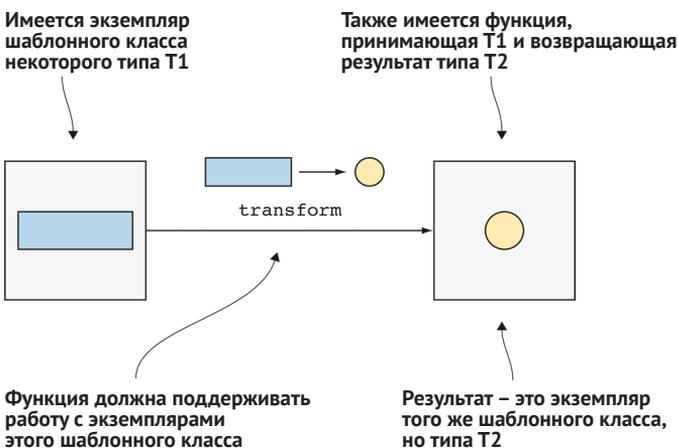


Рис. 10.1 Функтор – это шаблон класса, параметризуемый типом  $T$ . Для работы с экземплярами этого класса можно использовать любую функцию, способную обрабатывать значения типа  $T$

Функция `transform` должна соответствовать следующим двум правилам:

- тождественное преобразование экземпляра функтора должно возвращать тот же (*равный*) экземпляр функтора:

```
f | transform([])(auto value) { return value; } == f
```

- преобразование функтора с использованием одной, а затем другой функции должно давать тот же результат, что и преобразование с использованием композиции этих функций (см. рис. 10.2):

```
f | transform(t1) | transform(t2) ==
f | transform([=](auto value) { return t2(t1(value)); })
```

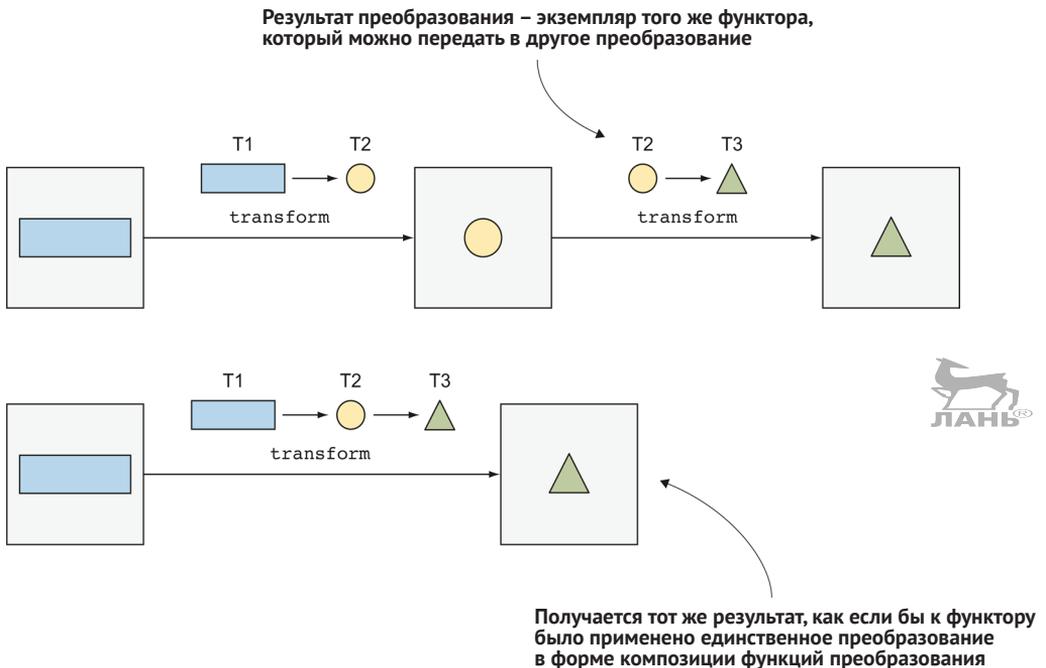


Рис. 10.2 Главное правило, которому должна следовать функция `transform`: последовательное выполнение двух преобразований, `f` и `g`, должно давать тот же эффект, что и одно преобразование, реализованное как композиция `f` и `g`

Это очень похоже на алгоритмы `std::transform` и `view::transform` из библиотеки диапазонов. И это сходство не случайно: обобщенные коллекции из STL и диапазоны являются функторами. Они все – типы-обертки, имеющие функцию `transform` с четко определенным поведением. Важно отметить, что обратное утверждение не всегда верно: не все функторы являются коллекциями или диапазонами.

### 10.1.1 Обработка необязательных значений

Одним из основных функторов является тип `std::optional` из главы 9. Ему просто необходима определенная функция преобразования.

**Листинг 10.1** Определение функции преобразования для `std::optional`

```

template <typename T1, typename F>
auto transform(const std::optional<T1>& opt, F f)
-> decltype(std::make_optional(f(opt.value())))
{
    if (opt) {
        return std::make_optional(f(opt.value()));
    } else {
        return {};
    }
}

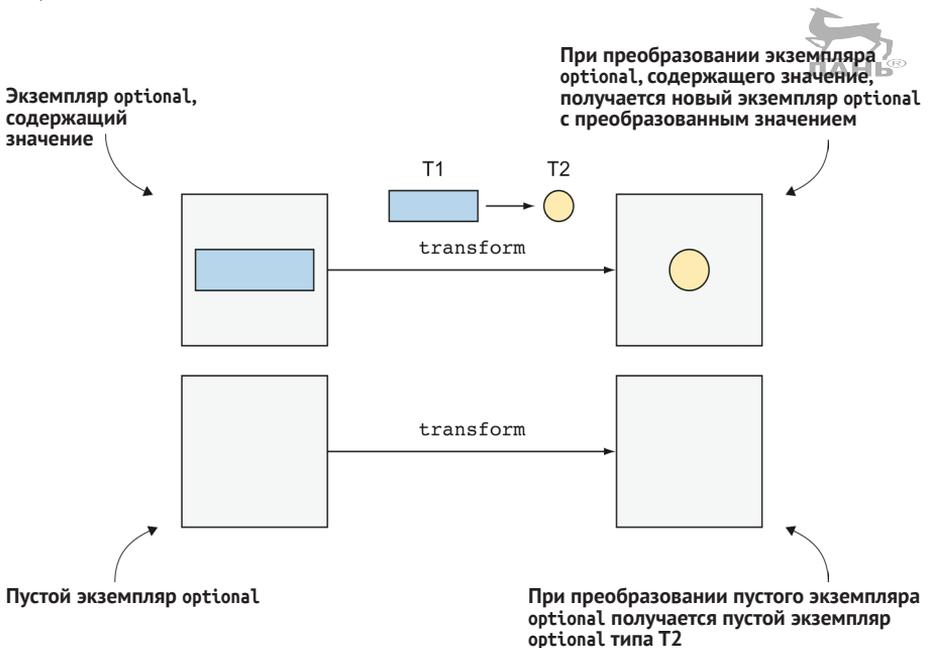
```

Указать возвращаемый тип, потому что в отсутствие значения возвращается пустой экземпляр {}

Если `opt` содержит значение, преобразовать его с помощью `f` и вернуть преобразованное значение в составе нового экземпляра `std::optional`

Если значение отсутствует, вернуть пустой экземпляр `std::optional`

Также можно создать представление диапазона, включающего единственный элемент, когда `std::optional` содержит значение, и пустой диапазон в противном случае (см. рис. 10.3). При таком подходе появляется возможность использовать синтаксис конвейера. (Загляните в пример кода `functors-optional`, где определяется функция `as_range`, которая преобразует `std::optional` в диапазон, содержащий не более одного элемента.)



**Рис. 10.3** `optional` – это тип-обертка, который может содержать одно значение или ничего не содержать. При попытке преобразования экземпляра `optional`, содержащего значение, получается новый экземпляр `optional` с преобразованным значением. Если экземпляр `optional` не содержит значения, в результате получится пустой экземпляр `optional`

В чем преимущество использования функции `transform` по сравнению с обработкой отсутствующих значений вручную с помощью инст-

рукции if-else? Рассмотрим следующий сценарий. Пусть есть система, поддерживающая учетные записи пользователей. Она может иметь два состояния: пользователь вошел в систему или нет. Естественно для этой цели было бы использовать переменную `current_login` типа `std::optional<std::string>`. Переменная `current_login` будет представлена пустым экземпляром `optional`, если пользователь не вошел в систему; иначе она будет содержать имя пользователя. Сделаем переменную `current_login` глобальной, чтобы упростить примеры.

Теперь представьте, что есть функция, извлекающая полное имя пользователя, и функция, создающая строку в формате HTML из всего, что ей будет передано:

```
std::string user_full_name(const std::string& login);
std::string to_html(const std::string& text);
```



Чтобы получить строку в формате HTML с именем текущего пользователя (см. рис. 10.4), можно вручную проверять наличие текущего пользователя, а можно создать функцию, возвращающую `std::optional<std::string>`. Функция должна возвращать пустое значение, если пользователь не вошел в систему, и полное имя в формате HTML, если пользователь вошел в систему. Теперь, когда у нас есть функция `transform`, способная работать с необязательными значениями, реализовать такую функцию тривиально просто:

```
transform(
    transform(
        current_login,
        user_full_name),
    to_html);
```



Имеется экземпляр `optional`, содержащий имя текущего пользователя

Сначала получим полное имя, а затем преобразуем его в формат HTML

В результате получается экземпляр `optional` с полным именем в формате HTML

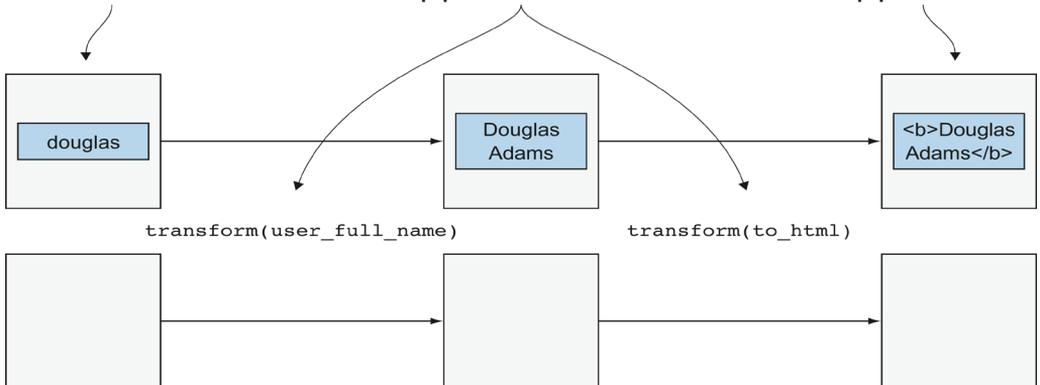


Рис. 10.4 К необязательному значению можно применить цепочку функций и в конце получить объект `optional`, содержащий результат всех преобразований

Если понадобится вернуть диапазон, преобразования можно выполнить с использованием синтаксиса конвейера:

```
auto login_as_range = as_range(current_login);
login_as_range | view::transform(user_full_name)
               | view::transform(to_html);
```

Рассматривая эти две реализации, можно заметить следующее: в этом коде ничто не говорит о том, что он работает с необязательными значениями. Он может работать с массивами, векторами, списками или чем-то еще, для чего определена функция `transform`. Вам не придется менять код, если вдруг решите заменить `std::optional` любым другим функтором.

### Особенность диапазонов

Важно отметить отсутствие автоматического преобразования из `std::optional` в диапазон и наоборот, поэтому такое преобразование нужно выполнять вручную. Строго говоря, функция `view::transform` определена неправильно с точки зрения функторов. Она всегда возвращает диапазон, а не тот тип, который ей передан.

Такое поведение может вызывать проблемы, потому что вынуждает преобразовывать типы вручную. Но это не самая большая неприятность, если учесть, какие преимущества дает диапазон.

Представьте, что нам нужно создать функцию, которая принимает список имен пользователей и возвращает список полных имен в формате HTML. Реализация этой функции будет идентична той, что работает с необязательными значениями. То же касается функции, которая использует `expected<T, E>` вместо `std::optional<T>`. Это та мощь, которую дают такие широко применяемые абстракции, как функторы: можно написать обобщенный код, работающий без изменений в разных сценариях.

## 10.2 Монады: расширение возможностей функторов

Функторы упрощают преобразование вложенных значений, но они имеют серьезное ограничение. Представьте, что функции `user_full_name` и `to_html` могут потерпеть неудачу, и поэтому при их реализации было решено вместо строки возвращать `std::optional<std::string>`:

```
std::optional<std::string> user_full_name(const std::string& login);
std::optional<std::string> to_html(const std::string& text);
```

Функция `transform` не способна помочь в этом случае. Если попытаться использовать ее и написать тот же код, что и в предыдущем примере, мы

получим составной результат. Напомню, что `transform` получает экземпляр функтора `F<T1>` и функцию преобразования из `T1` в `T2` и возвращает экземпляр `F<T2>`.

Взгляните на следующий фрагмент кода:

```
transform(current_login, user_full_name);
```

Значение какого типа вернет этот вызов? Это не `std::optional<std::string>`. Функция `user_full_name` принимает строку и возвращает необязательное значение `T2 = std::optional<std::string>`. По этой причине результатом `transform` является вложенное необязательное значение `std::optional<std::optional<std::string>>` (см. рис. 10.5). Чем больше преобразований будет выполнено, тем более глубоко вложенное значение вы получите, что очень неприятно.

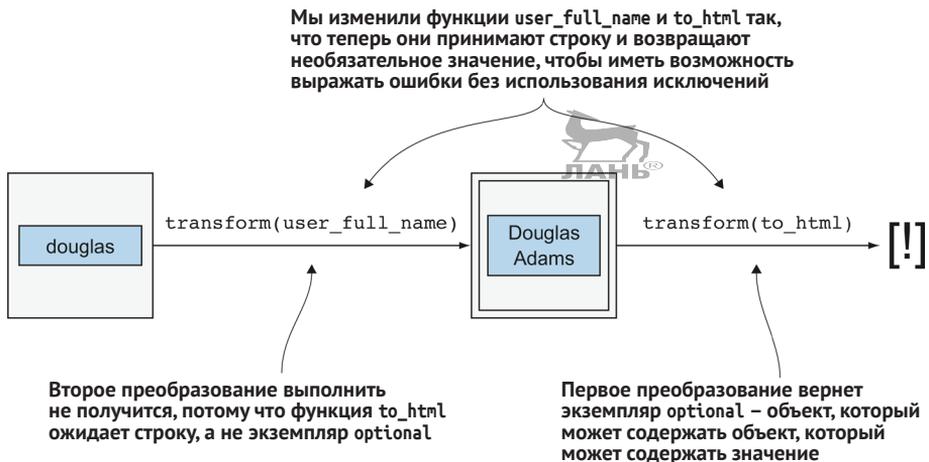


Рис. 10.5 Попытавшись объединить несколько функций, принимающих значение и возвращающих экземпляр функтора, вы получите вложенные функторы. В данном случае вы получите экземпляр `optional` с вложенным экземпляром `optional` с вложенным значением, который практически бесполезен. Более того, чтобы связать два преобразования, второе придется выполнить дважды

Помочь в этой ситуации могут монады. Монада `M<T>` – это функтор, имеющий дополнительную функцию, которая удаляет один уровень вложенности:

```
join: M<M<T>> → M<T>
```

Функция `join` (см. рис. 10.6 и 10.7) избавляет от проблем с использованием функций, которые возвращают не обычные значения, а экземпляры монады (функтора).



Рис. 10.6 Преобразование функтора с помощью функции, которая возвращает не значение, а новый экземпляр этого функтора, приводит к получению вложенных типов функторов. Мы можем создать функцию, которая удалит вложение

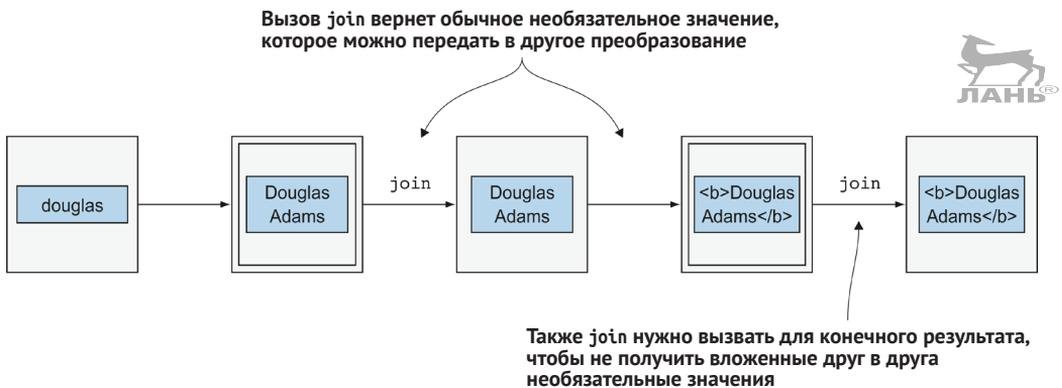


Рис. 10.7 Необязательные значения можно использовать для обозначения ошибок в вычислениях. Функция `join` позволяет легко объединить несколько преобразований, каждое из которых может вернуть ошибку

Теперь мы можем написать такой код:

```
join(transform(
  join(transform(
    current_login,
    user_full_name)),
  to_html));
```

Или такой, если вам больше нравится форма записи диапазонов:

```
auto login_as_range = as_range(current_login);
login_as_range | view::transform(user_full_name)
               | view::join
               | view::transform(to_html)
               | view::join;
```

Преобразование типа значения, возвращаемого функцией, является важным изменением. Если бы в реализации использовались только проверки `if-else`, пришлось бы существенно изменить код. В этом примере нам потребовалось избежать многократного обертывания значения.

Было бы хорошо, если бы была возможность упростить этот код еще больше. В примере выше к результату каждого преобразования применялась функция `join`. А можно ли объединить все эти преобразования в одну функцию?

Да, можно, и это самый типичный способ определения монад. Можно сказать, что монада  $M$  – это тип-обертка, который имеет функцию-конструктор `construct` (создающую экземпляр  $M<T>$  из значения типа  $T$ ) и функцию `mbind` (обычно ей дается более простое имя `bind`, но мы будем использовать имя `mbind`, чтобы избежать путаницы с `std::bind`), которая является композицией `transform` и `join`:

```
construct : T → M<T>
mbind     : (M<T1>, T1 → M<T2>) → M<T2>
```

Легко показать, что все монады являются функторами. Реализовать `transform` с помощью `mbind` и `construct` тривиально просто.

В отношении монад, как и функторов, действует несколько правил. Их соблюдение необязательно для использования монад в программах:

- если у вас есть функция  $f: T_1 \rightarrow M<T_2>$  и значение  $a$  типа  $T_1$ , обертывание его монадой  $M$  и последующее связывание с функцией  $f$  эквивалентно простому вызову функции  $f$  для этого значения:

```
mbind(construct(a), f) == f(a)
```

- это правило повторяет предыдущее с точностью до наоборот; привязка обернутого значения к функции `construct` дает в результате то же обернутое значение:

```
mbind(m, construct) == m
```

- это правило менее понятно, оно определяет ассоциативность операции `mbind`:

```
mbind(mbind(m, f), g) == mbind(m, [] (auto x) {
    return mbind(f(x), g) })
```

Эти правила могут показаться странными, но они точно определяют правильное поведение монад. Теперь вы можете положиться на свою интуицию: монада – это то, что можно сконструировать и связать с функцией.

## 10.3 Простые примеры

Рассмотрим несколько простых примеров. Обычно при изучении языка C++ принято сначала знакомиться с наиболее простыми типами, поэто-

му сначала возьмем *тип-обертку* `std::vector` и посмотрим, как из него создать функтор. Нам нужно выполнить два условия:

- 1 функтор – это шаблонный класс с одним параметром типа;
- 2 функтор должен иметь функцию `transform`, которая принимает вектор и функцию преобразования его элементов и возвращает вектор преобразованных элементов (см. рис. 10.8).

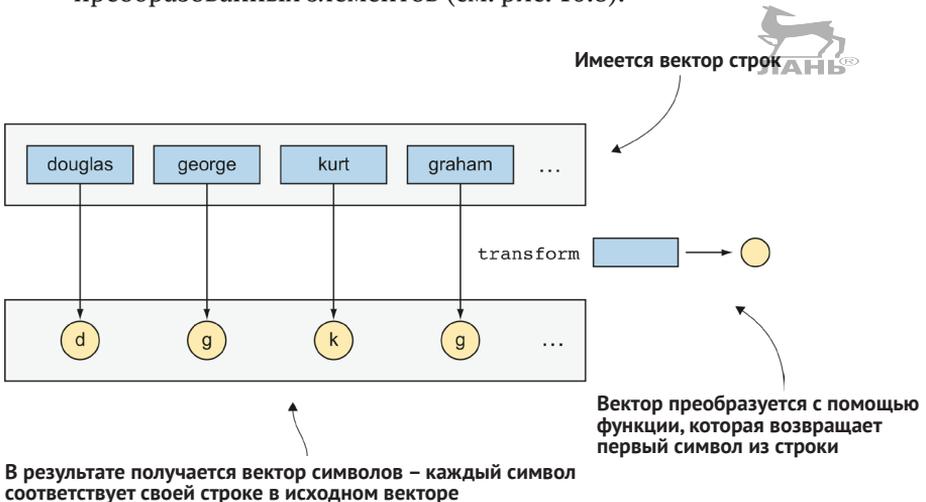


Рис. 10.8 Операция преобразования дает новый вектор с тем же количеством элементов. Каждому элементу в исходной коллекции соответствует свой элемент в результате

`std::vector` – это шаблонный класс, то есть первое условие уже выполнено. А выполнить второе условие и реализовать функцию `transform` легко можно с помощью диапазонов:

```
template <typename T, typename F>
auto transform(const std::vector<T>& xs, F f)
{
    return xs | view::transform(f) | to_vector;
}
```

Данный вектор рассматривается как диапазон, и с помощью `f` преобразуется каждый из его элементов. Чтобы функция вернула вектор, как того требует определение функтора, необходимо преобразовать результат обратно в вектор. Впрочем, для большей гибкости можно вернуть и диапазон.

Теперь, определив функтор, превратим его в монаду. Для этого нужно добавить функции `construct` и `mbind`. Функция `construct` должна принимать значение и создавать из него вектор. Вполне естественно для этой цели использовать фактический конструктор вектора. Вот как проще всего написать функцию, которая создает вектор из единственного значения:

```

template <typename T>
std::vector<T> make_vector(T&& value)
{
    return {std::forward<T>(value)};
}

```

Осталась функция `mbind`. Напомню, что она фактически является комбинацией функций `transform` и `join` (см. рис. 10.9).

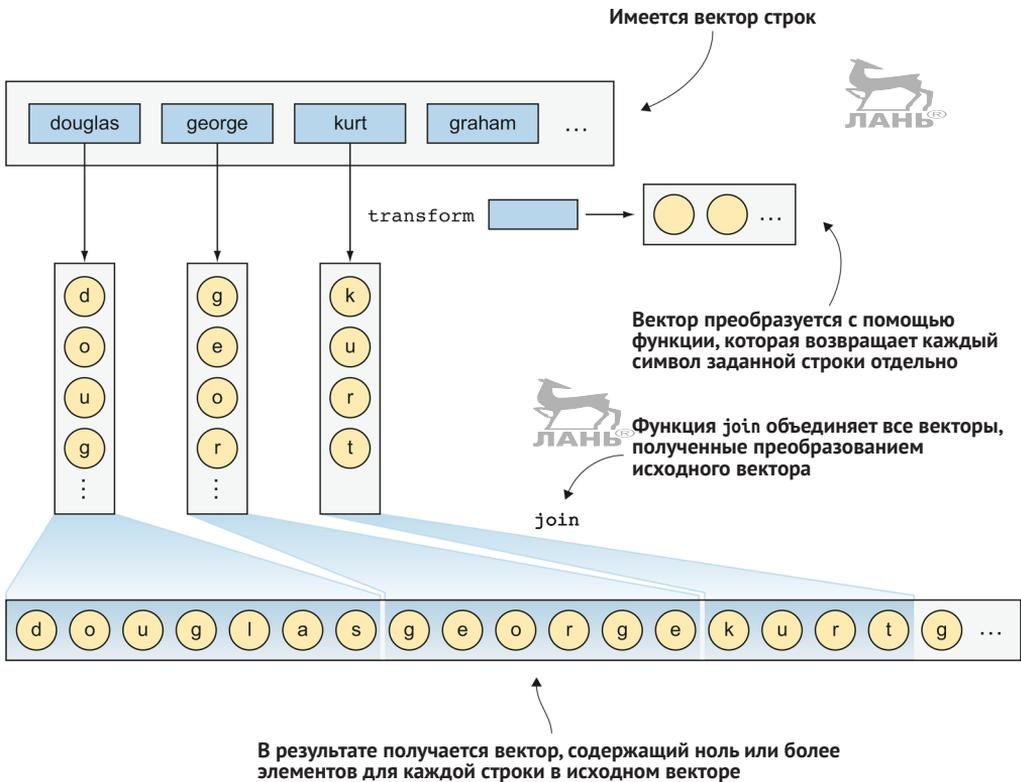


Рис. 10.9 При применении к векторам функция `mbind` вызывает функцию преобразования для каждого элемента в исходном векторе. Результатом каждого преобразования является вектор элементов. Все элементы из этих векторов объединяются в один вектор результата. В отличие от `transform`, функция `mbind` позволяет получить в результате не единственный элемент для каждого элемента в исходном векторе, а столько, сколько нужно

Функции `mbind` (в отличие от `transform`) требуется вызвать еще одну функцию, отображающую значения в экземпляры монады – в данном случае в экземпляр `std::vector`. То есть для каждого элемента в исходном векторе вместо единственного элемента будет получен новый вектор.

Листинг 10.2 Функция `mbind` для векторов

```

template <typename T, typename F>
auto mbind(const std::vector<T>& xs, F f)
{
    auto transformed =
        xs | view::transform(f) | to_vector;
    return transformed
        | view::join | to_vector;
}

```

`f` принимает значение типа `T` и возвращает вектор элементов типа `T` или другого типа

Вызовет `f` и вернет диапазон векторов, который затем преобразуется в вектор векторов

Нам нужен не вектор векторов, а единственный вектор, включающий содержимое всех векторов

Мы реализовали функцию `mbind` для векторов. Она не так эффективна, как могла бы быть, потому что сохраняет промежуточные результаты во временных векторах; но главная наша цель состояла совсем в другом – показать, что `std::vector` является монадой.

**ПРИМЕЧАНИЕ** В этом примере операция `mbind` определена как функция, которая принимает два аргумента и возвращает результат. Для удобства в оставшейся части главы функция `mbind` будет использоваться совместно с более удобочитаемым синтаксисом конвейеров. Я буду писать `xs | mbind(f)` вместо `mbind(xs, f)`. Этот синтаксис не получится использовать без выполнения предварительных условий – вам придется добавить немного типового кода, как показано в примерах `10-monad-vector` и `10-monad-range`, сопровождающих книгу.

## 10.4 Генераторы диапазонов и монад

Тот же подход, который выше использовался для векторов, можно применить к схожим коллекциям, таким как массивы и списки. Все эти коллекции похожи с точки зрения уровня абстракции, на котором мы работаем, – это плоские коллекции элементов одного типа.

Вы уже видели одну абстракцию, позволяющую единообразно работать со всеми этими типами: диапазоны. Мы использовали диапазоны во всех предыдущих функциях для `std::vector`.

Мы также не раз убеждались в полезности функции `transform`. Теперь у нас есть похожая на нее, но более мощная функция `mbind`. Но нужна ли эта дополнительная мощь – большой вопрос. Далее вы увидите, насколько выгодна эта мощь в других монадах, но сначала выясним, нужна ли она в обычных коллекциях и диапазонах.

Давайте взглянем на `mbind` с другой точки зрения – с точки зрения коллекций – и начнем с обсуждения `transform`, потому что знаем, как она работает. Функция `transform` принимает коллекцию и генерирует новую. Для этого она выполняет обход элементов исходной коллекции, преобразует их и помещает результаты в новую коллекцию.

Функция `mbind` действует аналогично, но имеет небольшое отличие. Как я уже говорил, ее можно рассматривать как композицию `transform` и `join`. Функция `transform` создает диапазон новых элементов для каждого элемента в исходной коллекции, а `join` объединяет все эти сгенерированные диапазоны. Другими словами, `mbind` позволяет сгенерировать для каждого элемента в исходной коллекции не один, а сколько угодно новых элементов: ноль или больше (см. рис. 10.10).

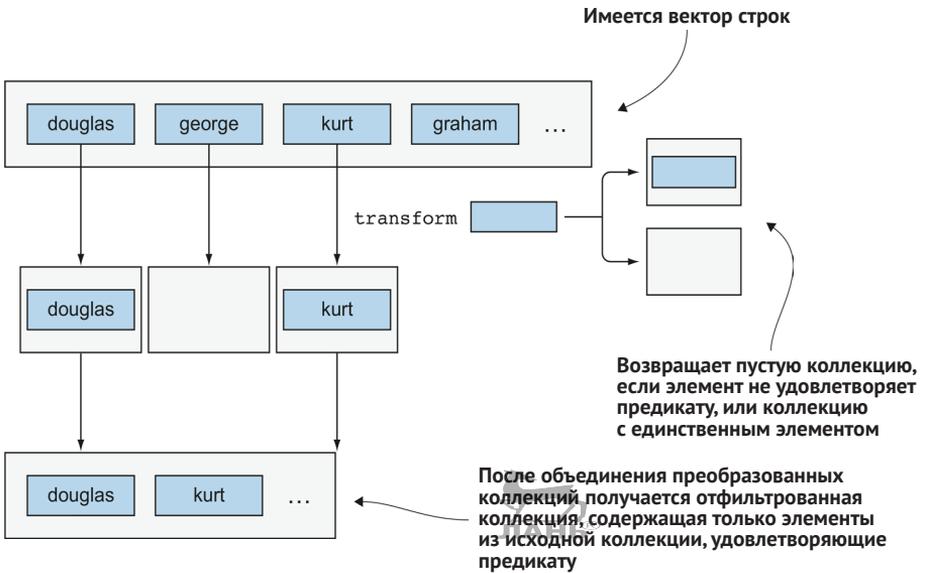


Рис. 10.10 Фильтрацию легко выразить с помощью `mbind`, передав ей функцию преобразования, которая возвращает пустую коллекцию, если исходный элемент не соответствует условию фильтрации, или коллекцию с одним элементом в противном случае

Где это может пригодиться? Есть еще одна функция, которую мы видели несколько раз, – `filter`. Ее легко реализовать в терминах `mbind`. Достаточно передать в `mbind` функцию, которая будет возвращать пустой диапазон, если текущий элемент не соответствует условию, или диапазон с единственным элементом в противном случае.

### Листинг 10.3 Фильтрация в терминах `mbind`

```
template <typename C, typename P>
auto filter(const C& collection, P predicate)
{
    return collection
        | mbind([=](auto element) {
            return view::single(element)
                | view::take(predicate(element)
                    ? 1 : 0);
        });
}
```

Создать диапазон с единственным элементом (сконструировать экземпляр монады из значения) и получить 1 или 0 элементов, в зависимости от соответствия текущего элемента предикату



Аналогично можно выполнить несколько преобразований диапазона. Список увеличится, если использовать изменяемые функциональные объекты. В функциональном программировании не рекомендуется применять изменяемое состояние, но в данном случае в этом нет никакой опасности, поскольку изменяемые данные нигде больше не используются (см. главу 5). Очевидно, что сама возможность что-то сделать не означает, что мы должны это делать; смысл в том, чтобы получить большую выразительность с помощью `mbind`.

Тот факт, что диапазоны являются монадами, не только позволяет переопределить преобразование диапазонов *крутым* способом, но также дает возможность иметь генераторы диапазонов.

Представьте, что в программе нужно сгенерировать список пифагоровых троек (в которых сумма квадратов двух первых чисел равна квадрату третьего числа). При классическом подходе придется написать три вложенных цикла `for`. Функция `mbind` позволяет выполнять аналогичное вложение с использованием диапазонов.

**Листинг 10.4 Генератор пифагоровых троек**

```

view::ints(1)
| mbind([])(int z) {
    return view::ints(1, z)
    | mbind([z])(int y) {
        return view::ints(y, z)
        | view::transform([y,z])(int x) {
            return std::make_tuple(x, y, z);
        });
    });
}
| filter([] (auto triple) {
    ...
});
    
```

Генерирует бесконечный список целых чисел

Для каждого числа z сгенерирует список целых чисел от 1 до z

Для каждого y из диапазона от 1 до z сгенерирует список целых чисел от y до z

Теперь из списка возможных троек нужно отфильтровать те, которые не являются пифагоровыми тройками

Хорошо бы сделать этот код более плоским. К счастью, в библиотеке диапазонов имеется пара специальных функций – `for_each` и `yield_if`, – помогающих писать более удобочитаемый код.

**Листинг 10.5 Получение пифагоровых троек с использованием генераторов диапазонов**

```

view::for_each(view::ints(1), [](int z) {
    return view::for_each(view::ints(1, z), [z](int y) {
        return view::for_each(view::ints(y, z), [y,z](int x) {
            return yield_if(
                x * x + y * y == z * z,
                std::make_tuple(x, y, z)
            );
        });
    });
});
    
```

Сгенерирует тройки (x, y, z), как в предыдущем примере

Если (x, y, z) – это пифагорова тройка, добавить ее в диапазон результата

Генератор диапазона состоит из двух компонентов. Первый – функция `for_each`, которая выполняет обход заданной коллекции и собирает все значения, возвращаемые указанной функцией. Если имеется несколько вложенных генераторов диапазонов, все полученные значения последовательно помещаются в диапазон результата. Генератор диапазона создает один плоский диапазон, а не диапазон диапазонов. Второй компонент – функция `yield_if`. Она помещает значение в диапазон результата, если предикат в ее первом аргументе вернет `true`.

Если говорить просто, генератор диапазона – это не что иное, как сочетание `transform` или `mbind c filter`. А поскольку эти функции существуют не только для диапазонов, но и для любых монад, мы можем назвать его *генератором монад*.

## 10.5 Обработка ошибок

В начале этой главы мы коснулись функций, сообщающих об ошибках посредством типа возвращаемого значения, без использования исключений. Основная и единственная задача функции в функциональном программировании – вычислить и вернуть результат. Если функция может потерпеть неудачу, она должна вернуть значение, если вычисления выполнены без ошибок, или ничего не вернуть, если произошла ошибка. Как было показано в главе 9, это можно реализовать, сделав возвращаемое значение необязательным.

### 10.5.1 `std::optional<T>` как монада

Необязательные значения позволяют сообщить, что возвращаемое значение может отсутствовать. Это хороший прием, но использование необязательных значений имеет существенный недостаток: необходимость постоянно проверять наличие значения перед его использованием. В результате код, использующий функции `user_full_name` и `to_html`, которые мы определили выше и которые возвращают `std::optional<std::string>`, оказывается испещренным проверками:

```
std::optional<std::string> current_user_html()
{
    if (!current_login) {
        return {};
    }

    const auto full_name = user_full_name(current_login.value());

    if (!full_name) {
        return {};
    }

    return to_html(full_name.value());
}
```

А теперь представьте, что нужно составить цепочку из большого числа вызовов подобных функций. Код будет выглядеть как старый код на С, где мы вынуждены были проверять наличие ошибок после вызова почти каждой функции.

Поступим умнее. Увидев значение с некоторым контекстом, который нужно удалить перед вызовом другой функции, вспомним о монадах. Контекстом для необязательных значений является информация о наличии значения. Поскольку другие функции принимают обычные значения, этот контекст необходимо удалить перед их вызовом (см. рис. 10.11).

Именно это и позволяют делать монады: комбинировать функции, не выполняя лишней работы для обработки контекстной информации. `std::make_optional` – это функция-конструктор, создающая монаду, а `mbind` легко определить самому:

```
template <typename T, typename F>
auto mbind(const std::optional<T>& opt, F f)
-> decltype(f(opt.value()))
{
    if (opt) {
        return f(opt.value());
    } else {
        return {};
    }
}
```

Указать возвращаемый тип, потому что в отсутствие значения возвращается пустой экземпляр {}

Если opt содержит значение, преобразовать его с помощью f и вернуть ее результат, потому что она сама возвращает тип optional

Если значение отсутствует, вернуть пустой экземпляр std::optional

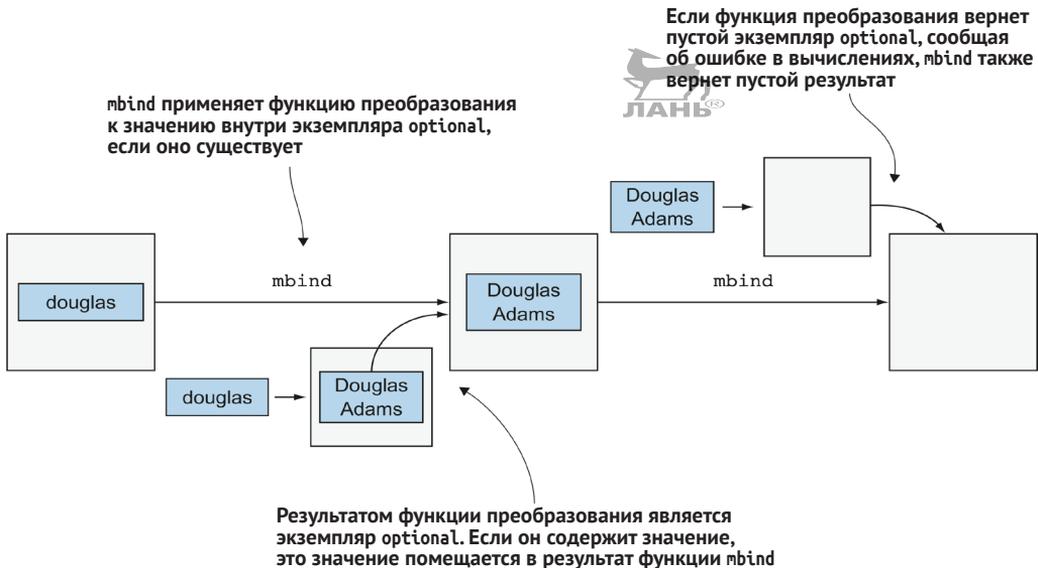


Рис. 10.11 Чтобы организовать обработку ошибок с использованием необязательных значений вместо исключений, функции, возвращающие необязательные значения, можно объединить с помощью `mbind`. Цепочка будет разорвана, как только любое из преобразований завершится неудачей и вернет пустое значение. Если все преобразования пройдут успешно, `mbind` вернет экземпляр `optional`, содержащий результат

Она вернет пустой результат, если исходное значение отсутствует или если функция `f` завершилась ошибкой и вернула пустой результат. В противном случае она вернет действительный результат. Объединив несколько функций, используя этот прием, можно получить автоматическую обработку ошибок: функции в цепочке будут вызываться до первой ошибки. Если все функции выполняются успешно, вы получите результат.

Теперь реализация `current_user_html` выглядит намного проще:

```
std::optional<std::string> current_user_html()
{
    return mbind(
        mbind(current_login, user_full_name),
        to_html);
}
```

Также можно создать преобразование `mbind`, поддерживающее синтаксис конвейеров и диапазонов, и упростить код еще больше:

```
std::optional<std::string> current_user_html()
{
    return current_login | mbind(user_full_name)
        | mbind(to_html);
}
```

Этот код напоминает пример функтора. В том примере мы использовали обычные функции и `transform`; здесь функции возвращают необязательные значения и используется `mbind`.

## 10.5.2 *expected<T, E>* как монада

Тип `std::optional` помогает обрабатывать ошибки, но он не передает никакой информации об ошибке. Тип `expected<T, E>` позволяет и обрабатывать ошибки, и узнать, какие ошибки возникли.

Так же как при использовании `std::optional<T>`, если вычисления прошли успешно, экземпляр `expected<T, E>` будет содержать значение, иначе – информацию об ошибке.

### Листинг 10.6 Использование монады `expected`

```
template <
    typename T, typename E, typename F,
    typename Ret = typename std::result_of<F(T)>::type
>
Ret mbind(const expected<T, E>& exp, F f)
{
    if (!exp) {
        return Ret::error(exp.error());
    }
    return f(exp.value());
}
```

← `f` может вернуть другой тип, поэтому его нужно определить, чтобы иметь возможность вернуть

← Если `exp` содержит ошибку, вернуть ошибку

← Иначе вернуть результат вызова `f`

Мы легко можем изменить функции так, чтобы проверять не только наличие значения, но и наличие ошибки. Для простоты обозначения ошибок используем целые числа:

```
expected<std::string, int> user_full_name(const std::string& login);
expected<std::string, int> to_html(const std::string& text);
```

В реализации `current_user_html` ничего менять не нужно:

```
expected<std::string, int> current_user_html()
{
    return current_login | mbind(user_full_name)
                       | mbind(to_html);
}
```

Как и прежде, в случае успеха эта функция вернет значение. Но если любая из функций в конвейере вернет ошибку, выполнение конвейера прервется и `current_user_html` вернет эту ошибку вызывающему коду (см. рис. 10.12).

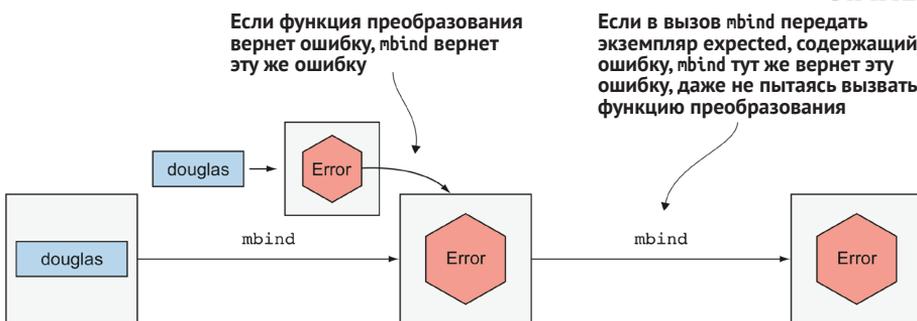


Рис. 10.12 При использовании типа `expected` для обработки ошибок можно связать в цепочку несколько преобразований, каждое из которых может завершиться неудачей, так же как при использовании типа `optional`. Как только возникнет первая ошибка, цепочка преобразований прервется, и вы получите эту ошибку в объекте `expected`, представляющем результат. Если вычисления выполняются успешно, вы получите преобразованное значение

Важно отметить, что монады должны иметь один шаблонный параметр, а здесь их два. Мы легко могли бы реализовать `mbind` для преобразования ошибки вместо значения, если бы потребовалось.

### 10.5.3 Исключения и монады

Тип `expected` позволяет использовать в роли ошибки любой тип: целочисленный, для передачи кода ошибки, строковый, для подробных описаний, и даже их комбинации. Также можно использовать ту же самую иерархию классов исключений, которая обычно применяется для обычной обработки исключений, указав тип ошибки `std::exception_ptr`.

### Листинг 10.7 Функции-обертки, использующие исключения в монаде `expected`

```

template <typename F,
         typename Ret = typename std::result_of<F()>::type,
         typename Exp = expected<Ret, std::exception_ptr>
Exp mtry(F f)
{
    try {
        return Exp::success(f());
    }
    catch (...) {
        return Exp::error(std::current_exception());
    }
}

```



f – это аргумент-функция без параметров. Чтобы вызвать ее с аргументами, нужно передать лямбда-выражение

Если исключение не возникнет, вернет экземпляр `expected` с результатом вызова `f`

Если возникнет любое исключение, вернет экземпляр `expected` с указателем на это исключение

Использование указателей на исключения с монадами `expected` позволяет легко совместить существующий код, использующий исключения, с обработкой ошибок на основе монады `expected`. Для примера ниже показано, как можно получить первого пользователя в системе. Функция, извлекающая список пользователей, может возбудить исключение, и код, вызывающий ее, тоже может возбудить исключение, если список пуст:

```

auto result = mtry( [= ] {
    auto users = system.users();

    if (users.empty()) {
        throw std::runtime_error("No users");
    }

    return users[0];
});

```



В результате вы получите значение или указатель на исключение.

Обратное также возможно: функцию, возвращающую экземпляр `expected` с указателем на исключение, легко интегрировать в код, использующий исключения. Например, вы можете создать функцию, возвращающую объект `expected` со значением или генерирующую исключение:

```

template <typename T>
T get_or_throw(const expected<T, std::exception_ptr>& exp)
{
    if (exp) {
        return exp.value();
    } else {
        std::rethrow_exception(exp.error());
    }
}

```

Эти две функции позволяют обрабатывать обе разновидности ошибок: и передаваемые в `expected`, как это принято в монадах, и исключения.

## 10.6 Обработка состояния с помощью монад

Одна из причин популярности монад в мире функционального программирования заключается в простоте реализации программ с состоянием. Нам этого не нужно, потому что состояние в C++ всегда было изменчивым.

Но если вы решите реализовать программу с использованием монад и различных цепочек их преобразования, вам пригодится возможность контролировать состояние каждой из этих цепочек. Как я уже говорил много раз, чтобы функции были чистыми, они не должны иметь побочных эффектов; они ничего не должны менять во внешнем мире. Как можно изменить состояние?

Нечистые функции могут изменять состояние неявно. Вызывая такую функцию, вы не видите, что происходит и что изменилось. Чтобы обеспечить чистоту изменениям в состоянии, они должны выполняться явно.

Самый простой способ – передавать всем функциям текущее состояние вместе с обычными аргументами, а функции должны возвращать новое состояние. Я говорил об этой идее в главе 5, когда мы развивали идею обработки изменяемого состояния путем создания новых миров вместо изменения текущего.

Рассмотрим пример. Вновь используем функции `user_full_name` и `to_html`, но теперь вместо обработки ошибок будем вести журнал отладки с описанием выполненных операций и их результатов. Этот журнал является образцом изменяемого состояния. Вместо типов-сумм `optional` и `expected`, которые могут содержать либо значение, либо что-то еще, обозначающее ошибку, используем тип-произведение, содержащий одновременно и значение, и дополнительную информацию (журнал отладки)<sup>1</sup>.

Самое простое решение – определить шаблонный класс:

```
template <typename T>
class with_log {
public:
    with_log(T value, std::string log = std::string())
        : m_value(value)
        , m_log(log)
    {
    }

    T value() const { return m_value; }
    std::string log() const { return m_log; }

private:
    T m_value;
    std::string m_log;
};
```

<sup>1</sup> В литературе такой тип обычно называют *монадой писателя*, потому что она просто записывает контекстную информацию, которая сама не используется в функциях `user_full_name` и `to_html`.

Теперь можно переопределить функции `user_full_name` и `to_html`, чтобы они возвращали значения вместе с журналом. Они обе будут возвращать результат вместе с их личными журналами выполненных действий:

```
with_log<std::string> user_full_name(const std::string& login);
with_log<std::string> to_html(const std::string& text);
```

Как и прежде, чтобы упростить комбинирование эти двух функций, нужно создать монаду из `with_log` (см. рис. 10.13). Функция конструирования монады имеет тривиально простую реализацию; в ее роли можно использовать конструктор `with_log` или написать функцию `make_with_log`, подобную `make_vector`, показанную выше в этой главе.

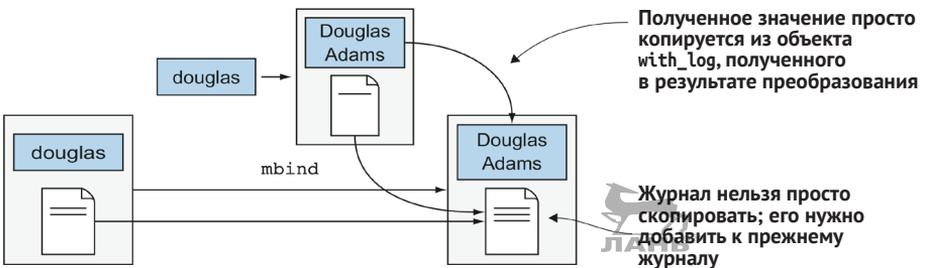


Рис. 10.13 В отличие от предыдущих монад, таких как `optional` и `expected`, где результат зависит только от последнего преобразования, `with_log` идет немного дальше. Она накапливает журнал из всех выполненных преобразований

Функция `mbind` – это место, где творится главное волшебство. Он должен принять экземпляр `with_log<T>`, содержащий значение и текущий журнал (состояние), функцию преобразования значения и вернуть преобразованное значение вместе с обновленным журналом. `mbind` должна вернуть новый результат вместе с новой отладочной записью, добавленной в конец старого журнала.

### Листинг 10.8 Обработка журнала в `mbind`

```
template <typename T,
         typename F>
         typename Ret = typename std::result_of<F(T)>::type
Ret mbind(const with_log<T1>& val, F f)
{
    const auto result_with_log = f(val.value());
    return Ret(result_with_log.value(),
              val.log() + result_with_log.log());
}
```

Преобразует заданное значение вызовом функции `f`, которая вернет результат и строку для записи в журнал

Нужно вернуть полученное значение, но запись для журнала, которую вернула `f`, нужно добавить в конец предыдущего журнала

Такой подход к журналированию имеет несколько преимуществ перед простой записью в стандартный вывод. Можно иметь сразу несколько журналов – по одному для каждой цепочки преобразования монад – без

использования любых специальных средств поддержки журналов. Одна функция может записывать информацию в несколько журналов, в зависимости от того, кто ее вызвал, и вам не придется указывать «эта запись пойдет в этот журнал, а эта – в тот». Кроме того, такой подход позволяет хранить журналы в цепочках асинхронных операций, и при этом они не будут пересекаться между собой.

## 10.7 Монады, продолжения и конкурентное выполнение

Итак, мы увидели несколько примеров монад. Все они содержали ноль или более значений и контекстную информацию. У кого-то из вас может сложиться представление, что монада – это своего рода контейнер, который знает, как работать со своими значениями, и если в программе есть экземпляр этого контейнера, она сможет получить доступ к этим элементам в любой момент.

Данная аналогия верна для многих монад, но не для всех. Возможно, вы помните из определения монады, что экземпляр монады можно создать из обычного значения или выполнить преобразование для значения, хранящегося в монаде. Но в нашем определении ничего не говорилось о функции извлечения значения из монады.

Вероятно, это звучит как простое упущение: как можно иметь контейнер, содержащий данные, но не иметь возможности получать эти данные? В конце концов, мы же можем получить доступ ко всем элементам в векторе, списке, экземпляре `optional` и т. д. Разве не так?

Нет, не так. Конечно, мы можем не называть *контейнерами* потоки ввода, такие как `std::cin`, но от этого они не перестанут быть таковыми. Они содержат элементы типа `char`. Также у нас есть тип `istream_range<T>`, который, по сути, является контейнером, содержащим ноль или больше значений типа `T`. Они отличаются от обычных контейнеров тем, что мы не знаем заранее их размеры и не можем получить доступ к элементам до того, как пользователь введет их.

С точки зрения человека, который пишет код, особой разницы нет. Можно написать обобщенную функцию, выполняющую такие операции, как `filter` и `transform`, которые могут работать как с векторами, так и с потоками ввода.

Но есть огромная разница в том, как такой код будет выполняться. При обработке потоков ввода код будет блокироваться, пока пользователь не введет все необходимые данные (см. рис. 10.14).

Интерактивные системы никогда не должны блокировать программы. Вместо того чтобы запрашивать данные и обрабатывать их, гораздо лучше сообщить программе, когда они станут доступны.

Представьте, что вам нужно извлечь заголовок веб-страницы. Вы подключаетесь к серверу, где находится страница, дожидаетесь ответа, а затем анализируете его, чтобы отыскать заголовок. Операция подключения к серверу и получения страницы может выполняться медленно,

но вы не должны блокировать программу в ожидании завершения этой операции.

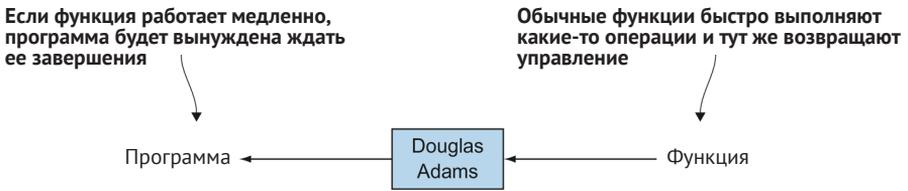


Рис. 10.14 При вызове функции из основной программы программа блокируется до ее завершения. Если функция работает медленно, программа может оказаться заблокированной на длительное время, хотя могла бы использовать это время для решения других задач

Вы должны отправить запрос, а затем продолжить выполнение других задач. Когда поступит ответ на запрос, вы извлечете данные и обработаете их.

Для этого нужен некий дескриптор, который даст доступ к данным после их получения. Назовем его *future* (будущее), потому что данные будут доступны не сразу, а в будущем (см. рис. 10.15). Эта идея будущих значений может пригодиться в самых разных ситуациях, поэтому не будем ограничиваться только строками (исходной веб-страницей); дескриптор должен быть универсального (обобщенного) типа `future<T>`.

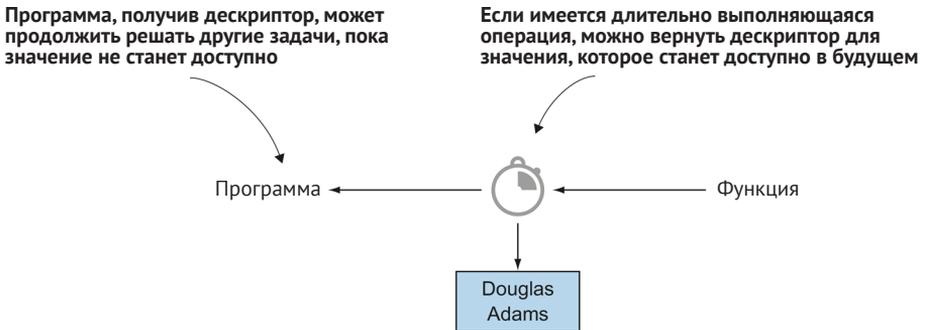


Рис. 10.15 Чтобы не ждать завершения медленной функции, эта функция должна вернуть дескриптор, который можно использовать для доступа к значению после его вычисления

Итак, дескриптор `future<T>` может не содержать значения, но оно может появиться в какой-то момент в будущем. Благодаря такому дескриптору можно разрабатывать программы, в которых разные компоненты выполняются одновременно или асинхронно. Каждый раз, когда потребуется выполнить продолжительную операцию или операцию, время выполнения которой неизвестно, можно заставить ее вернуть `future` вместо обычного значения. `future` – это контейнерный тип, но содержа-

щийся в нем элемент нельзя получить, пока асинхронная операция не завершилась и не поместила результат в контейнер.

### 10.7.1 *Tun future как монада*

Объект `future` по определению является монадой! Это разновидность типа-контейнера, который может содержать ноль или один результат, в зависимости от состояния асинхронной операции.

Давайте проверим, может ли `future` быть функтором. Чтобы называться функтором, объект должен иметь функцию `transform`, которая принимает `future<T1>` и функцию `f: T1 → T2` и возвращает экземпляр `future<T2>`.

Теоретически в этом нет никаких проблем. `future` – это дескриптор будущего значения. Если будет возможность получить значение, когда наступит будущее, его можно будет передать в функцию `f` и получить результат. В какой-то момент в будущем у вас появится преобразованное значение. Зная, как создать дескриптор, вы сможете создать функцию `transform` для `future`, а значит, `future` – это функтор.

Это довольно удобно, особенно когда важен не весь результат асинхронной операции, а только его часть, как в предыдущем примере получения заголовка веб-страницы. Имея тип `future` с функцией `transform` для него, легко можно сделать так:

```
get_page(url) | transform(extract_title)
```

В результате вы получите экземпляр `future` со строкой, из которой, когда она появится, будет извлечен заголовок веб-страницы.

Итак, выяснив, что `future` – это функтор, перейдем к следующему шагу и проверим, является ли он монадой. Сконструировать дескриптор, уже содержащий значение, просто. Гораздо интереснее другая часть – `mbind`. Давайте еще раз изменим тип значения, возвращаемого функциями `user_full_name` и `to_html`. Представим, что на этот раз для получения полного имени пользователя нужно подключиться к серверу и получить данные. Операция должна выполняться асинхронно. Также представим, что `to_html` выполняется медленно и тоже должна вызываться асинхронно. Обе операции должны возвращать `future` вместо нормальных значений:

```
future<std::string> user_full_name(const std::string& login);
future<std::string> to_html(const std::string& text);
```

Если использовать `transform` для комбинирования этих двух функций, в результате получится экземпляр `future` со значением внутри другого экземпляра `future`, что выглядит странно. Получается, что у нас появится дескриптор, который когда-нибудь в будущем даст вам другой дескриптор, который еще позже даст фактическое значение. Ситуация становится еще более сложной, если объединить больше двух асинхронных операций.

Решить эту проблему поможет `mbind`. Как и в предыдущих случаях, она позволит избежать вложения (см. рис. 10.16). На выходе вы всегда буде-

те получать дескриптор значения, а не дескриптор для дескриптора для дескриптора.

Функция `mbind` выполняет всю грязную работу. Она должна быть в состоянии узнать, наступило ли будущее, а затем вызвать функцию `transform` и получить дескриптор с конечным результатом. И самое главное: она должна немедленно вернуть дескриптор конечного результата.

С помощью `mbind` можно построить цепочку из любого количества асинхронных операций. Вот как это реализуется с использованием синтаксиса диапазонов:

```
future<std::string> current_user_html()
{
    return current_user() | mbind(user_full_name)
                          | mbind(to_html);
}
```

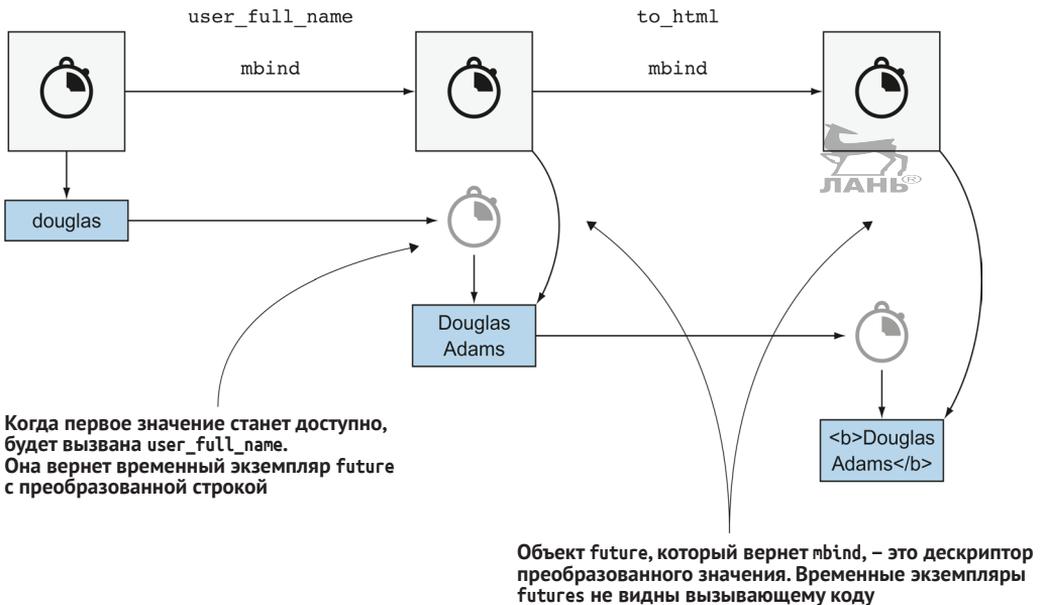


Рис. 10.16 Функция `mbind` в монадах позволяет связать несколько асинхронных операций. Результатом является объект `future`, представляющий результат последней асинхронной операции

Этот код выполняет три асинхронные операции в естественном порядке. Каждая следующая функция продолжает работу, начатую предыдущей. По этой причине функции, передаваемые в `mbind`, обычно называются *продолжениями*, а монаду `future` называют *монадой продолжения*.

Весь код сосредоточен в одном месте, читабелен и прост. Чтобы реализовать тот же процесс, используя привычные подходы, такие как функции обратного вызова или сигналы и слоты, эту единственную функцию

придется разбить на несколько отдельных функций. Каждый раз, вызывая асинхронную операцию, вам придется создать новую функцию для обработки результата.

## 10.7.2 Реализация *mona future*

Теперь, разобравшись с основной идеей, посмотрим, что мы имеем в мире C++. В версии C++11 появился тип `std::future<T>` – дескриптор будущего значения. Кроме фактического значения экземпляра `std::future` может содержать исключение, если асинхронная операция завершилась неудачей. В некотором смысле этот тип напоминает `expected<T, std::exception_ptr>`.

Недостаток этого типа состоит в том, что он не имеет механизма поддержки продолжений. Единственный способ получить значение – вызвать функцию-член `.get`, которая блокирует выполнение программы, если значение в `future` отсутствует (см. рис. 10.17). Чтобы избежать блокировки основной программы, нужно запустить отдельный поток выполнения, ожидающий, пока в `future` появится значение, или периодически опрашивать `future`.

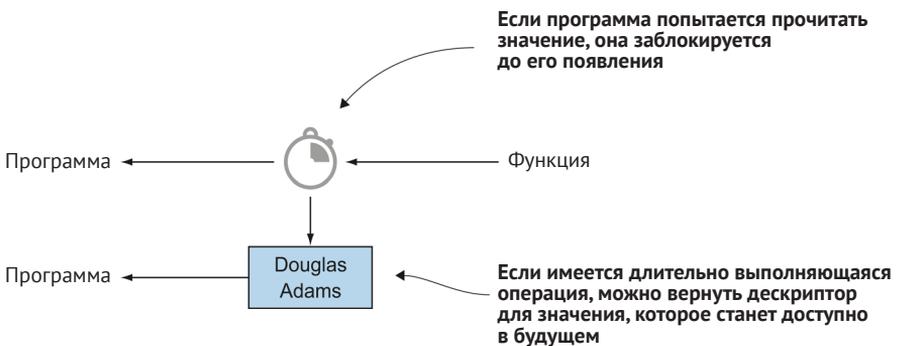


Рис. 10.17 Один из способов прочитать значение из `std::future` – вызвать функцию-член `.get`. К сожалению, она заблокирует вызывающий код, если значение еще недоступно. Этот тип можно с успехом использовать, когда требуется запустить параллельные вычисления и собрать результаты перед продолжением, но он плохо подходит для использования в интерактивных системах

Ни один из этих вариантов нам не подходит. В свое время было внесено предложение расширить тип `std::future` функцией-членом `.then`, которой можно передать функцию продолжения (см. рис. 10.18). Пока решение по этому предложению не принято, тем не менее многие поставщики стандартной библиотеки поддержали его и в C++17 реализовали расширенный класс `future` как `std::experimental::future`. В отсутствие доступа к компилятору, поддерживающего C++17, можно использовать класс `boost::future`, который тоже поддерживает продолжения.

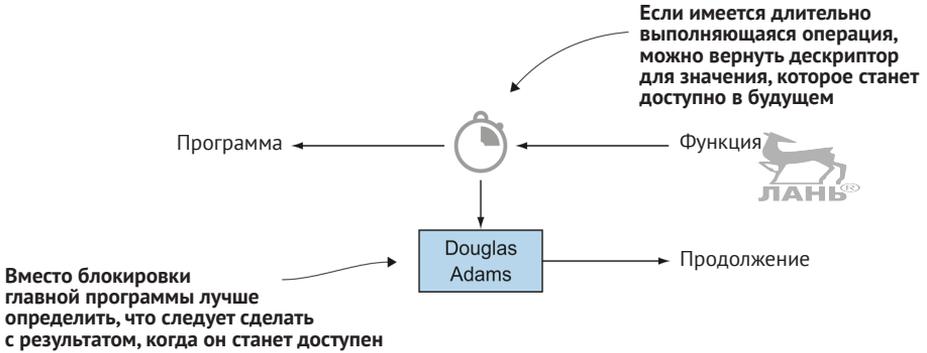


Рис. 10.18 Вместо блокировки программы лучше подключить функцию продолжения к объекту future. Когда значение станет доступно, для его обработки будет вызвана функция продолжения

Функция-член `.then` действует так же, как `mbind`, но с небольшим отличием: `mbind` в монадах принимает функцию, аргументом которой является обычное значение, а результатом – объект `future`, тогда как `.then` принимает функцию, аргументом которой является уже завершившийся экземпляр `future`, возвращающую новый экземпляр `future`. То есть `then` не делает `future` монадой, но упрощает правильную реализацию функции `mbind`.

**Листинг 10.9 Реализация mbind с использованием функции-члена .then**

```
template <typename T, typename F>
auto mbind(const future<T>& future, F f) ← | Принимает функцию преобразования из T
{                                       | в экземпляр future<T2>
    return future.then(
        [(future<T> finished) { | Принимает функцию, возвращающую future<T>,
            return f(finished.get()); | и возвращает future<T2>. Чтобы извлечь значение
        }];                          | из future для передачи в f, нужно использовать
    );                                 | лямбда-выражение
}
```

Не вызывает блокировки, потому что продолжение вызывается, только когда значение доступно (или если возникло исключение)

Некоторые другие библиотеки, находящиеся за рамками обычной экосистемы C++, реализуют свои типы, напоминающие `future`. Большинство из них использует базовую идею, добавляя поддержку ошибок, которые могут возникнуть во время асинхронных операций.

Наиболее яркими примерами, помимо стандартной библиотеки и `boost`, являются класс `Future` в библиотеке `Folly` и `QFuture` в библиотеке `Qt`. `Folly` предлагает ясную реализацию идеи будущих значений, которая никогда не блокирует выполнения (`.get` генерирует исключение, если значение пока недоступно, но не блокирует программу). В `QFuture` реализован способ объединения продолжений с помощью сигналов и слотов, но функция `.get` в ней также может заблокировать программу, подобно

своему аналогу в стандартной библиотеке. `QFuture` имеет дополнительные функции, выходящие за рамки основной идеи, позволяя собрать несколько значений с течением времени. Несмотря на эти различия, все классы можно использовать для объединения асинхронных операций с помощью `mbind`.

## 10.8 Композиция монад

До сих пор мы имели экземпляр объекта-монады и использовали функцию `mbind` для его передачи в функцию монады, возвращающую экземпляр того же типа, который можно связать с другой функцией, и т. д.

Это подобно обычному применению функции, когда функции передается обычное значение и она возвращает результат, который можно передать другой функции, и т. д. Если убрать исходное значение из уравнения, получится список функций, вызываемых друг за другом, и эта композиция дает новую функцию.

То же самое возможно с монадами. Поведение связывания можно выразить без исходного экземпляра монады и сосредоточиться на создании монадических функций.

В этой главе вы видели несколько вариантов функций `user_full_name` и `to_html`. Большинство из них получали строку и возвращали строку, завернутую в монадический тип. Они выглядели, как показано ниже (где вместо `M` использовались `optional`, `expected` и другие типы-обертки):

```
user_full_name : std::string → M<std::string>
to_html       : std::string → M<std::string>
```

Функция, объединяющая эти две функции, должна принимать экземпляр `M<std::string>`, представляющий пользователя, чье имя нужно получить. Внутри имя должно быть передано через два вызова `mbind`:

```
M<std::string> user_html(const M<std::string>& login)
{
    return mbind(
        mbind(login, user_full_name),
        to_html);
}
```

Эта рабочая реализация, но она избыточна. Было бы проще, если бы была возможность сказать, что `user_html` – это композиция функций `user_full_name` и `to_html`.

Мы легко можем создать такую обобщенную функцию композиции. Объединяя обычные функции, мы имеем две функции:  $f: T_1 \rightarrow T_2$  и  $g: T_2 \rightarrow T_3$ . А в результате получаем функцию  $T_1 \rightarrow T_3$ . В случае с монадами ситуация немного меняется. Функции возвращают не обычные значения, а значения, заключенные в монаду. Следовательно, мы имеем  $f: T_1 \rightarrow M<T_2>$  и  $g: T_2 \rightarrow M<T_3>$ .

## Листинг 10.10 Композиция двух монад

```
template <typename F, typename G>
auto mcompose(F f, G g) {
    return [=](auto value) {
        return mbind(f(value), g);
    };
}
```

Теперь функцию `user_html` можно определить так:

```
auto user_html = mcompose(user_full_name, to_html);
```

Функцию `mcompose` также можно использовать для *более простых* монад, таких как диапазоны (а также векторы, списки и массивы). Представьте, что есть функция `children`, которая возвращает диапазон, содержащий всех детей определенного человека. Она имеет правильную сигнатуру для монадической функции: принимает значение `person_t` и возвращает диапазон с экземплярами `person_t`. На ее основе можно создать функцию, возвращающую всех внуков:

```
auto grandchildren = mcompose(children, children);
```

Функция `mcompose` не только позволяет писать короткий и обобщенный код, но и дает одно теоретическое преимущество. Как вы помните, я перечислил три правила монад, которые трудно понять с первого раза. С помощью этой функции композиции их можно выразить намного проще.

Комбинация любой монадической функции с функцией-конструктором соответствующей монады дает в результате ту же функцию:

```
mcompose(f, construct) == f
mcompose(construct, f) == f
```

Правило ассоциативности утверждает, что если есть три функции `f`, `g` и `h`, которые нужно объединить, то не имеет значения, будут ли сначала объединены `f` и `g` и затем результат попадет в `h` или сначала будут объединены `g` и `h` и затем результат будет передан в `f`:

```
mcompose(f, mcompose(g, h)) == mcompose(mcompose(f, g), h)
```

Это также называют *композицией Клейсли* (Kleisli composition), и в целом она имеет те же характеристики, что и композиция обычных функций.

**СОВЕТ** За дополнительной информацией по теме, рассматривавшейся в этой главе, обращайтесь по адресу: <https://forums.manning.com/posts/list/43779.page>.



## Итоги

- Говоря о шаблонах программирования, обычно подразумевают объектно-ориентированное программирование, но мир функционального программирования также заполнен типичными идиомами и абстракциями, такими как функтор и монада.
- Функторы – это структуры, похожие на коллекции, которые знают, как применить функцию преобразования к своему содержимому.
- Монады умеют все то же, что и функторы, но имеют две дополнительные операции: операцию создания монадических значений из обычных и операцию устранения избыточных уровней вложенности монадических значений.
- Функторы позволяют легко обрабатывать и преобразовывать типы-обертки, а монады – комбинировать функции, возвращающие типы-обертки.
- Часто монады полезно представлять как контейнеры. Но термин «контейнер» используется здесь в более общем смысле и охватывает такие случаи, как монада продолжения – контейнер, который в конечном итоге будет содержать данные.
- В реальном мире вы можете открыть контейнер и посмотреть, что находится внутри, но с монадами это невозможно. Вы можете указать контейнеру, что делать со значениями, которые в нем имеются, но вы не всегда можете получить прямой доступ к значениям.

---

# 11

## Метапрограммирование на шаблонах

---

### О чем говорится в этой главе:

- манипулирование типами во время компиляции;
- использование `constexpr-if` для организации ветвления во время компиляции;
- статическая интроспекция для проверки типов свойств во время компиляции;
- использование `std::invoke` и `std::apply`;
- создание предметно-ориентированного языка (DSL) для определения транзакций с целью изменения данных.



Многие представляют себе программирование так: программист пишет код и компилирует его, а затем пользователь запускает скомпилированный двоичный файл. В целом это верное представление.

Но в C++ есть возможность писать программы другого типа, которые выполняются компилятором, пока он компилирует код. Это может показаться странным (какой смысл выполнять код, когда отсутствует ввод от пользователя и данные для обработки?). Дело в том, что задача кода, выполняющегося во время компиляции, состоит не в обработке пользовательских данных, которые будут доступны только после запуска скомпилированной программы, а в манипулировании компонентами будущей программы, доступными во время компиляции, – типами и сгенерированным кодом.

Это часто бывает необходимо при разработке оптимизированного универсального кода. Вам может понадобиться реализовать алгоритм

по-разному, в зависимости от типа входных данных. Например, при работе с коллекциями часто важно знать, поддерживает ли коллекция произвольный доступ к ее элементам, и в зависимости от этого вы можете выбрать совершенно разные реализации алгоритма.

Основным механизмом программирования (или метапрограммирования) кода, выполняющегося во время компиляции, в C++ являются шаблоны. Возьмем для примера определение шаблона класса `expected` из главы 9:

```
template<typename T, typename E = std::exception_ptr>
class expected
{
    ...
};
```

Этот шаблон параметризован двумя типами, `T` и `E`. Определяя эти два параметра, мы получаем конкретный тип, экземпляры которого можно создавать в программе. Например, если для `T` указать тип `std::string`, а для `E` – тип `int`, получится конкретный тип `expected<std::string, int>`. Если для обоих параметров, `T` и `E`, указать тип `std::string`, получится другой конкретный тип: `expected<std::string, std::string>`.

Эти два типа похожи и реализованы одинаково, но тем не менее являются разными типами. Мы не сможем преобразовать экземпляр из одного типа в другой. Кроме того, в скомпилированном двоичном коде эти два типа будут иметь отдельные реализации, не зависящие друг от друга.

Итак, у нас есть нечто с именем `expected`, принимающее два типа и дающее в результате некоторый тип. Это нечто похоже на функцию, только работает не со значениями, а с самими типами. Будем называть такие «функции» *метафункциями*, чтобы не путать их с обычными функциями.

Метапрограммирование с использованием шаблонов (или просто метапрограммирование) – обширная тема, заслуживающая отдельной книги. Поэтому в этой главе мы затронем лишь некоторые части из такой книги. Основное внимание мы сосредоточим на версии C++17, потому что в ней появились дополнительные возможности, значительно упрощающие метапрограммирование.

## 11.1 Манипулирование типами во время компиляции

Допустим, нам понадобилось реализовать обобщенный алгоритм суммирования всех элементов в данной коллекции. Он должен принимать аргумент с коллекцией и возвращать сумму всех ее элементов. Возникает вопрос: какой тип будет иметь значение, возвращаемое этой функцией?

Для алгоритма `std::accumulate` ответ на этот вопрос прост: тип результата совпадает с типом начального значения, используемого для накопления. Но у нас будет функция, которая получает только коллекцию, без начального значения:

```
template <typename C>
??? sum(const C& collection)
{
    ...
}
```

Ответ, казалось бы, очевиден: тип результата должен совпадать с типом элементов, содержащихся в данной коллекции. Если функция получает вектор целых чисел, она должна вернуть целое число. Если она получает связанный список вещественных чисел, она должна вернуть вещественное число; а для любой коллекции строк результатом должна быть строка.

Проблема в том, что нам известен только тип коллекции, а тип ее элементов – нет. Нам нужно написать метафункцию, получающую тип коллекции и возвращающую тип ее элементов.

Общим свойством большинства коллекций является возможность использовать итераторы для обхода их элементов. Операция разыменования итератора возвращает значение типа элемента. Например, вот как можно определить переменную для хранения первого элемента коллекции:

```
auto value = *begin(collection);
```

Компилятор способен автоматически определить тип переменной. Если функции передать коллекцию целых чисел, переменная `value` получит тип `int`, как раз то, что нам нужно. Теперь нам нужно воспользоваться этой способностью компилятора, чтобы определить метафункцию, которая делает то же самое.

Важно отметить, что во время компиляции еще не известно, содержит ли коллекция какие-либо элементы. Компилятор сможет определить тип `value`, даже если во время выполнения передать пустую коллекцию. Конечно, в этом случае попытка разыменовать итератор, возвращаемый функцией `begin`, приведет к ошибке, но в данный момент нас это не волнует.

Если есть выражение и нужно получить его тип, можно использовать спецификатор `decltype`. Остается только написать метафункцию, которая принимает коллекцию и возвращает тип ее элементов. Реализовать такую метафункцию можно как псевдоним обобщенного типа, например<sup>1</sup>:

```
template <typename T>
using contained_type_t = decltype(*begin(T()));
```

Рассмотрим этот код подробнее, чтобы понять происходящее. Спецификация шаблона `template <typename T>` сообщает, что далее следует метафункция с именем `contained_type_t`, которая принимает один аргумент:

<sup>1</sup> За дополнительной информацией о псевдонимах типов обращайтесь по адресу: [http://ru.cppreference.com/w/cpp/language/type\\_alias](http://ru.cppreference.com/w/cpp/language/type_alias).

тип `T`. Эта метафункция возвращает тип выражения, содержащегося в спецификаторе `decltype`.

В объявлении переменной `value` мы передали в вызов `begin` конкретную коллекцию. Здесь нет коллекции; есть только ее тип. Поэтому создаем экземпляр вызовом конструктора по умолчанию `T()` и передаем его в `begin`. Полученный итератор разыменовывается, и метафункция `contained_type_t` возвращает тип значения, на которое указывает итератор.

В отличие от предыдущего фрагмента, этот код не приведет к ошибке во время выполнения, потому что манипуляции с типами происходят во время компиляции. Спецификатор `decltype` не выполняет свой код; он просто возвращает тип выражения, не вычисляя его. Казалось бы, решение найдено, но у нашей метафункции есть две существенные проблемы.

Во-первых, она предполагает, что тип `T` имеет конструктор по умолчанию. Конечно, все коллекции в стандартной библиотеке имеют конструкторы по умолчанию, но коллекция может быть нестандартного типа и не иметь конструктора по умолчанию. Например, обсуждавшийся выше тип `expected<T, E>` можно представить как коллекцию, содержащую ноль или одно значение типа `T`, и она не имеет конструктора по умолчанию. Чтобы создать пустой экземпляр `expected<T, E>`, нужно явно определить ошибку, объясняющую, почему он пустой.

Метафункцию `contained_type_t` не получится использовать с этим типом; обращение к `T()` вызовет ошибку компилятора. Исправить проблему можно, заменив вызов конструктора обращением к вспомогательной метафункции `std::declval<T>()`. Она принимает произвольный тип `T`, будь то коллекция, целочисленный или любой нестандартный тип, такой как `expected<T, E>`, и имитирует создание экземпляра этого типа так, чтобы ее результат можно было использовать в метафункциях, требующих значений вместо типов, что часто имеет место при использовании `decltype`.

В первоначальном сценарии мы точно знали, как найти сумму элементов коллекции. Единственная проблема заключалась в том, что мы не знали тип возвращаемого значения. Метафункция `contained_type_t` помогает узнать тип элементов коллекции, а значит, ее можно использовать для определения типа возвращаемого значения функции, суммирующей эти элементы.

Вот как можно ее использовать:

```
template <typename C,  
         typename R = contained_type_t<C>>  
R sum(const C& collection)  
{  
    ...  
}
```

Мы называем эти конструкции *метафункциями*, но в действительности они являются обычными шаблонами, которые что-то определяют. Метафункции *вызываются* при создании шаблонов. В данном случае создается экземпляр шаблона `contained_type_t` для типа коллекции `C`.

### 11.1.1 Проверка правильности определения типа

Вторая проблема с реализацией `contained_type_t` заключается в том, что она делает не совсем то, что нам нужно. Начав использовать ее, вы вскоре столкнетесь с проблемами. Например, при попытке скомпилировать предыдущий код компилятор сообщит, что результатом `contained_type_t` для `std::vector<T>` является не `T`, а что-то другое.

В подобных случаях, когда ожидается один тип, а компилятор утверждает, что имеет другой, полезно иметь возможность проверить фактический тип. Можно, конечно, положиться на возможности механизма IDE, определяющего тип, который иногда ошибается, но лучше заставить компилятор сообщить получившийся тип.

Один из полезных приемов, которые можно использовать, – объявить шаблон класса без реализации. После этого, когда понадобится проверить определенный тип, попробуйте создать экземпляр этого шаблона, и компилятор сообщит об ошибке, точно указав переданный тип.

#### Листинг 11.1 Проверка типа, полученного метафункцией `contained_type_t`

```
template <typename T>
class error;

error<contained_type_t<std::vector<std::string>>>>();
```



При попытке скомпилировать этот код компилятор выведет сообщение, как показано ниже (которое может немного отличаться в разных версиях компилятора):

```
error: invalid use of incomplete type
'class error<const std::string>'
```

Метафункция `contained_type_t` определила, что элементы являются константными ссылками на строки, а не строками, как нам хотелось и как определяет объявление `auto value`.

Этого следовало ожидать, потому что объявление `auto` использует несколько иные правила определения типа, чем `decltype`. Спецификатор `decltype` возвращает точный тип заданного выражения, тогда как `auto` действует умнее и определяет типы практически так же, как это делается для параметров типов в шаблонах.

Чтобы преобразовать константную ссылку на тип в сам тип, нужно удалить ссылочную часть типа и спецификатор `const`. Для удаления квалификаторов `const` и `volatile` можно использовать метафункцию `std::remove_cv_t`, а для удаления ссылочной части – метафункцию `std::remove_reference_t`.

#### Листинг 11.2 Полная реализация метафункции `contained_type_t`

```
template <typename T>
using contained_type_t =
    std::remove_cv_t<
```

```

std::remove_reference_t<
    decltype(*begin(std::declval<T>()))
>
>;

```

Если теперь проверить результат `contained_type_t<std::vector<std::string>>`, вы получите `std::string`.



### Заголовок `<type_traits>`

Большинство стандартных метафункций определены в заголовке `<type_traits>`. В нем определено с десяток метафункций, предназначенных для манипулирования типами и имитации инструкций `if` и логических операций в метапрограммах.

Метафункции с окончанием `_t` в именах появились в C++14. Для выполнения аналогичных манипуляций с типами в старых компиляторах, поддерживающих только возможности C++11, придется использовать более подробные конструкции. Например, на странице [http://ru.cppreference.com/w/cpp/types/remove\\_cv](http://ru.cppreference.com/w/cpp/types/remove_cv) вы найдете пример реализации обычной `remove_cv`, действующей подобно одноименной метафункции с окончанием `_t`.

`static_assert` – еще одна утилита, которая может пригодиться при разработке метафункций. Статические утверждения (`static assertions`) могут обеспечить соблюдение определенного правила во время компиляции. Например, с их помощью можно написать серию тестов для проверки реализации `contained_type_t`:

```

static_assert(
    std::is_same<int, contained_type_t<std::vector<int>>>(),
    "std::vector<int> should contain integers");

static_assert(
    std::is_same<std::string, contained_type_t<std::list<std::string>>>(),
    "std::list<std::string> should contain strings");

static_assert(
    std::is_same<person_t*, contained_type_t<std::vector<person_t*>>>(),
    "std::vector<person_t> should contain people");

```

`static_assert` принимает значение типа `bool`, вычисляемое на этапе компиляции, и прерывает компиляцию, получив значение `false`. Предыдущий пример проверяет, что `contained_type_t` возвращает ожидаемый тип.

Типы нельзя сравнить с помощью оператора `==`. Нужно использовать его *метаэквивалент* `std::is_same`. Метафункция `std::is_same` принимает два типа и возвращает `true`, если типы идентичны, и `false` в противном случае.

### 11.1.2 Сопоставление с образцом во время компиляции

Теперь посмотрим, как реализованы метафункции, использованные выше. Начнем с реализации собственной версии метафункции `is_same`. Она должна принимать два аргумента и возвращать `true`, если типы совпадают, и `false` в противном случае. Метафункция, манипулирующая типами, не может вернуть значение `true` или `false`, но может вернуть тип `std::true_type` или `std::false_type`. Их можно рассматривать как константы типа `bool` для метафункций.

При определении метафункций часто полезно определить, какой должен получиться результат в общем случае, а затем определить конкретные случаи и рассчитать результаты для них. В случае с `is_same` у нас есть два варианта: она получила два разных типа и должна вернуть `std::false_type`, и она получила один и тот же тип в обоих параметрах и должна вернуть `std::true_type`. Первый вариант более общий, поэтому сначала рассмотрим его:

```
template <typename T1, typename T2>
struct is_same : std::false_type {};
```

Это определение создает метафункцию с двумя параметрами, которая всегда возвращает `std::false_type`, независимо от фактических типов `T1` и `T2`.

Теперь второй вариант:

```
template <typename T>
struct is_same<T, T> : std::true_type {};
```

Это – специализация предыдущего шаблона, которая будет использоваться, только если `T1` и `T2` представляют один и тот же тип. Когда компилятор встретит `is_same<int, contained_type_t<std::vector<int>>>`, он сначала вычислит результат метафункции `contained_type_t` (и получит `int`). Затем найдет все определения `is_same`, которые можно применить к `<int, int>`, и выберет наиболее конкретное (см. рис. 11.1).

В предыдущей реализации оба варианта подходят для `<int, int>` – и тот, что наследует `std::false_type`, и тот, что наследует `std::true_type`. Поскольку второй вариант более конкретный, компилятор выберет его.

А если компилятор встретит `is_same<int, std::string>?` Он сгенерирует список определений, применимых к `int` и `std::string`. В этом случае более конкретное определение неприменимо, потому что в `<T, T>` нельзя подставить вместо `T` ни один из аргументов и получить в результате `<int, std::string>`. Компилятор выберет единственное допустимое определение: первое, наследующее `std::false_type`.

`is_same` – это метафункция, возвращающая булеву константу во время компиляции. Аналогично можно реализовать функцию, возвращающую измененный тип. Для примера напишем функцию `remove_reference_t`, эквивалентную одноименной функции из стандартной библиотеки. На этот раз у нас есть три варианта:

- функции передан нессылочный тип (это самый общий вариант);
- функции передана ссылка на левостороннее значение (lvalue);
- функции передана ссылка на правостороннее значение (rvalue).

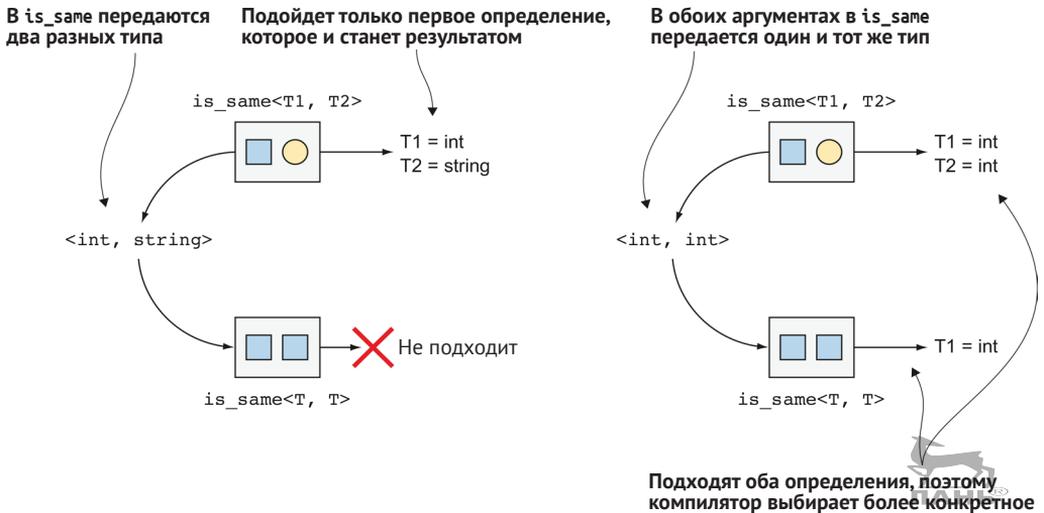


Рис. 11.1 Когда в `is_same` передаются разные аргументы, подходящим будет только первое определение, возвращающее `false_type`. Если в обоих параметрах передать один и тот же тип, подойдут оба определения, но использоваться будет более конкретное – возвращающее `true_type`

В первом случае нужно вернуть тип без изменения, а во втором и третьем случаях мы должны удалить ссылочные части.

В отличие от `is_same`, функцию `remove_reference` нельзя реализовать через наследование результата. Она должна возвращать точный тип, а не какой-то другой, унаследованный из результата. Поэтому создадим шаблон структуры, который будет содержать определение вложенного типа для получения точного результата.

**Листинг 11.3 Реализация метафункции `remove_reference_t`**

<pre>template &lt;typename T&gt; struct remove_reference {     using type = T; };</pre>	<p>  В общем случае <code>remove_reference&lt;T&gt;::type</code> – это тип T: эта версия возвращает полученный тип</p>
<pre>template &lt;typename T&gt; struct remove_reference&lt;T&amp;&gt; {     using type = T; };</pre>	<p>  Если получена левосторонняя (lvalue) ссылка T&amp;, убрать ссылочную часть и вернуть T</p>
<pre>template &lt;typename T&gt; struct remove_reference&lt;T&amp;&amp;&gt; {     using type = T; };</pre>	<p>  Если получена правосторонняя (rvalue) ссылка T&amp;&amp;, убрать ссылочную часть и вернуть T</p>

Работая над метафункцией `contained_type_t`, мы создали псевдоним шаблонного типа. Здесь используется другой подход. Шаблон структуры определяет вложенный псевдоним с именем `type`. Чтобы вызвать метафункцию `remove_reference` и получить результирующий тип, нужно использовать более подробный синтаксис, чем в `contained_type_t`: создать экземпляр шаблона `remove_reference` и получить определение вложенного типа. Для этого придется писать `typename remove_reference<T>::type` всякий раз, когда понадобится его использовать.

Чтобы избавиться от лишнего ввода и не писать каждый раз `typename ...::type`, можно определить вспомогательную метафункцию `remove_reference_t`, подобную той, что в C++ объявлена в заголовке `type_traits`:

```
template <typename T>
using remove_reference_t<T> =
    typename remove_reference<T>::type;
```



Встретив шаблон `remove_reference` с конкретным аргументом, компилятор отыщет все определения, соответствующие этому аргументу, и выберет наиболее конкретное.

Например, встретив код `remove_reference_t<int>`, компилятор проверит, какое из предыдущих определений можно применить. В этом случае применить можно только первое определение, и компилятор поймет, что `T` – это `int`. Второе и третье определения в данной ситуации неприменимы, потому что нет такого типа `T`, для которого ссылка на `T` была бы типом `int` (`int` не является ссылочным типом). Поскольку существует только одно соответствующее определение, оно и будет использовано, и результатом станет тип `int`.

Встретив код `remove_reference_t<int&>`, компилятор снова отыщет все применимые определения. На этот раз он найдет два подходящих определения. Первое, наиболее общее, когда тип `T` является типом `int&`. Второе определение тоже будет признано подходящим, потому что типу `T&` соответствует `int&`; т. е. типу `T` соответствует тип `int`. Третье определение будет признано неподходящим, потому что ожидает правостороннюю (`rvalue`) ссылку. Из двух совпадений второе является более конкретным, поэтому тип `T` (и результат) будет опознан как тип `int`. Тот же процесс повторится, когда компилятор встретит `remove_reference_t<int&&>`, с той лишь разницей, что второе определение будет признано неподходящим, а третье – наоборот, применимым.

Теперь, зная, как получить тип элемента коллекции, можно, наконец, реализовать функцию суммирования ее элементов. Предположим, что значение, созданное конструктором по умолчанию типа элемента, является единичным (нейтральным) значением для сложения и его можно передать как начальное значение в `std::accumulate`:

```
template <typename C,
         typename R = contained_type_t<C>>
R sum_iterable(const C& collection)
{
```

```

return std::accumulate(begin(collection),
                       end(collection),
                       R());
}

```

Если передать этой функции определенную коллекцию, компилятор определит типы `C` и `R`. Тип `C` будет определен как тип коллекции, переданной функции.

Поскольку тип `R` не задан явно, компилятор симитирует вызов конструктора в `contained_type_t<C>`, получит тип элементов, разыменовав итератор для коллекции `C`, а затем удалит из него спецификатор `const` и ссылочную часть.

### 11.1.3 Получение метаинформации о типах

В предыдущих примерах мы увидели, как узнать тип элемента коллекции. Проблема в том, что это очень утомительная задача и при ее реализации легко ошибиться. Поэтому обычно принято указывать такую информацию в классе коллекции. Обычно коллекции указывают тип своих элементов в виде определения вложенного типа с именем `value_type`. Эту информацию легко добавить в реализацию `expected`:

```

template <typename T, typename E>
class expected {
public:
    using value_type = T;
    ...
};

```

Все классы контейнеров в стандартной библиотеке – даже `std::optional` – предлагают такую возможность. Ее также должны предлагать все классы контейнеров из сторонних библиотек.

С помощью этой дополнительной информации можно избежать необходимости определять свои метафункции и просто писать такой код:

```

template <typename C,
         typename R = typename C::value_type>
R sum_collection(const C& collection)
{
    return std::accumulate(begin(collection),
                           end(collection),
                           R());
}

```

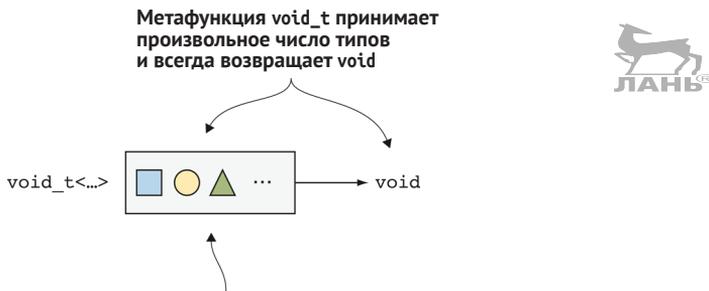
Использование вложенного типа `value_type` дает еще одно преимущество в случаях, когда итератор коллекции возвращает не сам элемент, а экземпляр типа-обертки. Если применить метафункцию `contained_type_t` к такой коллекции, она вернет тип обертки, тогда как вам, вероятно, нужен тип элемента. Добавляя определение типа `value_type`, коллекция сообщает, как именно будут выглядеть содержащиеся в ней элементы.

## 11.2 Проверка свойств типа во время компиляции

Итак, мы создали две функции `sum`: одну, более предпочтительную, для коллекций, имеющих вложенное определение типа `value_type`; и другую – для любых итерируемых коллекций. Было бы неплохо иметь возможность определить, имеет ли некоторая конкретная коллекция вложенный тип `value_type`, и выбрать соответствующую реализацию.

Но, прежде чем продолжить, я должен представить самую странную метафункцию, которая принимает произвольное количество типов и всегда возвращает `void` (см. рис. 11.2):

```
template <typename...>
using void_t = void;
```



Чтобы результат можно было вычислить, все переданные типы должны быть действительными

**Рис. 11.2** Метафункция `void_t` – странная: она игнорирует свои параметры и всегда возвращает `void`. Ее польза в том, что она завершится успехом, только если типы, переданные ей, являются действительными. Иначе `void_t` потерпит неудачу, и компилятор проигнорирует определение, где она использовалась

Эта метафункция может показаться бесполезной, но дело в том, что польза от данной функции заключена не в ее результате. `void_t` позволяет проверить правильность заданных типов или выражений в ситуациях, когда во время компиляции *невозможность подстановки не является ошибкой* (Substitution Failure Is Not An Error, SFINAE). SFINAE – это правило, которое используется при выборе перегруженной версии шаблона. Если замена параметра шаблона выведенным типом потерпит неудачу, компилятор не сообщит об ошибке; он просто проигнорирует эту конкретную перегруженную версию.

**ПРИМЕЧАНИЕ** Метафункция `void_t` появилась в стандартной библиотеке в версии C++17. Если вы используете более старый компилятор, прочитайте статью [http://en.cppreference.com/w/cpp/types/void\\_t](http://en.cppreference.com/w/cpp/types/void_t), чтобы узнать, как самому реализовать `void_t`.

Метафункция `void_t` – как раз то, что нам нужно. Мы можем передать ей любое число типов, и если какой-то тип окажется недопустимым, соответствующая перегруженная версия будет игнорироваться компи-

лятором. Напишем метафункцию, проверяющую наличие вложенного типа `value_type` в заданном типе.



#### Листинг 11.4 Метафункция, проверяющая наличие вложенного типа `value_type` в заданном типе

```
template <typename C,
          typename = void_t<>>
struct has_value_type
    : std::false_type {};

template <typename C>
struct has_value_type<C,
                    void_t<typename C::value_type>>
    : std::true_type {};
```

Общий случай: предполагается, что данный тип не имеет вложенного определения `value_type`

Особый случай: вступает в силу, только если `typename C::value_type` является существующим типом (если `C` имеет вложенное определение `value_type`)

Теперь можно определить функцию, суммирующую все элементы коллекции и действующую по-разному, в зависимости от наличия у коллекции вложенного определения `value_type`:

```
template <typename C>
auto sum(const C& collection)
{
    if constexpr (has_value_type<C>()) {
        return sum_collection(collection);
    } else {
        return sum_iterable(collection);
    }
}
```



Для коллекции, не имеющей вложенного определения `value_type`, нельзя вызвать `sum_collection`, потому что при попытке определить параметры шаблона компилятор потерпит неудачу.

Решить проблему поможет `constexpr-if`. Обычный оператор `if` проверяет условие во время выполнения и требует, чтобы обе ветви были компилируемыми. Оператор `constexpr-if`, напротив, требует, чтобы обе ветви имели правильный синтаксис, но компилироваться будет только одна из них. Вызов `sum_collection` с коллекцией, не имеющей вложенного определения `value_type`, приведет к ошибке; но в данном случае компилятор будет видеть только ветвь `else`, потому что `has_value_type<C>()` вернет `false`.

А что случится, если в `sum` передать что-то, что не имеет определения `value_type` и не является итерируемой коллекцией? Компилятор сообщит об ошибке невозможности вызова `sum_iterable` для этого типа. Однако было бы лучше добавить защиту от таких ситуаций, как подобную защиту от вызова `sum_collection`, когда это невозможно.

Для этого нужно проверить, является ли коллекция итерируемой, т. е. возможность вызова ее методов `begin` и `end`, а также возможность разыменования итератора, возвращаемого функцией `begin`. В данном случае не

важно, можно ли разыменовать итератор `end`, потому что это может быть специальный тип с ограничителем (см. главу 7).

Снова используем `void_t`. Метафункция `void_t` позволяет проверить правильность не только типов, но также выражений, если прибегнуть к помощи `decltype` и `std::declval`.

### Листинг 11.5 Метафункция, проверяющая поддержку типом итераторов

```
template <typename C,
          typename = void_t<>>
struct is_iterable
    : std::false_type {};

template <typename C>
struct is_iterable<
    C, void_t<decltype(*begin(std::declval<C>())),
          decltype(end(std::declval<C>()))>>
    : std::true_type {};
```

Общий случай: предполагается, что тип не поддерживает итерации

Особый случай: вступает в силу, только если `C` поддерживает итерации и итератор `begin` можно разыменовать

Теперь можно определить окончательную версию функции `sum`, которая проверяет допустимость типа коллекции перед вызовом любой из функций с этой коллекцией:

```
template <typename C>
auto sum(const C& collection)
{
    if constexpr (has_value_type<C>()) {
        return sum_collection(collection);
    } else if constexpr (is_iterable<C>()) {
        return sum_iterable(collection);
    } else {
        // ничего не делать
    }
}
```

Функция защищает вызовы всех своих версий и предусматривает возможность обработки варианта, когда функции передан тип, не являющийся итерируемой коллекцией. В этом случае можно сообщить об ошибке компиляции (как показано в сопровождающем примере `11-contained-type`).

## 11.3 Каррирование функций

В главе 4 мы познакомились с каррингом и с использованием этого приема для совершенствования API проектов. Также там упоминалось, что в этой главе мы реализуем универсальную функцию, которая превращает любой вызываемый объект в его каррированную версию.

Как вы наверняка помните, каррирование позволяет рассматривать функции с несколькими аргументами как унарные. То есть вместо функции с  $n$  аргументами, возвращающей окончательный результат, полу-



чается унарная функция, возвращающая другую унарную функцию, которая возвращает еще одну унарную функцию, и т. д., пока все  $n$  аргументов не будут обработаны и последняя функция не вернет окончательный результат.

Давайте вспомним пример из главы 4. Функция `print_person` принимает три аргумента: объект, представляющий человека, ссылку на поток вывода и желаемый формат вывода:

```
void print_person(const person_t& person,
                 std::ostream& out,
                 person_t::output_format_t format);
```

Реализовав каррированную версию вручную, мы получили цепочку из вложенных лямбда-выражений, каждое из которых должно захватить все ранее определенные аргументы:

```
auto print_person_cd(const person_t& person)
{
    return [&](std::ostream& out) {
        return [&](person_t::output_format_t format) {
            print_person(person, out, format);
        };
    };
}
```



Каррированная версия требует передавать аргументы по одному, потому что, как я говорил, все каррированные функции являются унарными:

```
print_person_cd(martha)(std::cout)(person_t::full_name);
```

Писать все эти скобки – довольно утомительное занятие, поэтому избавимся от этого требования. Позволим пользователю нашей функции указывать сразу несколько аргументов. Имейте в виду, что это всего лишь синтаксический сахар; каррированная функция все еще является унарной, просто мы сделаем ее более удобной в использовании.

Каррированная функция должна быть объектом-функцией с состоянием, чтобы запомнить исходную функцию и все аргументы, переданные ранее. Он будет хранить копии всех полученных аргументов в `std::tuple`. Для этого используем `std::decay_t`, чтобы гарантировать, что параметры типов для кортежа являются фактическими типами значений, а не ссылками:

```
template <typename Function, typename... CapturedArgs>
class curried {
private:
    using CapturedArgsTuple = std::tuple<
        std::decay_t<CapturedArgs>...>;

    template <typename... Args>
    static auto capture_by_copy(Args&&... args)
```

```

    {
        return std::tuple<std::decay_t<Args>...>(
            std::forward<Args>(args)...);
    }

public:
    curried(Function, CapturedArgs... args)
        : m_function(function)
        , m_captured(capture_by_copy(std::move(args)...))
    {
    }

    curried(Function, std::tuple<CapturedArgs...> args)
        : m_function(function)
        , m_captured(std::move(args))
    {
    }

    ...

private:
    Function m_function;
    std::tuple<CapturedArgs...> m_captured;
};

```



Итак, мы получили класс, который может хранить вызываемый объект и произвольное количество аргументов. Осталось превратить этот класс в объект-функцию – добавить оператор вызова.

Оператор вызова должен обрабатывать два случая:

- пользователь передал все аргументы, необходимые для вызова оригинальной функции; в этом случае ее нужно вызвать и вернуть результат;
- пока получены не все аргументы, необходимые для вызова оригинальной функции, поэтому нужно вернуть новый объект каррированной функции.

Для проверки достаточности числа аргументов используем метафункцию `std::is_invocable_v`. Она принимает тип вызываемого объекта и список типов аргументов и возвращает признак возможности вызова этого объекта с аргументами указанных типов.

Вот как можно проверить возможность вызова функции с уже полученными аргументами:

```
std::is_invocable_v<Function, CapturedArgs...>
```

В операторе вызова необходимо проверить возможность вызова функции не только с уже имеющимися аргументами, но и со вновь полученными. Здесь нужно использовать `constexpr-if`, потому что оператор вызова может возвращать разные типы, в зависимости от того, что он возвращает – конечный результат или новый экземпляр объекта каррированной функции:

```

template <typename... NewArgs>
auto operator()(NewArgs&&... args) const
{
    auto new_args = capture_by_copy(std::forward<NewArgs>(args)...);

    if constexpr(std::is_invocable_v<
        Function, CapturedArgs..., NewArgs...>) {
        ...
    } else {
        ...
    }
}

```



В ветви `else` нужно вернуть новый экземпляр `curried`, содержащий ту же функцию, что и текущий экземпляр, но с новыми аргументами, добавленными в кортеж `m_captured`.

### 11.3.1 Вызов всех вызываемых объектов

Ветвь `then` должна вызвать функцию с заданными аргументами. Обычно функции вызываются с использованием привычного синтаксиса вызова, поэтому его тоже можно использовать.

Проблема в том, что некоторые объекты в C++ выглядят как функции, но их нельзя вызвать непосредственно, к ним относятся: указатели на функции-члены и переменные-члены. Если, к примеру, имеется такой тип, как `person_t` с функцией-членом `name`, вы сможете получить указатель на эту функцию-член как `&person_t::name`. Но не сможете вызвать ее по этому указателю, как это возможно с указателями на обычные функции, потому что компилятор сообщит об ошибке:

```
&person_t::name(martha);
```

Это досадное ограничение зашито в ядро языка C++. Каждая функция-член похожа на обычную функцию, первым аргументом которой неявно передается указатель `this`. Но, даже зная это, вы не сможете вызвать функцию. То же касается переменных-членов. Их можно рассматривать как функции, принимающие экземпляр класса и возвращающие значение, хранящееся в его переменной-члене. Это ограничение усложняет разработку обобщенного кода, способного работать с любыми вызываемыми объектами – объектами-функциями и с указателями на функции-члены и переменные-члены.

Для устранения данного ограничения в стандартную библиотеку была добавлена функция `std::invoke`. С помощью `std::invoke` можно вызвать любой вызываемый объект, независимо от допустимости обычного синтаксиса вызова. В отличие от предыдущего фрагмента, который вызывает ошибку во время компиляции, следующий код компилируется и делает именно то, что ожидалось:

```
std::invoke(&person_t::name, martha);
```

Функция `std::invoke` имеет простой синтаксис. Первый аргумент – это вызываемый объект, за ним следуют аргументы для передачи этому вызываемому объекту:

```
std::less<>(12, 14)      std::invoke(std::less<>, 12, 14)
fmin(42, 6)             std::invoke(fmin, 42, 6)
martha.name()          std::invoke(&person_t::name, martha)
pair.first              std::invoke(&pair<int,int>::first, pair)
```

Использовать `std::invoke` имеет смысл только в обобщенном коде, когда тип вызываемого объекта не известен. Каждый раз, реализуя функцию высшего порядка, которая принимает аргумент с другой функцией, или в классе, таком как `curried`, хранящем вызываемый объект произвольного типа, вместо обычного синтаксиса вызова функции используйте `std::invoke`.

В классе `curried` не получится использовать `std::invoke` непосредственно, потому что кроме вызываемого объекта имеется также `std::tuple`, содержащий аргументы для передачи ему. Здесь нет отдельных аргументов. Поэтому необходимо использовать вспомогательную функцию `std::apply`. Он действует подобно `std::invoke` (и обычно реализуется с использованием `std::invoke`), но принимает не отдельные аргументы, а кортеж с аргументами – как раз то, что нам нужно.

### Листинг 11.6 Законченная реализация `curried`

```
template <typename Function, typename... CapturedArgs>
class curried {
private:
    using CapturedArgsTuple =
        std::tuple<std::decay_t<CapturedArgs>...>;

    template <typename... Args>
    static auto capture_by_copy(Args&&... args)
    {
        return std::tuple<std::decay_t<Args>...>(
            std::forward<Args>(args)...);
    }

public:
    curried(Function function, CapturedArgs... args)
        : m_function(function)
        , m_captured(capture_by_copy(std::move(args)...))
    {
    }

    curried(Function function,
             std::tuple<CapturedArgs...> args)
        : m_function(function)
        , m_captured(std::move(args))
```

```

{
}

template <typename... NewArgs>
auto operator()(NewArgs&&... args) const
{
    auto new_args = capture_by_copy(
        std::forward<NewArgs>(args)...);
    auto all_args = std::tuple_cat(
        m_captured, std::move(new_args));
    if constexpr(std::is_invocable_v<Function,
        CapturedArgs..., NewArgs...>) {
        return std::apply(m_function, all_args);
    } else {
        return curried<Function,
            CapturedArgs...,
            NewArgs...>(
                m_function, all_args);
    }
}

private:
    Function m_function;
    std::tuple<CapturedArgs...> m_captured;
};

```



Создать кортеж из новых аргументов



Объединить прежде полученные аргументы с новыми



Если возможно, вызвать m\_function с полученными аргументами



Иначе вернуть новый экземпляр curried со всеми полученными к данному моменту аргументами

Важно отметить, что оператор вызова возвращает разные типы, в зависимости от выбранной ветви `constexpr-if`.

Теперь можно без труда создать каррированную версию `print_person`:

```
auto print_person_cd = curried{print_person};
```

Аргументы сохраняются в экземпляре `curried` по значению, поэтому если в вызов каррированной функции понадобится передать объект, не поддерживающий копирование (например, поток вывода), или желательно избежать копирования (например, экземпляра `person_t`) для повышения производительности, такой аргумент можно завернуть в тип-обертку для ссылок:

```
print_person_cd(std::cref(martha))(std::ref(std::cout))(person_t::name_only);
```

Этот код вызовет функцию `print_person` и передаст ей константную (`const`) ссылку на `martha` и изменяемую ссылку на `std::cout`. Свойство `person_t::name_only` будет передано по значению.

Эта реализация каррирования будет работать с обычными функциями, с указателями на функции-члены, с обычными и обобщенными лямбда-выражениями, с классами, определяющими оператор вызова, – как обобщенными, так и конкретными – и даже с классами, имеющими несколько перегруженных версий оператора вызова.

## 11.4 Строительные блоки предметно-ориентированного языка



До сих пор мы занимались в основном утилитами, помогающими писать более лаконичный и безопасный код. Но иногда эти утилиты оказываются слишком обобщенными, и было бы желательно иметь что-то более конкретное.

Нередко, работая над проектом, можно заметить, что приходится снова и снова писать почти одинаковые фрагменты кода, но не настолько универсальные, чтобы оправдать создание таких библиотек, как диапазоны, которые весь мир считал бы полезными. Решаемые задачи могут оказаться слишком специализированными для конкретной области. Тем не менее, следуя мантре «*не повторяйся*», нужно что-то предпринять.

Представьте себе следующий сценарий. Есть набор записей, при изменении которых необходимо, чтобы все поля в них изменялись в одной транзакции. Если изменение какого-то поля завершится неудачей, запись должна остаться в прежнем состоянии. Для этого можно реализовать систему транзакций и повсюду в коде использовать `start-transaction` и `end-transaction`. Однако такой подход чреват ошибками – можно забыть завершить транзакцию или случайно изменить запись без запуска транзакции. Было бы гораздо лучше определить более удобный синтаксис, менее подверженный ошибкам и избавляющий от размышлений о транзакциях.

Это идеальная ситуация для создания небольшого *предметно-ориентированного языка* (Domain-Specific Language, DSL). Такой язык должен позволять определять изменения записей только правильным способом, и ничего более. Он не обязательно должен быть обобщенным и предназначен для использования лишь в этой небольшой предметной области – области изменения записей внутри транзакций. Когда вы определяете, что нужно изменить, реализация DSL должна автоматически выполнить все необходимые операции с транзакциями.

Вот пример, как мог бы выглядеть код, изменяющий запись с использованием этого языка:

```
with(martha) (  
    name = "Martha",  
    surname = "Jones",  
    age = 42  
);
```

Очевидно, что это *не обычный* синтаксис C++ – здесь нет ни фигурных скобок, ни точек с запятой. Но он может стать действительным синтаксисом для C++, если вы готовы потратить время на реализацию DSL. Реализация не обязательно должна быть красивой, главное, чтобы она скрывала внутренние сложности от основного кода – максимально упрощала написание основной логики программы, пряча скрытые части, которые большинство никогда не увидит.

Давайте исследуем синтаксис этого примера:

- здесь мы имеем функцию (или тип) с именем `with`; в данном случае известно, что это функция, которая вызывается с аргументом `martha`;
- результатом данного вызова является другая функция, которая должна принимать произвольное количество аргументов, – заранее может быть неизвестно, сколько полей потребуется изменить одновременно.

Здесь также есть сущности `name` и `surname`, которые используются в операциях присваивания. Результаты этих присваиваний передаются в функцию, возвращаемую вызовом `with(martha)`.

При реализации DSL, подобного этому, лучше начать с самых внутренних элементов и определить все типы, необходимые для представления абстрактного синтаксического дерева (Abstract Syntax Tree, AST), которое требуется реализовать (см. рис. 11.3). В данном случае нужно начать с сущностей `name` и `surname`. Очевидно, они предназначены для представления членов записей `person`. Чтобы изменить член класса, нужно либо объявить этот член общедоступным (`public`), либо вызвать функцию-член записи.

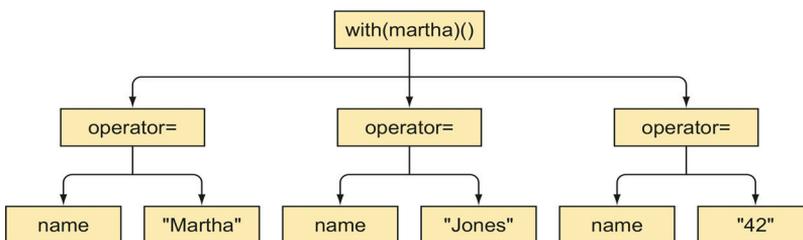


Рис. 11.3 Абстрактное синтаксическое дерево, которое мы должны построить, имеет три уровня: объект, подлежащий изменению, список изменений, которые необходимо выполнить (каждое изменение содержит два элемента: поле и новое значение для этого поля)

Определим простую структуру, способную хранить указатель на переменную-член или на функцию-член. Для этого создадим фиктивную структуру `field`, которая может хранить что угодно:

```

template <typename Member>
struct field {
    field(Member member)
        : member(member)
    {
    }
    Member member;
};
  
```

С ее помощью можно передавать поля для своих типов. Как определять поля для типов, можно увидеть в сопровождающем примере кода `11-dsl`.

Определив узел AST для хранения указателя на переменную-член или функцию-член для записи значения в поле, перейдем к реализации синтаксиса для его поддержки. Из примера выше следует, что структура `field` должна определять свой оператор присваивания. В отличие от обычного оператора присваивания, он не должен изменять данные, его задача – вернуть другой узел AST с именем `update`, определяющий одну операцию изменения. Этот узел должен хранить указатель на член и новое значение:

```
template <typename Member, typename Value>
struct update {
    update(Member member, Value value)
        : member(member)
        , value(value)
    {
    }

    Member member;

    Value value;
};

template <typename Member>
struct field {
    ...

    template <typename Value>
    update<Member, Value> operator=(const Value& value) const
    {
        return update{member, value};
    }
};
```



Теперь реализуем главный узел – функцию `with`. Она принимает экземпляр записи `person` и возвращает объект-функцию, представляющий транзакцию, которая принимает список изменений для выполнения. Поэтому дадим этому объекту-функции имя `transaction`. Он будет хранить ссылку на запись, чтобы иметь возможность изменить оригинал, и список экземпляров `update` для передачи оператору вызова типа `transaction`. Оператор вызова будет возвращать значение `bool` как признак успеха транзакции:

```
template <typename Record>
class transaction {
public:
    transaction(Record& record)
        : m_record(record)
    {
    }

    template <typename... Updates>
    bool operator()(Updates... updates)
```

```

    {
        ...
    }

private:
    Record& m_record;
};

template <typename Record>
auto with(Record& record)
{
    return transaction(record);
}

```



Мы определили все необходимые узлы AST, и теперь осталось лишь реализовать поведение DSL.

Давайте подумаем, что означает понятие «транзакция». При работе с базой данных нужно запустить транзакцию и подтвердить ее после внесения всех изменений. Если все измененные записи нужно отправить по сети, чтобы синхронизировать распределенные хранилища, можно подождать применения всех изменений, а затем отправить в сеть новую запись.

Для простоты будем считать, что внутри транзакции должны изменяться члены структуры C++. Если во время изменений возникнет исключение или функция записи вернет `false`, транзакцию следует отменить и оставить структуру в исходном состоянии. Самым простым способом реализации такого поведения является идиома *копирования и замены* (*copy-and-swap*), описанная в главе 9. Следуя ей, создадим копию текущей записи, внесем в нее изменения и заменим оригинальную запись, если все изменения выполнены успешно.



### Листинг 11.7 Реализация оператора вызова в классе `transaction`

```

template <typename Record>
class transaction {
public:
    transaction(Record& record)
        : m_record(record)
    {
    }

    template <typename... Updates>
    bool operator()(Updates... updates)
    {
        auto temp = m_record; ← Создать временную копию для выполнения изменений

        if (all(updates(temp)...)) {
            std::swap(m_record, temp);
            return true;
        }

        return false;
    }
}

```

Применить все изменения.  
Если все изменения выполнены успешно,  
заменить оригинал копией и вернуть true

```
private:
    template <typename... Updates>
    bool all(Updates... results) const
    {
        return (... && results);
    }

    Record &m_record;
};
```

Собрать все результаты применения отдельных изменений и вернуть true, если все они выполнены успешно

Все изменения выполняются с временной копией. Если какое-либо из изменений вернет false или возбудит исключение, оригинальная запись останется неизменной.

Единственное, что осталось реализовать, – оператор вызова для узла update. Он должен обрабатывать три случая:

- имеется указатель на переменную-член, которую можно изменить непосредственно;
- имеется обычная функция записи в свойство;
- имеется функция записи, которая возвращает значение bool как признак успешного изменения.

Выше было показано, как использовать `std::is_invocable` для проверки возможности вызова функции с определенным набором аргументов. В данном случае ее можно использовать, чтобы отличить указатель на функцию записи от указателя на переменную-член. Дополнительно мы должны различать функции записи, возвращающие `void` и `bool` (или другой тип, преобразуемый в тип `bool`). Сделать это можно с помощью `std::is_invocable_r`, которая проверяет возможность вызова функции возвращаемого ею типа.



### Листинг 11.8 Окончательная реализация структуры update

```
template <typename Member, typename Value>
struct update {
    update(Member member, Value value)
        : member(member)
        , value(value)
    {
    }
};

template <typename Record>
bool operator()(Record& record)
{
    if constexpr (std::is_invocable_r<
        bool, Member, Record, Value>()) {
        return std::invoke(member, record, value);
    } else if constexpr (std::is_invocable<
        Member, Record, Value>()) {
        std::invoke(member, record, value);
        return true;
    }
};
```

Если Member – вызываемый объект, принимающий запись и новое значение и возвращающий тип bool, значит, это функция записи, которая может потерпеть неудачу

Если тип результата отличается от bool и не может быть преобразован в bool, вызвать функцию записи и вернуть true

```

    } else {
        std::invoke(member, record) = value;
        return true;
    }
}

Member member;

Value value;
};

```

Если это указатель на переменную-член, изменить ее и вернуть true

Язык C++ обладает богатым арсеналом возможностей, которые могут пригодиться при реализации DSL, и основными из них являются перегрузка операторов и шаблоны с переменным количеством параметров (variadic templates). С их помощью можно определять весьма сложные DSL.

Основная проблема заключается в том, что реализация всех необходимых структур для представления узлов AST может оказаться утомительной задачей и потребовать большого количества типового кода. Этот недостаток снижает привлекательность идеи реализации DSL в C++, однако DSL предлагают два огромных преимущества: возможность лаконично описывать логику основной программы и переключаться между разными реализациями транзакций, не изменяя логики основной программы. Например, если вы решите сохранять все записи в базе данных, вам понадобится лишь переопределить оператор вызова в классе `transaction`, после чего остальная часть программы автоматически начнет сохранять данные в базе данных и вам не придется ничего менять в основной логике.

**СОВЕТ** За дополнительной информацией по теме, рассматривавшейся в этой главе, обращайтесь по адресу: <https://forums.manning.com/posts/list/43780.page>.

## Итоги

- Шаблоны предлагают полный по Тьюрингу язык программирования, код на котором выполняется во время компиляции программы. Он был случайно обнаружен Эрвином Унру (Erwin Unruh), написавшим программу на C++, которая выводит первые 10 простых чисел во время компиляции в виде сообщений об ошибках компиляции.
- Метаязык шаблонов – это не только полный по Тьюрингу, но также чисто функциональный язык. Все переменные являются неизменяемыми, и отсутствует изменяемое состояние в любой форме.
- Заголовок `type_traits` содержит много полезных метафункций для манипулирования типами.
- Иногда, из-за ограничений или отсутствия некоторых возможностей в основном языке программирования, в стандартную библиотеку добавляются обходные пути. Например, `std::invoke` позволяет вызывать

любые объекты-функции, даже те, которые не поддерживают обычный синтаксис вызова функций.

- Предметно-ориентированные языки (DSL) трудоемки в реализации, но позволяют значительно упростить основную логику программы. В некотором смысле диапазоны тоже являются предметно-ориентированным языком; они определяют AST для определения преобразований диапазонов с использованием синтаксиса конвейеров.



# 12

## Функциональный дизайн параллельных систем

---

### **О чем говорится в этой главе:**

- разделение программ на изолированные компоненты;
- обработка сообщений как потоков данных;
- преобразование реактивных потоков;
- использование программных компонентов с состоянием;
- преимущества реактивных потоков в параллельных и распределенных системах.



Самая большая проблема в разработке программного обеспечения – сложность обработки. Программные системы имеют свойство разрастаться с течением времени и быстро выходят за рамки первоначального замысла. Когда наши потребности вступают в противоречие с проектными решениями, приходится либо переписывать значительные фрагменты системы, либо внедрять не самые лучшие обходные решения.

Проблема со сложностью становится более очевидной в программном обеспечении с компонентами, выполняющимися параллельно – от простейших интерактивных пользовательских приложений до сетевых служб и распределенных программных систем.

*Многие недостатки в разработке программного обеспечения обусловлены недостаточно полным пониманием программистами всех возможных состояний их кода. В многопоточном окружении отсутствие понимания многократно усиливает*

*проблемы, которые могут вызвать панику, если обратить на них внимание.*

– Джон Кармак (John Carmack)<sup>1</sup>

Большинство проблем проистекают из-за образования тесных связей между различными компонентами системы. Наличие отдельных компонентов, использующих и изменяющих одни и те же данные, требует применения механизмов синхронизации. В роли таких механизмов традиционно используются мьютексы и другие похожие примитивы синхронизации. Этот подход работает, но создает проблемы с масштабируемостью и убивает параллелизм.

Одним из решений проблемы общих изменяемых данных является отказ от изменяемых данных вообще. Но есть и другой вариант: иметь изменяемые данные, но никогда не использовать их совместно с другими компонентами. Первое решение мы обсудили в главе 5, а теперь поговорим о втором.

## 12.1 Модель акторов: мышление в терминах компонентов

В этой главе вы познакомитесь с подходом к проектированию программного обеспечения в виде набора изолированных компонентов. Но сначала мы обсудим объектно-ориентированную реализацию, чтобы потом было проще понять, как то же самое реализовать в функциональном стиле.

При разработке классов обычно рекомендуется писать функции чтения/записи: методы чтения – для получения информации об объекте, а методы записи – для изменения атрибутов объекта допустимым образом, не нарушающим инварианты класса. Многие сторонники объектно-ориентированного проектирования считают, что этот подход противоречит философии ОО. Они склонны называть это *процедурным* программированием, потому что программист продолжает мыслить алгоритмическими шагами, а объекты воспринимать как контейнеры и валидаторы данных.

*Первый шаг на пути превращения успешного процедурного разработчика в успешного объектно-ориентированного разработчика – это лоботомия.*

– Дэвид Уэст (David West)<sup>2</sup>

Мы должны перестать думать о том, какие данные содержит объект, и начать думать о том, что он может сделать. Для примера рассмотрим класс, представляющий человека. Обычно в таких классах мы определя-

<sup>1</sup> John Carmack. In-Depth: Functional Programming in C++ // #altdevblogaday. April 30, 2012. Gamasutra, <http://mng.bz/OAzP>.

<sup>2</sup> David West. Object Thinking. Microsoft Press, 2004.

ем методы чтения и записи имени, фамилии, возраста и других атрибутов, которые используются как-то так:

```
douglas.set_age(42);
```

Этот код наглядно иллюстрирует проблему. Мы написали класс, который будет служить контейнером для данных, а не представлением человека. Можно ли в реальной жизни присвоить человеку возраст в 42 года по своему желанию? Нет, это невозможно, поэтому мы не должны проектировать классы так, чтобы они позволяли подобное.

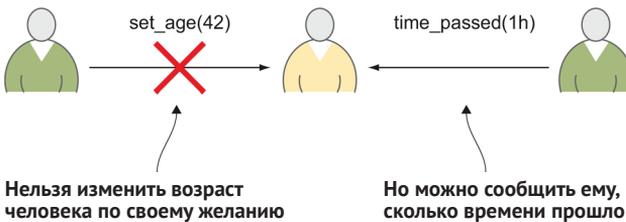


Рис. 12.1 Мы не можем изменять атрибуты реальных объектов, но можем посылать им сообщения и позволять реагировать на них

Мы должны проектировать классы как наборы действий, которые они могут выполнять, а затем добавлять в них данные, необходимые для выполнения этих действий. В случае с классом, моделирующим человека, вместо метода записи в атрибут возраста можно определить действие, сообщающее человеку, что прошло некоторое время, и объект должен среагировать соответственно (см. рис. 12.1). Вместо `set_age` объект мог бы иметь функцию-член `time_passed`:

```
void time_passed(const std::chrono::duration& time);
```

Получив уведомление, что прошло указанное время, объект, представляющий человека, сможет увеличить свой возраст и внести другие изменения, связанные с изменением возраста. Например, в результате старения может измениться рост или цвет волос человека. Поэтому вместо методов чтения и записи атрибутов класс должен определять только набор задач, известных объекту.

*Не запрашивайте информацию, необходимую для работы; попросите объект, у которого есть информация, сделать эту работу за вас.*

– Ален Голуб (Allen Holub)<sup>1</sup>

Продолжая моделировать объект, представляющий человека, следуя за аналогией реального мира, быстро становится понятно, что отдельные объекты персон не могут иметь общих данных. Реальные люди об-

<sup>1</sup> *Allen Holub. Holub on Patterns: Learning Design Patterns by Looking at Code. Apress, 2004.*

мениваются данными, общаясь друг с другом, но у них нет общих «переменных», к которым каждый может обратиться и изменить.

Эта идея была положена в основу *акторов*. В модели акторов каждый компонент – актер – полностью изолирован от других компонентов. Они не имеют общих данных, но могут посылать сообщения друг другу. То есть класс актора как минимум должен иметь способ получать и отправлять сообщения (см. рис. 12.2).

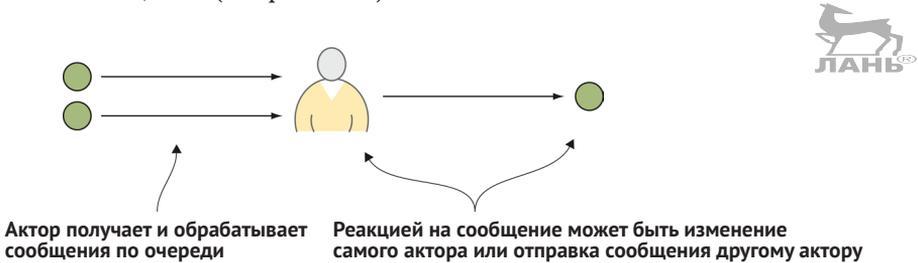


Рис. 12.2 Актер – это изолированный компонент, способный получать и отправлять сообщения. Он обрабатывает сообщения последовательно; в ответ на каждое сообщение он может изменить свое состояние или поведение либо отправить новое сообщение другому актору в системе

Традиционно акторы могут отправлять и получать разные типы сообщений, и каждый актер может выбирать, кому отправить сообщение. Кроме того, обмен сообщениями должен происходить асинхронно.

### Фреймворки акторов для C++

Законченную реализацию традиционной модели акторов для C++ можно найти на сайте <http://actor-framework.org> и использовать ее в своих проектах. Фреймворк C++ Actor Framework обладает впечатляющим набором возможностей. Акторы – это легковесные параллельные процессы (намного легче потоков), прозрачные для сети, что означает, что акторы можно распределить по нескольким компьютерам в сети и программа будет продолжать работать без изменения кода. Композиция традиционных акторов представляет определенные сложности, но их легко приспособить к дизайну, описанному в этой главе.

Альтернативой фреймворку C++ Actor Framework может служить библиотека SObjectizer (<https://sourceforge.net/projects/sobjectizer>). Она часто предлагает лучшую производительность, но не имеет встроенной поддержки распределения акторов по нескольким процессам.

В этой главе мы определим более простую реализацию акторов, по сравнению с реализациями в традиционной модели и во фреймворке C++ Actor Framework, чтобы в большей мере сосредоточиться на разработке программного обеспечения, а не на реализации настоящей модели акторов (см. рис. 12.3). Несмотря на то что дизайн акторов, представленный в этой главе, отличается от дизайна акторов в традиционной мо-

дели, описанные идеи в равной степени применимы к традиционным актерам.

Наши акторы будут обладать следующими возможностями:

- смогут получать и отправлять сообщения единственного типа (не обязательно одного и того же). Для поддержки нескольких типов входных или выходных сообщений можно использовать `std::variant` или `std::any`, как было показано в главе 9;
- выбор получателя каждого сообщения будут определять не сами авторы, а внешний контроллер, это даст нам возможность комбинировать акторы в функциональном стиле. Внешний контроллер также будет определять источники сообщений для каждого конкретного актора;
- выбор синхронного или асинхронного способа обработки того или иного сообщения мы оставим на усмотрение внешнего контроллера.

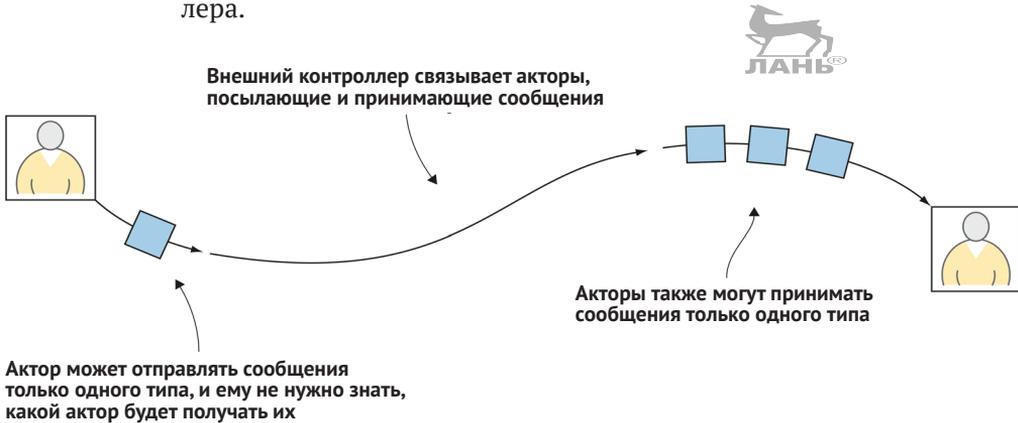


Рис. 12.3 Мы используем упрощенные, типизированные версии акторов, которым не нужно заботиться о том, кто кому посылает сообщения: эту задачу будет решать внешний контроллер

**ПРИМЕЧАНИЕ** Большинство современных программных систем используют или реализуют тот либо иной вид цикла обработки событий, который можно использовать для асинхронной доставки сообщений, поэтому мы не будем заботиться о реализации такой системы и сосредоточимся на дизайне, который легко адаптировать для работы в любой системе, основанной на событиях.

В листинге 12.1 показан интерфейс акторов.

**Листинг 12.1 Интерфейс минимального актора**

```
template <typename SourceMessageType,
         typename MessageType>
class actor {
public:
    using value_type = MessageType;
```

Актер может принимать сообщения одного типа и посылать сообщения другого типа

Определяет тип сообщений, посылаемых актором, что дает возможность проверить его позже, при связывании акторов

```

void process_message(SourceMessageType&& message);
Обрабатывает   template <typename EmitFunction>
поступившее   void set_message_handler(EmitFunction emit); ←
сообщение     private:
              std::function<void(MessageType&&)> m_emit; ←
              };

```

Определяет в m\_emit, какой обработчик должен вызвать актор для отправки сообщения



Этот интерфейс наглядно показывает, что актор знает только, как получить сообщение и как отправить сообщение. Он может иметь столько личных данных, сколько необходимо для работы, но никакие его данные не должны быть доступны внешнему миру. Поскольку общий доступ к данным отсутствует, отпадает необходимость в механизмах синхронизации.

Важно отметить, что акторы могут только получать сообщения (часто их так и называют – *приемниками*), только отправлять сообщения (их называют *источниками*) и получать и отправлять сообщения.



## 12.2 Простой источник сообщений

В этой главе мы реализуем простую веб-службу, которая получает и обрабатывает закладки (см. пример: `bookmarks-service`). Клиенты будут подключаться к ней и передавать информацию о закладках в формате JSON, например:

```
{ "FirstURL": "https://isocpp.org/", "Text": "Standard C++" }
```

Для реализации нам понадобится несколько сторонних библиотек. Для сетевых взаимодействий используем библиотеку Boost.Asio (<http://mng.bz/d62x>); а для работы с форматом JSON – библиотеку Нильса Ломанна (Niels Lohmann) JSON for Modern C++ (<https://github.com/nlohmann/json>).

Для начала определим актор, который будет принимать входящие сетевые подключения и собирать сообщения, отправляемые клиентами. Чтобы максимально упростить протокол, используем обычные текстовые сообщения; в каждом сообщении будет только один символ перевода строки – в конце сообщения.

Этот актор будет играть роль актора-источника. Его задача: получить сообщение извне (от сущностей, не являющихся частью веб-службы) и послать это сообщение всем, кто заинтересован в его получении. Остальным компонентам веб-службы не нужно беспокоиться о том, откуда поступают сообщения, поэтому клиентские подключения можно рассматривать как неотъемлемую часть этого актора-источника.

Актор-источник (рис. 12.4) будет иметь интерфейс, похожий на интерфейс обычного актора, кроме функции `process_message`, которая ему не нужна, потому что он играет роль лишь передаточного звена. Он не будет получать сообщений от других акторов (как я уже сказал, он получает

сообщения от внешних сущностей – клиентов, – которые не являются акторами этой веб-службы); актор службы просто отправляет сообщения.

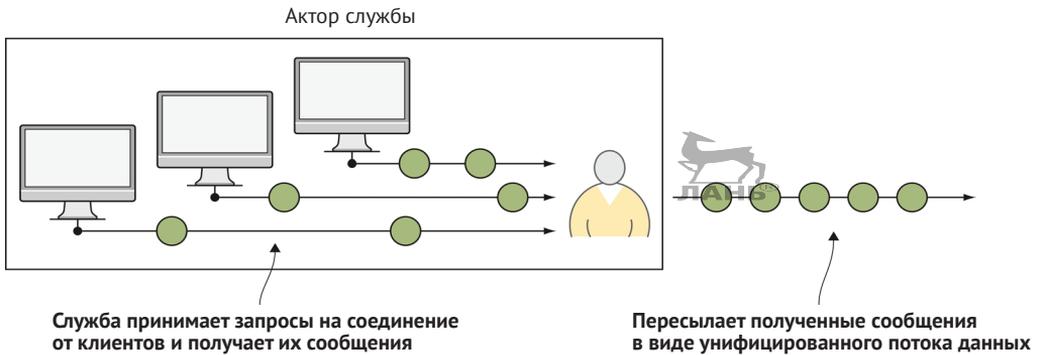


Рис. 12.4 Актор службы (актор-источник) принимает запросы на соединение от клиентов и их сообщения. Эта часть скрыта от остальной части программы, которая знает лишь, что откуда-то поступает поток строк

**Листинг 12.2 Прием запросов на соединение от клиентов**

```

class service {
public:
    using value_type = std::string;

    explicit service(
        boost::asio::io_service& service,
        unsigned short port = 42042)
        : m_acceptor(service,
            tcp::endpoint(tcp::v4(), port))
        , m_socket(service)
    {
    }

    service(const service& other) = delete;
    service(service&& other) = default;

    template <typename EmitFunction>
    void set_message_handler(EmitFunction emit)
    {
        m_emit = emit;
        do_accept();
    }

private:
    void do_accept()
    {
        m_acceptor.async_accept(
            m_socket,
            [this](const error_code& error) {
                if (!error) {

```

Читает сообщение клиента как строку, поэтому посылаемые сообщения должны быть строками

Создать экземпляр службы, которая принимает запросы на соединение на определенном порту (по умолчанию 42042)

Запретить копирование, но разрешить перемещение

Нет смысла принимать сообщения от клиентов, пока не зарегистрирован ни один актор-обработчик



```

        make_shared_session(
            std::move(m_socket),
            m_emit
        )->start();
    } else {
        std::cerr << error.message() << std::endl;
    }

    // Ждать запроса на соединение от другого клиента
    do_accept();
});

tcp::acceptor m_acceptor;
tcp::socket m_socket;
std::function<void(std::string&&)> m_emit;
};

```

Создать и запустить сеанс для нового клиента. Когда объект сеанса прочитает сообщение от клиента, он передаст его в m\_emit. Функция make\_shared\_session возвращает разделяемый указатель на экземпляр объекта сеанса

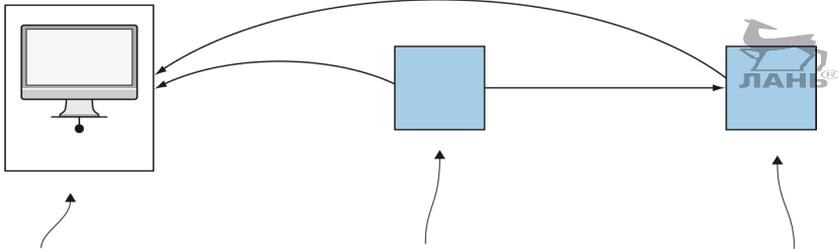
Служба имеет простую и понятную организацию. Единственная сложная часть – это функция do\_accept, но эта сложность обусловлена интерфейсом обратных вызовов библиотеки Boost.Asio. Если описывать кратко, она делает следующее:

- m\_acceptor.async\_accept планирует на выполнение переданное ей лямбда-выражение, когда появится запрос на соединение;
- лямбда-выражение проверит успех попытки соединиться с клиентом и создаст объект сеанса для клиента;
- так как веб-служба должна обслуживать множество клиентов, вновь вызывается do\_accept.

Большую часть работы выполняет объект сеанса. Он должен прочитать сообщения от клиента одно за другим и уведомить службу о них, чтобы та могла выступить в роли источника сообщений для остальной части программы.

Объект сеанса должен продолжать существовать, пока не возникнет ошибка или клиент не отсоединится, после чего он должен сам себя уничтожить. Для этого используется простой трюк; объект сеанса наследует std::enable\_shared\_from\_this. Это позволит экземпляру сеанса, которым управляет std::shared\_ptr, безопасно создавать дополнительные разделяемые указатели на себя. Наличие разделяемого указателя на сеанс позволяет обеспечить существование объектов сеансов, пока в системе существуют компоненты, использующие сеанс (рис. 12.5).

Мы захватываем разделяемый указатель на сеанс в лямбда-выражении, которое обрабатывает события, возникающие в соединении. Пока есть события, ожидаемые сеансом, объект сеанса будет существовать, потому что лямбда-выражение, которое обрабатывает события, будет хранить экземпляр разделяемого указателя. Когда события для обработки закончатся, объект будет удален автоматически.



Когда создается лямбда-выражение для обработки сообщения от клиента, оно захватывает разделяемый указатель на объект сеанса и тем самым обеспечивает его сохранность, пока существует само лямбда-выражение

После получения первого сообщения создается лямбда-выражение для обработки второго. Теперь уже это лямбда-выражение будет хранить разделяемый указатель и обеспечивать сохранность объекта сеанса

Если соединение будет разорвано, создание лямбда-выражений для обработки новых сообщений прекратится, и объект сеанса уничтожится

Рис. 12.5 Лямбда-выражение, предназначенное для обработки нового сообщения, когда оно поступит от клиента, захватывает разделяемый указатель на объект сеанса, что обеспечивает его существование до тех пор, пока клиент поддерживает соединение

### Листинг 12.3 Чтение и пересылка сообщений

```

template <typename EmitFunction>
class session:
public:
    session(tcp::socket&& socket, EmitFunction emit)
        : m_socket(std::move(socket))
        , m_emit(emit)
    {
    }

    void start()
    {
        do_read();
    }

private:
    using shared_session =
        std::enable_shared_from_this<session<EmitFunction>>;

    void do_read()
    {
        auto self = shared_session::shared_from_this();

        boost::asio::async_read_until(
            m_socket, m_data, '\n',
            [this, self](const error_code& error,
                std::size_t size) {
                Запланировать лямбда-выражение
                для выполнения после достижения
                символа перевода строки

                if (!error) {
                    std::istream is(&m_data);
                    std::string line;
                    std::getline(is, line);
                    m_emit(std::move(line));
                    Если никаких ошибок не возникло,
                    прочитать строку и послать ее любому
                    актору, зарегистрировавшемуся
                    для получения сообщений
                }
            });
    }
};
    
```



```

        do_read(); ←
    }
    });
}

tcp::socket m_socket;
boost::asio::streambuf m_data;
EmitFunction m_emit;
};

```

Если сообщение благополучно прочитано и отправлено, запланировать чтение следующего сообщения



Пользователи класса службы `service` не будут знать о существовании объекта сеанса `session`, тем не менее сообщения им будет посылать именно этот объект.

## 12.3 Моделирование реактивных потоков данных в виде монад



Мы создали службу, которая отправляет сообщения типа `std::string`. Она может отправить любое количество сообщений – ноль или больше. Поток сообщений выглядит как односвязный список – коллекция значений некоторого типа, по элементам которой можно перемещаться до самого конца. Единственное отличие от односвязных списков состоит в том, что в списке уже есть все значения для обхода, а в данном случае значения пока не известны – они поступают с течением времени.

Нечто подобное мы видели в главе 10. Там у нас имелись экземпляры `future` и монада продолжения. `future` – это структура-контейнер, которая будет содержать значение заданного типа в некоторый момент времени в будущем. Наша служба подобна этой структуре, с той лишь разницей, что не ограничивается одним значением и время от времени посылает новые значения (рис. 12.6). Мы будем называть такие структуры *асинхронными*, или *реактивными*, потоками. Важно отметить, что эта структура отличается от типа `future<list<T>>`, который означает, что все значения будут получены одновременно.

Реактивные потоки данных похожи на коллекции. Они содержат элементы одного типа; просто не все они доступны сразу. Подобный тип – поток ввода – мы видели в главе 7. Мы использовали поток ввода с библиотекой диапазонов и выполняли преобразования:

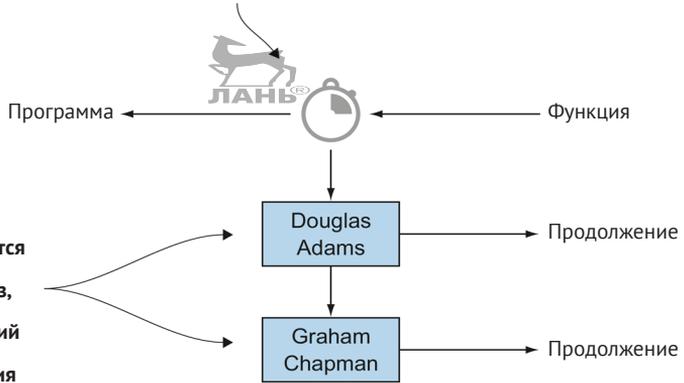
```

auto words = istream_range<std::string>(std::cin)
    | view::transform(string_to_lower);

```

Такие же преобразования мы реализовали для типа `future` и необязательных значений. Надеюсь, вы понимаете, куда я клоню; мы сумели создать похожие преобразования для всех монад, которые рассматривали. Остается ответить на вопрос: являются ли реактивные потоки данных монадами?

Напоминает монаду продолжения, когда вызывается функция, которая возвращает только дескриптор, который можно передать в продолжение



В отличие от ситуации, когда имеется только одно значение и функция продолжения вызывается один раз, реактивные потоки данных могут иметь произвольное число значений и для каждого вызывать функцию продолжения по мере их появления

Рис. 12.6 В отличие от монады продолжения, которая вызывает функцию продолжения только один раз, реактивные потоки данных могут иметь произвольное количество значений. Функция продолжения вызывается для каждого вновь поступившего значения

Концептуально они похожи. Давайте вспомним, какими чертами должен обладать тип, чтобы называться монадой:

- быть обобщенным;
- иметь конструктор – функцию, создающую экземпляр реактивного потока данных с заданным значением;
- иметь функцию `transform` – функцию, возвращающую реактивный поток с преобразованными значениями из исходного потока;
- иметь функцию `join`, которая принимает все сообщения из всех заданных потоков данных и возвращает поток с этими сообщениями;
- соответствовать правилам, установленным для монад (мы не будем обсуждать этот вопрос в книге).

Первый пункт выполняется: реактивные потоки данных являются обобщенными типами, параметризованными типом сообщений, которые несет поток (`value_type`). В следующих разделах мы превратим реактивные потоки данных в монады, создав:

- актора преобразования потока данных;
- актора, который будет конструировать поток из заданных значений;
- актора, который будет принимать сразу несколько потоков и посылать сообщения, содержащиеся в них.

### 12.3.1 Создание приемника для сообщений

Прежде чем реализовать все функции, чтобы показать, что реактивные потоки данных являются монадами, для начала определим простой объект-приемник, который можно использовать для тестирования веб-службы. Приемник – это актор, который только получает сообщения, но не отправляет их (рис. 12.7). Поэтому ему не нужна функция `set_message_handler`. Мы определим в нем только функцию `process_message`. Для прос-

тоты создадим обобщенный приемник, который будет вызывать любую заданную функцию для каждого нового сообщения.

#### Листинг 12.4 Реализация объекта-приемника

```
namespace detail {
    template <typename Sender,
              typename Function,
              typename MessageType = typename Sender::value_type>
    class sink_impl {
    public:
        sink_impl(Sender&& sender, Function function)
            : m_sender(std::move(sender))
            , m_function(function)
        {
            m_sender.set_message_handler(
                [this](MessageType&& message)
                {
                    process_message(
                        std::move(message));
                }
            );
        }

        void process_message(MessageType&& message) const
        {
            std::invoke(m_function,
                       std::move(message));
        }

    private:
        Sender m_sender;
        Function m_function;
    };
}
```

После создания приемник автоматически подключается к указанному отправителю

Полученное сообщение передается в функцию, заданную пользователем

#### Архитектура с единственным владельцем

В этой главе часто будут использоваться функция `std::move` и правосторонние (rvalue) ссылки. Например, `sink_impl` принимает объект отправителя `sender` как правостороннюю (rvalue) ссылку. Объект-получатель `sink` в этом случае становится единственным владельцем указанного вами отправителя. Аналогично другие акторы будут становиться владельцами своих отправителей.

Это решение подразумевает, что конвейер обработки данных будет владеть всеми акторами, участвующими в его работе, – когда конвейер уничтожится, автоматически уничтожатся все его акторы. Кроме того, сообщения будут передаваться акторам с использованием правосторонних (rvalue) ссылок, чтобы показать, что только один актор имеет доступ к сообщению в каждый конкретный момент времени. Это очень простая архитектура, и я считаю ее лучшей для демонстрации идеи акторов и потоков данных.

Недостаток данного подхода состоит в том, что в такой системе не может быть нескольких компонентов, принимающих сообщения от одного актора, и невозможно использовать одного и того же актора в разных потоках данных. Это легко исправить, разрешив общее владение акторами (`std::shared_ptr` был бы идеальным выбором) и позволив каждому отправителю иметь несколько получателей, храня коллекцию функций-обработчиков сообщений вместо одной.

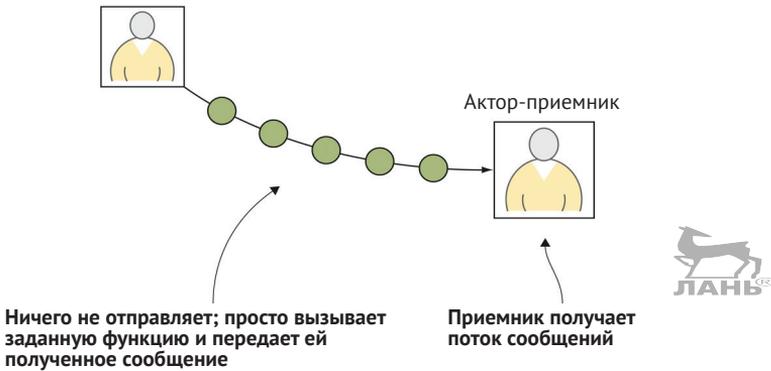


Рис. 12.7 Актор-приемник вызывает функцию для каждого полученного сообщения. Он ничего не отправляет. Такой актор можно использовать для вывода всех полученных сообщений в `std::cerr` или выполнения более сложных действий, например записи сообщения в файл или в базу данных

Теперь реализуем функцию, получающую отправителя, и функцию, создающую экземпляр `sink_impl`:

```
template <typename Sender, typename Function>
auto sink(Sender&& sender, Function&& function)
{
    return detail::sink_impl<Sender, Function>(
        std::forward<Sender>(sender),
        std::forward<Function>(function));
}
```

Работу объекта веб-службы легко проверить, связав его с приемником, который просто выводит все сообщения в `cerr`.

**Листинг 12.5 Запуск службы**

```
int main(int argc, char* argv[])
{
    boost::asio::io_service event_loop;
    auto pipeline =
        sink(service(event_loop),
            [](const auto& message) {
                std::cerr << message << std::endl;
            });
}
```

*io\_service* – это класс из библиотеки Boost.Asio, реализующий цикл обработки событий. Он принимает события и вызывает соответствующие лямбда-выражения

Создать службу и связать ее с приемником

```

    });
    event_loop.run(); ← Запустить обработку событий
}

```

### Автоматическое определение аргументов типов шаблонных классов в C++17



Версия C++17, необходимая для компиляции примеров в этой главе, поддерживает автоматическое определение аргументов типов шаблонных классов (class template argument deduction). Строго говоря, поддержка этой возможности компилятором не требуется для создания функции `sink`; мы могли бы определить класс с именем `sink`, и код в листинге 12.5 сохранил бы работоспособность.

Я разделил `sink` и `sink_impl`, чтобы получить возможность использовать синтаксис диапазонов с реактивными потоками данных. У нас будет две функции `sink`, возвращающие разные типы, в зависимости от числа переданных аргументов. Добиться этого было бы сложнее, если бы `sink` был классом, а не функцией.

Этот код напоминает вызов алгоритма `for_each` для коллекции: мы передаем коллекцию и функцию для обработки каждого элемента коллекции. Это не самый наглядный синтаксис, поэтому заменим его синтаксисом каналов, как это делали в библиотеке поддержки диапазонов.

Для этого нам понадобится функция `sink`, которая принимает только функцию для применения к каждому сообщению, без объекта отправителя. Она должна возвращать временный вспомогательный объект, хранящий эту функцию. Экземпляр класса `sink_impl` будет создан оператором конвейера, которому передается отправитель. Это решение можно рассматривать как частично примененную функцию – она связывает второй аргумент и оставляет первый, который будет определен позже. Единственное отличие состоит в том, что первый аргумент задается с использованием синтаксиса канала вместо обычного синтаксиса вызова функции, который мы использовали с частично примененными функциями в главе 4:

```

namespace detail {
    template <typename Function>
    struct sink_helper {
        Function function;
    };
}

template <typename Sender, typename Function>
auto operator|(Sender&& sender,
               detail::sink_helper<Function> sink)
{
    return detail::sink_impl<Sender, Function>(
        std::forward<Sender>(sender), sink.function);
}

```

Подобную функцию-член `operator|` нужно определить для каждого преобразования. Эта функция принимает любого отправителя в первом аргументе и экземпляр класса `_helper`, определяющего преобразование. Такой подход поможет сделать основную программу более читабельной:

```
auto sink_to_cerr =
    sink([](const auto& message) {
        std::cerr << message << std::endl;
    });

auto pipeline = service(event_loop) | sink_to_cerr;
```

Теперь у нас есть отличная возможность проверить правильность работы службы. Мы можем организовать любой поток сообщений и вывести его в `cerr`. Скомпилируйте программу, запустите ее и используйте `telnet` или похожее приложение для проверки простых текстовых сообщений, подключившись к программе через порт 42042. Все сообщения, отправленные любым клиентом, должны автоматически появиться в окне терминала, где запущена служба.

### 12.3.2 Преобразование реактивных потоков данных

Вернемся к нашей задаче превращения реактивных потоков данных в монады. Наиболее важной частью этой задачи является создание модификатора `transform` для преобразования потока. Он должен принимать реактивный поток и произвольную функцию и возвращать новый поток, содержащий сообщения из оригинального потока, но преобразованные с использованием заданной функции.

Роль модификатора `transform` в нашем примере будет играть актер, способный принимать и отправлять сообщения. Для каждого полученного сообщения он будет вызывать заданную функцию преобразования и посылать результат в выходной поток (см. рис. 12.8).

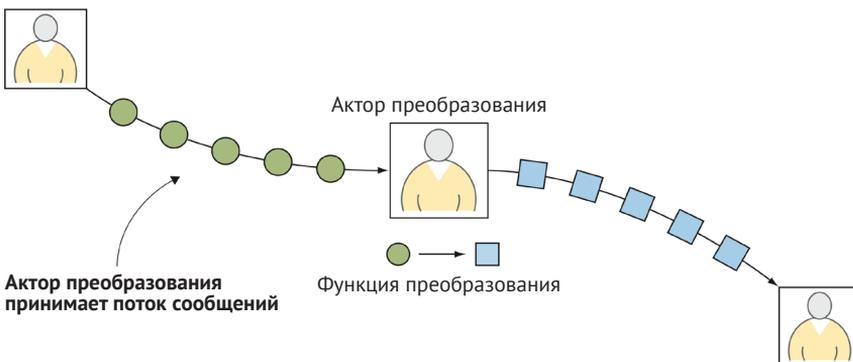


Рис. 12.8 Так же как в диапазонах, актер преобразования в реактивных потоках применяет заданную функцию к каждому полученному сообщению и отправляет результат следующему актору

## Листинг 12.6 Реализация модификатора transform для потока данных

```

namespace detail {
    template <
        typename Sender,
        typename Transformation,
        typename SourceMessageType =
            typename Sender::value_type,
        typename MessageType =
            decltype(std::declval<Transformation>()(
                std::declval<SourceMessageType>()))>
    class transform_impl {
    public:
        using value_type = MessageType;

        transform_impl(Sender&& sender, Transformation transformation)
            : m_sender(std::move(sender))
            , m_transformation(transformation)
        {
        }

        template <typename EmitFunction>
        void set_message_handler(EmitFunction emit)
        {
            m_emit = emit;
            m_sender.set_message_handler(
                [this](SourceMessageType&& message) {
                    process_message(
                        std::move(message));
                });
        }

        void process_message(SourceMessageType&& message) const
        {
            m_emit(std::invoke(m_transformation,
                std::move(message)));
        }

    private:
        Sender m_sender;
        Transformation m_transformation;
        std::function<void(MessageType&&)> m_emit;
    };
}

```

Для правильного определения функций приема сообщений и функций отправки нужно определить типы принимаемых и отправляемых сообщений

Подключить актор, заинтересованный в получении сообщений, к актору-отправителю

Преобразовать полученное сообщение вызовом заданной функции и послать результат следующему актору-приемнику



Обратите внимание на одно важное обстоятельство: в отличие от актора-приемника sink, актор преобразования transform не сразу подключается к отправителю. Если нет никого, кому можно посылать сообщения, было бы глупо тратить время на их обработку. Прием сообщений от отправителя начинается, только когда вызывается функция set\_message\_handler – когда появляется приемник для преобразованных сообщений.

После создания всех вспомогательных классов и оператора конвейера можно задействовать модификатор `transform`, как мы делали это с диапазонами. Например, вот как можно обрезать сообщения перед выводом на экран:

```
auto pipeline =
    service(event_loop)
    | transform(trim)
    | sink_to_cerr;
```



Потоки данных начинают походить на диапазоны. И это важно: реактивные потоки данных позволяют рассуждать о программном обеспечении как о совокупности входных потоков, преобразований, которые нужно выполнить, и выходных потоков, куда помещаются результаты, – все точно так же, как при использовании диапазонов.

### 12.3.3 Создание потока заданных значений



Функция `transform` превратила реактивный поток в функтор. Чтобы теперь превратить его в правильную монаду, нужно реализовать способ создания потока из значения, а также функцию `join`.

Сначала займемся самым простым: создадим поток данных на основе значения или списка значений (для удобства). Этот поток не принимает никаких сообщений, а только посылает их, подобно классу `service`, который вы видели раньше.

Нам не нужно принимать сообщения от других акторов. Пользователь должен иметь возможность указать значения при создании потока, а поток, в свою очередь, должен сохранить их и, когда к нему подключится актор-приемник, отправить эти значения ему:

```
template <typename T>
class values {
public:
    using value_type = T;

    explicit values(std::initializer_list<T> values)
        : m_values(values)
    {
    }

    template <typename EmitFunction>
    void set_message_handler(EmitFunction emit)
    {
        m_emit = emit;
        std::for_each(m_values.cbegin(), m_values.cend(),
            [&](T value) { m_emit(std::move(value)); });
    }

private:
    std::vector<T> m_values;
    std::function<void(T&&)> m_emit;
};
```

Этот класс можно использовать как конструктор монад для реактивных потоков. Его работоспособность легко проверить, передав конкретное значение непосредственно в объект-приемник:

```
auto pipeline = values{42} | sink_to_cerr;
```

Этот код создаст поток с единственным значением. Когда к нему подключится `sink_to_cerr`, он получит это значение и выведет его в `std::cerr`.



### 12.3.4 Объединение потоков в один поток

Последнее, что нужно сделать, чтобы превратить реактивный поток в монаду, – определить функцию `join`. Допустим, мы решили принимать соединения с нашей веб-службой с использованием нескольких портов. Для этого можно запустить несколько экземпляров службы, по одному для каждого порта, и затем объединить сообщения, получаемые всеми экземплярами, в один унифицированный поток.

То есть нам нужна возможность написать, например, такой код:

```
auto pipeline =
  values{42042, 42043, 42044}
  | transform([&](int port) {
    return service(event_loop, port);
  })
  | join()
  | sink_to_cerr;
```

Эта задача может показаться сложной, но, опираясь на уже имеющиеся знания, ее легко решить. Решение похоже на реализацию `transform`. Обе функции, `join` и `transform`, получают сообщения одного типа и выводят сообщения другого типа. Единственное отличие в том, что `join` получает сообщения, которые являются новыми потоками. Она должна принимать сообщения из этих потоков и посылать их в выходной поток.

#### Листинг 12.7 Реализация преобразования `join`

```
namespace detail {
  template <
    typename Sender,
    typename SourceMessageType =
      typename Sender::value_type,
    typename MessageType =
      typename SourceMessageType::value_type>
  class join_impl {
  public:
    using value_type = MessageType;

    ...

    void process_message(SourceMessageType&& source)
    {
```

Тип принимаемых потоков данных

Тип сообщений, получаемых из входных потоков, которые требуется передать получателям

```

    m_sources.emplace_back(std::move(source));
    m_sources.back().set_message_handler(m_emit);
}

private:
    Sender m_sender;
    std::function<void(MessageType&&)> m_emit;
    std::list<SourceMessageType> m_sources;
};

```

Получив новый входной поток, сохранить его и переслать сообщения из него в свой выходной поток

Хранит все входные потоки, чтобы не дать им уничтожиться. Здесь используется список для уменьшения числа перемещений

Теперь, когда у нас есть `join` и `transform`, можно, наконец, сказать, что реактивные потоки данных – это монады.

## 12.4 Фильтрация реактивных потоков

Итак, мы показали, что реактивные потоки с созданными нами преобразованиями являются монадами, и постарались сделать их максимально похожими на диапазоны. Теперь реализуем еще одну полезную функцию диапазонов.

В предыдущих примерах мы выводили в стандартный поток ошибок `cerr` все сообщения, поступающие от клиента. Предположим, что нам нужно отфильтровать некоторые из них. Пусть это будут пустые сообщения и сообщения, начинающиеся с символа решетки (`#`), потому что они представляют комментарии в данных, которые отправляет клиент.

Вот как мог бы выглядеть соответствующий код:

```

auto pipeline =
    service(event_loop)
    | transform(trim)
    | filter([](const std::string& message) {
        return message.length() > 0 &&
            message[0] != '#';
    })
    | sink_to_cerr;

```

Нам нужно создать новый модификатор потока, похожий на `transform`. Он будет получать сообщения и отправлять только те, которые удовлетворяют заданному предикату. Основное отличие от `transform` и `join` состоит в том, что `filter` принимает и отправляет сообщения одного и того же типа.

### Листинг 12.8 Актор преобразования для фильтрации сообщений в потоке

```

template <typename Sender,
         typename Predicate,
         typename MessageType =
             typename Sender::value_type>
class filter_impl {

```

Получаемые и отправляемые сообщения имеют один и тот же тип

```

public:
    using value_type = MessageType;

    ...

    void process_message(MessageType&& message) const
    {
        if (std::invoke(m_predicate, message)) {
            m_emit(std::move(message));
        }
    }
};

private:
    Sender m_sender;
    Predicate m_predicate;
    std::function<void(MessageType&&)> m_emit;
};

```

Если полученное сообщение соответствует предикату, отправить его дальше

Фильтрация может пригодиться, например, когда понадобится удалить недопустимые данные или данные, не представляющие интереса.

## 12.5 Обработка ошибок в реактивных потоках

Поскольку наша веб-служба должна принимать сообщения в формате JSON, она должна уметь обрабатывать синтаксические ошибки. В главах 9 и 10 мы обсудили несколько способов функциональной обработки ошибок. Там мы использовали тип `optional<T>`, значение которого можно оставить пустым, чтобы обозначить ошибку. Также мы использовали тип `expected<T, E>`, чтобы точно указать, какая ошибка произошла.

Библиотека, которую мы решили применить для обработки JSON, основана на исключениях. Поэтому для обработки ошибок используем тип `expected<T, E>`. Параметр `T` будет определять тип фактического сообщения, а параметр `E` будет типом указателя на исключение (`std::exception_ptr`). Каждое сообщение будет содержать либо значение, либо указатель на исключение.

Для вызова функций, способных сгенерировать исключения, используем функцию `try`, которую мы определили в главе 10. Напомним, что `try` – это вспомогательная функция, которая преобразует другие функции, генерирующие исключения, в функции, возвращающие экземпляр `expected<T, std::exception_ptr>`. В `try` можно передать любой вызываемый объект, и этот объект будет выполнен. Если вызов завершится успехом, `try` вернет значение, завернутое в объект `expected`. Если возникнет исключение, `try` вернет объект `expected`, содержащий указатель на это исключение.

Давайте завернем функцию `json::parse` в вызов `try` и используем `transform` для преобразования в объекты JSON всех сообщений, получаемых от клиента. В результате у нас получится поток объектов `expected<json, std::exception_ptr>`.

## Листинг 12.9 Парсинг строк в объекты JSON

```

auto pipeline =
  service(event_loop)
  | transform(trim)
  | filter([](const std::string& message) {
      return message.length() > 0 &&
         message[0] != '#';
    })
  | transform([](const std::string& message) {
      return mtry([&] {
          return json::parse(message);
        });
    })
  | sink_to_cerr;

```



Попытаться выполнить парсинг каждой полученной строки. В результате получится объект JSON или указатель на исключение (то есть `expected<json, std::exception_ptr>`)

Нам нужно извлечь данные из каждого объекта JSON в соответствующую структуру. Для этого определим структуру для хранения URL-адреса и текста закладки, а также напишем функцию, принимающую объект JSON и возвращающую закладку, если объект содержит требуемые данные, или ошибку, если это не так:

```

struct bookmark_t {
    std::string url;
    std::string text;
};

using expected_bookmark = expected<bookmark_t, std::exception_ptr>;

expected_bookmark bookmark_from_json(const json& data)
{
    return mtry([&] {
        return bookmark_t{data.at("FirstURL"), data.at("Text")};
    });
}

```



Библиотека для работы с форматом JSON генерирует исключение при попытке получить с помощью функции `at` доступ к атрибуту, отсутствующему в объекте JSON. Поэтому ее тоже нужно заключить в вызов `mtry`, по аналогии с `json::parse`. Теперь можно продолжить обработку сообщений.

Мы выполнили парсинг строки и получили `expected<json, ...>`. Теперь нужно отфильтровать недопустимые значения и попытаться создать значения `bookmark_t` из допустимых объектов JSON. Кроме того, поскольку преобразование в `bookmark_t` может завершиться неудачей, мы также должны исключить все ошибочные значения. Для этого используем комбинацию `transform` и `filter`:

```

auto pipeline =
  service(event_loop)
  | transform(trim)
  | filter(...)

```

```

| transform([](const std::string& message) {
|     return mtry([&] {
|         return json::parse(message);
|     });
| })
| filter(&expected_json::is_valid)
| transform(&expected_json::get)
| transform(bookmark_from_json)
| filter(&expected_bookmark::is_valid)
| transform(&expected_bookmark::get)

| sink_to_cerr;

```

← Передать дальше только допустимые объекты JSON

← Передать дальше только допустимые закладки

Предыдущий шаблон понятен: выполнить преобразование, которое может потерпеть неудачу, отфильтровать все недействительные результаты и извлечь значение из объекта `expected` для дальнейшей обработки. Проблема в том, что эта реализация выглядит слишком подробно. Но это не главная проблема. Еще большая проблема заключается в том, что мы отбрасываем информацию об ошибке, столкнувшись с ней. Если информация об ошибках не представляет интереса, используйте тип `optional` вместо `expected`.

Теперь пример становится более интересным. Мы получили поток значений, каждое из которых является экземпляром монады `expected`. До сих пор мы рассматривали потоки как монады, а все сообщения – как обычные значения. Станет ли этот код более читабельным, если интерпретировать `expected` как монаду?

Вместо включения полной цепочки `transform-filter-transform` в код преобразуем экземпляры `expected` монадическим образом. Если взглянуть на сигнатуру функции `bookmark_from_json`, можно заметить, что она принимает значение и возвращает экземпляр монады `expected`. Мы уже видели, что подобные функции можно комбинировать, используя прием композиции монад: `mbind`.

### Листинг 12.10 Интерпретация экземпляра `expected` как монады

```

auto pipeline =
    service(event_loop)
    | transform(trim)
    | filter(...)

    | transform([](const std::string& message) {
        return mtry([&] {
            return json::parse(message);
        });
    })

    | transform([](const auto& exp_json) {
        return mbind(exp_json, bookmark_from_json);
    })

```

До этого момента мы имеем поток обычных значений. Здесь мы получаем поток экземпляров `expected`: монад внутри другой монады

Если реализовать возможность применения `mbind` для преобразования экземпляров `expected`, ее можно использовать для преобразования потоков объектов `expected`

```
... ←
| sink_to_cerr; | При желании можно использовать сколько угодно
                 | преобразований, способных завершиться неудачей
```

Это отличный пример совместной работы подъема функций и связывания монад. Мы начали с функции, работающей с обычными значениями типа `json`, связали ее, чтобы получить возможность работать с `expected_json`, а затем подняли для работы с потоками объектов `expected_json`.

## 12.6 Возврат ответа клиенту

На настоящий момент служба принимает запросы от клиентов, но ничего не возвращает им. Этого вполне достаточно, если требуется лишь сохранить закладки, отправленные клиентами, и вместо `sink_to_cerr` записывать их в базу данных.

Но на практике часто бывает нужно отправить какой-то ответ клиенту, по крайней мере чтобы подтвердить получение сообщения. На первый взгляд это кажется проблемой, учитывая дизайн службы. Мы собрали все сообщения в один поток, и наша основная программа даже не подозревает о существовании клиентов.

Решить проблему можно двумя способами. Первый: вернуться к чертежной доске и изменить дизайн. Второй: прислушаться к внутреннему голосу, который шепчет: «Монады. Вы знаете, что это можно сделать с помощью монад». Вместо полной перedelки всего, что было реализовано, прислушаемся к внутреннему голосу.

Чтобы послать ответ клиенту, а не записывать закладки в `std::cerr` или в базу данных, нужно знать, какой клиент отправил то или иное сообщение. Единственный компонент в системе, который может это сказать, — объект `service`. Мы должны каким-то образом передать информацию о клиенте через весь конвейер — от `service(event_loop)` до объекта-приемника — без изменения на любом из этапов.

Для этого объект `service` должен передавать в конвейер сообщения, содержащие только строки, но и указатель на сокет, который можно использовать для связи с клиентом. Поскольку сокет должен проходить через все преобразования, изменяющие тип сообщения, мы создадим шаблонный класс, хранящий указатель на сокет вместе с сообщением.

### Листинг 12.11 Структура, хранящая указатель на сокет вместе с сообщением

```
template <typename MessageType>
struct with_client {
    MessageType value;
    tcp::socket* socket;

    void reply(const std::string& message) const
    {
        // Копировать и сохранять сообщение, пока async_write
        // не завершит асинхронную операцию
```

```

auto sptr = std::make_shared<std::string>(message);
boost::asio::async_write(
    *socket,
    boost::asio::buffer(*sptr, sptr->length()),
    [sptr](auto, auto) {});
}
};

```



Чтобы упростить основную программу и избавиться от зависимости от Boost.Asio, мы написали также функцию-член `reply` (см. полную реализацию в сопровождающем примере `bookmark-service-with-reply`), которую можно использовать для отправки сообщений клиенту.

`with_client` – это обобщенный тип, содержащий дополнительную информацию. Теперь вы знаете, что должны думать в терминах *функторов* и *монад*, когда видите что-то подобное. Мы легко можем написать необходимые функции, чтобы показать, что `with_client` является монадой.

### Функция `join` для `with_client`



Единственная функция, которая заслуживает особого внимания, – это `join`. Если вложить одну структуру `with_client` в другую, мы получим одно значение и два указателя на сокет, но нам нужно получить после объединения значение с единственным сокетом.

Мы можем сохранить сокет из самого внутреннего экземпляра `with_client`, если он не равен `null`, или из самого внешнего экземпляра. В данном случае, что бы мы ни делали, мы всегда должны вернуть ответ клиенту, инициировавшему соединение, а значит, сохраним самый внешний сокет.

Как вариант можно изменить класс `with_client` и сохранить коллекцию сокетов. В этом случае при объединении с вложенным экземпляром `with_client` потребуется объединить эти две коллекции.

Что еще нужно изменить в программе, чтобы она компилировалась, после того как мы заставим службу отправлять сообщения типа `with_client<std::string>` вместо простых строк? Прежде всего нужно изменить приемник. Он должен посылать сообщения клиенту, а не записывать их в `std::cerr`. Приемник будет получать сообщения типа `with_client<expected_bookmark>`. Мы должны проверить, содержит ли объект `expected` ошибку, и затем действовать соответствующим образом:

```

auto pipeline =
    service(event_loop)
    ...

    | sink([](const auto& message) {
        const auto exp_bookmark = message.value;

        if (!exp_bookmark) {
            message.reply("ERROR: Request not understood\n");

```

```
        return;
    }

    if (exp_bookmark->text.find("C++") != std::string::npos) {
        message.reply("OK: " + to_string(exp_bookmark.get()) +
            "\n");
    } else {
        message.reply("ERROR: Not a C++-related link\n");
    }
}
});
```

Если во время парсинга сообщения возникает какая-либо ошибка, мы уведомим клиента. Кроме того, поскольку служба должна принимать только закладки, связанные с языком C++, мы сообщаем об ошибке, если текст закладки не содержит «C++».

Мы изменили службу и приемник, добавив возможность послать ответ клиенту. Что еще нужно изменить?

Можно последовательно изменить все преобразования, чтобы они принимали только что добавленный тип `with_client`. Но можно пойти другим путем. По аналогии с обработкой ошибок, возникающих при преобразовании, когда мы использовали `mbind` вместо передачи каждого сообщения через цепочку модификаторов `transform-filter-transform`, попробуем реализовать что-то подобное.

Это еще один уровень монад. У нас есть поток (который является монадой) значений `with_client` (которые также являются монадами), каждое из которых содержит значение `expected<T, E>` (третья вложенная монада). Мы можем просто поднять все это на один уровень.

Для этого переопределим функции `transform` и `filter`, реализованные для реактивных потоков, которые находятся в пространстве имен `reactive::operators` (см. пример `bookmark-service-with-reply`), чтобы они могли работать с реактивным потоком значений `with_client`:

```
auto transform = [](auto f) {
    return reactive::operators::transform(lift_with_client(f));
};

auto filter = [](auto f) {
    return reactive::operators::filter(apply_with_client(f));
};
```

`lift_with_client` – это простая функция, которая поднимает любую функцию, преобразующую тип `T1` в тип `T2`, до функции, преобразующей тип `with_client<T1>` в тип `with_client<T2>`. Функция `apply_with_client` действует аналогично, но возвращает развернутое значение результата вместо объекта `with_client`.

Это все, что нужно сделать. Остальной код будет продолжать работать без каких-либо изменений. Код в следующем листинге доступен в примере `bookmark-service-with-reply/main.cpp`.

## Листинг 12.12 Окончательная версия сервера

```

auto transform = [](auto f) {
    return reactive::operators::transform(lift_with_client(f));
};
auto filter = [](auto f) {
    return reactive::operators::filter(apply_with_client(f));
};

boost::asio::io_service event_loop;

auto pipeline =
    service(event_loop)
    | transform(trim)

    | filter([](const std::string& message) {
        return message.length() > 0 && message[0] != '#';
    })

    | transform([](const std::string& message) {
        return mtry([&] { return json::parse(message); });
    })

    | transform([](const auto& exp) {
        return mbind(exp, bookmark_from_json);
    })

    | sink([](const auto& message) {
        const auto exp_bookmark = message.value;

        if (!exp_bookmark) {
            message.reply("ERROR: Request not understood\n");
            return;
        }

        if (exp_bookmark->text.find("C++") != std::string::npos) {
            message.reply("OK: " + to_string(exp_bookmark.get()) +
                "\n");
        } else {
            message.reply("ERROR: Not a C++-related link\n");
        }
    });

std::cerr << "Service is running...\n";
event_loop.run();

```



Этот пример наглядно демонстрирует силу обобщенных абстракций, таких как функторы и монады. Нам удалось реорганизовать весь конвейер простым изменением нескольких компонентов, оставив логику основной программы нетронутой.

## 12.7 Создание акторов с изменяемым состоянием

Мы всегда должны стараться избавляться от изменяемого состояния, но иногда оно может очень пригодиться. До этого момента такое изменяемое состояние, возможно, оставалось для вас незамеченным, тем не менее мы уже создали одно преобразование с изменяемым состоянием: преобразование `join`. Оно хранит список всех отправителей, чьи сообщения передаются.

В этом случае наличие изменяемого состояния в преобразовании `join` является деталью реализации – мы должны запоминать отправителя. Но в некоторых ситуациях необходимы акторы с явным изменяемым состоянием.

Чтобы обеспечить отзывчивость службы, мы не можем присвоить всем сообщениям одинаковый приоритет. Предположим, некоторый клиент пытается выполнить атаку типа «отказ в обслуживании» (DoS), посылая массу сообщений, из-за чего служба оказывается не в состоянии отвечать другим клиентам.

Есть разные подходы к решению подобных проблем. Наиболее простым из них является регулирование количества сообщений, которое клиент может послать в единицу времени. Получив сообщение от клиента, служба может отклонять все последующие его сообщения, пока не пройдет определенный интервал времени. Например, мы можем установить предел, равный одному сообщению в секунду, то есть получив сообщение от клиента, служба будет игнорировать этого клиента в течение 1 секунды.

Для этого можно создать актора, который будет принимать сообщения и запоминать клиента, отправившего сообщение, а также абсолютное время, когда служба сможет снова начать принимать сообщения от него. Для этого такой актор должен иметь изменяемое состояние; он должен запоминать и обновлять тайм-ауты для каждого клиента.

В обычных программных системах с параллельной обработкой наличие изменяемого состояния требует синхронизации. Но это не относится к акторам. Актор – это однопоточный компонент, полностью изолированный от других акторов. Изменяемое состояние актора не может быть изменено параллельными процессами. А поскольку в этом случае нет общего изменяемого состояния, выполнять синхронизацию не требуется.

Как упоминалось выше, в примере службы закладок использовались упрощенные акторы. Служба может обслуживать несколько клиентов одновременно, но вся обработка по-прежнему выполняется в одном потоке; асинхронный обмен сообщениями используется только при взаимодействии с клиентом (где применяется библиотека `Boost.Asio`).

В более универсальной модели каждый актор действует в отдельном процессе или потоке (или в чем-то еще, более легковесном, действующем подобно потоку выполнения). Поскольку каждый актор имеет свой маленький мир, в котором время течет независимо от времени других акторов, сообщения не могут обрабатываться синхронно.

Всем актерам нужны свои очереди сообщений, в которые можно добавлять столько сообщений, сколько понадобится. Актор (как однопоточный компонент) будет обрабатывать сообщения из очереди одно за другим.

В нашем примере сообщения обрабатываются синхронно. Вызов `m_emit` для одного из акторов немедленно вызывает функцию `process_message` для другого. Чтобы создать многопоточную систему, эти вызовы нужно сделать косвенными. А для этого понадобится организовать цикл обработки сообщений в каждом потоке, который будет доставлять сообщения нужному актору.

Изменить инфраструктуру будет непросто, но идея актора как изолированного компонента, который получает и отправляет сообщения, останется прежней. Изменится только механизм доставки сообщений.

Реализация, лежащая в основе, изменится, но дизайн самой программы останется прежним. Проектируя конвейер обработки сообщений, мы не полагались на однопоточность системы. Мы организовали его как набор изолированных компонентов, которые обрабатывают сообщения друг друга, – мы не ставили перед собой задачу доставлять эти сообщения немедленно. Разработанный нами конвейер сообщений можно оставить полностью нетронутым, как концептуально, так и с точки зрения реализации, даже если полностью изменить основу системы.

## 12.8 *Распределенные системы на основе акторов*

Есть еще одно преимущество подхода к разработке параллельных программных систем в виде набора акторов, посылающих сообщения друг другу. Я уже говорил, что все акторы изолированы; они не имеют ничего общего друг с другом. Единственное, что гарантируется, – сообщения в очереди актора будут обрабатываться в том же порядке, в каком они были отправлены.

Акторам не важно, действуют они в одном потоке выполнения, в разных потоках в одном процессе, в разных процессах на одном компьютере или на разных компьютерах, для них важно иметь возможность посылать сообщения друг другу. Из этого следует, что службу закладок легко можно масштабировать по горизонтали, не меняя основной ее логики. Каждый из акторов может действовать на отдельном компьютере и отправлять сообщения по сети.

Так же, как переход с однопоточной системы на многопоточную не влечет за собой никаких изменений в логике основной программы, переход с обычной системы, основанной на акторах и реактивных потоках, на распределенную тоже не повлияет на нее. Единственное, что понадобится изменить, – систему доставки сообщений. В многопоточном приложении нужно создать циклы обработки сообщений во всех потоках и передавать сообщения в соответствующий цикл и соответствующему актору. В распределенных системах все то же самое, просто появляется еще один уровень косвенности. Сообщения должны перемещаться не

только между потоками, но и между компьютерами, по сети, в сериализованном виде.



**СОВЕТ** За дополнительной информацией по теме, рассматривавшейся в этой главе, обращайтесь по адресу: <https://forums.manning.com/posts/list/43781.page>.

## Итоги

- Большинство программистов на C++ пишут процедурный код. Я советую прочитать книгу Дэвида Уэста (David West) «Object Thinking» (Microsoft Press, 2004), которая поможет вам начать писать более качественный код. Она поможет вам, даже если вы решите писать программы исключительно в функциональном стиле.
- Люди способны решать сложные задачи, общаясь друг с другом. Мы не можем читать мысли друг друга, но способность общаться помогает нам достигать своих целей. Это рассуждение привело к изобретению модели акторов.
- Монады прекрасно сотрудничают друг с другом. Не бойтесь вкладывать их друг в друга.
- Для реактивных потоков можно реализовать такие же преобразования, как для диапазонов ввода. Но реализовать такие функции, как сортировка, потому что для сортировки необходим произвольный доступ ко всем элементам, а мы даже не знаем, сколько элементов будет иметь реактивный поток – они могут оказаться бесконечными.
- Как и в случае с типом `future`, обобщенная реализация реактивных потоков не ограничивается отправкой значений; она также может отправлять специальные сообщения, такие как «конец потока». Это может пригодиться для более эффективного использования памяти: поток можно уничтожить, если знать, что он больше не будет доставлять сообщений.

---

# 13

## Тестирование и отладка

---



### **О чем говорится в этой главе:**

- предотвращение ошибок времени выполнения заменой ошибками времени компиляции;
- преимущества чистых функций в модульном тестировании;
- автоматическое генерирование тестовых случаев для чистых функций;
- тестирование кода сравнением с существующими решениями;
- тестирование параллельных систем на основе монад.

Компьютеры становятся все более вездесущими. У нас уже есть умные часы (смарт-часы), телевизоры, тостеры и многие другие приборы и устройства, внутри которых работают специализированные компьютеры. Последствия ошибок в программном обеспечении в настоящее время варьируются от небольших неприятностей до серьезных проблем, как то: кража личных данных и даже опасность для жизни.

Поэтому в наши дни как никогда важно, чтобы программное обеспечение, которое мы пишем, работало правильно: оно должно делать именно то, что должно, и не содержать ошибок. На первый взгляд это простая задача, потому что кто, находясь в здравом уме, захочет писать программы с ошибками? Однако все не так просто, широко распространено мнение, что все нетривиальные программы содержат ошибки. Мы настолько привыкли к этому факту, что подсознательно готовы искать обходные пути с целью избежать ошибок, которые обнаруживаем в используемых программах.



Все вышесказанное – грустная правда, но это не оправдание для отказа от попыток писать правильные программы. Проблема лишь в том, что это сложно.

Большинство возможностей языков программирования высокого уровня проектируются с учетом этой проблемы. Это особенно верно для C++, при разработке которого значительные усилия были направлены на то, чтобы упростить создание безопасных программ или, если быть более точным, помочь программистам избежать распространенных ошибок программирования.

Мы видели, что для большей безопасности в язык были добавлены: новые способы управления динамической памятью с использованием умных указателей, автоматическое определение типов с применением спецификатора `auto`, предотвращение случайного неявного приведения типа и монады, такие как `std::future`, упрощающие разработку параллельных программ без использования низкоуровневых примитивов синхронизации, таких как мьютексы. Мы также увидели смещение в сторону более строгого программирования на основе алгебраических типов данных `std::option` и `std::variable`, единиц измерения и пользовательских литералов (например, `std::chrono::duration`) и т. д.

### 13.1 Программа, которая компилируется, – правильная?



Все эти возможности призваны помочь нам избежать распространенных ошибок программирования и перенести обнаружение ошибок со времени выполнения на время компиляции. Одним из самых известных примеров, как простая ошибка может привести к огромным потерям, является ошибка в космической автоматической станции Mars Climate Orbiter; в большей части кода предполагаемые расстояния измерялись в метрических единицах, но в одном месте использовались имперские (английские) единицы.

*Комиссия по расследованию аварии МСО определила, что основной причиной потери космического корабля МСО стало использование единиц, отличных от метрических, в программном обеспечении наземного комплекса «Small Forces», участвующего в вычислении параметров траекторий. В частности, параметры производительности двигателя были представлены в английских единицах вместо метрических, которые, в свою очередь, использовались в программном коде приложения под названием SM\_FORCES (small forces).*

– NASA<sup>1</sup>

Этой ошибки можно было бы избежать, если бы код использовал более строгую типизацию вместо простых значений. Мы легко можем опреде-

<sup>1</sup> NASA: Mars Climate Orbiter Mishap Investigation Board Phase I Report. November 10, 1999. <http://mng.bz/YO17>.

лить тип для обработки расстояний, требующий использования определенной единицы измерения:

```
template <typename Representation,
         typename Ratio = std::ratio<1>
class distance {
    ...
};
```



Этот тип позволяет создавать разные типы для разных единиц измерения и представлять число единиц в виде произвольного числового типа – целочисленного, вещественного или специального типа, созданного нами. Если предположить, что единицей по умолчанию является метр, мы могли бы создать другие единицы (например, выражающие мили в метрах):

```
template <typename Representation>
using meters = distance<Representation>;

template <typename Representation>
using kilometers = distance<Representation, std::kilo>;

template <typename Representation>
using centimeters = distance<Representation, std::centi>;

template <typename Representation>
using miles = distance<Representation, std::ratio<1609>>;
```



Мы также могли бы упростить их использование, определив для них специальные литералы:

```
constexpr kilometers<long double> operator ""_km(long double distance)
{
    return kilometers<long double>(distance);
}
... // Аналогично для других единиц
```

После этого в программе можно использовать любые единицы, какие заблагорассудится. Но при попытке смешать их или сравнить разные единицы компилятор сообщит о несоответствии типов:

```
auto distance = 42.0_km + 1.5_mi; // ошибка!
```

Также мы могли бы добавить функции преобразования для удобства, но главное, чего мы достигли бы, – за счет небольшой абстракции с нулевыми затратами мы заменили бы ошибку времени выполнения на ошибку времени компиляции. Это имеет огромное значение для разработки программного обеспечения – разница между потерей космической станции и обнаружением ошибки задолго до того, как станция даже будет запланирована к запуску, очень велика.

Используя высокоуровневые абстракции, описанные в этой книге, многие распространенные ошибки можно обнаруживать во время ком-

пиляции. По этой причине некоторые даже говорят, что после успешной компиляции функциональной программы она обязательно будет работать правильно.

Очевидно, что все нетривиальные программы содержат ошибки. Это относится и к функциональным программам. Но чем короче код (а, как вы видели, функциональный стиль программирования и абстракций, которые в нем используются, позволяют писать код, намного более короткий, чем аналогичный ему императивный код), тем меньше мест, где может быть допущена ошибка. И чем больше ошибок обнаружится во время компиляции, тем меньше останется ошибок времени выполнения.

## 13.2 Модульное тестирование и чистые функции

Мы всегда должны стараться писать код так, чтобы потенциальные ошибки обнаруживались во время компиляции, однако это не всегда возможно. Во время выполнения программы должны обрабатываться реальные данные, к тому же мы сами можем допускать логические ошибки или возвращать неправильные результаты.

По этой причине мы должны писать тесты, которые автоматически проверяли бы наше программное обеспечение. Автоматические тесты также полезны для регрессионного тестирования после изменения существующего кода.

Самый нижний уровень в иерархии тестирования занимают *модульные тесты*. Цель модульного тестирования состоит в том, чтобы выделить небольшие части программы (модули), проверить их по отдельности и убедиться в их правильности. Они проверяют правильность работы самих частей программы (модулей), а не их взаимодействия друг с другом.

Самое замечательное, что модульное тестирование в функциональном программировании похоже на модульное тестирование императивных программ. При создании модульных тестов можно использовать привычные библиотеки. Разница лишь в том, что тестировать чистые функции немного проще.

Традиционный модульный тест для объекта с состоянием включает инициализацию состояния объекта, выполнение действия над этим объектом и проверку результата. Представьте, что у нас есть класс, обрабатывающий текстовый файл. Он может иметь несколько функций-членов и среди них, например, подсчитывающую количество строк в файле, как было показано в главе 1 – путем подсчета количества символов перевода строки в файле:

```
class textual_file {
public:
    int line_count() const;
    ...
};
```

Чтобы выполнить модульное тестирование этой функции, нужно создать несколько файлов, для каждого создать объект `textual_file` и про-

верить результат функции `line_count`. Это типичный подход к тестированию классов с состоянием. Мы должны инициализировать состояние и только потом выполнить тестирование. Часто желательно выполнить один и тот же тест несколько раз, с разными состояниями, которые может иметь класс.

Обычно это означает, что для надежного тестирования нужно знать, какие части состояния класса могут повлиять на исход тестирования. Например, состояние класса `textual_file` может включать флаг, указывающий доступность файла для записи. Мы должны знать внутреннее устройство класса, чтобы понять, влияет ли этот флаг на результат `line_count`, и при необходимости создать тесты, которые проверяют работу функции со всеми файлами – доступными как для записи, так и только для чтения.

Чистые функции существенно упрощают тестирование. Результат такой функции зависит лишь от аргументов. Если только вы не добавили лишних аргументов в определение функции исключительно ради своего удовольствия, то сможете предположить, что для вычисления результата используются все аргументы.

Нам не нужно настраивать какое-либо внешнее состояние перед запуском тестов, и мы можем писать тесты, не задумываясь о внутренней реализации тестируемой функции. Такое разделение функций и внешнего состояния также позволяет писать более обобщенные функции, что расширяет возможности повторного их использования и тестирования одной и той же функции в разных контекстах.

Взгляните на следующую чистую функцию:

```
template <typename Iter, typename End>
int count_lines(const Iter& begin, const End& end)
{
    using std::count;
    return count(begin, end, '\n');
}
```

Будучи чистой, эта функция не нуждается в каком-либо внешнем состоянии для вычисления результата, она не использует ничего, кроме своих аргументов, и не изменяет аргументы.

При тестировании эту функцию можно вызывать без любых предварительных приготовлений и по нескольким типам – от списков и векторов до диапазонов и потоков ввода:

```
std::string s = "Hello\nworld\n";
assert(count_lines(begin(s), end(s)) == 2); | Тестирование со строкой

auto r = s | view::transform([](char c) { return toupper(c); });
assert(count_lines(begin(r), end(r)) == 2); | Тестирование с диапазоном

std::istrstream ss("Hello\nworld\n");
assert(count_lines(std::istreambuf_iterator<char>(ss),
                  std::istreambuf_iterator<char>()) == 2); | Тестирование с потоком ввода (чтобы не ограничиваться файлами)
```

```
std::forward_list<char> l;
assert(count_lines(begin(l), end(l)) == 0);
```

Тестирование  
с односвязным списком



При желании мы могли бы реализовать перегруженные версии `count_lines`, более удобные в использовании, чтобы не вызывать ее с парой итераторов. Это были бы однострочные обертки, не требующие тщательно-го тестирования.

Наша задача при написании модульных тестов состоит в выделении небольших частей программы и их проверке по отдельности. Каждая чистая функция сама по себе является изолированной частью программы, а это, кроме того что чистые функции легко тестируются, делает каждую такую функцию идеальным кандидатом на звание «модуля» в контексте модульного тестирования.

## 13.3 Автоматическое генерирование тестов

Модульные тесты полезны (и необходимы), но их основная проблема заключается в том, что мы должны писать их вручную. Это означает, что есть вероятность допустить ошибку в самом тесте и риск написать неправильный или неполный тест. Насколько сложнее найти орфографические ошибки в своем собственном письме, чем в чужом, настолько же труднее писать тесты для своего кода. Скорее всего, вы пропустите проверку тех же крайних случаев, которые забыли охватить в реализации. Было бы гораздо удобнее, если бы тестовые случаи можно было генерировать автоматически, опираясь на тестируемый код.

### 13.3.1 Генерирование тестовых случаев

Реализуя функцию `count_lines`, мы определили ее спецификацию: вернуть количество символов перевода строки, содержащихся в заданном наборе символов. Как сформулировать обратную задачу? По заданному числу сгенерировать все коллекции, количество строк которых равно этому числу. Это ведет к функции, возвращающей коллекцию строк:

```
std::vector<std::string> generate_test_cases(int line_count);
```

Если эти задачи действительно обратны друг другу, тогда для любой коллекции, сгенерированной методом `generate_test_cases(line_count)`, функция `count_lines` должна вернуть то же значение `line_count`, которое передано в `generate_test_cases`. И это условие должно соблюдаться для любого значения `line_count`, от нуля до бесконечности. Это правило можно записать так:

```
for (int line_count : view::ints(0)) {
    for (const auto& test : generate_test_cases(line_count)) {
        assert(count_lines(test) == line_count);
    }
}
```

Это был бы идеальный тест, но у него есть одна маленькая проблема. Количество проверяемых случаев бесконечно, потому проверяется диапазон всех целых чисел, начиная с нуля.

И для каждого из них можете получить бесконечное количество строк, которые имеют заданное количество символов перевода строки.

Поскольку все эти случаи проверить невозможно, нужно сгенерировать подмножество задач и проверить, выполняется ли правило для этого подмножества. Сгенерировать отдельный образец строки с заданным количеством символов перевода строки тривиально просто. Для этого достаточно сгенерировать достаточное количество случайных строк и объединить их в одну строку, поместив символы перевода строки между ними. Каждая строка будет иметь случайную длину и случайные символы внутри, от нас требуется лишь гарантировать, что они не содержат символа перевода строки:

```
std::string generate_test_case(int line_count)
{
    std::string result;

    for (int i = 0; i < line_count; ++i) {
        result += generate_random_string() + '\n';
    }

    result += generate_random_string();
    return result;
}
```

Эта функция генерирует один тестовый случай: единственную строку, содержащую точно `line_count` символов перевода строки. Далее можно определить функцию, возвращающую бесконечный диапазон этих примеров:

```
auto generate_test_cases(int line_count)
{
    return view::generate (std::bind(generate_test_case, line_count));
}
```

Теперь нужно ограничить количество тестов. Вместо того чтобы охватывать все целые числа и обрабатывать бесконечное количество коллекций для каждого, можно добавить predefined ограничения.

### Листинг 13.1 Тестирование `count_lines` на множестве случайно сгенерированных тестов

```
for (int line_count :
    view::ints(0, MAX_NEWLINE_COUNT)) {
    for (const auto& test :
        generate_test_cases(line_count)
        | view::take(TEST_CASES_PER_LINE_COUNT)) {
        assert(line_count ==
            count_lines(begin(test), end(test)));
    }
}
```

Вместо охвата всех целых чисел проверяется только predefined часть из них

Определяет число тестовых случаев для каждого числа строк

Эта проверка охватывает только подмножество всех возможных входных данных, но каждый раз, когда тест будет выполняться, он сгенерирует новый набор случайных примеров. С каждым новым прогоном пространство проверенных входных данных будет расширяться.

Недостаток случайного подхода состоит в том, что в некоторых вызовах тесты могут терпеть неудачу, а в других – нет. Это может привести к неверному выводу, что ошибка тестирования вызвана последними изменениями в программе. Поэтому всегда полезно в вывод с результатами тестирования добавлять начальное число, использованное для инициализации генератора случайных чисел, чтобы впоследствии легко было воспроизвести ошибку и найти версию программы, в которой она появилась.

### 13.3.2 Тестирование на основе свойств

Иногда приходится сталкиваться с проблемами, для которых проверки уже известны или они намного проще, чем сама проблема. Представьте, что мы решили протестировать функцию, которая переставляет элементы в векторе в обратном порядке:

```
template <typename T>  
std::vector<T> reverse(const std::vector<T>& xs);
```

Для этого можно создать несколько тестовых случаев и проверить, правильно ли работает обратная задача. Но, опять же, она должна охватывать только часть возможных случаев. Можно попытаться найти правила, применимые к обратной функции.

Во-первых, определим обратную задачу перестановки элементов заданной коллекции *xs*. Нам нужно найти все коллекции, которые после перестановки дают оригинальную коллекцию *xs*. Существует только одна такая коллекция, и это *reverse(xs)*. Обращение, или сторнирование, коллекции – это обратная задача:

```
xs == reverse(reverse(xs));
```

Это верно для любой коллекции *xs*. Также можно добавить еще несколько свойств функции *reverse*:

- количество элементов в обращенной коллекции должно совпадать с количеством элементов в исходной коллекции;
- первый элемент исходной коллекции должен совпадать с последним элементом обращенной коллекции;
- последний элемент исходной коллекции должен совпадать с первым элементом обращенной коллекции.

Все эти условия должны соблюдаться для любой коллекции. Мы можем создать столько случайных коллекций, сколько пожелаем, и проверить выполнение всех этих правил для каждой.

## Листинг 13.2 Генерирование тестовых случаев и проверка свойств

```

for (const auto& xs : generate_random_collections()) {
    const auto rev_xs = reverse(xs);
    assert(xs == reverse(rev_xs));
    assert(xs.length() == rev_xs.length());
    assert(xs.front() == rev_xs.back());
    assert(xs.back() == rev_xs.front());
}

```

Если обратить коллекцию дважды, должна получиться оригинальная коллекция

Оригинальная и обращенная коллекции должны иметь одинаковое число элементов

Первый элемент оригинальной коллекции должен совпадать с последним элементом обращенной коллекции, и наоборот

Как и в предыдущем случае, когда мы проверяли правильность функции `count_lines`, при каждом новом запуске тестов будем проверять разные части пространства входных данных функции. Разница лишь в том, что нам не нужно создавать функцию – генератор тестовых примеров. Любой случайно сгенерированный пример должен удовлетворять всем свойствам функции `reverse`.

Так же можно поступать с другими задачами, которые не являются обратными к самим себе, но имеют свойства, которые должны сохраняться. Представьте, что нам нужно проверить правильность работы функции сортировки. Есть несколько способов реализации сортировки; некоторые более эффективны при сортировке данных в памяти, а некоторые лучше справляются с задачей при сортировке данных на устройствах хранения. Но все они должны соблюдать одни и те же правила:

- оригинальная коллекция должна иметь то же количество элементов, что и отсортированная;
- минимальный элемент в исходной коллекции должен совпадать с первым элементом в отсортированной коллекции;
- максимальный элемент в исходной коллекции должен совпадать с последним элементом в отсортированной коллекции;
- каждый последующий элемент в отсортированной коллекции должен быть больше или равен предшествующему элементу;
- сортировка обращенной коллекции должна дать тот же результат, что и сортировка оригинальной коллекции.

Мы определили набор свойств, которые легко проверить (этот список не является исчерпывающим, но его достаточно для демонстрации главной идеи).

## Листинг 13.3 Генерирование тестовых случаев и проверка свойств функции сортировки

```

for (const auto& xs : generate_random_collections()) {
    const auto sorted_xs = sort(xs);
    assert(xs.length() == sorted_xs.length());
}

```

Проверить совпадение количества элементов в исходной и отсортированной коллекциях

```

assert(min_element(begin(xs), end(xs)) ==
       sorted_xs.front());
assert(max_element(begin(xs), end(xs)) ==
       sorted_xs.back());

assert(is_sorted(begin(sorted_xs),
                 end(sorted_xs)));
assert(sorted_xs == sort(reverse(xs)));
}

```

Проверить совпадение наименьшего и наибольшего элементов в исходной коллекции с первым и последним элементами в отсортированной коллекции

Проверить, что каждый последующий элемент в отсортированной коллекции больше или равен предыдущему

Проверить, что сортировка обращенной коллекции дает тот же результат, что и сортировка оригинальной коллекции (имеется в виду общий порядок элементов)

Определив набор свойств функции и реализовав их проверку, можно сгенерировать случайные входные данные и передать их для проверки. Если какое-либо из свойств нарушится в любом из случаев, значит, реализация содержит ошибку.

### 13.3.3 Сравнительное тестирование

Теперь вы знаете, как автоматически генерировать тестовые случаи для функций, для которых известно, как решить обратную задачу, и как проверять свойства функций, которые должны соблюдаться независимо от входных данных. Существует еще один вариант, когда случайно сгенерированные тестовые случаи можно с успехом использовать в модульных тестах.

Представим, что нам нужно проверить реализацию структуры префиксного дерева (Bitmapped Vector Trie, BVT) из главы 8. Мы написали ее как неизменяемую (постоянную) структуру данных. Она выглядит и ведет себя как стандартный вектор с одним исключением: она оптимизирована для копирования и не допускает изменений на месте.

Самый простой способ проверить такую структуру – сравнить ее со структурой, которую она имитирует, в данном случае с обычным вектором. Мы должны протестировать все операции, которые определили в структуре, и сравнить результат с такими же или эквивалентными операциями, выполняемыми стандартным вектором. Для этого нужна возможность преобразовать стандартный вектор в вектор BVT и обратно, а также возможность сравнить содержимое структуры BVT и стандартного вектора.

Затем можно определить набор правил для проверки. И снова эти правила должны выполняться для любого случайного сбора данных. Первое, что нужно проверить, – что структура BVT, построенная на основе стандартного вектора, содержит те же данные, что и вектор, и наоборот. Затем протестировать все операции, применив их и к BVT и к стандартному вектору, и после каждой сравнить содержимое получившихся коллекций.

### Листинг 13.4 Генерирование тестовых случаев и сравнение содержимого BVT и вектора

```

for (const auto& xs : generate_random_vectors()) {
    const BVT bvt_xs(xs);
    assert(xs == bvt_xs);
    {
        auto xs_copy = xs;
        xs_copy.push_back(42);
        assert(xs_copy == bvt_xs.push_back(42));
    }
    if (xs.length() > 0) {
        auto xs_copy = xs;
        xs_copy.pop_back();
        assert(xs_copy == bvt_xs.pop_back());
    }
    ...
}

```

Если обе коллекции поддерживают итераторы, сравнение легко реализовать с помощью `std::equal`

Так как BVT – неизменяемая структура, это же поведение нужно имитировать для стандартного вектора. Сначала создать копию, а потом изменить ее




Эти приемы автоматической генерации тестовых случаев не являются взаимоисключающими. Их можно комбинировать для проверки одной функции.

Например, для тестирования своей реализации алгоритма сортировки можно использовать все три подхода:

- для каждого отсортированного вектора создать несортированный вектор, перетасовав его; сортировка перетасованной версии должна вернуть исходный вектор;
- проверить несколько свойств, которые вы уже видели и которые должны соблюдаться всеми реализациями сортировки;
- сгенерировать случайные данные и отсортировать их с помощью своего алгоритма сортировки и `std::sort`, а затем сравнить их.

Написав комплект проверок, вы сможете передавать им случайно сгенерированные примеры. И снова, если какая-то проверка потерпит неудачу, значит, ваша реализация содержит ошибку.

## 13.4 Тестирование параллельных систем на основе монад

В главе 12 мы рассмотрели реализацию простой веб-службы. Она была основана на реактивных потоках, но в ней также использовалось несколько других монадических структур: `expected<T, E>` для обработки ошибок и `with_socket<T>` для передачи указателя на сокет через логику программы, чтобы иметь возможность в любой момент отправить ответ клиенту.

Такая архитектура монадических потоков данных имеет несколько преимуществ, которые мы уже обсудили. Она легко реструктурируется; логика программы определена как набор полностью изолированных преобразований, которые легко можно задействовать в других частях той же программы или в иных программах.

Еще одно большое преимущество – простота изменения исходной реализации сервера без изменения основной логики программы в конвейере потока данных, как мы сделали это, добавив возможность ответить клиенту, – мы просто подняли преобразования на уровень выше, чтобы научить их работать с типом `with_socket<T>`, а вся остальная логика осталась прежней.

В этом разделе мы воспользуемся сходством всех монадических структур – все они имеют функции `mbind`, `transform` и `join`. Если основать логику программы на этих функциях (или функциях, построенных на их основе), можно свободно переключаться между монадами, не меняя логики основной программы, для тестирования этой программы. Одна из основных проблем при тестировании параллельных программных систем или программного обеспечения, в котором есть части, выполняющиеся асинхронно, заключается в сложности создания тестов, охватывающих все возможные взаимодействия между параллельными процессами. Если двум параллельным процессам нужно взаимодействовать друг с другом или совместно использовать одни и те же данные, многое может измениться, если один из них вдруг будет выполняться дольше, чем ожидалось.

Смоделировать это во время тестирования сложно, и почти невозможно обнаружить все проблемы, возникающие в процессе эксплуатации. Кроме того, воспроизведение проблемы, обнаруженной в промышленном окружении, может быть настоящей головной болью, потому что трудно воспроизвести одинаковые временные характеристики всех процессов.

При разработке небольшой веб-службы мы не делали никаких предположений (явных или неявных) о том, сколько времени займет то или иное преобразование. Мы даже не предполагали, что преобразования выполнялись синхронно.

Единственное, что мы предположили, – что у нас есть поток сообщений от клиента. Хотя этот поток имел асинхронную природу, сконструированный нами поток данных не выдвигал такого требования – он может обрабатывать даже синхронные потоки данных.

Давайте вспомним, как выглядел конвейер потока данных:

```
auto pipeline =
  source
  | transform(trim)

  | filter([](const std::string& message) {
    return message.length() > 0 && message[0] != '#';
  })
```

```

| transform([](const std::string& message) {
    return mtry([&] { return json::parse(message); });
})

| transform([](const auto& exp) {
    return mbind(exp, bookmark_from_json);
})

| sink([](const auto& message) {
    const auto exp_bookmark = message.value;

    if (!exp_bookmark) {
        message.reply("ERROR: Request not understood\n");
        return;
    }

    if (exp_bookmark->text.find("C++") != std::string::npos) {
        message.reply("OK: " + to_string(exp_bookmark.get()) +
            "\n");
    } else {
        message.reply("ERROR: Not a C++-related link\n");
    }
});

```



Поток строк поступает из источника, и мы преобразуем их в закладки. Получив этот код без сопроводительных пояснений, вы бы наверняка подумали, что источником является не какая-то служба на основе Boost.Asio, а коллекция, и для ее обработки используется библиотека range-v3.

Это главное преимущество данной конструкции с точки зрения тестирования: мы можем включать и выключать асинхронность по своему усмотрению. В промышленном окружении система будет использовать реактивные потоки, а в окружении тестирования мы сможем использовать обычные коллекции данных.

Давайте посмотрим, что нужно изменить, чтобы создать тестовую программу из веб-службы. Поскольку для тестирования конвейера компонент самой веб-службы нам не нужен, можно удалить весь код, который использует библиотеку Boost.Asio. Единственное, что нужно оставить, – тип-обертку, который используется для отправки сообщений клиенту. Поскольку у нас больше нет клиентов, вместо указателя на сокет будем хранить в этом типе ожидаемое сообщение-ответ. Затем, когда конвейер вызовет функцию `reply`, мы проверим, получили ли ожидаемое сообщение:

```

template <typename MessageType>
struct with_expected_reply {
    MessageType value;
    std::string expected_reply;

    void reply(const std::string& message) const
    {
        REQUIRE(message == expected_reply);
    }
};

```

Как и `with_socket`, эта структура хранит сообщение с контекстной информацией. Мы можем использовать этот класс в качестве замены `with_socket`; в самом конвейере ничего не изменилось.

Следующий шаг – переопределение преобразований в конвейере, чтобы использовать преобразования из библиотеки диапазонов вместо преобразований из библиотеки простых реактивных потоков. И снова нам не нужно менять конвейер; достаточно просто изменить определения `transform` и `filter` из исходной программы. Их нужно поднять для работы с `with_expected_reply<T>`:

```
auto transform = [](auto f) {
    return view::transform(lift_with_expected_reply(f));
};

auto filter = [](auto f) {
    return view::filter(apply_with_expected_reply(f));
};
```

Также нужно определить преобразование `sink`, потому что у диапазонов его нет. Преобразование `sink` должно вызвать заданную функцию для каждого значения из исходного диапазона. Для этого можно использовать `view::transform`, но с небольшим изменением. Функция, которую мы передаем в `sink`, возвращает `void`, поэтому ее нельзя передать напрямую непосредственно в `view::transform`, потому что это приведет к созданию диапазона из `void`. Мы должны завернуть функцию преобразования в функцию, которая возвращает фактическое значение:

```
auto sink = [](auto f) {
    return view::transform([](auto&& ws) {
        f(ws);
        return ws.expected_reply;
    });
};
```

Вот и все! Теперь можно создать вектор с экземплярами `with_expected_reply<std::string>` и передать его в конвейер. По мере обработки каждого элемента в коллекции будет проверяться правильность ответа. Полную реализацию этого примера вы найдете в прилагаемом примере `bookmark-service-testing`.

Важно отметить, что этот тест проверяет только логику основной программы. Он не освобождает от необходимости писать тесты для компонента службы и отдельных преобразований, таких как `filter` и `transform`. Обычно тесты для таких небольших компонентов легко написать, и большинство ошибок возникает не в результате реализации компонента, а в результате взаимодействия различных компонентов. И тестирование именно этих взаимодействий мы сейчас упростили.

**СОВЕТ** За дополнительной информацией по теме, рассматривавшейся в этой главе, обращайтесь по адресу: <https://forums.manning.com/posts/list/43782.page>.

## Итоги

- Всякая чистая функция является хорошим кандидатом для модульного тестирования. Мы знаем, что она используется для вычисления результата и не меняет внешнего состояния – ее единственный эффект заключается в том, что она возвращает результат.
- QuickCheck для Haskell – одна из самых известных библиотек, которая проверяет свойства случайно сгенерированного набора данных. По ее образу и подобию созданы аналогичные проекты многих других языков программирования, включая C++.
- Изменяя функцию `gandom`, можно управлять созданием тестовых случаев. Например, при генерации случайных строк можно создавать более короткие строки, используя функцию `gandom` с нормальным распределением и средним значением, равным нулю.
- Нечеткое тестирование (fuzzing) – еще один метод тестирования, использующий случайные данные. Его идея заключается в проверке правильности работы программного обеспечения при вводе ошибочных (случайных) данных. Это очень полезно для программ, которые принимают неструктурированный ввод.
- Запоминание начального числа, использовавшегося для инициализации генератора случайных чисел, позволит повторить тесты, потерпевшие неудачу.
- Правильно спроектированные монадические системы должны продолжать правильно работать, если монаду продолжения или реактивные потоки заменить обычными значениями и обычными коллекциями данных. Это позволяет включать и выключать параллелизм и асинхронное выполнение на лету. Данный переключатель можно использовать во время тестирования.



---

# Предметный указатель

---

## Символы

---

++ (оператор инкремента), 195  
== (оператор) для итераторов  
ограниченных диапазонов, 202  
= (оператор присваивания), копирование  
при записи, 217  
\* (оператор разыменования), 196  
> (больше), 109  
|= (комбинированный оператор  
присваивания с конвейером), 200  
| (оператор конвейера)  
для диапазонов, 194  
| (операция конвейера), 29

## A

---

accumulate (функция), 51, 73  
АСМ (Ассоциация вычислительной  
техники), 132  
any класс (std::), 235  
apply (std::), метафункция, 300  
Association for Computing Machinery, 132  
AST (абстрактное синтаксическое  
дерево), 303

## B

---

back\_inserter (итератор), 63  
bind (функция высшего порядка), 114  
Boost, библиотека, 100  
Boost, библиотеки  
    Boost.Asio, 314  
    Boost.Range, 193  
    Boost.Phoenix, 100

## C

---

C++ Actor Framework, 312  
сборка мусора, 215  
call\_once, 171  
call\_on\_object, 78  
company\_t (класс), 89  
const (ключевое слово), 156  
constexpr (ключевое слово), 156  
contained\_type\_t, метафункция, 286, 290  
copy\_if (функция), 63  
count (функция), 27  
count\_if (функция), 83, 90  
count\_lines (функция), 56  
count\_lines\_in\_files (функция), 26  
count\_occurrences (функция), 134  
current\_exception, функция (std::), 272

## D

---

decay\_t (std::), метафункция, 297  
decltype (ключевое слово), 78  
decltype, спецификатор, 286  
declval (std::), метафункция, 287  
do\_accept, функция, 316

## E

---

enable\_shared\_from\_this (std::), 316  
equal\_to оператор (std::), 207  
error (функция-член), 95  
error\_test\_t (класс), 98  
exception\_ptr, тип (std::), 240  
expected<T, E>, 240  
expected<T, E> как монада, 270

**F**

false\_type (std::), метазначение, 290  
 fibmemo, 181  
 find\_if, алгоритм (std::), 195  
 find\_if (функция), 59  
 fmap (функция), 28  
 for\_each, диапазоны (view::), 267  
 for\_each (функция), 92  
 forward (функция), 78  
 forward, функция (std::), 321  
 function (шаблон класса), 103  
 future, тип, как монада, 277  
   реализация, 279

**G**

generate, диапазоны (view::), 344  
 get\_if, функция (std::), 240  
 get\_if (std::), 235  
 get (std::), 235  
 group\_by, диапазоны (view::), 207

**I**

ifstream (класс), 26  
 ints, диапазоны (view::), 204  
 invoke (std::), метафункция, 300  
 is\_invocable\_r (std::), метафункция, 306  
 is\_invocable\_v (std::), метафункция, 298  
 is\_same (std::), метафункция, 290  
 istream\_range, шаблонный класс, 205

**J**

join, функция для потоков данных, 325  
 JSON, формат, 314

**L**

lazy\_string\_concat\_helper, 185  
 lazy\_val, 188  
 line\_count, функция, 342

**M**

Mach7, библиотека, 251  
 make\_memoized, 190  
 max\_element (функция), 147  
 mbind, функция (монадическое  
 связывание), 262

move (функция), 60, 93  
 moving\_accumulate (функция), 73  
 mtry, функция (для монад), 328  
 mutex (класс), 161

**N**

names\_for (функция), 67  
 new оператор, размещающая версия, 242  
 NRVO, 164

**O**

older\_than (класс), 82  
 once\_flag, 171  
 optional (std::), 237  
 overloaded, шаблон, 250

**P**

partition, 173  
 partition (функция), 60  
 print\_name (функция), 158  
 print\_person (функция), 118  
 process\_message, функция, 314  
 propagate\_const (функция), 165

**Q**

query (функция), 128  
 quicksort, 172, 173

**R**

RANGES\_FOR, макрос, 207  
 read\_text, функция, 199  
 reduce (функция), 52  
 ref (функция), 105  
 remove\_cv\_t (std::), метафункция, 288  
 remove\_if, диапазоны (view::), 205  
 remove\_reference\_t (std::),  
 метафункция, 288  
 rethrow\_exception (std::), 272

**S**

set\_message\_handler, функция, 319  
 SFINAE (невозможность подстановки  
 не является ошибкой), 294  
 shared\_ptr (std::), 214  
 SObjectizer, библиотека, 312

stable\_partition (функция), 60  
static\_assert, 289  
STL (Standard Template Library),  
алгоритмы find\_if (std::), 195  
string\_only\_alnum, функция, 206  
string\_to\_lower, функция, 206

## T

take, диапазоны (view::), 204  
take, функция, диапазоны (view::), 204  
Tennis kata, упражнение, 245  
tie (std::), 228  
to\_vector, диапазоны, 207  
transaction, тип, 304  
transform (функция), 63  
true\_type (std::), метазначение, 290  
tuple (std::), 228

## U

unique\_ptr (std::), 214

## V

variant (std::), 232  
visit (std::), 250

## W

with\_client, класс, 332

## Y

yield\_if, диапазоны, 267

## Z

zip, диапазоны (view::), 204  
zip, функция, диапазоны (view::), 204

## A

Абстрактное синтаксическое дерево  
(AST), 303  
Автоматическое определение  
аргументов типов шаблонных  
классов, 322  
Актеры  
    обзор, 310  
    приемники сообщений, 320

    простой источник сообщений, 314  
    с изменяемым состоянием, 335  
Алгебраические типы, необязательные  
значения как монады, 268  
Алгебраические типы данных, 227  
    моделирование предметной  
    области, 245  
    необязательные значения  
    как диапазоны, 259  
    как функторы, 256  
    optional (std::), 237  
    сопоставление с образцом, 249  
    типы-суммы  
        expected<T, E>, 240  
Амдала закон, 155  
Арифметические операции,  
объекты-обертки, 99  
Асинхронные потоки, 318  
Ассоциативность, 57  
Ассоциативный массив, 181  
Ассоциация вычислительной  
техники, 132

## Б

Бесконечные диапазоны, 203

## В

Внутренняя константность const  
(ключевое слово), 159  
Время выполнения замены ошибками  
времени компиляции, 340  
Время компиляции  
    манипулирование типами во, 285  
    проверка правильности определения  
    типов, 288  
    проверка свойств типов во, 294  
    сопоставление с образцом, 290  
Вызов метафункций, 286  
Вызываемые объекты, 298, 299  
Вычисление частоты слов, 204

## Д

Декларативное программирование, 24  
Диапазоны, 191  
    бесконечные, 203  
    вычисление частоты слов, 204  
    изменение значений, 199

action::, 200  
 обзор, 193  
 ограниченные, 201  
 ints (view::), 204  
 istream\_range, шаблонный класс, 205  
 take (view::), 204  
 to\_vector, 207  
 yield\_if, 267  
 zip (view::), 204

Динамический полиморфизм, 234  
 Динамическое программирование, 178

### З

Замыкание, 91  
 Замыкание лямбда-выражения, 88  
 Значения
 

- изменение с помощью диапазонов, 199
- необязательные
  - как диапазоны, 259
  - как монады, 268
  - как функторы, 256
  - optional (std::), 237
- создание потоков, 325

### И

Идиома копирования и замены, 243  
 Изменение значений, 199  
 Изменяемое состояние, 142  
 Изменяемое состояние, акторы, 335  
 Императивное программирование, 24  
 Инкремент, оператор, 195

### К

Карринг, 124
 

- каррирование функций, 296

 Комбинирование преобразований диапазонов, 199  
 Конкатенация, 184  
 Конкурентное выполнение, 275
 

- future как монада, 277
- future, реализация, 279

 Копирование при записи, 217  
 Копирования и замены (copy-and-swap) идиома, 243, 305

### Л

Левенштейн, 178  
 Ленивая сортировка, 172

Ленивое вычисление, 167
 

- с диапазоном значений, 197

 Логическая константность const (ключевое слово), 159  
 Логические операции, 99  
 объекты-обертки, 99  
 Лямбда-выражение, 87, 249

### М

Массив ассоциативный, 181  
 Матрица, 167  
 Мемоизация, 168, 177, 182  
 Метаинформация о типах, 293  
 Метапрограммирование, 284
 

- каррирование функций, 296
- манипулирование типами во время компиляции, 285
- метаинформация о типах, 293
- метафункции, 285
- предметно-ориентированные языки, 302
- проверка свойств типов, 294
- сопоставление с образцом, 290

 Метафункции, 285
 

- вызов, 286
- манипулирование типами во время компиляции, 285

 Метафункции и значения
 

- apply (std::), 300
- decay\_t (std::), 297
- declval (std::), 287
- false\_type (std::), 290
- invoke (std::), 300
- is\_invocable\_r (std::), 306
- is\_invocable\_v (std::), 298
- is\_same (std::), 290
- remove\_cv\_t (std::), 288
- remove\_reference\_t (std::), 288
- true\_type (std::), 290

 Моделирование предметной области, 245  
 Модульное тестирование, 341  
 Монады, 254
 

- генераторы диапазонов и монад, 265
- и исключения, 271
- и конкурентное выполнение, 275
- и продолжения, 275
- и функторы, 259
- обработка ошибок, 268
- обработка состояния, 273

примеры, 263  
 реактивные потоки, 318  
   объединение, 326  
   приемник, 320  
   создание потока заданных значений, 325  
 тестирование параллельных систем, 348  
 expected<T, E>, 270  
 optional<T> (std::), 268

## Н

Невозможность подстановки не является ошибкой (SFINAE), 294  
 Неизменяемое состояние, 149  
 Неизменяемые векторы, 216  
 Неизменяемые связанные списки, 210  
   добавление и удаление элемента  
     в конце списка, 212  
     в начале списка, 211  
     в середине списка, 213  
   управление памятью, 214  
 Необязательные значения  
   как монады, 268  
   как функторы, 256, 259  
   optional (std::), 237  
 Нотация ромбическая, 100

## О

Обобщенное программирование, 28  
 Обработка ошибок  
   в реактивных потоках, 328  
   и монады, 271  
   и типы-суммы, 240  
   expected<T, E>, 270  
   std::optional<T>, 268  
 Обработка состояния с помощью монад, 273  
 Объединения, 232  
 Объектно-ориентированное программирование, 25  
 Объект функциональный, 76  
 Объявление типа функции ->, 76  
 Ограниченные диапазоны, 201  
 Ограничители, 201  
 Односвязные списки, 213  
 Оператор вызова в классе  
 transaction, 305

Оператор вызова для каррированной функции, 298  
 Оператор разыменования (\*), 196  
 Операции вызова, перегрузка, 81  
 Операция вызова для частичного применения, 111  
 Оптимизация программ, 172

## П

Параллельность стандартных алгоритмов, 52<sup>®</sup>  
 Параллельные системы  
   на основе монад, тестирование, 348  
   функциональный дизайн  
     акторы с изменяемым состоянием, 335  
     возврат ответа клиенту, 331  
     модель акторов, 310  
   обработка ошибок в реактивных потоках, 328  
   преобразование реактивных потоков, 323  
   приемники сообщений, 320  
   простой источник сообщений, 314  
   реактивные потоки как монады, 318  
   фильтрация реактивных потоков, 327  
 Пары, типы-произведения, 227  
 Перегруженная операция вызова, 81  
 Перегрузка функции, 163  
 Перемещающая операция присваивания, 160  
 Перестановка аргументов, бинарная функция, 116  
 Перечисления, 228  
 Побитовые операции,  
 объекты-обертки, 99  
 Подъем функции, 37  
 Предикат, 48, 98  
 Предметно-ориентированные языки, 302  
   строительные блоки, 302  
 Представления данных только для чтения, 194  
   ленивые вычисления, 197  
   filter (view::), 194  
   generate (view::), 344  
   group\_by (view::), 207  
   remove\_if (view::), 205  
   reverse (view::), 207



transform (view::), 196  
 unique (view::), 200  
 Преобразование бинарной функции  
 в унарную, 110  
 Преобразование диапазонов,  
 view::transform, 196  
 Преобразование реактивных потоков, 323  
 Префиксные деревья, 217  
 добавление элементов в конец, 220  
 изменение элементов в конец, 223  
 удаление элементов в конец, 223  
 эффективность, 224  
 Применение частичное, 108  
 Проблемы, const (ключевое слово), 163  
 Проверка правильности определения  
 типов, 288  
 Проектирование сверху вниз, 247

## P

Разделение по предикату, коллекции, 60  
 Размещающая версия new, 242  
 Расстояние Левенштейна, 178  
 Реактивные потоки, 318  
 как монады, 318  
 возврат ответа клиенту, 331  
 обработка ошибок, 328  
 объединение, 326  
 создание потока заданных  
 значений, 325  
 фильтрация, 327  
 преобразование, 323, 325  
 Рекурсия, 68

## C

Свертка, 53  
 Сверху вниз, проектирование, 247  
 Семантика перемещения, const  
 (ключевое слово), 164  
 Семафор, 154  
 Сообщения  
 приемник, 320  
 простой источник, 314  
 Сопоставление с образцом, 226, 249  
 во время компиляции, 290  
 Mach7, библиотека, 251  
 Сортировка (диапазоны, action::sort), 200  
 Состояние  
 обработка с помощью монад, 273

реализация в виде типа-суммы, 235  
 Спецификация техническая, 44  
 Списки  
 добавление и удаление элемента  
 в конце списка, 212  
 в начале списка, 211  
 в середине списка, 213  
 неизменяемые связанные, 211  
 управление памятью, 214  
 Сравнения (операции,  
 объекты-обертки), 99  
 Сравнительное тестирование, 347  
 Среднее арифметическое, вычисление, 51  
 Статический полиморфизм, 43  
 Стереть и удалить (идиома), 62  
 Структуры данных, 209

## T

Теория категорий, 255  
 Тернарная операция, 150  
 Тестирование, 338  
 автоматическое генерирование  
 тестовых случаев, 343  
 модульное, 341  
 на основе свойств, 345  
 параллельных систем на основе  
 монад, 348  
 сравнительное, 347  
 чистых функций, 341  
 Типы  
 манипулирование во время  
 компиляции, 285  
 метаинформация о типах, 293  
 проверка правильности  
 определения, 288  
 проверка свойств, 294  
 Типы и утилиты  
 current\_exception (std::), 272  
 enable\_shared\_from\_this (std::), 316  
 exception\_ptr тип (std::), 240  
 forward (std::), 321  
 get\_if (std::), 235  
 get (std::), 235  
 optional (std::), 237, 256  
 rethrow\_exception (std::), 272  
 shared\_ptr (std::), 214  
 tie (std::), 228  
 tuple (std::), 228  
 unique\_ptr (std::), 214



variant (std::), 232

visit (std::), 250

Типы-произведения, 227

Типы-суммы, 227

определение через наследование, 229

expected<T, E>, 240

## у

Указатель на функцию, 80

Утиная типизация, 75

## ф

Фибоначчи числа, 176, 181

Фильтрация

диапазонов, view::filter, 194

реактивных потоков, 327

Функторы, 255

и монады, 259

обработка необязательных

значений, 256

Функции, каррирование, 296

Функциональный дизайн параллельных систем

акторы с изменяемым состоянием, 335

возврат ответа клиенту, 331

модель акторов, 310

обработка ошибок в реактивных

потоках, 328

преобразование реактивных

потоков, 323

приемники сообщений, 320

простой источник сообщений, 314

реактивные потоки как монады, 318

фильтрация реактивных потоков, 327

Функция сору\_if, фильтрация, 63

Функция remove\_if, фильтрация, 109

## х



Хвостовая рекурсия, 71

## ц

Циклы for для диапазонов, 202

## ч

Числа Фибоначчи, 176, 181

Чистые функции, тестирование, 341

## ш

Шаблоны и метапрограммирование, 284

## э



Элементы

добавление в конец префиксных

деревьев, 220

добавление и удаление

в конце списка, 212

в начале списка, 211

в середине списка, 213

изменение в префиксных деревьях, 223

удаление в конце префиксных

деревьев, 223

---

Книги издательства «ДМК ПРЕСС»  
можно купить оптом и в розницу в книготорговой компании «Галактика»  
(представляет интересы издательств «ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;  
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью),  
по которому должны быть высланы книги;  
фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.a-planet.ru**.



Иван Чукич

## Функциональное программирование на языке C++

Главный редактор *Мовчан Д. А.*  
dmkpress@gmail.com  
Перевод *Винник В. Ю., Киселев А. Н.*  
Корректор *Синяева Г. И.*  
Верстка *Чаннова А. А.*  
Дизайн обложки *Мовчан А. Г.*

Формат 70 × 100 1/16.  
Гарнитура PT Serif. Печать офсетная.  
Усл. печ. л. 29,25. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**