

# Паттерны проектирования *Rust*

Пишем  
идиоматический код  
как профи

Бренден Мэтьюз

 MANNING



# *Idiomatic Rust*

CODE LIKE A RUSTACEAN

BRENDEN MATTHEWS



MANNING  
SHELTER ISLAND

# *Паттерны проектирования Rust*

Пишем идиоматический код как профи

Бренден Мэтьюз



Санкт-Петербург • Москва • Минск

2026

**Бренден Мэтьюз**  
**Паттерны проектирования Rust**

Серия «Библиотека программиста»

Перевел с английского Д. Брайт  
Научный редактор Д. Бардин

ББК 32.973.2-018.1  
УДК 004.43

**Мэтьюз Бренден**

M97 Паттерны проектирования Rust. — СПб.: Питер, 2026. — 256 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-4389-4

Как только вы освоитесь с синтаксисом Rust, с его уникальным и мощным компилятором и начнете использовать его в реальных проектах, перед вами откроется совершенно новое измерение. Как правильно применять стандартные паттерны проектирования в приложениях на Rust? Как и когда следует использовать IntoIterator? Почему Rust-разработчики любят тип PhantomData? Ответы на эти и многие другие вопросы вы найдете в книге.

Познакомьтесь с паттернами программирования и проектирования, необходимыми для использования уникальной архитектуры языка Rust. Понятные объяснения и примеры кода помогут вам освоить метaprogramмирование, позволят создавать собственные библиотеки, программируя удобные интерфейсы и делать многое другое. Попутно, вникая в особенности языка, вы будете учиться писать эффективный и идиоматический код на Rust, который легко поддерживать и развивать.

**16+** (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1633437463 англ.

Authorized translation of the English edition © 2024 Manning Publications.

This translation is published and sold by permission of Manning Publications, the owner of all rights to publish and sell the same.

ISBN 978-5-4461-4389-4

© Перевод на русский язык ООО «Прогресс книга», 2025

© Издание на русском языке, оформление ООО «Прогресс книга», 2025

© Серия «Библиотека программиста», 2025

Права на издание получены по соглашению с Manning Publications. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 12.2025. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 12.11.25. Формат 70×100/16. Бумага офсетная. Усл. п. л. 20,640. Тираж 500. Заказ 0000.

# *Краткое содержание*

---

## **Часть I Структурные элементы Rust**

Глава 1. Введение в паттерны Rust.....	22
Глава 2. Базовые структурные компоненты Rust.....	30
Глава 3. Поток кода.....	55

## **Часть II Основные паттерны**

Глава 4. Паттерны в Rust.....	86
Глава 5. Паттерны проектирования: расширяем функционал .....	114
Глава 6. Проектирование библиотеки .....	153

## **Часть III Более сложные паттерны**

Глава 7. Использование трейтов, дженериков и структур для особых задач.....	176
Глава 8. Конечные автоматы, корутины, макросы и прелюдии .....	195

## **Часть IV Предотвращение проблем и создание надежного ПО**

Глава 9. Неизменяемость .....	214
Глава 10. Антипаттерны .....	231

Приложение. Установка Rust .....	253
----------------------------------	-----

# Оглавление

---

Предисловие .....	12
Благодарности.....	13
О книге.....	14
В чем особенность книги .....	14
Целевая аудитория.....	15
Структура издания.....	15
Как читать книгу.....	16
О коде .....	16
Об авторе.....	18
Иллюстрация на обложке .....	19
От издательства .....	20
О научном редакторе русскоязычного издания .....	20

## Часть I Структурные элементы Rust

<b>Глава 1.</b> Введение в паттерны Rust.....	22
1.1. О чем эта книга.....	23
1.2. Что такое паттерны проектирования.....	24
1.3. Чем эта книга отличается от других.....	28
1.4. Инструменты, необходимые для работы с Rust.....	28
Резюме .....	29
<b>Глава 2.</b> Базовые структурные компоненты Rust.....	30
2.1. Дженирики.....	31
2.1.1. Система типов, отвечающая принципу полноты по Тьюрингу .....	31
2.1.2. Почему дженерики.....	32
2.1.3. Основы дженериков .....	33
2.1.4. Знакомимся с Option в Rust .....	35
2.1.5. Структуры маркеров и фантомные типы.....	36
2.1.6. Ограничения трейтов обобщенных параметров.....	39

2.2. Трейты .....	40
2.2.1. Почему трейты не относятся к объектно-ориентированному программированию.....	40
2.2.2. Что содержится в трейте.....	41
2.2.3. Разбираемся в трейтах на примере объектно-ориентированного кода .....	42
2.2.4. Совмещение дженериков и трейтов.....	45
2.2.5. Автоматический вывод трейтов .....	49
2.2.6. Трейт-объекты.....	50
Резюме .....	54
<b>Глава 3. Поток кода .....</b>	<b>55</b>
3.1. Что такое сопоставление с шаблоном.....	55
3.1.1. Основы сопоставления с шаблоном .....	56
3.1.2. Чистые сопоставления с помощью оператора ? .....	61
3.2. Функциональный Rust .....	64
3.2.1. Основы функционального программирования в Rust .....	65
3.2.2. Захват переменных замыкания .....	66
3.2.3. Знакомство с итераторами .....	68
3.2.4. Получение итератора с помощью with iter(), into_iter() и iter_mut().....	72
3.2.5. Возможности итераторов .....	80
Резюме .....	84

## **Часть II** **Основные паттерны**

<b>Глава 4. Паттерны в Rust.....</b>	<b>86</b>
4.1. Получение ресурса есть инициализация .....	87
4.1.1. RAII в С и С++ .....	87
4.1.2. RAII в Rust.....	91
4.2. Передача аргументов по значению и по ссылке.....	96
4.2.1. Передача по значению .....	96
4.2.2. Передача по ссылке.....	97
4.2.3. Когда передавать по значению, а когда — по ссылке .....	99
4.3. Конструкторы .....	100
4.4. Видимость членов объекта и доступ к ним.....	102
4.5. Обработка ошибок.....	104
4.6. Глобальное состояние .....	107
4.6.1. Крейт lazy-static.rs.....	110
4.6.2. Крейт once_cell.....	111

4.6.3. Крейт static_init .....	112
4.6.4. std::cell::OnceCell.....	112
Резюме .....	113
 <b>Глава 5.</b> Паттерны проектирования: расширяем функционал .....	114
5.1. Метапрограммирование с помощью макросов.....	115
5.1.1. Базовый декларативный макрос в Rust.....	116
5.1.2. Когда использовать макросы.....	118
5.1.3. Использование макросов для написания мини-DSL .....	122
5.1.4. Использование макросов для соблюдения принципа «Не повторяйся».....	124
5.2. Необязательные аргументы функций .....	127
5.2.1. Необязательные аргументы в Python .....	127
5.2.2. Необязательные аргументы в C++ .....	128
5.2.3. Необязательные аргументы в Rust, вернее, их отсутствие .....	128
5.2.4. Эмуляция необязательных аргументов с помощью трейтов .....	128
5.3. Паттерн «Строитель» .....	131
5.3.1. Реализация паттерна «Строитель» .....	132
5.3.2. Расширение строителя трейтами.....	134
5.3.3. Расширение строителя макросами.....	135
5.4. Паттерн «Гибкий интерфейс» .....	138
5.4.1. Гибкий строитель .....	138
5.4.2. Тестирование гибкого строителя.....	141
5.5. Паттерн «Наблюдатель» .....	141
5.5.1. А почему не обратные вызовы? .....	141
5.5.2. Реализация наблюдателя.....	142
5.6. Паттерн «Команда» .....	145
5.6.1. Определение паттерна «Команда» .....	145
5.6.2. Реализация паттерна «Команда» .....	146
5.7. Паттерн «Новый тип» .....	149
Резюме .....	152
 <b>Глава 6.</b> Проектирование библиотеки .....	153
6.1. Обдумайте дизайн библиотеки .....	154
6.2. Делайте что-то одно и делайте это качественно и правильно.....	155
6.3. Избегайте излишнего абстрагирования.....	155
6.4. Страйтесь использовать простые типы.....	156
6.5. Пользуйтесь инструментами .....	157
6.6. Хорошие художники копируют; великие — воруют (из стандартной библиотеки).....	157
6.7. Документируйте каждый нюанс и приводите примеры .....	158

6.8. Не ломайте код пользователя.....	158
6.9. Помните о состоянии.....	159
6.10. Помните об эстетичности .....	160
6.11. Оценка эргономичности библиотеки Rust .....	160
6.11.1. Вернемся к связанным спискам.....	161
6.11.2. Улучшение дизайна API с помощью rustdoc .....	162
6.11.3. Улучшение связанного списка с помощью дополнительных тестов.....	169
6.11.4. Упрощение отладки библиотеки .....	171
Резюме .....	174

### **Часть III Более сложные паттерны**

<b>Глава 7.</b> Использование трейтов, дженериков и структур для особых задач .....	176
7.1. Постоянные дженерики.....	177
7.2. Реализация трейтов для типов из других крейтов.....	179
7.2.1. Структуры-обертки .....	179
7.2.2. Использование трейта Deref для развертывания обернутой структуры .....	180
7.3. Трейты-расширения .....	181
7.4. Blanket-трейты.....	183
7.5. Трейты-маркеры.....	185
7.6. Использование структур в качестве тегов .....	187
7.7. Объекты-ссылки .....	189
Резюме .....	194

<b>Глава 8.</b> Конечные автоматы, корутины, макросы и прелюдии .....	195
8.1. Конечный автомат на основе трейтов.....	196
8.2. Корутины.....	200
8.3. Процедурные макросы .....	205
8.4. Прелюдии .....	209
Резюме .....	212

### **Часть IV Предотвращение проблем и создание надежного ПО**

<b>Глава 9.</b> Неизменяемость .....	214
9.1. Преимущества неизменяемости .....	215
9.2. Почему неизменяемость не панацея .....	217
9.3. Как воспринимать неизменяемые данные .....	218
9.4. Принцип неизменяемости в Rust .....	219

## **10**    Оглавление

9.5. Анализ основ неизменяемости в Rust .....	220
9.6. Обеспечение неизменяемости с помощью трейтов .....	223
9.7. Использование типа Cow для обеспечения неизменяемости .....	224
9.8. Получение неизменяемых структур данных с помощью крейтов.....	227
9.8.1. Использование крейта im .....	228
9.8.2. Использование крейта rpds.....	229
Резюме.....	230
<b>Глава 10. Антипаттерны .....</b>	<b>231</b>
10.1. Что такое антипаттерн.....	232
10.2. Использование unsafe .....	233
10.2.1. Что делает unsafe .....	234
10.2.2. Где можно использовать unsafe .....	235
10.2.3. Когда следует использовать unsafe .....	238
10.2.4. Нужно ли беспокоиться насчет unsafe.....	238
10.3. Использование метода unwrap().....	239
10.4. Пренебрежение Vec .....	239
10.5. Излишнее клонирование .....	243
10.6. Использование трейта Deref для эмуляции полиморфизма .....	244
10.7. Глобальные данные и синглтоны.....	248
10.8. Слишком много умных указателей.....	249
10.9. Что дальше?.....	250
Резюме.....	251
<b>Приложение. Установка Rust .....</b>	<b>253</b>
П.1. Установка инструментов для работы с книгой.....	253
П.1.1. Установка для macOS с помощью Homebrew.....	253
П.1.2. Установка для систем Linux.....	253
П.1.3. Установка для Windows .....	254
П.2. Управление rustc и прочими компонентами Rust через rustup .....	254
П.2.1. Установка rustc и других компонентов.....	254
П.2.2. Переключение штатного набора инструментов с помощью rustup.....	255
П.2.3. Обновление компонентов Rust .....	255

*Посвящается моим лучшим друзьям, Дожу и Уолтеру,  
без которых я бы не смог написать эту книгу. Благодарю  
их за неиссякаемый оптимизм и поддержку.*

# *Предисловие*

---

Когда я только начинал изучать программирование в далеком 1990 году, у меня не было доступа ко всевозможным ресурсам, которые легко найти сегодня, — интернет только начинал развиваться. К сожалению, и в библиотеке моей школы не было книг по информатике и программированию. Так что мне оставалось идти путем проб и ошибок.

Доступ к таким образовательным ресурсам, как книги, у меня появился лишь спустя многие годы. К тому моменту я уже неплохо освоил программирование, просто читая исходный код, экспериментируя и задавая вопросы в ретранслируемом интернет-чате (Internet Relay Chat, IRC) и на форумах. Моими онлайн-учителями преимущественно были незнакомцы, которым я очень признателен за помощь.

К счастью, сегодня изучать программирование стало гораздо проще, поскольку существует огромное количество отличных ресурсов. При написании этой книги я старался собрать такой материал, который бы пригодился мне самому в те годы, когда я только учился программировать. Надеюсь, она поможет вам стать более эффективным программистом и достичь желаемых целей, как помогли мне в свое время добрые незнакомцы в интернете.

# Благодарности

---

Хочу поблагодарить своих близких друзей Джавви Шейкха (Javeed Shaikh) и Бена Лина (Ben Lin) за их обратную связь по первым наброскам книги и помочь в проработке различных идей. Благодарю также Элеанор Сей (Eleanor Seay) за ее вдохновение и поддержку. Отдельное спасибо Аве (Ava) и Тобиасу (Tobias) за терпение и понимание.

Спасибо издательству Manning Publications и его сотрудникам за предоставленную поддержку и помощь. Огромная благодарность редактору Карен Миллер (Karen Miller), техническому корректору Джерри Кучу (Jerry Kuch) и всем членам выпускающей команды.

Отдельную признательность выражаю научному редактору Аллену Кунио (Alain Couniot), профессиональному, искренне интересующемуся инновациями и языками программирования — в частности функциональными. В область его интересов входят встраиваемые системы и распределенные корпоративные приложения, а также облачные, высокопроизводительные и квантовые вычисления. На данный момент Rust — любимый язык Алена.

Благодарю всех рецензентов: Александро Кампейса (Alessandro Campeis), Энди Стайнера (Andy Stainer), Чарльза Чана (Charles Chan), Дэвида Пакуда (David Paccoud), Дэвида Уайта (David White), Эдера Андреса Авильо Ниньо (Eder Andrés Ávila Niño), Филипа Мечанта (Filip Merchant), Флориана Брауна (Florian Braun), Герта Ван Летема (Geert Van Laethem), Джорджа Рейли (George Reilly), Джузеппе Каталано (Giuseppe Catalano), Гийома Шмидта (Guillaume Schmid), Джона Гатри (John Guthrie), Йона Кристиансена (Jon Christiansen), Лева Вейде (Lev Veyde), Мартина Новака (Martin Nowack), Скотта Линга (Scott Ling), Серхио Бритоса (Sergio Britos), Сын-джина Кима (Seung-jin Kim), Стефана Вершера (Stefaan Verschueren), Стивена Уэйкли (Stephen Wakely), Томаса Локни (Thomas Lockney), Фолькера Рота (Volker Roth), Уолтера Александра Мата Лопеса (Walter Alexander Mata López), Уильяма Уиллера (William Wheeler) и Ива Дорфсмана (Yves Dorfsman). Ваши рекомендации помогли улучшить эту книгу.

Паттерны, представленные в книге, были выработаны на основе усердных трудов других людей, которых я упоминаю везде, где это уместно. Книгу я писал, стоя на плечах гигантов, чаще всего случайных людей из интернета, которые привили мне страсть к созданию качественного программного обеспечения. Я восхищен и польщен тем, что сегодня так много гениальных людей пишут замечательные вещи и делятся ими с миром.

# О книге

---

В книге представлены коллекция паттернов проектирования и лучшие практики программирования на языке Rust. Она писалась для широкой аудитории — начиная с джуниоров и заканчивая продвинутыми разработчиками. Некоторые ее части опираются на теорию, но в основном она посвящена практической работе. Моя задача — помочь вам стать более сильным программистом на Rust, умеющим писать идиоматический код и эффективно использовать возможности этого языка.

Эта книга стала продолжением другой моей книги, *Code Like a Pro in Rust*<sup>1</sup> (Manning Publications, 2024), которая является более обобщенным руководством по Rust и прекрасно подойдет для новичков.

Кстати, рабочим названием этой книги было *Rust Design Patterns*, а вдохновением к ее написанию послужила классическая работа «Банды четырех» *Design Patterns: Elements of Reusable Object-Oriented Software*<sup>2</sup> (Addison-Wesley Professional, 1994). И хотя я неставил своей задачей напрямую перевести описанные там паттерны на Rust, здесь вы найдете набор специфичных для этого языка паттернов и практик, основанных на классических паттернах проектирования.

## В чем особенность книги

Эта книга не является полноценным руководством по Rust или справочником по синтаксису и функциям стандартной библиотеки. Моя задача здесь — помочь вам повысить мастерство написания кода на этом языке, лучше понять его и более эффективно использовать.

Значительная часть книги посвящена паттернам и практикам, которые могут не быть описаны в официальной документации Rust и иных ресурсах. Тем не менее вы встретите эти паттерны во многих кодовых базах. И хотя они не всегда являются уникальными конкретно для Rust, здесь они приводятся в контексте именно этого языка.

---

<sup>1</sup> Мэттьюз Б. Rust. Профессиональное программирование. — Астана, 2025.

<sup>2</sup> Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Паттерны объектно-ориентированного программирования. — СПб.: Питер, 2025.

## **Целевая аудитория**

Книга предназначена для программистов с любым опытом работы, но новичкам некоторые разделы могут показаться сложными. Это не вводный учебник по Rust, и предполагается, что вы уже в какой-то степени знакомы с этим языком.

В добавок будет весьма кстати, если вы читали классическую работу «Паттерны объектно-ориентированного программирования», поскольку описанные там паттерны проектирования упоминаются и в этой книге.

## **Структура издания**

Книга разделена на четыре части, каждая из которых состоит из глав, описывающих конкретную тему программирования на Rust.

В части I представлен обзор основных функций и структурных элементов Rust.

- В главе 1 разбирается содержание книги и дается общее описание паттернов проектирования.
- В главе 2 описываются базовые элементы структуры Rust.
- В главе 3 говорится о сопоставлении с шаблоном и функциональном программировании.

Часть II посвящена основным паттернам Rust и проектированию библиотеки.

- В главе 4 описываются базовые паттерны в Rust.
- В главе 5 разбираются конкретные паттерны проектирования.
- В главе 6 представлен процесс проектирования библиотеки.

Часть III знакомит с более сложными паттернами.

- В главе 7 описываются продвинутые приемы и паттерны в Rust.
- В главе 8 продолжается обсуждение тем из главы 7.

Часть IV посвящена тому, как избегать проблем и создавать надежное программное обеспечение.

- В главе 9 разбирается неизменяемость и ее использование в Rust.
- В главе 10 описываются несколько антипаттернов и рассказывается, как избегать их использования.

В приложении описаны нюансы установки Rust и инструментов, необходимых для работы с ним.

## Как читать книгу

Читать ее можно как последовательно, от начала и до конца, так и переключаясь между главами, наиболее интересными для вас. Каждая глава самодостаточна, и их можно изучать в любом порядке. Однако в некоторых главах есть ссылки к принципам или паттернам, описанным в предыдущем тексте. Если у вас мало опыта работы с Rust, то лучше читать книгу последовательно, так как в основе одних паттернов лежат другие.

Я рекомендую читать книгу, параллельно работая на компьютере: пробуйте выполнить фрагменты кода и экспериментируйте с описанными здесь паттернами. Лучший способ научиться программировать — делать это, так что я призываю вас экспериментировать с примерами кода и применять освоенные паттерны и практики в своих проектах. Приведенные в книге фрагменты кода не имеют строгих лицензионных требований, поэтому вы можете использовать их в своей работе.

Как советует Мортимер Дж. Адлер (Mortimer J. Adler) в своей работе *How to Read a Book* (Touchstone, 1974), максимум пользы из книги вы можете получить, перечитав ее несколько раз. Сначала вы можете сфокусироваться на понимании представленных в ней паттернов и практик, а затем на их применении в ваших проектах и экспериментах с фрагментами кода.

## О коде

В книге очень много оригинальных фрагментов кода, и чтобы получить копию всего исходного кода, клонируйте репозиторий книги с GitHub на свой компьютер (<https://github.com/brndnmthws/idiomatic-rust-book>). Приведенные фрагменты нередко обрывочны, поэтому для изучения полноценных листингов вам потребуется обращаться к исходникам.

Код в книге может слегка отличаться от кода в репозитории ввиду форматирования и прочих нюансов, таких как перенос строк, отступы и компиляция (в книге намеренно показаны ошибки).

Примеры исходного кода будут встречаться как в нумерованных листингах, так и среди обычного текста. В обоих случаях он будет отформатирован моноширинным шрифтом. В некоторых случаях части кода также будут выделены **жирным моноширинным шрифтом**, обозначая фрагменты, которые изменились после предыдущих шагов, например после добавления новой функциональности.

Нередко изначальный исходный код переформатировался. В него добавлялись разрывы строк и измененные отступы, чтобы полноценно задействовать пространство страницы. Но иногда даже этих изменений было недостаточно, поэтому листинги содержат маркеры разрыва строки (→). Кроме того, если код листинга описан в тексте, то из него могут быть удалены комментарии. Многие листинги снабжены аннотациями, в которых описаны важные концепции.

Со временем приведенные здесь фрагменты кода устареют, так как и сам язык Rust, и его экосистема развиваются. Тем не менее код в репозитории книги будет обновляться, чтобы отражать все последние изменения. Рекомендуем обращаться к нему для получения наиболее свежих фрагментов кода.

Чтобы клонировать этот код на свой ПК, выполните в Git следующую команду:

```
$ git clone https://github.com;brndnmthws/idiomatic-rust-book
```

Код в репозитории распределен по каталогам, обозначающим главы книги; при этом каждый из них, в свою очередь, разбит по темам в соответствии с разделами. Весь код распространяется по лицензии Массачусетского технологического института (Massachusetts Institute of Technology, MIT). Это довольно свободная лицензия, которая позволяет копировать фрагменты кода и использовать их по своему усмотрению, даже в качестве основы собственной работы.

Весь код доступен для скачивания с сайта Manning по ссылке <https://www.manning.com/books/idiomatic-rust> и с GitHub: <https://github.com;brndnmthws/idiomatic-rust-book>.

## *Об авторе*

---



**Бренден Мэтьюз** (Brenden Matthews) — разработчик программного обеспечения, предприниматель и активный участник проектов с открытым исходным кодом. Начал использовать Rust еще на ранних этапах его развития и, помимо профессионального применения, участвовал в разработке нескольких инструментов и открытых проектов для него. Автор Conky, популярного системного монитора, а также член Apache Software Foundation (опыт работы в области программирования — более 25 лет). Помимо этого, Бренден выступает в роли инструктора на YouTube, а также написал множество статей по Rust и другим языкам программирования. Выступал на нескольких технологических конференциях, таких как Qcon, LinuxCon, ContainerCon, MesosCon и All Things Open, а также на неформальных встречах, посвященных Rust. Более 14 лет является контрибьютором GitHub и опубликовал для Rust множество крейтов. Внес вклад в несколько проектов open source на Rust и на профессиональном уровне разрабатывал приложения промышленных масштабов.

## *Иллюстрация на обложке*

---

Иллюстрация называется «Биржевой брокер» (*L'agent de change*) и взята из книги Анри-Леона Кюрмье, опубликованной в 1841 году. Каждая иллюстрация в этой книге была тщательно прорисована и раскрашена вручную.

В те времена можно было по одной только одежде легко определить, где жил человек и чем занимался. Издательство Manning делает акцент на изобретательности и инициативности в компьютерной сфере, оформляя обложки своих книг в стиле богатого разнообразия региональных культур, существовавших столетия назад. Мы освежаем это разнообразие в памяти, используя картины из различных памятных коллекций.

# *От издательства*

---

Мы выражаем огромную благодарность клубу рецензентов ИТ-литературы ReadIT Club за помощь в работе над русскоязычным изданием книги и их вклад в повышение качества переводной литературы.

Ваши замечания, предложения, вопросы отправляйте по адресу [comp@piter.com](mailto:comp@piter.com) (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства [www.piter.com](http://www.piter.com) вы найдете подробную информацию о наших книгах.

## ***О научном редакторе русскоязычного издания***

**Дмитрий Бардин** — ведущий разработчик, архитектор решений, один из авторов курса «Архитектор ПО» от «Яндекс Практикума». В настоящее время занимается разработкой бэкенда «КиноПоиска» с применением языков Go и Java. В прошлом руководитель службы продуктовой разработки и ресурс-менеджер. Опыт в ИТ — более 16 лет.

## *Часть I*

# *Структурные элементы Rust*

Мы начнем с рассмотрения некоторых основных структурных элементов проектирования, используемых в Rust. Эти базовые элементы позволяют понять сложные паттерны, о которых пойдет речь позднее, и помогут писать более идиоматический код. Одни из этих элементов являются специфичными для Rust, а другие представляют более универсальные концепции программирования, особенно важные в этом языке.

Эти структурные элементы, по сути, составляют словарь книги и относятся к основным особенностям Rust. Можно рассматривать их как атомы в молекуле, которые мы будем комбинировать разными способами, создавая сложные субстанции (или паттерны). А паттерны можно совмещать и корректировать, чтобы создавать бесконечное разнообразие программных систем.

Имея под ногами твердый фундамент, мы можем достигать великих вершин, главное — внимательно подойти к возведению надежной и продуманной структуры. Rust обеспечивает такой фундамент, но в конечном счете именно мы, разработчики, несем ответственность за решения, позволяющие эффективно использовать те или иные доступные компоненты и инструменты.

# 1

## *Введение в паттерны Rust*

### **В этой главе**

- ✓ Что такое паттерны проектирования.
- ✓ Чем эта книга отличается от других.
- ✓ Инструменты, необходимые для работы с Rust.

Кем бы вы ни были: начинающим, продвинутым или профессиональным программистом Rust, чтение этой книги станет отличным способом расширить навыки работы с этим языком. Если вы новичок, то изучение паттернов проектирования — прекрасная возможность повысить ваш уровень мастерства. Некоторые части книги могут показаться вам сложными, так что будьте готовы обращаться к сторонним ресурсам за дополнительной информацией. В книге приводятся различные приемы написания качественного кода на Rust, но мы сосредоточимся на паттернах, идиомах и соглашениях, которые широко используются в этом языке и хорошо известны в его сообществе.

Паттерны проектирования — это мощная абстракция, которую любой программист может использовать для написания качественного кода. Люди прекрасно распознают закономерности, поэтому использование понятных и легко узнаваемых паттернов в коде позволяет решить две задачи.

- Во-первых, паттерны дают возможность понять, является ли конкретная структура кода удачной (использование известных паттернов позволяет избежать написания плохого кода).
- Во-вторых, паттерны помогают другим людям понять наш код.

Читать код зачастую сложнее, чем писать. Когда мы читаем чей-то код, в котором используются определенные, известные нам паттерны, то становится проще понимать происходящее в нем. Если вы научите свой мозг узнавать наиболее типичные паттерны, то сможете упростить процесс оценки качества кода, а значит, и ошибок станет меньше.

Что касается программирования, то, зная, какие паттерны подходят для тех или иных ситуаций, вы сможете получить хороший результат за меньшее время. И это знание ничем не отличается от знания того, какие структуры данных или алгоритмы использовать в других обстоятельствах и к каким компромиссам для этого нужно прийти.

В этой книге вы практически не встретите догм. При описании всех паттернов я постараюсь подробно пояснить причины их использования. Будучи программистом, вы вольны экспериментировать, отклоняясь от представленных паттернов, чтобы создавать собственные структуры. Тем не менее я буду приводить продуманные условия, в целом предпочитая их возможности настройки.

Если использовать аналогию, то я предпоchitaю пойти в ресторан, где шеф-повар предлагает всего пару блюд, заранее утвержденных как лучшие для этого сезона, а не просматривать десятки или сотни позиций меню, пытаясь понять, какое блюдо вкуснее. Лучшие рестораны обычно предлагают рекомендательный выбор (то есть основанный на доверии к вкусу шеф-повара), и я в этой книге постараюсь сделать так же.

Многие из фрагментов кода в ней являются выдержками из листингов, но вы всегда можете найти рабочие примеры кода на GitHub по адресу <https://github.com;brndnmthws/rust-advanced-techniques-book>. Этот код доступен по лицензии Массачусетского технологического института (MIT), которая разрешает его использование, копирование и изменение без каких-либо ограничений. Если у вас есть такая возможность, то я рекомендую в процессе чтения книги работать с полноценными листингами кода (чтобы получить максимальную пользу от прочтения). Все фрагменты есть в репозитории GitHub, где они распределены по каталогам в соответствии с главами. Тем не менее некоторые примеры охватывают несколько разделов или даже глав и названы в зависимости от темы. Отдельные фрагменты кода в книге могут немного отличаться от представленных в репозитории, так как были отредактированы из соображений читабельности, краткости и удобства печати.

## **1.1. О чём эта книга**

В книге мы рассмотрим различные идиомы, паттерны и схемы проектирования. Одни из них будут относиться исключительно к Rust. Другие будут служить иллюстрацией старых концепций, представленных в новом формате в рамках системы уникальных возможностей, синтаксиса и грамматики Rust.

Цель книги — помочь вам понять эти паттерны и научиться с их помощью улучшать программную архитектуру. Освоение и использование паттернов Rust позволит вам писать более эффективный, удобный в сопровождении и масштабируемый код. С каждым новым разделом я буду подробнее объяснять рассматриваемый паттерн и рассказывать, почему он важен и как применять его в реальных сценариях. В добавок мы обсудим компромиссы и важные нюансы, связанные с использованием каждого паттерна.

При всем этом важно понимать, что не нужно слепо использовать предложенные паттерны. Это лишь инструменты, которые вы можете адаптировать и изменять под свои конкретные нужды. Как я уже сказал выше, будучи программистом, вы вольны экспериментировать и отклоняться от описанных паттернов, создавая собственные уникальные структуры. Дочитав книгу до конца, вы будете твердо понимать различные идиомы и паттерны проектирования в Rust, а также получите знания и навыки для их эффективного применения в собственных проектах.

Многие из паттернов, рассматриваемых в классической работе «Банды четырех» «Паттерны объектно-ориентированного проектирования», относятся строго к объектно-ориентированному программированию (ООП) на C++. И разработчики Rust проделали прекрасную работу, сделав эти паттерны неактуальными за счет предоставления более эффективных альтернатив или добавления их в стандартную библиотеку (например, итераторов). И хотя предвестие грядущей смерти ООП сильно преувеличено, используемые в Rust абстракции после их освоения оказываются более понятными интуитивно.

Объектно-ориентированное программирование зачастую ведет к написанию лишнего шаблонного кода и использованию сложных паттернов. Иногда эту сложность ООП мы оправдываем некой необходимостью, однако, по сути, просто увеличиваем когнитивную нагрузку. Но сложные системы склонны чаще давать сбои, причем более критические, в сравнении с простыми. Кроме того, их сложнее понимать.

Я нахожу подход Rust к проектированию программного обеспечения и архитектуре более актуальным и надеюсь, что вы тоже. Разработчики этого языка отказались от множества устаревших элементов ООП, сосредоточившись на том, что необходимо для создания качественного ПО. Rust не страдает культом сложности, которым славятся языки наподобие C++ и Java.

## 1.2. Что такое паттерны проектирования

Дать определение понятию «паттерны проектирования» непросто — зачастую это понимаешь, только когда сталкиваешься с ними на практике. Чем больше паттернов вы изучаете, тем проще становится узнавать их в чужом коде или реализовывать самостоятельно. Освоение наиболее распространенных видов

паттернов позволит вам быстро узнавать их и повторять. *Паттернами* они называются потому, что часто повторяются в различных контекстах, а *паттернами проектирования* — потому, что представляют собой высокоуровневые абстракции, которые помогают разработчикам проектировать и создавать программное обеспечение.

Некоторые свойства паттернов проектирования типичны для всех паттернов и не ограничены конкретным языком программирования. Вот эти свойства (хотя их список может быть неполным).

- Их можно использовать повторно.
- Спектр их применения очень широк.
- Они решают задачи таким образом, который позволяет легко прослеживать механизм работы чужого кода.
- Они легко узнаваемы и понятны для опытных разработчиков.
- Код, в котором не используются устоявшиеся паттерны, может попадать в категорию *антипаттернов*.

В отношении последнего пункта вы можете подумать: «Но я же вот только что придумал такой классный паттерн!» Возможно, так и есть, но пока он не станет широко используемым и известным, *не стоит* ожидать, что другие будут понимать или применять его. Хорошие паттерны проектирования со временем становятся широко распространенными и являются легкими для понимания.

При этом, как я уже упоминал выше, паттерны лишь предоставляют знакомый шаблон для нового дизайна программного обеспечения, оставляя за вами выбор деталей реализации. Хороший паттерн применим к широкому спектру приложений и накладывает минимум ограничений на их авторов. Такие паттерны развиваются по мере появления в языке новых функций и парадигм, а суть многих фундаментальных паттернов за последние десятилетия изменилась несильно.

В книге я буду использовать широкое определение паттернов и паттернов проектирования. *Паттернами* я буду называть приемы, идиомы и соглашения, которые широко используются и известны в сообществе Rust. Эти паттерны могут быть как большими и сложными, имеющими в своем составе несколько структур и компонентов, так и весьма скромными, состоящими из одной функции или метода. Что же касается *паттернов проектирования*, то под ними я понимаю широко применимые паттерны, которые служат в качестве шаблонов для проектирования кода и решают распространенные задачи программирования. Я буду использовать термины «паттерны» и «паттерны проектирования» взаимозаменяя, но обычно под первыми имеется в виду подмножество вторых.

### Что такое антипаттерны

Антипаттерны — это «горе-родственники» паттернов проектирования. Обычно мы рассматриваем паттерны проектирования как правильный способ решения определенного класса задач. Исходя из этого, антипаттерны можно назвать ошибочным способом решения. В книге мы не будем разбирать их подробно, поскольку Rust разработан так, чтобы реализовать антипаттерны в нем было сложно в принципе.

Антипаттерны в большинстве случаев являются неправильным инструментом для выполнения задачи. Вы же не станете закручивать шуруп молотком, как и забивать гвоздь отверткой.

Речь об антипаттернах пойдет в главе 10. Но по ходу повествования я буду показывать, где не стоит использовать конкретные паттерны.

Кроме того, следует сразу проговорить отличие *паттернов* от *идиом* в контексте этой книги. Существует несколько определений различий этих терминов, но я сосредоточусь на двух основных моментах: идиомы обычно относятся к самому коду, а паттерны — к дизайну и архитектуре вашего ПО. Иными словами, паттерны состоят из идиом. Некоторые из них также могут быть идиомами (например, в них отдается предпочтение итераторам вместо циклов `for`), но идиома — это не паттерн, поскольку, например, использование змеиного регистра для имен переменных паттерном не является. Идиомы обычно имеют отношение к синтаксису и форматированию кода, например к соглашениям по именованию, стилю кода и прочим низкоуровневым деталям.

В иерархическом смысле можно рассматривать идиомы как нижний уровень абстракции, паттерны проектирования — как средний, а общую архитектуру — как верхний (рис. 1.1). Архитектура любой системы состоит из множества небольших единиц паттернов проектирования, которые сами состоят из множества идиом.

Паттерны проектирования и языки программирования можно рассматривать аналогично тому, как мы рассматриваем письменные и разговорные языки. Любой язык развивается, в нем появляются новые слова, а старые выходят из моды.

Но если вы попробуете изобрести собственные слова или фразы, то они могут показаться другим людям бессмыслицей. Весь смысл любого языка в том, чтобы отчетливо доносить некий смысл, быть понятным для других и создавать чувство связи с остальными людьми. Если говорить в контексте программирования, то вы можете решить



**Рис. 1.1.** Иерархия идиом, паттернов и архитектуры

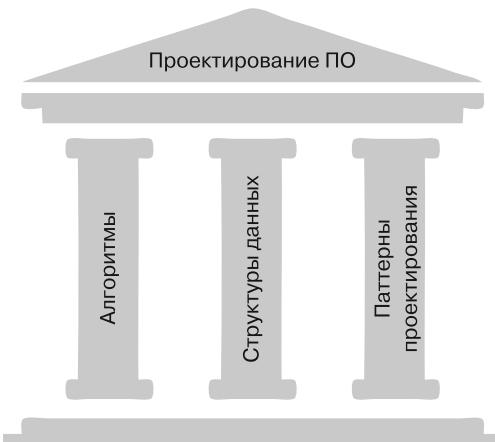
отказаться от устоявшихся норм и двигаться в собственном ритме. И это может быть нормально, но тогда велик шанс, что другие не смогут понять ваш код и не захотят с ним работать. В некоторых случаях такой компромисс приемлем, но программное обеспечение часто используется в социальных контекстах, подразумевающих участие клиентов, пользователей, менеджеров, коллег и т. д. Один в поле не воин.

Об изучении паттернов проектирования сложно говорить, не упоминая книгу «Банды четырех» «Паттерны объектно-ориентированного проектирования», широко известную среди программистов как канонический учебник по данной теме. Эта книга была написана Эрихом Гаммой, Ричардом Хельмом, Ральфом Джонсоном и Джоном Влиссидесом и выпущена издательством Addison-Wesley Professional в 1994 году. В ней приводятся примеры, написанные на C++ и Smalltalk.

Некоторые представленные в ней паттерны с тех пор были привнесены во многие языки программирования в качестве основных составляющих. Лучшими из примеров, пожалуй, будут итераторы, которые есть практически в любом языке программирования и ключевой библиотеке. Объясняется это полезностью паттерна, качеством решения с его помощью задачи перебора элементов в структуре данных и понятностью. Реализовывать итераторы с нуля, чтобы понять принцип их работы, до сих пор весьма интересно, но в большинстве языков можно использовать их встроенные эквиваленты.

Паттерны проектирования являются одним из трех столпов создания хорошего программного обеспечения; два других — алгоритмы и структуры данных (рис. 1.2). Как разработчику ПО, вам нужно хорошо разбираться в каждой из этих областей. Освоения одних только паттернов проектирования будет недостаточно.

Таким образом, паттерны проектирования — это высокоуровневые абстракции над основной грамматикой и синтаксисом языка программирования, которые позволяют эффективно передавать идеи и создавать качественный код. Правильная передача смысла является ответственностью отправителя сообщения, а не его получателя, хотя желательно, чтобы и тот и другой говорили на одном языке.



**Рис. 1.2.** Три столпа разработки хорошего программного обеспечения

### 1.3. Чем эта книга отличается от других

После выхода книги «Банды четырех», ставшей впоследствии канонической, появилось много других книг по данной теме, и в этом смысле наша ничем от них не отличается. Тем не менее в ней представлены идеи, которые касаются конкретно Rust. Этот язык становится все более популярным, поэтому очень важно каталогизировать, документировать и описывать используемые в нем паттерны.

В отличие от работы «Банды четырех» моя книга не является каталогом, а содержит описание, примеры и реализацию конкретных паттернов. Есть две причины, по которым я не хочу собирать паттерны в каталог и классифицировать: они не являются просто шаблонами или шаблонным кодом, и, просто их копируя, вы приблизитесь к полноценному завершенному коду не более чем на 10 %. Эта книга для читателей, которые стремятся к знаниям и личному росту.

Приведу аналогию с едой. Конкретное блюдо (например, лазанья) можно рассмотреть в роли паттерна проектирования. Оно является частью большого обеда, состоящего из нескольких смен блюд, напитков и безупречного сервиса. Реальная же сложность для повара в том, чтобы решить, как изготовить свою версию блюда, где взять ингредиенты и как все объединить, а затем преподнести в аппетитном виде. (Как известно любому, кто работал в ресторане, презентация — самый важный элемент.) Программирование же одновременно и наука, и искусство. Это очень креативный процесс, который выходит за рамки простого написания кода. И подражанием здесь мало добьешься.

Уникальные особенности Rust требуют уделять больше внимания проектированию API и созданию качественного кода. В частности, нужно продумывать управление памятью и временем жизни объектов, передавать значения между контекстами, избегать состояний гонки и следить, чтобы API были эргономичными. Кроме того, Rust полон возможностей для создания или открытия новых паттернов, которые определенно продолжат развиваться и после выхода этой книги. Прежде чем мы сможем отправиться на Марс, нам нужно создать ракету, которая нас туда доставит, а также решить множество задач, которые возникнут на протяжении семи месяцев полета.

Rust — удивительный язык, уникальность которого отчасти связана с тем, как он развивался, — это полностью заслуга участников сообщества. Его абстракции одновременно открывают новые паттерны, делая старые неактуальными. Изучить синтаксис языка — это одна задача, а вот писать качественный код, используя корректные паттерны в нужных местах, причем делая это правильно, — совсем другая.

### 1.4. Инструменты, необходимые для работы с Rust

В книге приводятся примеры кода, которые свободно доступны по лицензии MIT. Для получения копии кода вам потребуется подключенный к интернету компьютер с поддерживаемой операционной системой (<https://mng.bz/JZpa>),

а также инструменты, перечисленные в табл. 1.1. Подробнее об установке этих инструментов читайте в приложении.

**Таблица 1.1.** Необходимые инструменты

Название	Описание
git	Исходный код книги хранится в открытом репозитории на GitHub по адресу <a href="https://github.com/brndnmthws/idiomatic-rust-book">https://github.com/brndnmthws/idiomatic-rust-book</a>
rustup	Инструмент Rust для управления компонентами языка. Будет регулировать установку rustc и прочих компонентов
gcc или clang	Сборка некоторых примеров кода потребует наличия копии GCC или Clang. Для большинства Clang наверняка станет более подходящим выбором, поэтому по умолчанию мы будем иметь в виду его. Но при желании, встречая команду clang, можете свободно заменять ее на gcc

## Резюме

- Хорошие паттерны проектирования можно использовать повторно, они применимы во множестве ситуаций и могут решать типичные задачи программирования.
- Отличительные особенности хорошего паттерна проектирования — его понятность и то, что со временем он становится широко распространенным.
- Антипаттернами называются сложные, вредные или нечетко определенные паттерны.
- В этой книге приводятся характерные для Rust паттерны проектирования, которые используют уникальные особенности языка и его инструментов.
- Вам потребуется свежая версия Rust, Git и современный компилятор наподобие GNU GCC или LLVM Clang.
- Если вы хотите получить максимум пользы от прочтения книги, то рекомендую параллельно изучать примеры кода, доступные по адресу <https://github.com/brndnmthws/idiomatic-rust-book>.



# *Базовые структурные компоненты Rust*

---

## **В этой главе**

- ✓ Дженерики.
- ✓ Трейты.
- ✓ Совмещение обобщенных типов и трейтов.
- ✓ Автоматический вывод трейтов.

В этой главе я познакомлю вас с наиболее важными абстракциями и функциями Rust, которые буду называть *структурными элементами* и которые служат основой практических всех паттернов проектирования в этом языке. Очень важно сначала изучить эти элементы и только потом переходить к освоению паттернов. Некоторые читатели считают, что в этой главе просто описываются базовые понятия Rust. Однако она закладывает фундамент для более сложных тем, поэтому не рекомендую пропускать ее.

Начнем мы с обсуждения дженериков и трейтов. Наряду с сопоставлением с шаблоном и функциональными возможностями (будут описаны в главе 3), дженерики и трейты выступают в качестве строительных блоков практически во всех паттернах проектирования Rust. Эти элементы — «кровь и плоть» Rust.

## 2.1. Джениерики

Когда вы освоите базовый синтаксис, именно джениерики (generics — обобщенные типы, или обобщения) должны стать первой важной темой, которую вам нужно изучить далее. В Rust они относятся к этапу компиляции и являются типобезопасными абстракциями, способствующими метапрограммированию. Джениерики позволяют вместо конкретных типов использовать в функциях и определениях структур плейсхолдеры. При совмещении с трейтами (об этом процессе речь пойдет в разделе 2.2) джениерики дают возможность писать код типобезопасным способом, не требующим явного определения всех возможных типов.

Чаще всего джениерики используются для определения структур, функций и трейтов, которые работают с любым типом. К примеру, у вас может быть функция, которая работает с целыми числами, числами с плавающей запятой или строками и вы не хотите раз за разом переписывать ее для каждого типа. В этом случае джениерики позволят вам написать эту функцию один раз и использовать ее с любым типом.

Джениерики дают возможность создавать типы, состоящие из других, не требуя знания всех возможных комбинаций этих типов или вариантов их последующего использования. Джениерики — абстракция этапа компиляции, поэтому их применение не влечет за собой никаких затрат или издержек в среде выполнения. Хотя при компиляции они все же повышают сложность.

Джениерики Rust аналогичны шаблонам в C++ и дженерикам Java, поэтому покажутся вам знакомыми, если у вас есть опыт работы с этими двумя языками. В языке C иногда для обобщенного метапрограммирования используются макросы, вот только, в отличие от дженериков в Rust, C++ и Java, они *не являются* типобезопасными.

И если в некоторых языках джениерики появились в качестве добавочной функциональности, то Rust изначально создавался с их учетом. В итоге они прекрасно вписываются в этот язык, используются практически везде и не воспринимаются как нечто неуклюжее или неуместное.

### 2.1.1. Система типов, отвечающая принципу полноты по Тьюрингу

Система типов в Rust является полной по Тьюрингу, и с помощью дженериков вы можете писать программы, выполняющиеся на этапе компиляции. Это довольно ловкий прием, схожий с использованием компилятора в качестве процессора. Под «*полной по Тьюрингу*» я подразумеваю, что система типов Rust позволяет выражать любые вычисления, которые можно произвести с помощью машины Тьюринга. То есть можно вычислить все, что поддается вычислению.

Для системы типов полнота по Тьюрингу важна, так как позволяет вычислить что угодно на этапе компиляции, а не в среде выполнения, тем самым открывая ряд очень интересных возможностей.

Один из примеров использования типов для вычисления — реализация машины Минского, которую можно найти по адресу <https://github.com/paholg/minsky>. Машина Минского — это просто счетчик на основе регистров, который в вычислительном смысле равнозначен машине Тьюринга, и его можно рассматривать как аналог простого процессора. Таким образом, если мы можем создать машину Минского, используя систему типов Rust, то, по сути, можем эффективно использовать систему типов Rust для вычисления всего, что поддается вычислению.

Чтобы получить пользу от работы с Rust, не обязательно озадачиваться тем, полна ли по Тьюрингу его система типов, и на практике она может не понадобиться для вычислений. Для большинства программистов основные преимущества системы, полной по Тьюрингу, заключаются в безопасности и функциональных возможностях, которые она предоставляет.

### 2.1.2. Почему дженерики

В статически типизированных языках наподобие Rust компилятору нужно знать типы всех элементов программы на этапе компиляции. Такая необходимость предоставления информации о типах до этапа выполнения идет вразрез с динамически типизируемыми языками вроде Python и Ruby, где типы определяются во время выполнения. Дженерики же позволяют писать код, который будет работать с любым типом, не требуя от разработчика понимания, какой это будет тип на этапе компиляции. Вместо этого поручить определение типов можно самому компилятору.

За счет использования дженериков мы обеспечиваем соблюдение принципа DRY (Don't Repeat Yourself — «Не повторяйся») по всей базе кода. Писать один и тот же код во множестве мест, изменяя только сигнатуру типа, — подход, который в дальнейшем неизбежно приведет к проблемам.

Недостаток дженериков в том, что они могут усложнять чтение и написание кода. Поэтому важно сохранять баланс между их использованием и написанием прозрачного, читаемого кода. Сложность использования дженериков вызвана внесением дополнительных слоев абстракции, в частности абстракций, требующих дополнительной когнитивной нагрузки со стороны программиста. Кроме того, компилятор не всегда может точно угадать нужные вам типы, поэтому иногда нужно давать ему подсказки о том, чего конкретно вы хотите, а это усложняет дженерики и делает их более громоздкими. Но чаще всего эта дополнительная нагрузка оказывается оправданной, так как дженерики позволяют создавать более гибкое, повторно используемое и надежное программное обеспечение.

### 2.1.3. Основы дженериков

Перейдем к знакомству с синтаксисом. Базовая структура с одним обобщенным полем выглядит так:

```
struct Container<T> {
    value: T,
}
```

Здесь есть простой контейнер, который содержит значение типа `T`, определенное в качестве обобщенного параметра в угловых скобках. Дженерики можно использовать в структурах, перечислениях, функциях, блоках `impl` и не только. Этот синтаксис встречается в Rust повсеместно. Использование угловых скобок (`< ... >`) означает, что вы работаете с дженериками.

Создать экземпляр обобщенной структуры относительно просто. Зачастую компилятор справляется с выводом этого параметра типа автоматически:

```
let str_container = Container { value: "Thought is free." };
println!("{}", str_container.value);
```

Контейнер имеет тип `Container<&str>`,  
 но нам не нужно указывать этот  
 обобщенный тип явно, так как  
 компилятор выведет его сам

Этот фрагмент кода создает экземпляр `Container<&str>`, называемый `str_container`. Выполнение кода ожидаемо приведет к выводу `Thought is free.`

Иногда определить обобщенный тип компилятор может только с помощью подсказок. Предположим, что мы хотим сохранить конструкцию `Option<String>` в контейнере, но инициализируем его с `None`. Если мы попробуем выполнить такой код:

```
let ambiguous_container = Container { value: None };
```

то компилятор выдаст следующую ошибку:

```
error[E0282]: type annotations needed for `Container<Option<T>>`
--> src/main.rs:8:50
 |
8 |     let ambiguous_container = Container { value: None };
   |           ----- ^^^ cannot infer type for type parameter
   |           `T` declared on the enum `Option`
   |
   |     consider giving `ambiguous_container` the explicit type
   |     `Container<Option<T>>`, where the type parameter `T`
   |     is specified
```

К счастью, в сообщении конкретно указывается, что нужно сделать. Мы можем обновить код, указав компилятору, что хотим использовать `Option<String>`:

```
let ambiguous_container: Container<Option<String>> =
    Container { value: None };
```

Единственное различие в том, что мы указываем целевой тип слева от знака присваивания. Типы должны совпадать, чтобы компилятор мог понять, что конкретно нас интересует.

Еще один вариант — использовать паттерн конструктора `fn new()` (к которому мы еще вернемся в главе 4). В Rust он используется часто, но необходимым не является:

```
impl<T> Container<T> {
    fn new(value: T) -> Self {
        Self { value }
    }
}
```

Обобщенный параметр `T` встречается дважды — в блоке `impl` и `Container`. У вас могут быть и более сложные конструкции (например, плейсхолдеры, конкретные реализации и предустановленные типы), но та, что представлена здесь, является простейшей структурой дженерика

Мы переносим значение в структуру, то есть никаких ссылок, копий или клонирования

Здесь можно использовать короткую форму присваивания, так как значение локальной переменной совпадает со значением в структуре. Более длинным эквивалентом будет `value: value`

Далее можно вызвать `new()`. Только на этот раз мы сообщаем компилятору, какой целевой тип должен присутствовать на правой стороне выражения присваивания, явно вызывая функцию с этим типом:

```
let short_alt_ambiguous_container =
    Container::<Option<String>>::new(None);
```

Такая форма во многих случаях получается более чистой и понятной. Однако есть ситуации, когда такую форму присваивания использовать *обязательно*, поскольку эта операция по-прежнему слишком неоднозначна, чтобы компилятор мог вывести в ней целевой тип. Тогда он попросит внести ясность.

Как я уже говорил, обобщенные параметры в Rust можно добавлять во все типы структур и функций. Использование дженериков позволяет выполнять разные действия, например выстраивать рекурсивные структуры. В качестве примера создадим структуру, которая содержит экземпляр самой себя. Это будет связанный список, в котором есть обобщенный параметр:

```
#[derive(Clone)]
struct ListItem<T>
where
    T: Clone,
{
    data: Box<T>,
    next: Option<Box<ListItem<T>>>,
}
```

Можно реализовать трейт `Clone` автоматически, используя атрибут `#[derive]`

Этот же паттерн можно использовать с перечислениями. Возьмем следующий пример, с помощью которого можно создавать связанные списки (хоть и бесполезные):

```
enum Recursive<T> {
    Next(Box<Recursive<T>>),
    Boxed(Box<T>),
    Optional(Option<T>),
}
```

Здесь перечисление `Recursive` может содержать указатель на другой `Recursive`, размещенный в куче (`Boxed`) `T` или необязательный `T`. Этот пример откровенно бесполезен, но показывает, что можно делать с помощью дженериков.

**ПРИМЕЧАНИЕ** Я буду использовать пример со связанным списком и далее, чтобы показывать различные возможности Rust, опираясь на текущий вариант примера. На тот случай, если вы не знаете, что такое связанные списки, скажу, что их односвязная форма — это структура данных, состоящая из последовательности элементов, каждый из которых содержит ссылку на следующий элемент:  $A \rightarrow B \rightarrow C \rightarrow \dots \rightarrow Z$ .

Итак, мы можем применить этот паттерн к нашему связанному списку, используя вместо `Option` такую структуру:

```
enum NextNode<T> {
    Next(Box<ListNode<T>>),
    End,
}

struct ListNode<T> {
    data: Box<T>,
    next: NextNode<T>,
}
```

Наш узел списка содержит размещенные в куче данные типа `T` и необязательный `next`, указывающий на следующий узел в списке. Это красивый сжатый код. Однако в целях ясности будет лучше использовать `Option`, а не создавать подобный эквивалент.

**ПРИМЕЧАНИЕ** Подобающая реализация связанных списков в Rust сложнее, чем я показываю в данной главе. Я еще вернусь к этой теме позднее и продемонстрирую более эффективный способ создания связанных списков с помощью указателей `Rc` и `RefCell`. Пример, показанный выше, будет бесполезен в большинстве практических случаев.

## 2.1.4. Знакомимся с `Option` в Rust

Перейдем к `Option` и начнем с определения:

```
pub enum Option<T> {
    None,
    Some(T),
}
```

В Rust `Option` — один из самых ярких примеров применения дженериков на практике. Его определение простое и элегантное, но при этом обеспечивает неизменно мощную абстракцию.

### 2.1.5. Структуры маркеров и фантомные типы

Иногда может возникнуть такая ситуация: вам необходимо создать структуру с обобщенными параметрами, но вы не хотите использовать эти параметры в самой структуре. В таком случае помогут *фантомные типы*, которые позволяют использовать обобщенные параметры, не являющиеся членами структуры. Они дают возможность применять паттерн тегирования с помощью структур (поговорим о нем в главе 7).

Во фрагменте кода ниже мы видим структуру, которая содержит параметр типа, но в самой структуре этот тип не используется (на этапе компиляции мы имеем только информацию типа):

```
struct Dog<Breed> {
    name: String,
}
```

Структура `Dog` содержит имя собаки. Мы хотим отслеживать породу собак, но эти детали интересуют нас только при компиляции (не при выполнении), поэтому мы можем, по сути, сохранить эту информацию в виде параметра типа, не добавляя в структуру поле `breed: Breed`. Для этого потребуется создать типы, которые будут обозначать породы:

```
struct Labrador {}
struct Retriever {}
struct Poodle {}
struct Dachshund {}
```

Для каждой породы мы используем пустую структуру. Можно задействовать любой тип, но мы возьмем именно пустую структуру. Тем не менее попытка скомпилировать код в его текущем виде приведет к следующей ошибке:

```
error[E0392]: parameter `Breed` is never used
--> src/main.rs:27:12
   |
27 | struct Dog<Breed> {
   |         ^^^^^ unused parameter
   |
= help: consider removing `Breed`, referring to it in a field,
or using a marker such as `PhantomData`
= help: if you intended `Breed` to be a const parameter,
use `const Breed: usize` instead
```

Компилятор ругается, так как мы добавили в структуру неиспользуемый обобщенный параметр, на что компилятор обоснованно указывает. Сообщить ему

о том, что этот параметр нам нужен, мы можем с помощью фантомного поля, но его значение нас интересует только при компиляции, поэтому хранить это значение в структуре нет необходимости.

```
use std::marker::PhantomData;

struct Dog<Breed> {
    name: String,
    breed: PhantomData<Breed>,
}
```

При создании структуры `Dog` нам все равно нужно предоставить фантомные данные, хотя в ходе оптимизации компилятор их удалит:

```
use std::marker::PhantomData;

let my_poodle: Dog<Poodle> = Dog {
    name: "Jeffrey".into(),
    breed: PhantomData,
};
```

`PhantomData` — особый вид маркера, с которым вы встретитесь при работе с Rust. Маркеры обычно используются в качестве *трейтов-маркеров*, но в данном случае `PhantomData` — это *структурно-маркер*. В стандартной библиотеке Rust есть несколько трейтов-маркеров. Подробнее о них мы поговорим в главе 7.

Наиболее важный случай использования структур-маркеров — определение отдельных типов на этапе компиляции, что бывает весьма полезно. При желании мы можем добавлять специализированные варианты структуры `Dog` для каждой породы и возвращать ее название, не сохраняя ее значение в виде состояния или отдельного поля структуры:

```
impl Dog<Labrador> {
    fn breed_name(&self) -> &str {
        "labrador"
    }
}

impl Dog<Retriever> {
    fn breed_name(&self) -> &str {
        "retriever"
    }
}

impl Dog<Poodle> {
    fn breed_name(&self) -> &str {
        "poodle"
    }
}

impl Dog<Dachshund> {
    fn breed_name(&self) -> &str {
        "dachshund"
    }
}
```

Impl `Dog <Labrador>` — это конкретный вид структуры `Dog` с типом породы `Labrador`. `impl` не требует обобщенного параметра `Breed`, так как это конкретная специализация

Мы можем возвращать породу, не сохраняя ее название в качестве поля структуры. Оно будет являться частью сегмента данных скомпилированной программы

Для каждого блока `impl` мы создаем конкретную специализацию структуры `Dog` с указанным типом. Можно добавлять сколько угодно таких специализаций, и если одной будет не хватать, то компилятор сообщит об этом. Обратите внимание, что мы не используем `impl<T>`, так как это не обобщенное инстанцирование. Специализация происходит в конкретном типе.

Теперь, чтобы вернуть название породы, можно вызвать `breed_name()` для экземпляра `Dog`. Обратите внимание: в методах `breed_name()` не нужно указывать для ссылки `&str` время жизни `'static`, поскольку эти методы получают `&self`. Таким образом, компилятор может обоснованно заключить, что время жизни возвращаемой строки соответствует `&self`.

### Время жизни и `'static`

Время жизни в Rust — это мощная функциональность, которая позволяет указывать, как долго ссылка (или заимствование) является действительной. Ссылки равнозначны указателям, но в отличие от тех указателей, которые вы можете знать по C или C++, с ними нельзя выполнять арифметических действий. Указание времени жизни позволяет обеспечить валидность ссылок на протяжении их использования.

Основной смысл параметра времени жизни (который начинается с одинарной кавычки) заключается в том, что он позволяет отметить ссылку именем, с помощью которого компилятор сможет отслеживать ее время жизни в ходе использования. Время жизни похоже на обобщенные параметры, поскольку также указывается в угловых скобках, но все же это два разных механизма. Следующая структура содержит параметр времени жизни `'a`:

```
struct Dog<'a> {
    name: &'a str,
}
```

В этом коде мы указываем, что поле `name` структуры `Dog` содержит ссылку на строку с временем жизни `'a`. Таким образом мы сообщаем компилятору, что ссылка должна оставаться действительной как минимум до тех пор, пока действительна структура `Dog`.

В Rust `'static` — это особое время жизни, которое длится в течение всего периода выполнения программы. Такое время жизни присуще всем строковым литералам, поэтому для них указывать его не обязательно. Если вы возвращаете строковый литерал из функции, то для явного указания его времени жизни нужно вернуть этот литерал как `&'static str`.

Как я только что сказал, добавлять время жизни `'static` для строкового литерала не обязательно, но это может оказаться полезным при его возвращении из функции, так как позволит показать, что он будет валидным в течение всего периода выполнения программы.

Итак, в завершение проверим наш код:

```
let my_poodle: Dog<Poodle> = Dog {
    name: "Jeffrey".into(),
    breed: PhantomData,
};

println!(
    "My dog is a {}, named {}",
    my_poodle.breed_name(),
    my_poodle.name,
);
```

Он дает такой вывод:

```
My dog is a poodle, named Jeffrey
```

Мой пудель Джейфри был корректно определен как пудель, и здесь мы успешно применили фантомный тип для специализации структуры `Dog`.

### **2.1.6. Ограничения трейтов обобщенных параметров**

Прежде чем переходить к самим трейтам (что мы сделаем в разделе 2.2), нужно вкратце затронуть тему их ограничений. *Ограничения трейтов* — это особенность дженериков, благодаря которой можно контролировать, какие типы могут быть использованы в конкретной структуре или функции, путем указания трейтов, которые должны быть реализованы. В частности, такой механизм позволяет указывать, какие возможности должны быть доступны конкретному параметру обобщенного типа. Можно устанавливать несколько ограничений трейтов, каждый из которых будет применяться к какому-то отдельному параметру. Если вы вернетесь к примеру связанного списка, который приводился в подразделе 2.1.3, то заметите в структуре `ListItem` два нюанса:

- мы вывели трейт `Clone`, который позволяет вызывать для структуры метод `clone()`, чтобы ее скопировать;
- прописав ограничение трейта `where T: Clone`, мы указали, что обобщенный тип `T` также должен реализовывать трейт `Clone`.

Если мы хотим потребовать обязательной реализации `Clone` и `Debug`, то используем следующий код:

```
#[derive(Clone)]
struct ListItem<T>
where
    T: Clone + Debug,
{
    data: Box<T>,
    next: Option<Box<ListItem<T>>>,
```

## 2.2. Трейты

Потратив некоторое время на изучение Rust и освоив его синтаксис, механизмы заимствования и время жизни, вы поймете, что трейты вкупе с дженериками являются базовыми элементами в этом языке. Трейты — очень мощная абстракция, которая формирует основу многих библиотек Rust. Но обеспечиваемая ими широта возможностей влечет за собой ответственность. Использование трейтов чревато двумя серьезными проблемами: загрязнением и дублированием. Чуть позже мы разберем, как их избежать.

Трейты позволяют определять общую функциональность для типов. Экземпляры типов (объекты) содержат состояние (например, структуру), и трейты определяют поверх него совместно используемую функциональность, не привязанную ни к какому конкретному типу.

Трейты существуют не только в Rust. Изначально они появились в малоизвестном языке Self, но присутствуют и в других языках, таких как Scala, Julia, TypeScript, Kotlin (в качестве интерфейсов), Haskell (в качестве классов типов) и Swift (в качестве расширений протоколов).

И хотя трейты зачастую используются для управления состоянием, они отличаются от своей реализации, которая привязывается к конкретному типу. То есть сами по себе трейты являются обобщенными, но их реализация конкретна. При этом их можно автоматически выводить с помощью атрибута `#[derive]`. Библиотеки могут экспорттировать трейты, их реализаций или и то и другое.

### 2.2.1. Почему трейты не относятся к объектно-ориентированному программированию

Rust не является языком для объектно-ориентированного программирования (ООП), но его код может казаться похожим на эту парадигму в плане эргономики. В нем есть объекты, которые могут содержать методы. *Объект* — это экземпляр типа, такого как структура или перечисление, который представляет некое состояние. При вызове методов объекта используется синтаксис, аналогичный синтаксису из объектно-ориентированных языков (`object.method()`). Однако в Rust нет одной важной особенности, присущей таким языкам, — *наследования*.

Это отсутствие компенсируется трейтами. По своей сути трейты отличаются от классов (или наследования классов), но при этом решают аналогичный набор задач. В ООП объекты расширяются за счет наследования. В случае же программирования на основе трейтов можно добавлять трейты поверх любой структуры или типа данных, обеспечивая таким образом конкретные возможности. При наследовании между объектами определяется *связь по типу is-a*, в то время как трейты определяют *функциональность*.

Иными словами, если сравнивать трейты с ООП, то они добавляют совместно используемые возможности поверх разных видов состояний. От классов трейты

отличаются тем, что их функциональность не связывается с конкретными типами (или состоянием). И хотя классы в C++ тоже можно делать обобщенными с помощью шаблонов, в этом языке подобное разделение произвести сложно.

### 2.2.2. Что содержится в трейте

Трейты содержат определение и любое количество необязательных реализаций. Определение обычно состоит из таких компонентов, как:

- имя трейта;
- необязательный набор методов (с необязательными предустановленными реализациями);
- необязательные обобщенные типы-плейсхолдеры;
- необязательный набор необходимых трейтов.

Как минимум трейту необходимо имя, поэтому следующий фрагмент кода представляет рабочее определение:

```
trait MinimalTrait { }
```

*Реализации* трейтов применяют определение трейта к конкретному типу. Обычно мы пишем такие реализации для отдельных типов, но система трейтов Rust достаточно гибкая, чтобы нам не приходилось реализовывать трейт для каждого возможного типа. Вдобавок трейты могут использовать обобщенные типы данных (поговорим о них в подразделе 2.2.4), которые предоставляют иной способ установки сложных связей. И хотя реализации трейтов конкретны, также можно предоставлять *blanket*-реализации (обобщенные реализации), которые будут охватывать все типы, удовлетворяющие прописанным условиям. (Подробнее такой вид реализаций обсудим в главе 7.) Ниже представлен пример трейта с реализацией в Rust:

```
trait DoesItBark {           ← Блок определения трейта
    fn it_barks(&self) -> bool; ← Сигнатура метода
}                           трейта

struct Dog;

impl DoesItBark for Dog {   ← Блок реализации
    fn it_barks(&self) -> bool {
        true                  ← Можно прописать возвращаемое значение
    }                         жестко, так как собаки точно лают
}
```

Определения трейтов могут быть пустыми, что позволяет использовать их для метапрограммирования, например, в виде трейтов-маркеров. О расширенном использовании трейтов мы поговорим в главах 7–9.

В ООП дополнительные возможности добавляются путем наследования в иерархии (класс B <– класс Б <– класс А). В случае же трейтов никакая структура

наследования не подразумевается. Их можно применять к любому типу внутри крейта. Трейты могут иметь зависимости, указанные в виде их ограничений (то есть трейт B требует реализации трейта A), но даже с ограничениями их можно применять к любому типу, который соответствует этим ограничениям.

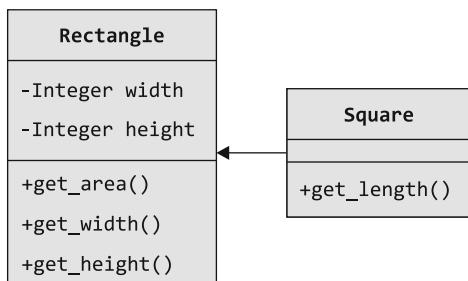
В ООП связи определяются с помощью самих объектов. Если же мы используем трейты, то для определения связей указываем, какие трейты реализует объект, а не то, для какого поведения он реализован, — тонкое, но важнейшее различие.

**ПРИМЕЧАНИЕ** Очень прошу вас не думать о трейтах в разрезе принципов ООП, таких как классы и наследование. Но в книге я привел сравнения, чтобы внести большую ясность для тех, кто пришел из мира ООП. Попытка соотнести эти принципы один в один на практике не будет иметь смысла. Трейты требуют иного подхода. Так что советую скорректировать свою модель мышления и исключить из нее лишние понятия доктрины ООП.

### 2.2.3. Разбираемся в трейтах на примере объектно-ориентированного кода

Трейты обеспечивают куда большую гибкость, нежели наследование, которое требует построения связи снизу вверх. (То есть при наследовании общее поведение определяется на нижних уровнях иерархии.) Для начала рассмотрим фрагмент, написанный на C++, в котором используется связь по типу *is-a*. Затем я объясню, как сделать то же самое в Rust. Начнем с реализаций связи, показанной на рис. 2.1.

Соответствующий код C++ для UML с рис. 2.1 показан в листинге 2.1.



**Рис. 2.1.** Схема на унифицированном языке моделирования (Unified Modeling Language, UML) для геометрических фигур на C++

#### Листинг 2.1. Моделирование геометрических фигур на C++

```

class Rectangle {
protected:
    int width;
    int height;

public:
    Rectangle(int width, int height) : width(width), height(height) {}
    int get_area() { return width * height; }
    int get_width() { return width; }
    int get_height() { return height; }
};
  
```

← Моделирует простой прямоугольник,  
используя ширину и высоту

```
class Square : public Rectangle {
public:
    Square(int length) : Rectangle(length, length) {}
    int get_length() { return width; }
};
```

Моделирует квадрат, представляющий простой прямоугольник, ширина и высота которого равны. Значит, можно выполнить наследование от класса Rectangle

Написать равнозначный код на Rust будет не так просто. Прямой перевод получится неуклюжим. Вместо этого мы изменим структуру кода. Для начала разберем код, в котором моделируется прямоугольник (листинг 2.2).

### Листинг 2.2. Реализация прямоугольника на Rust

```
struct Rectangle {
    width: i32,
    height: i32,
}

impl Rectangle {
    pub fn new(width: i32, height: i32) -> Self {
        Self { width, height }
    }
}
```

Моделируется простой прямоугольник, имеющий только ширину и высоту

Здесь мы используем подобный конструктору метод new(), который возвращает новый прямоугольник. В Rust создание конструкторов new() является распространенным паттерном

Далее смоделируем квадрат (листинг 2.3).

### Листинг 2.3. Реализация квадрата на Rust

```
struct Square {
    length: i32,
}

impl Square {
    pub fn new(length: i32) -> Self {
        Self { length }
    }
    pub fn get_length(&self) -> i32 {
        self.length
    }
}
```

Моделировать квадрат еще проще; нужен всего один атрибут

Предоставляет конструктор, использующий паттерн new()

Добавляет аксессор для получения длины квадрата, если известно, что у нас квадрат

Теперь можно создать трейт Rectangular (листинг 2.4).

### Листинг 2.4. Реализация трейта Rectangular

```
pub trait Rectangular {
    fn get_width(&self) -> i32;
    fn get_height(&self) -> i32;
    fn get_area(&self) -> i32;
}

impl Rectangular for Rectangle {
    fn get_width(&self) -> i32 {
        self.width
    }
}
```

Здесь мы определяем Rectangular, который предоставляет аксессоры для свойств, имеющихся и у прямоугольников, и у квадратов

Реализует трейт Rectangular для Rectangle

```

fn get_height(&self) -> i32 {
    self.height
}
fn get_area(&self) -> i32 {
    self.width * self.height
}
}

impl Rectangular for Square {
    fn get_width(&self) -> i32 {
        self.length
    }
    fn get_height(&self) -> i32 {
        self.length
    }
    fn get_area(&self) -> i32 {
        self.length * self.length
    }
}

```

Реализует трейт  
Rectangular для Square

На рис. 2.2 показан результат, отрисованный в UML.

В завершение протестируем код (листинг 2.5).

### Листинг 2.5. Тестирование трейта Rectangular

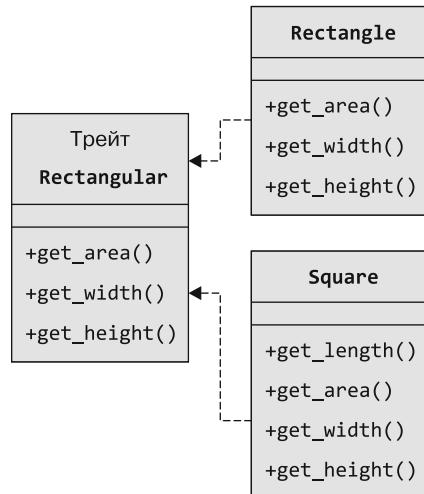
```

fn main() {
    let rect = Rectangle::new(2, 3);
    let square = Square::new(5);

    println!(
        "rect has width {}, height {}, and area {}",
        rect.get_width(),
        rect.get_height(),
        rect.get_area()
    );
    println!(
        "square has length {} and area {}",
        square.get_length(),
        square.get_area()
    );
}

```

Версия на Rust поначалу кажется более длинной. Здесь нам пришлось реализовать трейт `Rectangular` дважды, нарушив принцип DRY (Don't Repeat Yourself — «Не повторяйся»). Но при этом мы проделали нечто фундаментальное: отделили состояние (в данном случае измерения) от функциональности, отвечающей за предоставление ширины, высоты, площади и т. д. По мере усложнения это разделение



**Рис. 2.2.** UML-диаграмма для геометрических фигур на Rust

аспектов будет масштабироваться гораздо лучше. Выполнив код из листинга 2.5, мы получим такой ожидаемый вывод:

```
$ cargo run
rect has width 2, height 3, and area 6
square has length 5 and area 25
```

Польза трейтов становится очевидной, если учесть сложность изменения существующего кода. В подразделе 2.2.4 мы разберем еще один пример, но подойдем к нему с точки зрения Rust.

## 2.2.4. Совмещение дженериков и трейтов

Допустим, мы хотим создать функцию, которая получает любой тип и возвращает его описание. Ниже мы напишем такую функцию, получающую обобщенный параметр `T` и возвращающую его описание. Предположим, что эти типы — например, `Dog` и `Cat` — где-то определены. Описания для них нам нужно дать самостоятельно, поскольку компилятор их сформировать не сможет. Для этого мы используем такое определение функции:

```
fn describe_type<T>(t: &T) -> String { ... }
```

Теперь нужно понять, как получить описание типа `T`. Все просто — нам нужен трейт, который будет его предоставлять. Выглядеть он будет так:

```
pub trait SelfDescribing {
    fn describe(&self) -> String;
}
```

Отлично. Теперь у нас есть трейт, содержащий описание типа. Как же заставить функцию использовать его? Например, следующий код не сработает:

```
fn describe_type<T>(t: &T) -> String {
    t.describe()
}
```

Компилятор выдаст ошибку:

```
error[E0599]: no method named `describe` found for reference `&T` in the current scope
--> src/main.rs:6:7
   |
6 |     t.describe()
   |     ^^^^^^^^ method not found in `&T`
   |
   = help: items from traits can only be used if the type parameter is
     bounded by the trait
help: the following trait defines an item `describe`, perhaps you need to
restrict type parameter `T` with it:
   |
5 | fn describe_type<T: SelfDescribing>(t: &T) -> String {
   | ~~~~~~
```

For more information about this error, try `rustc --explain E0599`.

Неплохо! Компилятор сообщает, что конкретно необходимо сделать. Нам нужно сказать ему, что мы хотим использовать метод `describe()` из трейта `SelfDescribing`, для чего мы создадим ограничение трейта. Такие ограничения говорят компилятору, что указанный тип должен реализовывать конкретный трейт. В Rust этот прием встречается часто, в том числе вместе с дженериками.

Обратите внимание, что указать ограничение можно двумя способами: встроенно (как в выводе ошибки компилятором) или в явном условии `where`, которое следует за определением функции. Так выглядит встроенная форма:

```
fn describe_type<T: SelfDescribing>(t: &T) -> String {
    t.describe()
}
```

И хотя такая форма лаконичнее, в случае сложных ограничений я предпочитаю вариант `where`, поскольку для разработчика он чуть более читабелен:

```
fn describe_type<T>(t: &T) -> String
where
    T: SelfDescribing,
{
    t.describe()
}
```

Теперь наша программа компилируется и можно написать код для ее тестирования:

```
struct Dog;
struct Cat;

fn main() {
    let dog = Dog;
    let cat = Cat;
    println!("I am a {}", describe_type(&dog));
    println!("I am a {}", describe_type(&cat));
}
```

Попытка скомпилировать этот код приведет к ошибке, так как здесь нет реализации:

```
error[E0277]: the trait bound `Dog: SelfDescribing` is not satisfied
--> src/main.rs:15:41
   |
15 |     println!("I am a {}", describe_type(&dog));
   |             ----- ^^^^ the trait `SelfDescribing`
   |             is not implemented for `Dog`
   |             |
   |             required by a bound introduced by this call
   |
note: required by a bound in `describe_type`
```

```
--> src/main.rs:5:21
| fn describe_type<T: SelfDescribing>(t: &T) -> String {
|     ^^^^^^^^^^^^^^ required by this bound in
`describe_type`  

error[E0277]: the trait bound `Cat: SelfDescribing` is not satisfied
--> src/main.rs:16:41
|     println!("I am a {}", describe_type(&cat));
|             ^^^^^ the trait `SelfDescribing`  

is not implemented for `Cat`  

|             |
|             required by a bound introduced by this call  

note: required by a bound in `describe_type`  

--> src/main.rs:5:21
| fn describe_type<T: SelfDescribing>(t: &T) -> String {
|     ^^^^^^^^^^^^^^ required by this bound in
`describe_type`
```

For more information about this error, try `rustc --explain E0277`.

И снова компилятор точно сообщает нам, чего не хватает. (Нужно реализовать `SelfDescribing` для `Dog` и `Cat`.) Добавим эти реализации:

```
impl SelfDescribing for Dog {
    fn describe(&self) -> String {
        "happy little dog".into()
    }
}

impl SelfDescribing for Cat {
    fn describe(&self) -> String {
        "curious cat".into()
    }
}
```

Теперь выполнение кода дает такой результат:

```
$ cargo run
I am a happy little dog
I am a curious cat
```

Обратите внимание, что этот код использует параметр `&self` в `fn describe(&self)`, требуя наличия в трейте экземпляра типа. Можно ли обойтись без этого требования? Попробуем. Мы изменим наш трейт так:

```
pub trait SelfDescribing {
    fn describe() -> String;
}
```

В этом варианте мы исключили `&self` из метода `describe()`. Теперь нужно обновить функцию `describe_type()`:

```
fn describe_type<T: SelfDescribing>() -> String {
    T::describe()
}
```

...и реализации (исключив параметр `&self`):

```
impl SelfDescribing for Dog {
    fn describe() -> String {
        "happy little dog".into()
    }
}

impl SelfDescribing for Cat {
    fn describe() -> String {
        "curious cat".into()
    }
}
```

И в завершение изменяем вызов к `describe_type()`:

```
fn main() {
    println!("I am a {}", describe_type::<Dog>());
    println!("I am a {}", describe_type::<Cat>());
}
```

Обе рассмотренные формы валидны, но подходят в разных случаях. Если мы требуем наличия параметра `&self` в вызове метода, то должны иметь экземпляр типа для его описания. Если же опустить данный параметр, то можно будет описать тип при отсутствии экземпляра объекта.

Получив базовое представление о трейтах, вы сможете применять их в различных задачах. Чаще всего они используются для обеспечения *обобщенной функциональности*, совместно применяемой разными типами. Однако этот пример использования — лишь вершина айсберга, так как на основе трейтов можно создавать весьма обширные паттерны этапа компиляции, о которых пойдет речь в главах 7–9.

Трейты очень интересны, но их нужно правильно использовать. Для меня самыми серьезными проблемами являются загрязнение и дублирование. *Загрязнение трейтами* происходит, когда их у вас слишком много. *Дублирование* же относится к случаям, когда несколько трейтов предоставляют одинаковую (или похожую) функциональность. В типичных паттернах обычно уже есть трейты, и по возможности лучше использовать их. Сторонние библиотеки зачастую определяют собственные трейты, а иногда даже конкурирующие, и вы можете потратить немало времени на написание связующего кода, который подружит вашу программу с трейтами разных библиотек.

### 2.2.5. Автоматический вывод трейтов

Если вы новичок в Rust, то должны ознакомиться с часто используемыми трейтами в стандартной библиотеке, такими как `Clone`, `Debug`, `Default`, трейты-итераторы и трейты-равенства. Кроме того, в Rust есть особые трейты наподобие `Drop`, предоставляющего деструктор, и тех, которые компилятор выводит автоматически, например `Send` и `Sync`. Весь список особых трейтов можно найти в документации Rust на странице <https://mng.bz/wxKa>.

Для некоторых из наиболее распространенных трейтов вы будете использовать атрибут `#[derive]`, чтобы создавать реализации автоматически. Определения структур, использующие `#[derive]` для автоматического вывода трейтов и шаблонного кода, встречаются довольно часто. В примере ниже показаны `Clone`, `Debug` и `Default` с нашей структурой `Pumpkin`:

```
use std::fmt::Debug;

#[derive(Clone, Debug, Default)]
struct Pumpkin {
    mass: f64,
    diameter: f64,
}
```

Здесь у нас есть `Pumpkin`, которую с помощью `Debug` можно отформатировать в виде строки, с помощью `Clone` клонировать, а с помощью `Default` создать ее базовый экземпляр:

```
fn main() {
    let big_pumpkin = Pumpkin {
        mass: 50.,
        diameter: 75.,
    };
    println!("Big pumpkin: {:?}", big_pumpkin);
    println!("Cloned big pumpkin: {:?}", big_pumpkin.clone());
    println!("Default pumpkin: {:?}", Pumpkin::default());
}
```

Вот вывод после выполнения этого кода:

```
$ cargo run
Big pumpkin: Pumpkin { mass: 50.0, diameter: 75.0 }
Cloned big pumpkin: Pumpkin { mass: 50.0, diameter: 75.0 }
Default pumpkin: Pumpkin { mass: 0.0, diameter: 0.0 }
```

На практике вам потребуется применять эти трейты довольно часто, поскольку они широко используются как в стандартной библиотеке Rust, так и в сторонних библиотеках. К счастью, с помощью атрибута `#[derive]` сделать это

несложно. В определении `Option` из стандартной библиотеки Rust мы видим следующее:

```
#[derive(Copy, PartialEq, PartialOrd, Eq, Ord, Debug, Hash)]
pub enum Option<T> {
    None,
    Some(T),
}
```

Здесь `Option` обеспечивает реализации трейтов для `Copy`, `PartialEq`, `PartialOrd`, `Eq`, `Ord`, `Debug` и `Hash`. Вы можете заметить, что трейт `Clone` отсутствует. Он реализуется без атрибута `#[derive]`.

Вам не обязательно выводить реализации трейтов, что часто является самым простым способом. Вдобавок вы всегда можете написать собственные. Предположим, вам нужно, чтобы базовая структура `Pumpkin` имела `diameter`, равный 5, и `mass`, равную 2. Тогда вы исключите `Default` из `#[derive]` и добавите следующую реализацию:

```
impl Default for Pumpkin {
    fn default() -> Self {
        Self {
            mass: 2.0,
            diameter: 5.0,
        }
    }
}
```

Теперь при выполнении кода вы получите такой вывод:

```
$ cargo run
Big pumpkin: Pumpkin { mass: 50.0, diameter: 75.0 }
Cloned big pumpkin: Pumpkin { mass: 50.0, diameter: 75.0 }
Default pumpkin: Pumpkin { mass: 2.0, diameter: 5.0 }
```

## 2.2.6. Трейт-объекты

В Rust есть удобная функциональность под названием *трейт-объекты*. Она позволяет управлять объектами как трейтами, а не типами. Трейт-объекты можно рассматривать как аналог виртуальных методов в C++ или Java, только наследование к ним неприменимо. Что касается деталей реализации, то в Rust внутренне они реализуются с помощью *vtable*<sup>1</sup> — таблицы поиска, генерируемой компилятором и позволяющей производить динамическую диспетчеризацию в среде выполнения.

Некоторые участники сообщества Rust считают трейт-объекты, динамическую диспетчеризацию и *vtable* своеобразной формой полиморфизма среды выполнения. В некоторых случаях использование динамической диспетчеризации

---

<sup>1</sup> *Vtable* — таблица виртуальных методов, координирующая таблица. — Примеч. пер.

можно рассматривать как антипаттерн, о которых мы будем говорить в главе 10. Я же считаю, что трейт-объекты — это инструменты, которые программист по своему усмотрению может использовать как во зло, так и во благо.

Для определения трейт-объектов применяется ключевое слово `dyn`, и вместо того, чтобы использовать имя типа, мы предоставляем трейт. Предположим, нам нужно сохранить какой-нибудь тип в контейнере. Это можно сделать при условии, что все типы будут реализовывать некий указанный трейт, как в этом примере:

```
trait MyTrait {
    fn trait_hello(&self);
}

struct MyStruct1;

impl MyStruct1 {
    fn struct_hello(&self) {
        println!("Hello, world! from MyStruct1");
    }
}

struct MyStruct2;

impl MyStruct2 {
    fn struct_hello(&self) {
        println!("Hello, world! from MyStruct2");
    }
}

impl MyTrait for MyStruct1 {
    fn trait_hello(&self) {
        self.struct_hello();
    }
}

impl MyTrait for MyStruct2 {
    fn trait_hello(&self) {
        self.struct_hello();
    }
}
```

В этом коде мы объявляем `MyTrait`, предоставляющий метод `trait_hello()`. Метод реализован как для `MyStruct1`, так и для `MyStruct2`, которые вызывают собственные отдельные методы `struct_hello()`, выводящие `Hello, world!`. Теперь можно протестировать этот код:

При выполнении этого тестового кода мы получаем такой вывод:

```
Hello, world! from MyStruct1
Hello, world! from MyStruct2
```

Мы не можем сохранить трейт в виде объекта напрямую, поскольку трейт-объекты не имеют размера (не реализуют трейт `Sized`). Иными словами, нам нужно сохранять объекты в типе контейнера, который может содержать объекты, не реализующие `Sized`. К таким контейнерам относятся умные указатели `Box`, `Rc`, `Arc`, `RefCell` и `Mutex`. Но при этом мы не можем сохранить безразмерный объект непосредственно внутри `Vec`. У `Box` (и других видов умных указателей) среди ограничений трейтов есть `where T: ?Sized`, говорящий о том, что `Sized` является необязательным (то есть указатель может содержать трейт-объекты). В Rust по умолчанию для любого обобщенного типа `T` трейт `Sized` является необходимым (эквивалент `where T: Sized`).

Мы не можем создать, например, `Vec<dyn MyTrait>`, так как `Vec` не умеет создавать безразмерные объекты. `Box` же, напротив, разграничивает выделение элемента и его локализацию. То есть при создании объекта с помощью `Box` мы одновременно предоставляем конкретный тип. Тогда компилятор может автоматически привести этот объект к типу трейт-объекта (имеется в виду от `Box<MyStruct1>` к `Box<dyn MyTrait>`) при его передаче или присваивании.

**СОВЕТ** Подробнее о трейт-объектах читайте в документации Rust по адресу <https://mng.bz/qOp6>.

### Поникающее приведение трейт-объектов

Помимо издержек координирующих таблиц, еще одним ограничением трейт-объектов является возможность вызова методов только в трейте, но не в конкретном типе. Если мы хотим привести трейт-объект к конкретному типу, то используем поникающее приведение (уточнение типа). Для этого можно использовать `Box`, `Rc` и `Arc`, а также трейт `Any`, который предоставляет метод для поникающего приведения. Если же мы хотим получить ссылку, то должны использовать именно `Any`. Метод `downcast()` в `Box`, `Rc` и `Arc` будет поглощать объект, а `Any` предоставляет `downcast_ref()`, который возвращает ссылку.

Трейт `Any` выводится автоматически для любых типов, которые имеют ограничение `'static`. Это означает, что они свободны от нестатических ссылок, поэтому данный прием работает только для объектов `dyn Any + 'static`.

Чтобы получить объект `Any` в трейт-объекте, нужно сначала предоставить способ для его извлечения из `Box`. Нельзя просто вызвать `downcast_ref()` для `Box<dyn MyTrait>`, поскольку `Box` сам реализует `Any` и мы получим не тот объект. Вместо этого нужно добавить в наш трейт метод `as_any()` для получения внутреннего объекта. Обновим код:

```

trait MyTrait {
    fn trait_hello(&self);
    fn as_any(&self) -> &dyn Any;
}

impl MyTrait for MyStruct1 {
    fn trait_hello(&self) {
        self.struct_hello();
    }
    fn as_any(&self) -> &dyn Any {
        self
    }
}

impl MyTrait for MyStruct2 {
    fn trait_hello(&self) {
        self.struct_hello();
    }
    fn as_any(&self) -> &dyn Any {
        self
    }
}

```

← Этот метод трейта позволяет получать &dyn Any

← Возвращает экземпляр Any для self

← Возвращает экземпляр Any для self

Теперь можно получить ссылку на исходный тип объекта:

```

println!("With a downcast:");
v.iter().for_each(|i| {
    if let Some(obj) = i.as_any().downcast_ref::<MyStruct1>() {
        obj.struct_hello();
    }
    if let Some(obj) = i.as_any().downcast_ref::<MyStruct2>() {
        obj.struct_hello();
    }
});

```

← Здесь вместо iter() можно было использовать into\_iter().  
В полноценном фрагменте кода это последний случай  
использования объекта v. Значит, можно поглотить этот  
объект, а не использовать ссылку, задействуя iter()

И последнее. Некоторые крейты предоставляют расширенные возможности уточнения типов, например `downcast`, `downcast-rs` и `Anyhow`. Подробнее о крейтах речь пойдет в главе 4.

Еще один нюанс динамической диспетчеризации: тщательно обдумывайте, насколько вы хотите использовать трейты таким образом. Например, вам не следует злоупотреблять этой возможностью, чтобы реализовывать полиморфизм в стиле ООП. Мы будем разбирать такой вариант в качестве антипаттерна в главе 10.

Полноценного руководства по трейтам Rust не существует, но очень хорошей отправной точкой будет документация (<https://doc.rust-lang.org/std/prelude/index.html>). В ней перечисляются трейты и типы, доступные в базовом пространстве имен Rust.

Напоследок добавлю, что нельзя реализовывать внешние трейты для типов, находящихся за пределами крейта. Но этот нюанс можно обойти, используя оберточные структуры или трейты-расширения, о которых мы будем говорить в главах 5 и 7. Вы по-прежнему можете реализовывать локальные трейты (определенные внутри крейта) для любого типа, даже из внешних крейтов. Можете реализовывать внешние трейты с несколькими параметрами типов для внешних типов при условии, что один из этих параметров является локальным. Более подробная информация доступна в документации Rust (<https://mng.bz/7dA7>), где описываются правила сиротства (orph rules).

## Резюме

- Дженики (обобщения) — ключевые абстракции в Rust, которые позволяют повторно использовать код типобезопасным способом.
- Дженики позволяют добавлять параметры типов при определении структур, перечислений и функций для создания объектов и функций, способных обрабатывать множество типов значений, а не какой-то один конкретный тип.
- Как правило, дженики используются для создания типов контейнеров (которые содержат другие виды произвольных данных).
- Трейты в Rust позволяют добавлять совместно используемую функциональность поверх различных типов.
- Можно совмещать дженики и трейты, чтобы создавать небольшие библиотеки, которые будут прекрасно справляться со своими функциями, в противовес более крупным приложениям.
- При определении обобщенных параметров с помощью ограничений трейтов можно указывать, какие трейты должны реализовывать эти параметры. Это позволит создавать обобщенный код, зависящий от общего поведения, без указания конкретных типов.
- Можно выводить трейты автоматически с помощью атрибута `#[derive(...)]`, избегая лишнего ввода, в том числе шаблонного кода.

# 3

## Поток кода

### В этой главе

- ✓ Что такое сопоставление с шаблоном и как с его помощью обрабатывать ошибки.
- ✓ Обзор функционального программирования в Rust.

Прежде чем переходить к паттернам проектирования, вам нужно познакомиться и с другими фундаментальными возможностями Rust. В этой главе вы начнете изучать сопоставление с шаблоном и функциональное программирование. *Сопоставление с шаблоном* позволяет управлять потоком выполнения кода, разворачивать или деструктурировать значения, а также обрабатывать необязательные случаи. *Функциональное программирование* дает возможность создавать ПО на основе единичной функции, которая является одной из самых базовых и простых для понимания абстракций.

Это разные структурные элементы, которые можно по-разному совмещать, создавая новые абстракции. Впоследствии мы будем связывать их воедино для получения более сложных паттернов проектирования.

### 3.1. Что такое сопоставление с шаблоном

До этого момента мы говорили о дженериках и трейтах, определяющих основные возможности Rust на этапе компиляции. *Сопоставление с шаблоном* — это функциональность среды выполнения, которая позволяет применять эффективные паттерны выполнения кода. Сопоставлять можно типы, значения, варианты

перечислений и не только. В Rust эта возможность весьма эффективна, поскольку поддерживает несколько видов сопоставления (как значений, так и типов). И самое важное — сопоставление с шаблоном позволяет создавать чистые функциональные паттерны программирования.

**ПРИМЕЧАНИЕ** Сопоставление с шаблоном не нужно путать с паттернами проектирования. Первое — это фундаментальная функциональность Rust (и других языков), и хотя ее можно использовать для создания вторых, но в строгом смысле она к ним не относится.

Если вы использовали инструкцию `switch/case`, то сопоставление с шаблоном в Rust покажется вам знакомым процессом, но при этом оно обладает куда большим потенциалом. В некоторых языках программирования есть аналогичные возможности, но сопоставление с шаблоном все равно остается чем-то нишевым, и встретить эту функциональность можно далеко не везде. Впервые она получила широкое распространение в Prolog и является важнейшей особенностью функциональных языков, таких как Haskell, Scala, Erlang (создавался на базе Prolog и под его влиянием), Elixir и OCaml.

Простейшее сопоставление с шаблоном начинается с ключевого слова `match`, которое позволяет легко перестраивать структуру. Как и в случае с инструкцией `switch/case`, здесь мы перечисляем все шаблоны, которые хотим сопоставить с необязательным блоком `catch-all` в конце. В Rust же нужно сопоставлять все возможные шаблоны или предоставить блок `catch-all`. Об отсутствии этого блока компилятор Rust сообщает, выдавая ошибку.

### 3.1.1. Основы сопоставления с шаблоном

Простым примером сопоставления с шаблоном будет раскрытие структуры `Option` и вывод информации о том, содержит ли он значение (листинг 3.1).

#### Листинг 3.1. Сопоставление Option с шаблоном

```
fn some_or_none<T>(option: &Option<T>) {
    match option {
        Some(_v) => println!("is some!"),
        None => println!("is none :("),
    }
}
```

Разворачиваем значение `option` в `_v`.  
 Предваряющее переменную нижнее  
 подчеркивание говорит компилятору,  
 что это необязательное значение

Разворачивание `Option`, `Result` или других структур, содержащих необязательные данные, является типичным случаем сопоставления с шаблоном. Подобное разворачивание данных — главное достоинство этого приема, так как компилятор здесь требует обработки всех случаев, полностью исключая сомнения в том, сделали вы это или нет. Сопоставление с шаблоном не может гарантировать отсутствие в коде логических ошибок, зато упрощает его анализ.

Внимательный читатель мог заметить, что в листинге 3.1 мы отбросили значение `Some(_v)`, но будет неплохо вывести его. Для этого мы используем в сопоставлении с шаблоном привязку и обновим обобщенный параметр `T`, добавив ограничение трейта для `std::fmt::Display` (листинг 3.2).

### Листинг 3.2. Сопоставление Option с шаблоном и вывод ограничения трейта

```
fn some_or_none_display<T: std::fmt::Display>(option: &Option<T>) {
    match option {
        Some(v) => println!("is some! where v={v}"),
        None => println!("is none :("),
    }
}
```

Теперь можно вызвать `some_or_none_display()` с `Option`, содержащим любое значение, реализующее `std::fmt::Display`, и вывести данное значение, если это `Some`.

### Получение информации об уязвимостях безопасности

Подавляющее большинство критических уязвимостей в программном обеспечении связаны с одним и тем же классом проблем: безопасностью памяти. Исследователи из Microsoft выяснили (<http://mng.bz/>), что 70 % всех уязвимостей в их продуктах были связаны с багами в обработке памяти кодом C и C++. Вот примеры подобных проблем:

- ◆ чтение/запись за пределами массива;
- ◆ разыменование недействительных указателей, таких как нулевые;
- ◆ использование памяти после ее освобождения;
- ◆ попытка освободить память, которая уже была освобождена (двойное освобождение);
- ◆ отсутствие обработки ошибочных случаев.

В Rust функциональность защиты нацелена на исключение этих случаев, и сопоставление с шаблоном является ключевой особенностью, которая помогает программистам избегать проблем, требуя обработки всех случаев. Сопоставление с шаблоном в примере с `Option`, где проверяются варианты `Some` и `None`, — отличный пример того, как Rust заставляет нас обрабатывать все возможные кейсы.

Выбор Rust для создания критически важного программного обеспечения подобен покупке страховки или пут-опциона (финансовый инструмент, защищающий от крупных потерь). Использование этого языка — это способ избежнуть рисков в плане уязвимостей безопасности, а также защитить ваших пользователей и репутацию. В качестве оплаты вы вкладываете время и силы, необходимые для освоения механизмов защиты в Rust, соблюдаете дисциплину и прикладываете дополнительные усилия. Что взамен? Вы можете быть спокойны, осознавая, что ваше программное обеспечение с меньшей вероятностью попадет в заголовки новостей об очередном взломе систем безопасности. В качестве простейшего компромисса вы заранее предельываете некоторую дополнительную работу, избавляя себя от большого количества будущих хлопот, которые могут возникнуть из-за ошибки.

## 58 Глава 3. Поток кода

Сопоставление с шаблоном не ограничивается разворачиванием типов Option, хотя это распространенный случай. Кроме того, можно сопоставлять конкретные значения интегралов, добавляя диапазоны:

```
fn what_type_of_integer_is_this(value: i32) {  
    match value {  
        1 => println!("The number one number"),  
        2 | 3 => println!("This is a two or a three"),  
        4..=10 => println!("This is a number between 4 and 10 (inclusive)"),  
        _ => println!("Some other kind of number"),  
    }  
}
```

Сопоставление с шаблоном часто используется для декомпозиции структур, кортежей и перечислений. Можно разбирать кортежи частично или вытаскивать из них отдельные элементы, что иногда является удобным способом получения доступа к ним:

```
fn destructure_tuple(tuple: &(i32, i32, i32)) {  
    match tuple {  
        (first, ..) => {  
            println!("First tuple element is {first}") ← Выполняет сопоставление  
        } ← только с первым элементом  
    } ← кортежа любой длины  
  
    match tuple {  
        (.., last) => {  
            println!("Last tuple element is {last}") ← Выполняет сопоставление  
        } ← только с последним элементом  
    } ← кортежа любой длины  
  
    match tuple {  
        (_, middle, _) => {  
            println!()  
            "The middle tuple element is {middle}" ← Выполняет сопоставление  
        } ← только со средним элементом  
    } ← кортежа из трех элементов  
  
    match tuple {  
        (first, middle, last) => {  
            println!("The whole tuple is ({first}, {middle}, {last})") ← Выполняет сопоставление  
        } ← с каждым элементом кортежа  
    } ← из трех элементов  
}
```

У вас может быть несколько равнозначных выражений `match`, но блок в целом всегда будет возвращать выражение из первого совпавшего паттерна. В примере выше мы для каждого случая используем отдельный блок `match`, так как все сопоставления валидны. Если у вас в блоке `match` есть несколько равнозначных паттернов, то код будет компилироваться, но будет выдано предупреждение. Вот пример:

```
fn unreachable_pattern_match(value: i32) {  
    match value {  
        1 => println!("This value is equal to 1"),  
    }
```

```

    1 => println!("This value is equal to 1"),
    _ => println!("This value is not equal to 1"),
}
}

```

Компиляция этого кода приведет к следующему предупреждению для второго случая `match`:

```

warning: unreachable pattern
--> src/main.rs:56:9
56 |     1 => println!("Second match: This value is equal to 1"),
|     ^
|
= note: `#[warn(unreachable_patterns)]` on by default

```

Использование `guard` (защитного условия) позволяет делать сопоставление условно. Для этого после паттерна указывается оператор `if`, который может использовать сопоставляемое или отдельное значение, передаваемое в защитное условие. В коде ниже `guard` используется при сопоставлении со значением `i32` и логическим значением:

```

fn match_with_guard(value: i32, choose_first: bool) {
    match value {
        v if v == 1 && choose_first => {
            println!("First match: This value is equal to 1")
        }
        v if v == 1 && !choose_first => {
            println!("Second match: This value is equal to 1")
        }
        v if choose_first => {
            println!("First match: This value is equal to {v}")
        }
        v if !choose_first => {
            println!("Second match: This value is equal to {v}")
        }
        _ => println!("Fell through to the default case"),
    }
}

```

С помощью оператора `match` нельзя сопоставлять значения разных типов. Все ветки сопоставления внутри блока `match()` должны относиться к одному типу. Блок `match` — это выражение, поэтому каждая его ветвь (и каждое внутреннее выражение) должна возвращать один и тот же тип. Можно разворачивать структуры, которые содержат разные типы (например, перечисления), но сопоставление нельзя делать обобщенным образом. К примеру, код ниже не является валидным:

```

fn invalid_matching<T>(value: &T) {
    match value {
        "is a string" => println!("This is a string"),
        1 => println!("This is an integral value"),
    }
}

```

## 60 Глава 3. Поток кода

При попытке его скомпилировать возникнет ошибка:

```
error[E0308]: mismatched types
--> src/lib.rs:3:9
|
1 | fn invalid_matching<T>(value: &T) {
|         - this type parameter
2 |     match value {
|             ----- this expression has type `&T`
3 |         "is a string" => println!("This is a string"),
|             ^^^^^^^^^^^^^^ expected `&T`, found `&str`
|
4 | = note: expected reference `&T`
|           found reference `&'static str`
```

```
error[E0308]: mismatched types
--> src/lib.rs:4:9
|
1 | fn invalid_matching<T>(value: &T) {
|         - this type parameter
2 |     match value {
|             ----- this expression has type `&T`
3 |         "is a string" => println!("This is a string"),
|             ^ => println!("This is an integral value"),
|                 ^ expected type parameter `T`, found integer
|
4 | = note: expected type parameter `T`
|           found type `{integer}`
```

For more information about this error, try `rustc --explain E0308`.

Можно деструктурировать различные внутренние типы, если использовать перечисление. `DistinctTypes` аналогично `Option` позволяет сопоставлять отдельные именованные типы в `match_enum_types()`:

```
enum DistinctTypes {
    Name(String),
    Count(i32),
}

fn match_enum_types(enum_types: &DistinctTypes) {
    match enum_types {
        DistinctTypes::Name(name) => println!("name={name}"),
        DistinctTypes::Count(count) => println!("count={count}"),
    }
}
```

Можно разбирать структуры, чтобы извлекать конкретные значения, и даже сопоставлять с конкретными значениями внутри них, что я покажу в примере ниже. Этот фрагмент кода создает перечисление окрасов кошек, структуру, которая содержит имя кошки и ее цвет, а также функцию `match_on_black_cats()`, которая выводит это имя и сообщает о том, является ли кошка черной:

```

enum CatColor {
    Black,
    Red,
    Chocolate,
    Cinnamon,
    Blue,
    Cream,
    Cheshire,
}

struct Cat {
    name: String,
    color: CatColor,
}

fn match_on_black_cats(cat: &Cat) {
    match cat {
        Cat {
            name,
            color: CatColor::Black,
        } => println!("This is a black cat named {name}"),
        Cat { name, color: _ } => println!("{} is not a black cat"),
    }
}

```

Можно тут же протестировать этот код так:

```

let black_cat = Cat {
    name: String::from("Henry"),
    color: CatColor::Black,
};

let cheshire_cat = Cat {
    name: String::from("Penelope"),
    color: CatColor::Cheshire,
};

match_on_black_cats(&black_cat);
match_on_black_cats(&cheshire_cat);

```

В результате получим следующее:

```

This is a black cat named Henry
Penelope is not a black cat

```

### **3.1.2. Чистые сопоставления с помощью оператора ?**

Сопоставление с шаблоном — прекрасный способ обработки ошибок. Но при очень большом количестве сопоставлений или их излишне глубокой вложенности код становится беспорядочным. Этот прием можно совмещать с использованием оператора ?, чтобы чисто обрабатывать функции, возвращающие **Result** или **Option**. В этом случае возврат происходит, как только **Result** или **Option** вернут ошибку или **None** соответственно. Использовать оператор ? можно

только изнутри функции, которая возвращает `Result` или `Option`. Он позволяет существенно выровнять многоуровневый код, повысив его читаемость:

```
fn write_to_file() -> std::io::Result<()> {
    use std::fs::File;
    use std::io::prelude::*;

    let mut file = File::create("filename")?;
    file.write_all(b"File contents")?;
    Ok(())
}

fn try_to_write_to_file() {
    match write_to_file() {
        Ok(_) => println!("Write succeeded"),
        Err(err) => println!("Write failed: {}", err.to_string()),
    }
}
```

В коде выше мы обертываем вызов `write_to_file()` в выражение сопоставления с шаблоном. Если функция вернет `Ok()`, то выводим `Write succeeded`. В случае ошибки выводим `Write failed: ...` с соответствующим сообщением.

Оператор `?` очень удобен тем, что позволяет сохранять чистоту кода в случае применения `Result`. Обратите внимание: я использовал единичный тип `()`, особый тип в Rust, по сути являющийся плейсхолдером, который не несет никакого значения и удаляется компилятором при оптимизации. Единичный тип `()` часто называется просто *unit*. Ниже показан эквивалентный код без оператора `?`, содержащий дублирование на случай вывода возможной ошибки:

```
fn write_to_file_without_result() {
    use std::fs::File;
    use std::io::prelude::*;

    let create_result = File::create("filename");
    match create_result {
        Ok(mut file) => match file.write_all(b"File contents") {
            Err(err) => {
                println!("There was an error writing: {}", err)
            }
            _ => println!("Write succeeded"),
        },
        Err(err) => println!(
            "There was an error opening the file: {}",
            err
        ),
    }
}
```

Если нужно связать в цепочку множество вызовов с помощью `?`, то следует обратить внимание на возвращаемые ими типы. Оператор `?` работает только

с функциями, которые возвращают `Result<T, E>` либо `Option<T>`, соответствующий типу оператора, к которому был применен `?`. В случае `Result<T, E>` типы ошибок всех функций, использующих `?`, должны соответствовать родительской функции или предоставлять реализацию трейта `From`, чтобы их можно было преобразовать в целевой тип ошибки. По этой причине для преобразования одного типа ошибки в другой вам зачастую придется писать `impl From for ... { }`.

**СОВЕТ** При создании цепочек с использованием оператора `?` преобразование между `Result` и `Option` можно делать не только с помощью трейта `From`, но и другими способами. В случае `Result<T, E>` можете использовать метод `ok()` для отображения в `Option<T>`, `err()` для отображения в `Option<E>` и `map_err()` для отображения ошибки в другой тип. Если же вам нужно отобразить `Option<T>` в `Result<T, E>`, используйте `ok_or()`.

В предыдущем примере если мы захотим использовать вместо `std::io::Error` собственный тип ошибки, например, для расширения исходного дополнительной информацией, то нам нужно будет сделать следующее:

```
enum ErrorTypes {
    IoError(std::io::Error),
    FormatError(std::fmt::Error),
}

struct ErrorWrapper {
    source: ErrorTypes,
    message: String,
}
```

Далее следует реализовать для обертки ошибки `From<std::io::Error>`:

```
impl From<std::io::Error> for ErrorWrapper {
    fn from(source: std::io::Error) -> Self {
        Self {
            source: ErrorTypes::IoError(source),
            message: "there was an IO error!".into(),
        }
    }
}
```

Теперь можно обновить код записи в файл, чтобы он использовал новый тип ошибки, возвращая `ErrorWrapper` в функции `write_to_file()`:

```
fn write_to_file() -> Result<(), ErrorWrapper> { ←
    use std::fs::File;
    use std::io::prelude::*;

    let mut file = File::create("filename")?;
    file.write_all(b"File contents")?;
    Ok(())
}
```

Вместо `std::io::Result` возвращает простой `Result`, используя наш тип ошибки

```
fn try_to_write_to_file() {
    match write_to_file() {
        Ok(()) => println!("Write succeeded"),
        Err(err) => {
            println!("Write failed: {}", err.message)
        }
    }
}
```

← Выводит наше сообщение вместо того, которое предоставляет std::io::Error

При вызове наша функция `try_to_write_to_file()` в стандартных условиях должна выводить `Write succeeded`. В случае же ошибки (например, при отсутствии разрешения на запись в файл) она будет выводить `Write failed: ...` с сообщением, предоставленным `File`.

Подобная обработка ошибок очень распространена в Rust и может значительно сократить объем необходимого синтаксиса. Это относительно простой способ интеграции ошибок из стороннего кода. В главе 4 мы еще вернемся к оператору `?` и обработке ошибок в Rust.

## 3.2. Функциональный Rust

До этого момента мы рассматривали основы: дженерики, трейты и сопоставление с шаблоном. Теперь же перейдем к функциональным возможностям Rust и к одной из моих любимых тем: функциональному программированию. В этом языке двумя его ключевыми элементами являются *замыкания* и *итераторы*.

Замыкания становятся все популярнее, поэтому многие из вас наверняка их так или иначе уже использовали. К примеру, в языках JavaScript и TypeScript, а также в их библиотеках замыкания применяются повсеместно. Итераторы же настолько распространены, что большинство программистов воспринимают их уже не как абстракции, а как фундаментальную особенность всех современных языков.

Функциональное программирование — это парадигма, в которой программы строятся из декларативных функций, и изменение состояния в них не приветствуется, хотя в зависимости от строгости языка может допускаться. Некоторые же языки строго функциональны, то есть изменять состояние в них нельзя. И единственный способ как-то повлиять на него — использовать функцию, отображающую одно значение в другое. Кроме того, функциональные языки ориентированы на исключение побочных эффектов, то есть таких действий внутри функции, которые могут иметь недетерминированные результаты. Это может быть, например, ввод/вывод или изменение локального состояния.

В целях поддержки функционального программирования некоторые языки содержат инструменты, созданные непосредственно для взаимодействия с функциями и обработки неизменяемого состояния. Rust не является строго функциональным, но позволяет применять функциональные паттерны, так как

по умолчанию изменение состояния в нем не допускается, и для этого нужно использовать ключевое слово `mut`. К тому же в нем предоставляются основные функциональные возможности, такие как замыкания и итераторы.

Тема функционального программирования очень обширна, поэтому я затрону лишь высокоуровневые возможности Rust. Более углубленно с этой темой можно познакомиться в прекрасной книге *Groking Functional Programming* Михала Плахты<sup>1</sup> (Michał Płachta).

### **3.2.1. Основы функционального программирования в Rust**

Начнем с разбора простого (но не чистого) замыкания:

```
let bark = || println!("Bark!"); ←
    bark();
```

Вызов `println!()` создает побочные эффекты,  
так как это операция ввода/вывода, а значит,  
замыкание не является чистым

Это функция, которая лает подобно собаке, выводя "Bark!". При этом на функцию она не очень похожа, так как аргументов в ней нет, и скобки ввиду их необходимости были удалены. В Rust замыкания начинаются со списка аргументов, прописанных между двумя вертикальными чертами (||), сопровождаемыми блоком кода. Добавим параметр, чтобы придать функции более аутентичный вид:

```
let increment = |value| value + 1;
increment(1);
```

Здесь функция получает целочисленное `value` и возвращает сумму его сложения с 1. Тип параметра `value` указывать не нужно, так как компилятор может вывести его сам. Теперь, используя блок кода, создадим замыкание, которое будет еще больше похоже на функцию:

```
let print_and_increment = |value| {
    println!("{} will be incremented and returned");
    value + 1
};
print_and_increment(5);
```

Но все эти примеры не слишком интересны. Наиболее четко польза замыканий становится видна, когда речь идет о *функциях высшего порядка*, которые получают в качестве параметров другие функции. В Rust вы могли встречать такие, когда работали с итераторами, в частности при использовании методов `map()`, `for_each()`, `find()`, `fold()` и аналогичных. Функции высшего порядка дают удобную возможность делегировать операции вызывающему коду, с помощью которого можно передавать вызываемому внутреннюю логику. Замыкания делают синтаксис более удобным, приятным и гибким. В этом примере использования

---

<sup>1</sup> Плахта М. Гроаем функциональное программирование. – СПб.: Питер, 2024.

функции высшего порядка создается сумматор (adder), получающий значения из других функций:

```

Замыкание, которое возвращает 1
и предоставляет impl Fn() -> i32
let left_value = || 1;
let right_value = || 2;
let adder = |left: fn() -> i32,
            right: fn() -> i32| {
    left() + right()
};

println!(
    "{} + {} = {}",
    left_value(),
    right_value(),
    adder(left_value, right_value)
);

```

Замыкание, которое возвращает 2  
и предоставляет impl Fn() -> i32

Замыкание, которое получает две функции  
и складывает их результаты, предоставляя  
impl Fn(fn() -> i32, fn() -> i32) -> i32

В этом примере два замыкания, которые присвоены `left_value` и `right_value` и возвращают жестко прописанное целое число. Далее создается `adder`, который получает два параметра, имеющих специальный функциональный тип `fn() -> i32`. Мы можем передать любую функцию, соответствующую сигнатуре сумматора. В этом случае мы складываем левое и правое значение, а именно `1 + 2`, поэтому функция возвращает `3`. При выполнении вышеприведенного кода мы получим такой вывод:

`1 + 2 = 3`

Вы можете поэкспериментировать с изменением значений, возвращаемых `left_value` и `right_value`. Вывод будет меняться соответственно. Кроме того, вы можете попробовать заменить сумматор мультипликатором, который будет не складывать, а умножать значения.

### 3.2.2. Захват переменных замыкания

Если мы хотим вызвать сумматор с функцией, не имеющей подобающей сигнатуры, то можем обернуть ее другим замыканием, которое обеспечит эту сигнатуру. Рассмотрим захват переменных в замыканиях, чтобы понять, почему это может понадобиться.

Облегчить функциональное программирование в Rust призваны три трейта: `Fn`, `FnMut` и `FnOnce`. Они реализуются автоматически там, где это возможно, и описать их можно так:

- `Fn` применяется для функций формы `Fn(&self)`. Их можно вызывать повторно, так как они не поглощают переменные, которые захватывают. Все аргументы в них неизменяемы (иммутабельны);
- `FnMut` применяется для изменяемых (мутабельных) функций, имеющих форму `FnMut(&mut self)`. Их тоже можно вызывать повторно по той же причине, но при этом они могут содержать изменяемые ссылки;

- FnOnce применяется для функций, которые используются только один раз (буквально: consume themselves), таких как FnOnce(self). Эти функции можно вызвать только раз, так как они поглощают переменные, которые захватывают.

В случае замыканий трейт FnOnce реализуется всегда, если замыкание поглощает любую из захватываемых переменных, о чём сообщает ключевое слово move, стоящее перед определением замыкания. Взглянем на замыкание в листинге 3.3.

### Листинг 3.3. Замыкание с move

```
let consumable = String::from("cookie");
let consumer = move || consumable;
consumer();
// consumer(); error!
```

В этом примере четвертая строка выдаст ошибку, поскольку consumable можно переместить лишь раз, а значит, второй вызов consumer() окажется недействительным. Если попробовать скомпилировать код, не закомментировав второй вызов consumer(), то компилятор выдаст следующее:

```
error[E0382]: use of moved value: `consumer`
--> src/main.rs:22:5
21 |     consumer();
|     ----- `consumer` moved due to this call
22 |     consumer();
|     ^^^^^^^^ value used here after move
|
note: closure cannot be invoked more than once because it moves the
variable `consumable` out of its environment
--> src/main.rs:20:28
|
20 |     let consumer = move || consumable;
|     ^^^^^^^^^^
note: this value implements `FnOnce`, which causes it to be moved when called
--> src/main.rs:21:5
|
21 |     consumer();
|     ^^^^^^^^
For more information about this error, try `rustc --explain E0382`.
error: could not compile `closures` (bin "closures") due to 1 previous error
```

Чаще всего move [...] используется (как в листинге 3.3), когда нужно передать владение объектом в замыкание, избежав его копирования или клонирования. Ключевое слово move является необязательным. Если его не использовать, то Rust сам разберется, нужно ли перемещать переменную, которую вы захватываете. И все же будет лучше выразить ваши намерения явно, чтобы исключить двусмысленность. Естественно, в случае ошибки компилятор выдаст предупреждение. В примере с consumable можно было безопасно опустить слово

`move`, на результат это не повлияло бы. Для получения различных обобщенных функциональных паттернов можно совместно использовать замыкания, дженерики, а также трейты `Fn`, `FnMut` и `FnOnce`.

### 3.2.3. Знакомство с итераторами

Теперь поговорим об итераторах Rust, которые дополняют замыкания. В этом языке они предстаются трейтом `Iterator`, который содержит множество функций: `map()`, `for_each()`, `take()`, `fold()`, `filter()`, `find()`, `zip()` и др. Если вы реализуете трейт `Iterator` для своего типа, то получите все эти итераторы (и не только).

Итераторы — один из оригинальных паттернов проектирования, предложенных «Бандой четырех», который можно считать самым продуктивным. Они служат прекрасным наглядным примером как самих паттернов проектирования, так и их применения в языке Rust. Основа трейта `Iterator` выглядит так:

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Этот трейт содержит намного больше, нежели показано здесь, но если вы захотите реализовать `Iterator` для своего типа, то вам потребуется предоставить только `next()` и `Item`. Рассмотрим пример связанного списка в Rust, реализовав трейт `Iterator`. Начнем с написания реализации нового связанного списка (листинг 3.4).

#### Листинг 3.4. Реализация `LinkedList`

```
use std::cell::RefCell;
use std::rc::Rc;

type ItemData<T> = Rc<RefCell<T>>;
type ListItemPtr<T> = Rc<RefCell<ListItem<T>>>;

struct ListItem<T> {
    data: ItemData<T>,           Указатель
    next: Option<ListItemPtr<T>>, ←   на наши данные
}                                     Указатель на следующий элемент
                                         связанного списка

impl<T> ListItem<T> {
    fn new(t: T) -> Self {       Создает новый элемент
        Self {                   (или узел) списка
            data: Rc::new(RefCell::new(t)),
            next: None,
        }
    }
}
```

```

struct LinkedList<T> {
    head: ListItemPtr<T>, ← Указатель на первый элемент
}
}

impl<T> LinkedList<T> {
    fn new(t: T) -> Self { ← Создает новый список, где head
        Self { ← указывает на первый элемент
            head: Rc::new(RefCell::new(ListItem::new(t))),
        }
    }
}

```

В этом листинге показан незавершенный связанный список, который имеет нужную структуру, но не предоставляет способ перебора или добавления новых элементов. Я намеренно опустил функциональность добавления, так как хочу реализовать ее с помощью итератора. Если сначала реализовать трейт `Iterator`, то добавить остальную функциональность связанного списка будет несложно. Попробуем сделать это.

### Rc и RefCell

На случай, если вы не знакомы с `Rc` или `RefCell` (есть в листинге 3.4), я дам краткое пояснение о них. Это умные указатели, которые обеспечивают важные (но различные) возможности.

`Rc` предоставляет указатель с подсчетом ссылок, аналогичный `std::shared_ptr` в C++, а `RefCell` — особый тип указателя, делающий возможными *внутренние изменения*.

`Rc` позволяет хранить несколько ссылок (или указателей) на одну область памяти, а `RefCell` дает возможность проверять заимствование в среде выполнения. Обычно модуль проверки заимствований в Rust работает на этапе компиляции, но иногда эту проверку нужно производить при выполнении. Примером может быть случай, когда необходимо хранить несколько ссылок на один объект, оставляя возможность их изменения (что при компиляции невозможно).

В примере связанного списка выше нам нужно хранить несколько ссылок на один объект (что обеспечивает `Rc`). Вдобавок нам нужна возможность изменять внутренний объект (что позволяет делать `RefCell`).

В пятой главе книги *Code Like a Pro in Rust*, которую я уже упоминал, я тщательно разбираю умные указатели Rust. Что касается `Rc` и `RefCell`, то подробнее читайте о них в документации стандартной библиотеки на страницах <https://doc.rust-lang.org/std/rc/index.html> и <https://doc.rust-lang.org/std/cell/index.html> соответственно.

Отмечу, что итераторы *имеют состояние*. То есть итератор понимает, где конкретно он находится в последовательности элементов, в связи с чем может при каждом очередном вызове `next()` переходить к следующему.

**ПРИМЕЧАНИЕ** Даже в чистейших функциональных языках при достаточном усердии можно найти внутреннее состояние, так как программное обеспечение в конечном счете всегда разбивается на строго императивный машинный код.

Мы пока будем хранить состояние в самом связанном списке. Общую структуру вместе с методом fn new() можно обновить так:

```
struct LinkedList<T> {
    head: ListItemPtr<T>,
    cur_iter: Option<ListItemPtr<T>>,
}

impl<T> LinkedList<T> {
    fn new(t: T) -> Self {
        Self {
            head: Rc::new(RefCell::new(ListItem::new(t))),
            cur_iter: None,
        }
    }
}
```

Прекрасно! Теперь у нас есть указатель на текущую позицию итератора в cur\_iter(), который может быть инициализирован как None. Реализуем первый вариант трейта Iterator для нашего связанного списка (это не идеальный подход, к которому мы придем чуть позже):

```
impl<T> Iterator for LinkedList<T> {
    type Item = ListItemPtr<T>;
    fn next(&mut self) -> Option<Self::Item> {
        match &self.cur_iter.clone() {
            None => {
                self.cur_iter = Some(self.head.clone());
            }
            Some(ptr) => {
                self.cur_iter = ptr.borrow().next.clone();
            }
        }
        self.cur_iter.clone()
    }
}
```

В этой реализации итератора мы будем возвращать указатель на элемент списка, а не сами данные

Здесь нужно клонировать cur\_iter, поскольку мы пытаемся изменить указатель, который позднее заимствуется

Если cur\_iter является None, значит, итератор не инициализирован и мы начинаем с начала

cur\_iter необходимо обновлять, чтобы он указывал на очередной элемент в последовательности

В завершение мы клонируем и возвращаем текущую позицию в последовательности

Теперь найти последний элемент списка с помощью итератора будет несложно:

```
let dinosaurs = LinkedList::new("Tyrannosaurus Rex");
let last_item = dinosaurs.last()
    .expect("couldn't get the last item");
println!("last_item='{}'", last_item.borrow().data.borrow());
```

Реализовав трейт Iterator, мы получаем возможность вызывать last() для извлечения последнего элемента списка. При выполнении предыдущего кода

мы ожидаемо получим вывод `last_item='Tyrannosaurus Rex'`. А теперь добавим в оригинальный `LinkedList` наш метод `append()`:

```
impl<T> LinkedList<T> {
    fn new(t: T) -> Self {
        Self {
            head: Rc::new(RefCell::new(ListItem::new(t))),
            cur_iter: None,
        }
    }
    fn append(&mut self, t: T) {
        self.last()
            .expect("List was empty, but it should never be")
            .as_ref()
            .borrow_mut()
            .next = Some(Rc::new(RefCell::new(ListItem::new(t))));
    }
}
```

Чтобы получить доступ к внутреннему `ListItem`, необходимо заимствовать внутренний `RefCell`

Чтобы изменить внутренний указатель `next`, необходимо выполнить соответствующее заимствование

Теперь можно дополнить, а затем перебрать наш список, используя `for_each` с замыканием:

```
let mut dinosaurs = LinkedList::new("Tyrannosaurus Rex");
dinosaurs.append("Triceratops");
dinosaurs.append("Velociraptor");
dinosaurs.append("Stegosaurus");
dinosaurs.append("Spinosaurus");
dinosaurs
    .iter()
    .for_each(|ptr| {
        println!("data={}", ptr.borrow().data.borrow());
    });

```

Здесь нам все равно нужно разворачивать внутренний указатель, и наш вызов `for_each()` будет поглощать `dinosaurs`

При выполнении этого кода мы получим такой вывод:

```
data=Tyrannosaurus Rex
data=Triceratops
data=Velociraptor
data=Stegosaurus
data=Spinosaurus
```

**ПРИМЕЧАНИЕ** Код в этом примере не соответствует финальной реализации, а значит, отличается от кода в репозитории. Но вскоре мы дойдем до итоговой версии.

Неплохо, как считаете? Это интересный пример, но наш итератор неидеален, поскольку для доступа к полезной нагрузке в каждом узле связанного списка все равно нужно разворачивать внутренний указатель. Я считаю, что для типа-коллекции этот интерфейс неудобен. Если мы будем писать библиотеку, то вряд ли захотим раскрывать внутренние типы.

### 3.2.4. Получение итератора с помощью `with_iter()`, `into_iter()` и `iter_mut()`

Чтобы сделать наш связанный список более идиоматическим, нужно перебирать элементы, не раскрывая его внутреннюю структуру. Кроме того, нам нужно перебирать изменяемые ссылки на элементы списка, получить владение списком и перебирать его элементы. Иными словами, нам может потребоваться перебирать список тремя способами:

- используя `iter()` — перебирать неизменяемые ссылки на элементы списка;
- используя `iter_mut()` — перебирать изменяемые ссылки на элементы списка;
- используя `into_iter()` — получить владение списком и перебирать его элементы.

В подразделе 3.2.3 я реализовал трейт `Iterator` непосредственно в `LinkedList`, но это не идиоматический вариант и так делать нежелательно. Вместо этого мы создадим отдельную структуру для обработки итерации, что является типичным паттерном в Rust и более грамотным решением в плане структуры. Встроенные в Rust типы-коллекции обычно предоставляют три итератора:

- для перебора `T`; предоставляемый `into_iter(self)`, который получает владение (`consume`) `self`;
- для перебора `&T`; предоставляемый `iter(&self)`;
- для перебора `&mut T`; предоставляемый `iter_mut(&mut self)`.

Вы заметите, что `Vec` не реализует трейт `Iterator` напрямую. Вместо этого он реализует для `T`, `&T` и `&mut T` трейт `IntoIterator`. `Vec` для реализации `Iterator` использует собственные внутренние объекты `Iter`, `IterMut` и `IntoIter` (<https://doc.rust-lang.org/std/vec/struct.IntoIter.html>). Мы в нашем связанном списке можем поступить аналогично, создав для обработки итерации отдельные структуры вместо реализации `Iterator`.

Скопируем этот паттерн и применим его к нашему связанному списку. Для начала создадим новые структуры итератора с сохранением состояния:

```
struct Iter<T> {
    next: Option<ListItemPtr<T>>,
}
struct IterMut<T> {
    next: Option<ListItemPtr<T>>,
}
struct IntoIter<T> {
    next: Option<ListItemPtr<T>>,
}
```

Каждая структура несет в себе указатель на следующий элемент списка. Для реализации связанного списка мы используем `Rc` и `RefCell`, поэтому управлять

указателями будет легко и беспокоиться об отслеживании времени жизни особо не придется.

Мы инициализируем эти итераторы, добавив методы `iter()`, `iter_mut()` и `into_iter()` в `LinkedList`, возвращающий новый экземпляр. Кроме того, мы обновим `append()`, чтобы он снова заработал:

```
impl<T> LinkedList<T> {
    fn new(t: T) -> Self {
        Self {
            head: Rc::new(RefCell::new(ListItem::new(t))),
        }
    }
    fn append(&mut self, t: T) {
        let mut next = self.head.clone();
        while next.as_ref().borrow().next.is_some() { ←
            let n = next
                .as_ref()
                .borrow()
                .next
                .as_ref()
                .unwrap()
                .clone();
            next = n;
        }
        next.as_ref().borrow_mut().next =
            Some(Rc::new(RefCell::new(ListItem::new(t))));
    }
    fn iter(&self) -> Iter<T> {
        Iter {
            next: Some(self.head.clone()),
        }
    }
    fn iter_mut(&mut self) -> IterMut<T> {
        IterMut {
            next: Some(self.head.clone()),
        }
    }
    fn into_iter(self) -> IntoIter<T> {
        IntoIter {
            next: Some(self.head.clone()),
        }
    }
}
```

Нам необходимо развернуть Option внутри RefCell и Rc, в связи с чем для доступа к внутреннему указателю next нужно получить ссылку с помощью `as_ref()` и выполнить заимствование с помощью `borrow()`

Всего нужно выполнить три заимствования: два из текущего next и одно из следующего. После этого можно будет клонировать указатель

Отлично! Мы обновили `append()`, и он больше не использует старую реализацию `Iterator`, которую мы уже определили как неполноценную. Теперь осталось реализовать трейт `Iterator` для `Iter`, `IterMut` и `IntoIter`:

```
impl<T> Iterator for Iter<T> {
    type Item = ItemData<T>;
    fn next(&mut self) -> Option<Self::Item> {
        match self.next.clone() {
            Some(ptr) => {
```

```

        self.next.clone_from(&ptr.as_ref().borrow().next);
        Some(ptr.as_ref().borrow().data.clone())
    }
    None => None,
}
}
impl<T> Iterator for IterMut<T> {
    type Item = ItemData<T>;
    fn next(&mut self) -> Option<Self::Item> {
        match self.next.clone() {
            Some(ptr) => {
                self.next.clone_from(&ptr.as_ref().borrow().next);
                Some(ptr.as_ref().borrow().data.clone())
            }
            None => None,
        }
    }
}
impl<T> Iterator for IntoIter<T> {
    type Item = ItemData<T>;
    fn next(&mut self) -> Option<Self::Item> {
        match self.next.clone() {
            Some(ptr) => {
                self.next.clone_from(&ptr.as_ref().borrow().next);
                Some(ptr.as_ref().borrow().data.clone())
            }
            None => None,
        }
    }
}
}

```

Реализация `next()` весьма проста: мы возвращаем указатель на данные структуры `ListItem`, обновляем `self.next` с помощью очередного элемента списка и возвращаем `None`, когда записей не остается. Вы могли заметить, что все три реализации идентичны. Но ситуация еще хуже: все они возвращают `Rc<RefCell<T>`, а не нужные нам `T`, `&T` и `&mut T`. В возвращении `Rc<RefCell<T>` нет ничего плохого, но он не соответствует шаблону, и нам все равно нужно развернуть данные, чтобы получить доступ к ним.

Решить эту проблему несложно, но мы попробуем исправить ситуацию, используя `IntoIter` и `Vec`. Сигнатура метода `into_iter()` в `Vec` выглядит так:

```
fn into_iter(self) -> slice::IterMut<'a, T>;
```

Если мы посмотрим внимательно, то увидим, что метод получает `self` по значению. Иными словами, вызов `into_iter()` получает владение `Vec` (потребляет его). Зная это, можно изменять `IntoIter` таким образом, чтобы он потреблял каждый элемент списка:

```
impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<Self::Item> {
```

```

        match self.next.clone() {
            Some(ptr) => {
                self.next = ptr.as_ref().borrow().next.clone();
                let listitem =
                    Rc::try_unwrap(ptr).map(|refcell| refcell.into_inner());
                match listitem {
                    Ok(listitem) => Rc::try_unwrap(listitem.data)
                        .map(|refcell| refcell.into_inner())
                        .ok(),
                    Err(_) => None,
                }
            }
            None => None,
        }
    }
}

```

Код становится все сложнее. Разберем его.

- Оба наших указателя на каждый элемент (или узел) в связанном списке, а также данные хранятся в RefCell внутри Rc (то есть Rc<RefCell<T>>).
- Нам нужно использовать `try_unwrap()` в отношении Rc, чтобы извлечь внутренний RefCell из Rc, так как мы хотим применить его. `try_unwrap()` действует на Rc только при отсутствии других ссылок. Мы не собираемся раскрывать эти ссылки за пределами связанного списка, поэтому можем быть уверены, что других ссылок нет.
- Когда мы достанем RefCell из Rc с помощью `try_unwrap()`, нужно будет переместить T из RefCell<T>. Для этого мы вызываем `into_inner()`, который потребляет RefCell, возвращающий находящийся во владении T.
- Возвращаемый тип определяется type Item = T, являющимся ассоциируемым типом, и мы ссылаемся на него с помощью Self::Item, которому необходим трейт Iterator.

Протестируем наш код:

```

let mut dinosaurs = LinkedList::new("Tyrannosaurus Rex");
dinosaurs.append("Triceratops");
dinosaurs.append("Velociraptor");
dinosaurs.append("Stegosaurus");
dinosaurs.append("Spinosaurus");
dinosaurs
    .into_iter()
    .for_each(|data| println!("data={}", data));

```

Тест проходит ожидаемым образом и дает следующий результат:

```

data=Tyrannosaurus Rex
data=Triceratops
data=Velociraptor
data=Stegosaurus
data=Spinosaurus

```

Неплохо! Еще раз взглянем на реализации `Iter` и `IterMut`, так как они по-прежнему не возвращают `&T` или `&mut T` нужным нам образом. В отличие от `into_iter()` методы `iter()` и `iter_mut()` в `LinkedList` не потребляют `self`. Они получают ссылки на `self` (`&self` и `&mut self` соответственно), что все немного усложняет.

В стабильном Rust `RefCell` не дает способа получить простую ссылку на объект, который он содержит. В ночной версии Rust обертки `Ref` и `RefMut` предоставляют метод `leak()`, но мы попробуем обойтись без него.

**ПРИМЕЧАНИЕ** Есть три канала выпуска Rust:

- ночной (nightly);
- бета (beta);
- стабильный (stable).

К сожалению, единственный способ реализовать желаемое — использовать `unsafe`. Если мы взглянем на реализации из стандартной библиотеки коллекций Rust, то увидим, что в них в разных местах используется `unsafe`, например во внутренней реализации `next()` из трейта `Iterator`.

Нам нужно обновить структуры `Iter` и `IterMut`, чтобы добавить время жизни '`a` для возвращаемой ссылки. Вдобавок мы сохраним копию указателя на возвращаемые данные, чтобы они существовали до тех пор, пока итератор находится в области действия. Для отображения времени жизни '`a` в структуре мы используем поле `PhantomData`:

```
struct Iter<'a, T> {
    next: Option<ListItemPtr<T>>,
    data: Option<ItemData<T>>,
    phantom: PhantomData<&'a T>,
}
struct IterMut<'a, T> {
    next: Option<ListItemPtr<T>>,
    data: Option<ItemData<T>>,
    phantom: PhantomData<&mut 'a T>,
}
```

### Время жизни

Механизм времени жизни позволяет сохранить валидность ссылок в течение определенного срока, препятствуя появлению висячих ссылок (аналог висячих указателей в C и C++). В Rust принцип времени жизни был введен, чтобы модуль проверки заимствований мог убеждаться в валидности ссылки на этапе компиляции и программисты могли сообщать эту информацию компилятору. Обозначается время жизни апострофом ('), сопровождаемым именем, например, так: '`a`', '`b`' и '`c`'.

Поначалу принцип времени жизни понять сложно, но по мере наработки опыта вы поймете, что он достаточно прост. Вот несколько важных моментов, которые нужно иметь в виду.

- ◆ Время жизни переменной — это период, в течение которого она остается валидной, начиная с момента ее создания и заканчивая моментом уничтожения.
- ◆ Ссылка валидна в течение времени '`a`', где '`a`' — произвольно выбранное имя, смысл которого состоит только в обозначении этого времени жизни.
- ◆ Ссылка валидна в течение времени жизни объекта, на который она ведет, или времени существования области, в которой была создана, — в зависимости от того, какой из этих периодов окажется короче.
- ◆ Иногда нужно объявлять время жизни явно, чтобы компилятор понял связь между ссылками. В иных случаях он может вывести его самостоятельно (обычно это происходит по умолчанию).
- ◆ Если компилятор не может вывести время жизни, то выдает ошибку, требуя указать его явно.
- ◆ Время жизни всегда существует в контексте ссылки и всегда связано с ней. При отсутствии ссылки время жизни не нужно, и если его не определить явно, то компилятор выведет его автоматически.

Время жизни обычно прописывается в функции, структуре или трейте. Место указания определяет область действия этого времени. Если вы введете время на уровне функции, то оно окажется действительным в течение ее жизни (то же касается структуры, трейта и т.д.). Рассмотрим небольшую программу, в которой вводятся функции `print_without_lifetim()` и `print_with_lifetim()`:

```
fn print_without_lifetim(s: &str) {  
    println!("{}", s);  
}  
  
fn print_with_lifetim<'a>(s: &'a str) {  
    println!("{}", s);  
}  
  
fn main() {  
    print_without_lifetim("calling print_without_lifetim()");  
    print_with_lifetim("calling print_with_lifetim()");  
}
```

Эти функции идентичны, за исключением того, что в `print_with_lifetim()` для ссылки в строке `s` явно указано время жизни '`a`'. Компилятор выведет время жизни для `print_without_lifetim()`, но для `print_with_lifetim()` мы определяем его явно.

При добавлении времени жизни '`a`' в сигнатуру функции компилятор получает сообщение, что ссылка будет валидна на протяжении существования этой функции, а в нашем случае это означает просто один ее вызов.

Если вы решите добавить время жизни к определению структуры, то оно будет валидно в течение существования объекта этой структуры. Разберем пример:

```
struct RefStruct<'a> {
    s_ref: &'a str,
}

fn main() {
    let dog = "dog";
    let dog_struct = RefStruct { s_ref: dog }; ← Структура dog_struct  
не должна существовать  
дольше dog
    println!("I am a {}", dog_struct.s_ref)
}
```

В этом коде время жизни 'a вводится на уровне структуры, то есть ссылка s\_ref будет валидна, пока существует структура RefStruct. Теперь можно поместить ссылку на dog в структуру RefStruct и выводить при условии, что dog существует дольше dog\_struct.

Если этот принцип пока вам не до конца понятен, то не волнуйтесь. По мере получения опыта ситуация прояснится. Дополнительную информацию по теме времени жизни можно найти на странице <https://mng.bz/QZ91>.

Помимо всего прочего, нам нужно инициализировать в `iter()` и `iter_mut()` новые поля `data` и `phantom`:

```
impl<T> LinkedList<T> {
    fn iter(&self) -> Iter<T> {
        Iter {
            next: Some(self.head.clone()),
            data: None,
            phantom: PhantomData,
        }
    }

    fn iter_mut(&mut self) -> IterMut<T> {
        IterMut {
            next: Some(self.head.clone()),
            data: None,
            phantom: PhantomData,
        }
    }
}
```

Теперь можно реализовать метод `next()` для обоих этих методов:

```
impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;
    fn next(&mut self) -> Option<Self::Item> {
        match self.next.clone() {
            Some(ptr) => {
                self.next = ptr.as_ref().borrow().next.clone();
                self.data = Some(ptr.as_ref().borrow().data.clone());
            }
        }
    }
}
```

```

        unsafe { Some(&*self.data.as_ref().unwrap().as_ptr()) }
    }
    None => None,
}
}
impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;
    fn next(&mut self) -> Option<Self::Item> {
        match self.next.clone() {
            Some(ptr) => {
                self.next = ptr.as_ref().borrow().next.clone();
                self.data = Some(ptr.as_ref().borrow().data.clone());
                unsafe { Some(&mut *self.data.as_ref().unwrap().as_ptr()) }
            }
            None => None,
        }
    }
}
}

```

Как видите, для получения желаемого нам нужно выполнить приведение указателей. Сначала мы используем в отношении `RefCell` метод `as_ptr()`, чтобы получить `*mut T`. Затем разыменовываем этот указатель. Потом получаем другую ссылку. Это не очень красивое решение, но оно работает. Нужно помнить, что такая структура не является потокобезопасной. Наконец, мы можем ее протестировать и получить ожидаемый результат:

```

let mut dinosaurs = LinkedList::new("Tyrannosaurus Rex");
dinosaurs.append("Triceratops");
dinosaurs.append("Velociraptor");
dinosaurs.append("Stegosaurus");
dinosaurs.append("Spinosaurus");
dinosaurs
    .iter()
    .for_each(|data| println!("data={}", data));

dinosaurs
    .iter_mut()
    .for_each(|data| println!("data={}", data));

```

Есть еще один нюанс. Нам нужно добавить трейт `IntoIterator` и удалить предыдущий блок `impl<T> Iterator for LinkedList<T> {}`. Так мы сможем перебрать список, используя цикл `for`:

```

impl<'a, T> IntoIterator for &'a LinkedList<T> {
    type IntoIter = Iter<'a, T>;
    type Item = &'a T;
    fn into_iter(self) -> Self::IntoIter {
        self.iter() ←
            Обертывает iter()
            вокруг LinkedList
    }
}

```

```

impl<'a, T> IntoIterator for &'a mut LinkedList<T> {
    type IntoIter = IterMut<'a, T>;
    type Item = &'a mut T;
    fn into_iter(self) -> Self::IntoIter {
        self.iter_mut() ← Обертывает iter_mut()
    }
}

impl<T> IntoIterator for LinkedList<T> {
    type IntoIter = IntoIter<T>;
    type Item = T;
    fn into_iter(self) -> Self::IntoIter {
        self.into_iter() ← Обертывает into_iter()
    }
}

```

Здесь параметр времени жизни 'a не нужен, так как далее этот код не используется

Далее мы протестируем этот код, используя обычный цикл `for`:

```

for data in &linked_list {
    println!("with for loop: data={}", data);
}

```

Компилятор понимает, какую из реализаций `IntoIterator` использовать, исходя из переданного в цикл `for` типа. Здесь мы передаем `&linked_list`, поэтому компилятор использует форму, которая возвращает `&T`, вызывая метод `iter()` для `LinkedList`.

Реализация итераторов открывает доступ ко множеству встроенных функций, таких как `for_each()`, `map()`, `reduce()`, `filter()`, `zip()` и `fold()`. Кроме того, вы можете использовать `for ... {}` со структурами, которые реализуют `IntoIterator` или `Iterator`.

**ПРИМЕЧАНИЕ** Я обычно предпочитаю использовать вместо синтаксиса цикла `for...{}` метод `for_each()`, хотя с функциональной точки зрения оба этих подхода равнозначны. Метод `for_each()` получает в качестве аргумента функцию, то есть вы можете напрямую передать ей замыкание или другую функцию. Тем не менее в особых случаях, например при использовании `async/await`, необходимо использовать цикл `for`, а не метод `for_each()`.

### 3.2.5. Возможности итераторов

Кратко рассмотрим их. Вот пример `map()`:

```

let arr = [1, 2, 3, 4];
println!("{:?}", arr);
let vec: Vec<_> = arr.iter().map(|v| v.to_string()).collect();
println!("{:?}", vec);

```

Для начала инициализируем массив с целыми числами. Затем преобразуем эти числа в строки целых чисел (то есть выведем их в виде строк). Для этого отобразим каждое значение в строку, используя `map()`. Последняя получает

в качестве аргумента функцию, то есть является функцией высшего порядка. Ее сигнатура выглядит так:

```
fn map<B, F>(self, f: F) -> Map<Self, F>
where
    F: FnMut(Self::Item) -> B,
{ ... }
```

Метод `map()`, согласно ограничениям трейта, получает функцию с одним параметром, `Self::Item`. Как мы помним из разбора трейта `Iterator`, `Self::Item` определяется самим итератором. В случае среза, массива или `Vec` это будет `&T`. Эта функция может возвращать любой тип, о чем говорит обобщенный параметр `B`. Самое же интересное в `map()` то, что функция просто возвращает другой итератор, на этот раз особый, с именем `Map`. Его нам предоставляет Rust. Мы передаем в `map()` замыкание, но могли также передать функцию `i32::to_string()` непосредственно в качестве аргумента.

**СОВЕТ** Итераторы в Rust всегда по возможности используют отложенное вычисление, как в случае с `map()`. То есть результат не вычисляется, пока вы этого не потребуете явно (например, вызвав `collect()`).

Наконец, мы вызываем `collect()`, преобразуя итератор в коллекцию — обычно `Vec`. Вы заметите, что нам нужно сообщить компилятору целевой тип, поскольку сам он его не определит. При выполнении предыдущего кода мы получим такой вывод:

```
[1, 2, 3, 4]
["1", "2", "3", "4"]
```

Теперь предположим, что нам нужно проделать нечто более сложное. Возможно, мы хотим преобразовать `Vec` в `LinkedList` из стандартной библиотеки Rust, попутно применив трансформацию. Повторно используем второй `Vec` из предыдущего примера и превратим строки обратно в целые числа, применяя парсинг:

```
let linkedlist: LinkedList<i32> =
    vec.iter().flat_map(|v| v.parse::<i32>()).collect();
println!("{}:", linkedlist);
```

Здесь мы сделали кое-что новое, использовав `flat_map()` вместо `map()`. Почему мы так поступили? Потому что `String::parse()` возвращает `Result`, и нам нужно выровнять результат парсинга. Можно было вызвать `unwrap()` после этой операции, но `flat_map()` немного чище и обрабатывает ошибки более изящно (отбрасывая их).

Если точнее, то `flat_map()` выравнивает `Result`, вызывая метод `Result::into_iter()`, который возвращает итератор для значения `Ok`, если оно есть, и пустой итератор, если нет. Когда `Result` выровнен, значение `Err` игнорируется.

Проблема в том, что если в процессе парсинга возникнет ошибка, то мы можем ее не перехватить. Но волноваться не стоит, нас подстрахует `partition()`:

```
let arr = ["duck", "1", "2", "goose", "3", "4"];
let (successes, failures): (Vec<_>, Vec<_>) = arr
    .iter()
    .map(|v| v.parse::<i32>())
    .partition(Result::is_ok);
println!("successes={:?}", successes);
println!("failures={:?}", failures);
```

Здесь мы получаем список строк и пытаемся спарсить каждую в целое число. Мы смогли получить `duck` и `goose` (это не целые числа), поэтому их парсинг не увенчается успехом. Нам нужно *разделить* результат парсинга, что мы и сделаем с помощью `Result::is_ok()`, которая возвращает `true`, если результат `Ok`. При выполнении предыдущего кода мы получим такой вывод:

```
successes=[Ok(1), Ok(2), Ok(3), Ok(4)]
failures=[Err(ParseIntError { kind: InvalidDigit }), Err(ParseIntError { kind: InvalidDigit })]
```

Как мы видим, наши успехи и провалы по-прежнему обернуты в `Result`, что вполне разумно, поскольку мы их не разворачивали. Сделать это можно вот так:

```
let successes: Vec<_> =
    successes.into_iter().map(Result::unwrap).collect();
let failures: Vec<_> =
    failures.into_iter().map(Result::unwrap_err).collect();
println!("successes={:?}", successes);
println!("failures={:?}", failures);
```

Обратите внимание, что мы вызываем для нашего `Vec` итератор `into_iter()`, поскольку при развертывании `Result` нам также нужно получить владение им. Если помните, `into_iter()` потребляет `Vec` и его содержимое. При выполнении предыдущего кода мы получим такой вывод:

```
successes=[1, 2, 3, 4]
failures=[ParseIntError { kind: InvalidDigit }, ParseIntError { kind: InvalidDigit }]
```

Отлично! Все так, как и должно быть.

**СОВЕТ** Страйтесь избегать использования конструкций наподобие циклов `for` и `while`, задействуя коллекции с итераторами. Вместо цикла `for` можете применять `for_each()`, а вместо `while` – `map_while()`.

С помощью итераторов можно создавать довольно сложные цепочки операций. Кроме того, в Rust есть ряд специализированных итераторов для обработки более сложных задач, таких как подсчет с помощью `Enumerate`. В примере ниже показан вариант использования `Enumerate` со списком пород собак:

```

let popular_dog_breeds = vec![
    "Labrador",
    "French Bulldog",
    "Golden Retriever",
    "German Shepherd",
    "Poodle",
    "Bulldog",
    "Beagle",
    "Rottweiler",
    "Pointer",
    "Dachshund",
];
let ranked_breeds: Vec<_> =
    popular_dog_breeds.into_iter().enumerate().collect();
println!("{:?}", ranked_breeds);

```

При выполнении этого кода мы получим такой вывод:

```

[(0, "Labrador"), (1, "French Bulldog"), (2, "Golden Retriever"),
(3, "German Shepherd"), (4, "Poodle"), (5, "Bulldog"), (6, "Beagle"),
(7, "Rottweiler"), (8, "Pointer"), (9, "Dachshund")]

```

Довольно близко, но не совсем то, что нужно. Есть смысл начать отсчет не с 0, а с 1. Для получения нужного результата достаточно внести небольшое изменение:

```

let ranked_breeds: Vec<_> = popular_dog_breeds
    .into_iter()
    .enumerate()
    .map(|(idx, breed)| (idx + 1, breed))
    .collect();

```

Мы добавили после `enumerate()` метод `map()`, чтобы распаковать созданный этим итератором кортеж и вернуть его с добавлением в индекс 1. Теперь мы получили нужный результат:

```

[(1, "Labrador"), (2, "French Bulldog"), (3, "Golden Retriever"),
(4, "German Shepherd"), (5, "Poodle"), (6, "Bulldog"), (7, "Beagle"),
(8, "Rottweiler"), (9, "Pointer"), (10, "Dachshund")]

```

А что, если мы захотим выполнить не прямой отсчет, а обратный? Тогда можно реверсировать список с помощью `rev()`:

```

let ranked_breeds: Vec<_> = popular_dog_breeds
    .into_iter()
    .enumerate()
    .map(|(idx, breed)| (idx + 1, breed))
    .rev()
    .collect();

```

Итераторы — одна из моих любимых абстракций в Rust. Поразительно, насколько быстро можно получить из довольно небрежной структуры данных полноценную коллекцию, просто реализовав несколько трейтов итератора.

**СОВЕТ** Полный список всех возможностей, предоставляемых итераторами Rust, доступен в документации стандартной библиотеки по ссылке <https://doc.rust-lang.org/std/iter/index.html>.

Благодаря итераторам и замыканиям Rust предоставляет все необходимое для легкого написания чисто функционального кода. Модель использования памяти в Rust несколько усложняет выполнение определенных задач, которые в других языках реализуются проще, но почти ни один язык не может соперничать с Rust в плане возможностей, безопасности и производительности.

## Резюме

- Сопоставление с шаблоном позволяет разворачивать структуры данных и обрабатывать различные сценарии гораздо чаще, чем при использовании комбинаций операторов `if/else`.
- Сопоставление с шаблоном можно использовать вместе с оператором `?`, чтобы изящно обрабатывать ошибки и разворачивать или деструктурировать значения.
- При сопоставлении с шаблоном можно деструктурировать вложенные структуры и перечисления, а также делать сопоставление со значениями.
- Rust допускает применение паттернов функционального программирования, в частности, с помощью итераторов и замыканий. Освоение этих паттернов поможет вам использовать его более эффективно.
- Итераторы используют гибкий (fluent) интерфейс и вместе с замыканиями позволяют легко выражать операции и мутации в структурах данных.
- Итераторы обычно содержат ссылку на данные (например, заимствованные) или используют перемещение для извлечения элементов из обрабатываемой последовательности.
- Как правило, метод `iter()` возвращает итератор со ссылками, а `into_iter()` дает итератор, который забирает владение.

## Часть II

# *Основные паттерны*

Основные паттерны — это те, которые программисты используют очень часто, поэтому крайне важно хорошо в них разобраться. Кроме того, нужно научиться говорить на языке паттернов так, чтобы создавать системы и структуры, понятные другим людям.

Но порой следует вспоминать, что паттерны не являются целью как таковой. Иногда нужно взглянуть на проделанную работу с более высокого уровня, чтобы убедиться в адекватном использовании паттернов и правильном понимании решаемой задачи.

Задача проектирования программного обеспечения редко заключается в использовании всех возможностей языка или написании максимального количества строк кода. Напротив, она состоит в решении задач и создании долгосрочной ценности. Паттерны — это инструменты, которые помогают достичь этой цели, но в нашем распоряжении есть не только они. Иногда оптимальным решением будет самое простое или понятное. К тому же порой мы пишем программы просто ради развлечения, и это тоже вполне нормально.

# 4

## Паттерны в Rust

### В этой главе

- ✓ Принцип «Получение ресурсов есть инициализация».
- ✓ Передача аргументов по значению и по ссылке и различия этих способов.
- ✓ Использование конструкторов.
- ✓ Видимость членов объекта и доступ к ним.
- ✓ Обработка ошибок.
- ✓ Управление глобальным состоянием с помощью `lazy-static.rs`, `OnceCell` и `static_init`.

Теперь вы готовы к знакомству с конкретными паттернами. Начнем с простейших тем: RAII, передача значений, конструкторы и видимость. В этой главе мы обсудим множество тем, однако основное внимание уделим небольшим паттернам, которые будем часто использовать.

Кроме того, мы поговорим о *крейтах* — библиотеках, разрабатываемых участниками сообщества Rust. Этот язык основывается на крейтах, и они играют в нем важнейшую роль. Без них вам не обойтись. Конечно, можно поддаться синдрому неприятия чужой разработки, отказавшись от использования крейтов, но я не советую так делать. Даже крупнейшие организации с огромным финансированием при формировании технологических стеков активно используют программное обеспечение с открытым исходным кодом.

Работая с Rust, вы быстро поймете, что стандартная библиотека неполноценна и лишена множества возможностей, которыми должен обладать современный язык. И эта ограниченность заложена самим дизайном. Команда Rust предпочла сделать стандартную библиотеку минималистичной и обеспечивать дополнительную функциональность с помощью крейтов. Такой подход имеет несколько преимуществ:

- стандартная библиотека получилась небольшой и ее легче обслуживать;
- она более стабильна и изменения в ней менее вероятны;
- она больше ориентирована на основную функциональность;
- сообщество может создавать и обслуживать отдельные конкурирующие крейты, реализующие специальную функциональность, что позволит разработчикам выбирать из них наиболее подходящие для своих задач.

Если вы хотите работать исключительно с проприетарным программным обеспечением, то обращайте внимание на лицензии, закрепленные за каждым крейтом. Эта книга скорее образовательный ресурс, поэтому предположу, что вы готовы использовать ПО с открытым исходным кодом, лицензии которого могут запрещать коммерческое или проприетарное применение. Основная же часть крейтов Rust доступна по свободным лицензиям, которые разрешают практически любое использование.

## **4.1. Получение ресурса есть инициализация**

Идиома «*получение ресурса есть инициализация*» (resource acquisition is initialization, RAII) появилась в C++ и является, пожалуй, одной из важнейших в современном программировании. RAII — ключевая особенность Rust. Она позволяет уверенно реализовывать другие паттерны и играет важнейшую роль в обеспечении безопасности.

В сообществе витают некоторые сомнения относительно того, чем же конкретно является RAII: идиомой или паттерном. Я буду описывать этот принцип как паттерн, поскольку он представляет собой формализованный способ обработки ресурсов в программе, отличаясь от более неформального подхода к формированию кода. Кроме того, RAII влияет на общую структуру и архитектуру программы, что больше характеризует паттерн, чем идиому.

### **4.1.1. RAII в C и C++**

В этом подразделе я вкратце объясню принцип RAII на случай, если ранее вы с ним не сталкивались. Для любого опытного программиста эта информация окажется просто напоминанием хорошо известного механизма. Здесь мы рассмотрим код на C и C++, поскольку паттерн RAII зародился именно в C++ в качестве улучшения механизмов C. Не беспокойтесь, если эти языки

вам незнакомы. Примеры будут простыми, и глубоко вникать в них не потребуется.

RAII с помощью стека внутри конкретной области определяет, когда можно освободить ресурсы (такие как переменные). Название этого паттерна может сбивать с толку, так как RAII обычно рассматривался как способ обработки освобождения ресурсов, а не их получения и инициализации, как предполагает его название. Но эти процессы связаны, поэтому я дам небольшое пояснение. Для начала посмотрим, что происходит, если объявить простую переменную внутри функции на C:

```
void func() {
    int a;
    // Здесь идет код, выполняющий какие-то действия с a.
}
```

Здесь мы объявляем переменную `a`. Несмотря на то что мы *объявили* переменную в функции, мы ее не *инициализировали*, для чего нужно присвоить ей значение. То есть значение `a` в нашем примере не определено. Как правило, вы будете встречать код, подобный следующему фрагменту C, обрабатывающему и объявление, и инициализацию:

```
void func() {
    int a = 0;
}
```

Этот код *объявляет и инициализирует* `a` со значением `0`. Теперь мы знаем, что в момент объявления `a` равна `0`. Когда функция вернет результат, `a` выйдет из области видимости и будет удалена из стека, то есть переменная освободится. В C при освобождении переменной ничего особенного не происходит.

Хорошо, а если `a` является указателем? Иными словами, если `a` указывает на какую-то область памяти, то что произойдет при ее освобождении? В C у нас может быть такой код:

```
void func() {
    int *a = malloc(sizeof(int));
}
```

Этот код вызывает утечку памяти, так как мы выделяем ее из кучи с помощью `malloc()` и присваиваем возвращенный этой функцией адрес переменной `a`. Обратите внимание, что `sizeof(int)` содержит размер `int` в байтах. Количество байтов зависит от платформы, но обычно составляет 4.

Когда эта функция возвращает результат, указатель `a` освобождается, но блоки памяти, которые он адресует, — нет. То есть мы создали утечку памяти. Решить это можно так: вызвать `free()`, чтобы освободить адрес, на который ссылается `a`, перед возвращением ответа функции.

Но есть проблема. Что, если мы можем возвращать результаты из нескольких мест функции? Предположим, мы написали такой код:

```
void leaky_func() {
    FILE *fp;
    int *a = malloc(sizeof(int));
    *a = 0; ← Инициализирует значение a равным 0

    // Попытка открыть файл для чтения.
    fp = fopen("file.txt", "r");
    if (fp == NULL) {
        // Ошибка!
        return;
    }

    // Теперь можно прочитать файл по указателю fp.
    // ...

    fclose(fp); ← Закрывает указатель на файл
    free(a); ← Освобождает память, на которую указывает a
}
```

Функция `leaky_func` открывает файл для чтения, но в случае сбоя (например, если файл не существует) завершается раньше. В добавок мы ввели утечку памяти, поскольку при сбое не освобождаем ее из-под `a`. Это классический случай утечки памяти и один из недостатков работы с языками наподобие С.

Разработчики C++, помимо прочего, также хотели затруднить возникновение утечек памяти, и одним из способов добиться этого стало использование *конструкторов и деструкторов*. При создании в C++ класса или структуры всегда вызывается конструктор, а при уничтожении объекта — деструктор. Когда вы создаете объект в стеке, конструктор и деструктор вызываются автоматически. Если же объект создается в куче, то нужно использовать ключевые слова `new` и `delete` для освобождения памяти и вызова конструкторов или деструкторов соответственно. Эти ключевые слова в C++ аналогичны `malloc()` и `free()` в С. Они не решают проблему с утечкой памяти, но RAII помогает избежать этого с помощью умных указателей.

*Умный указатель* — особый вид указателя, который предоставляет конструктор, обертывающий `new`, и деструктор, обертывающий `delete`. Компилятор гарантирует, что для любой переменной, выходящей из области видимости, будет вызван деструктор, поэтому на основе этого поведения можно устраниить целый класс утечек памяти, но только если всегда использовать умные указатели.

Ситуацию еще больше усложняет то, что C++ обратно совместим с С, то есть код С можно считать полностью валидным кодом C++. В связи с этим C++ позволяет вам совершить ошибку в той же степени, что и С, вопреки всем этим конструкторам, деструкторам и умным указателям.

Как вы могли догадаться, хоть C++ и предоставил инструменты для решения одного класса проблем с утечкой памяти, разработчики не всегда использовали эти инструменты правильно (а порой и не использовали вовсе). Так что этот язык добился лишь небольших успехов в решении этой проблемы. Код C++, эквивалентный предыдущему коду C, на этот раз с использованием `std::shared_ptr` вместо простого указателя C, будет выглядеть так:

```
#include <fstream>
#include <memory>

void func() {
    std::shared_ptr<int> a(new int(0));
    std::ifstream stream("file.txt");
    if (!stream.is_open()) {
        // Ошибка!
        return;
    }
    // Теперь можно выполнять чтение файла.
    // ...
}
```

Обратите внимание, что мы используем для нашего указателя `a` тип `std::shared_ptr`, исключая утечку памяти. Теперь неважно, когда функция вернет результат, так как компилятор гарантирует, что при возвращении для `a` всегда будет выполняться деструктор, освобождающий память. Даже если возникнет исключение, деструктор гарантированно выполнится.

### Области видимости в C

В старых версиях C вы могли объявлять переменные только в начале функции или на уровне файлов. Объявлять их внутри цикла `for` было нельзя. Вот пример:

```
void old_C_func() {
    int a;

    for (a = 0; a < 10; a++) {
        // OK
    }

    for (int b = 0; b < 10; b++) {
        // Нельзя. b находится в области видимости блока.
    }
}
```

Принципа «область видимости блока», как в этом примере, не было в C вплоть до 1989 года, пока не появился ANSI C, хотя некоторые компиляторы могли придерживаться его и ранее. В C есть три основных типа области видимости:

- ◆ *область функции* — переменные, объявленные на уровне функции;
- ◆ *область блока* — переменные, объявленные в блоке кода;
- ◆ *область файла* — переменные, объявленные в файле.

Переменные внутри блоков можно вкладывать и затенять (*shadowing*). К примеру, следующий код является валидным:

```
void shadowing() {
    int a = 0;
    {
        int a = 1;
        printf("inner a=%d\n", a);
    }
    printf("outer a=%d\n", a);
}
```

В этом примере мы затеняем `a`, объявляя ее дважды: на уровне функции и снова внутри блока кода. При выполнении этого кода мы получим такой вывод:

```
inner a=1
outer a=0
```

В Rust используется блочная область видимости, а также есть возможность затенять переменные. Правила области видимости в этом языке аналогичны правилам современного C и C++. К тому же в нем есть дополнительные правила обработки перемещений, времени жизни и заимствования. Примечательно, что в Rust переменная может существовать дальше области, в которой была объявлена, если ее переместить, и этим Rust принципиально отличается от C и C++.

Как компилятор реализует RAII? Он делает это, используя стек, который находится в области функции или блока и обычно обозначается фигурными скобками (`{ ... }`). Каждая новая переменная передается в стек при входе в конкретную область видимости (например, функцию). При выходе из этой области переменная выбрасывается из стека. Компилятору нужно сохранять вместе с каждой переменной немного дополнительных данных, чтобы понимать, как безопасно уничтожить каждое значение. Издержки при этом минимальны и обычно сводятся к использованию дополнительного указателя для всего, что требует очистки.

### **4.1.2. RAII в Rust**

В Rust управление объектами выполняется в соответствии с правилами RAII за двумя исключениями: небезопасный код и значения `Copy`. Переменные нужно инициализировать со значением в момент их объявления, и, когда переменная выходит из своей области, она уничтожается через вызов деструктора (о чем мы поговорим ниже).

И хотя процесс инициализации переменных и вызова деструктора может быть скрыт абстракциями слоев косвенной адресации, переменную всегда нужно инициализировать со значением (в отличие от C или C++, где переменные могут быть не инициализированы), и при ее выходе из области видимости всегда вызывается деструктор объекта. В случае простых переменных (которые не являются указателями, включая `Rc` и `Arc`) модуль проверки заимствования

и семантика перемещения помогают понять, когда переменные или объекты выходят из области видимости, а значит, уничтожаются.

Для объектов `Copy` (включая примитивные типы наподобие целых чисел, чисел с плавающей запятой и логических значений, а также простых структур, состоящих только из примитивов) нельзя вызвать деструктор, поскольку они копируются по значению, а не перемещаются. Отсутствие деструкторов для объектов `Copy` — особый случай, и вам нужно иметь его в виду при работе с ними. Для такого объекта нельзя определить или вызвать деструктор. В листинге 4.1 показан принцип работы RAII.

#### Листинг 4.1. Получение ресурсов

```
fn main() {
    let status = String::from("Active"); ← Конструктор строк получает и выделяет
    let statuses = vec![status]; ← память для хранения строки
    println!("{}:?", statuses);
} ← При инициализации владение строкой
      statuses выходит из области
      видимости, освобождая Vec и String
      теперь строка status хранится в statuses
```

Этап конструирования показан на рис. 4.1. Создаются новые объекты, которые скрытым образом выделяют память в куче (один раз для `String` и один для `Vec`). Они инициализируются предоставленными нами значениями и передаются в стек для локальной области видимости. Мы передаем владение оригинальной `status` переменной `statuses`, поэтому `status` в стеке фактически становится недействительной ссылкой. Тем не менее компилятор Rust обрабатывает эту ситуацию прозрачно, так что можно не беспокоиться об этом.

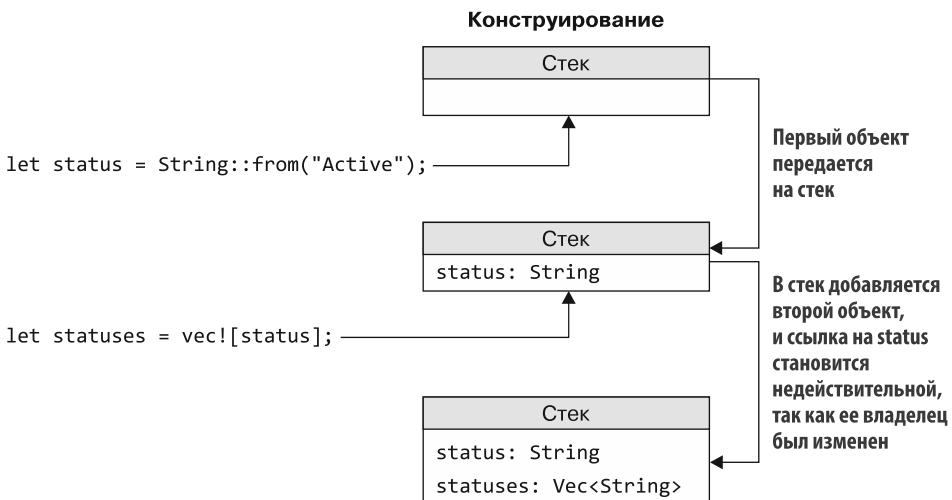
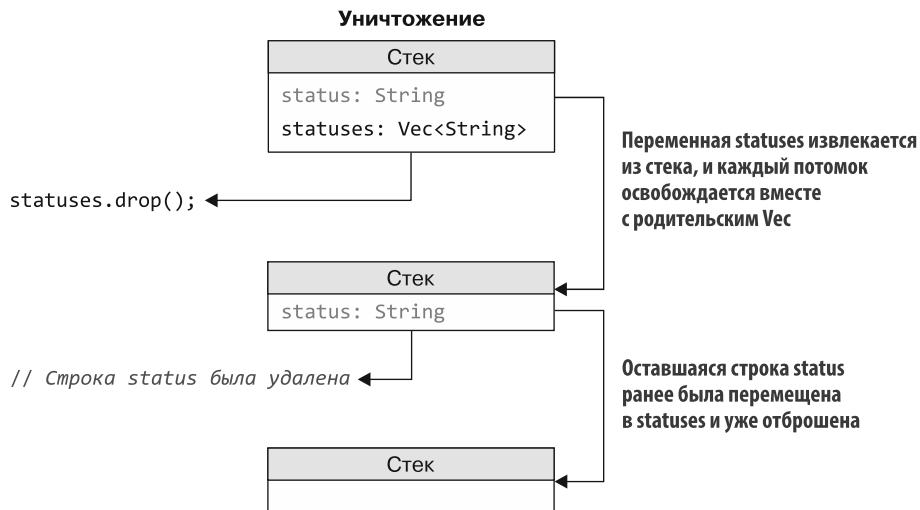


Рис. 4.1. Запись и конструирование RAII

Этап уничтожения схематично показан на рис. 4.2. Новые объекты при удалении из стека уничтожаются по одному. Для контейнеров наподобие `Vec` деструктор автоматически вызывается также и для всех потомков. В итоге наша первичная строка `status` уничтожается вместе со `statuses`, хотя изначальная ссылка `status` больше не актуальна, так как была перемещена.



**Рис. 4.2.** Завершение и уничтожение RAII

В Rust уничтожение обрабатывается автоматически генерируемым деструктором, который также рекурсивно вызывает деструктор каждого члена объекта. Сначала он вызывает метод `drop()` для указанного типа, который определяется третим `Drop` так:

```
pub trait Drop {
    fn drop(&mut self);
}
```

При реализации `Drop` для любого типа его соответствующий метод `drop()` будет гарантированно вызываться, когда переменная этого типа выходит из области видимости. После этого автоматический деструктор рекурсивно вызывает деструкторы каждой переменной из объекта.

В Rust деструкторы всегда вызываются для всех объектов, когда те выходят из области видимости, поэтому вручную вызывать `drop()` не нужно. Кроме того, вы не можете переопределить это поведение, не используя `unsafe`. (То есть в Rust отключить вызов деструкторов нельзя.)

**Особенности использования RAII в Rust.** В отношении RAII нужно помнить ряд ключевых моментов, многие из которых окажутся интуитивно

понятными для тех, кто уже знаком с этой концепцией (например, из C++ и других языков).

- *RAII активно используется в Rust.*
  - В Rust нет функции сбора мусора. Управление памятью происходит явно. Выделение памяти в куче обычно реализуется с помощью `Box` или `Vec`.
  - Сроки жизни объектов детерминированы и известны на этапе компиляции (за исключением случаев использования умных указателей).
  - Объекты, созданные в стеке, подчиняются тем же правилам RAII, что и созданные в куче.
- *Объекты, участвующие в управлении памятью, используют RAII.*
  - `Box` и `Vec` используют RAII для получения, инициализации и освобождения ресурсов памяти.
  - Умные указатели, такие как `Rc` и `Arc`, используют RAII для реализации подсчета ссылок при каждом клонировании и уничтожении указателя.
  - `RefCell` возвращает ссылки на заимствованные данные `Ref` и `RefMut`, которые используют RAII для защиты от одновременного создания нескольких ссылок.
- *RAII использует несколько примитивов синхронизации.*
  - `Mutex::lock()` в случае успеха возвращает `MutexGuard`. `MutexGuard` — это основанный на RAII механизм защиты путем блокировки (lock guard), который автоматически разблокирует мьютекс при уничтожении.
  - `RwLock` возвращает `RwLockReadGuard` или `RwLockWriteGuard`, когда в случае блокировки чтения-записи вы получаете доступ к совместному чтению или эксклюзивной записи соответственно.
  - `Condvar` требует, чтобы `MutexGuard` ожидал уведомления этой условной переменной, как показано в листинге 4.2.

Чтобы продемонстрировать работу RAII в Rust, мы создадим простой пример с двумя потоками, используя `Mutex` и `Condvar` (листинг 4.2). Один поток будет инкрементировать значение и уведомлять об этом второй, основной поток.

В этом примере показаны сразу несколько случаев использования RAII, что может легко запутать. Обобщим все это для лучшего понимания:

- `Mutex` обертывает произвольное значение (в данном случае целое число, но это может быть любой объект), которое освобождается, когда выходит из области видимости;
- `Mutex` и `CondVar` используют обеспечиваемый `MutexGuard` принцип RAII для передачи владения заблокированным мьютексом;
- `Arc` предоставляет потокобезопасный указатель с подсчетом ссылок на мьютекс и условную переменную.

### Листинг 4.2. RAII с использованием Mutex и Condvar

```

use std::sync::{Arc, Condvar, Mutex};
use std::thread;

fn main() {
    let outer = Arc::new(
        (Mutex::new(0), Condvar::new())
    );
    let inner = outer.clone();

    thread::spawn(move || {
        let (mutex, cond_var) = &*inner;
        let mut guard = mutex.lock().unwrap();
        *guard += 1;
        println!("inner guard={guard}");
        cond_var.notify_one();
    });
}

let (mutex, cond_var) = &*outer;
let mut guard = mutex.lock().unwrap();
println!("outer before wait guard={guard}");
while *guard == 0 {
    guard = cond_var.wait(guard).unwrap();
}
println!("outer after wait guard={guard}");
}

```

Объявляем мьютекс для целого числа и условную переменную в кортеже, который на следующей строке клонируем

Перемещаем и разворачиваем внутренний (inner) Arc и его кортеж в отдельную пару мьютекса и условной переменной

Обеспечиваем защиту мьютекса, блокируя его

Инкрементируем целое число, которое обернули мьютексом, чтобы видеть, когда оно изменится

В последнюю очередь во внутреннем (inner) потоке сообщаем условной переменной, что данные готовы

В этот момент запущенный (spawn) поток завершается и внутренний Arc вместе с блокировкой мьютекса покидают область видимости, в результате чего блокировка снимается и указатель inner освобождается

Чтобы считать значение, необходимо получить блокировку мьютекса в основном (outer) потоке

Выполняем бесконечный цикл в потоке outer, пока значение мьютекса не изменится

Для реализации ожидания в условной переменной мы передаем владение блокировкой этой переменной, которая возвращает ее, когда получает уведомление

Когда внутренний (inner) поток завершает выполнение, MutexGuard освобождается, разблокируя мьютекс, и Arc отбрасывается, освобождая указатель на мьютекс и условную переменную. В то же время блокировку мьютекса получает внешний (outer) поток, ожидает уведомления условной переменной и освобождает блокировку, когда находящийся под ней объект (guard) покидает область видимости. Обратите внимание, что мы не знаем, какой поток выполнится первым, поэтому продолжить выполнение можно только после получения уведомления условной переменной. При этом гарантировать определенный порядок выполнения нельзя.

RAII — мощный паттерн, который дает возможность безопасно управлять ресурсами и автоматически производить очистку. Строгие правила владения и заимствования в Rust позволяют легко понимать, когда объекты покидают область видимости и вызываются их деструкторы.

## 4.2. Передача аргументов по значению и по ссылке

На первый взгляд может показаться, что это тема начального уровня. Но при более глубоком знакомстве с программированием на Rust вы поймете, что очень важно хорошо обдумывать, как именно передавать аргументы: по значению или по ссылке. В этом вопросе множество нюансов, но я подскажу, какие распространенные паттерны можно использовать и когда.

### 4.2.1. Передача по значению

В Rust передача аргументов по значению обычно подразумевает перемещение. Проще говоря, *перемещение* происходит, когда вы передаете владение объектом из одной области в другую. Оно может произойти при вызове функции, создании замыкания, присваивании объекта или возвращении значения из функции. Еще одно интересное свойство передачи по значению заключается в том, что оно подчиняется принципу RAII. Простой пример кода, демонстрирующий этот механизм, показан в листинге 4.3.

#### Листинг 4.3. Реверсирование строки, передача по значению

```
fn reverse(s: String) -> String {
    let mut v = Vec::from_iter(s.chars());
    v.reverse(); ← Реверсирует созданный вектор по месту
    String::from_iter(v.iter()) ← Возвращает новую строку путем перебора
} ← Выстраивает Vec путем
      перебора символов в s
```

Перед вами пример функции, которая реверсирует символы строки. Она получает эту строку по значению и возвращает новую. Теперь убедимся, что возвращенное значение действительно является реверсированной версией предоставленного:

```
assert_eq!("abcdefg", reverse(String::from("gfedcba")));
```

Иногда удобно перемещать значения в функцию и сразу же их оттуда извлекать, как в предыдущем примере. Это можно делать во избежание заимствования или клонирования значения. Если нужно вернуть несколько значений, то можно сделать это в форме кортежа:

```
fn reverse_and_uppercase(s: String) -> (String, String) {
    let mut v = Vec::from_iter(s.chars());
    v.reverse();
    let reversed = String::from_iter(v.iter());
    let uppercased = reversed.to_uppercase();
    (reversed, uppercased)
}
```

И хотя в этом примере в функцию передается лишь один аргумент, их можно передать несколько, после чего вернуть. Проверим этот код:

```
assert_eq!(
    reverse_and_uppercase("abcdefg".to_string()),
    ("gfedcba".to_string(), "GFEDCBA".to_string())
);
```

Не бойтесь передавать аргументы по значению, но имейте в виду, что это подразумевает перемещение для любого типа, который не реализует `Copy`, хотя иногда это может быть плюсом.

### 4.2.2. Передача по ссылке

Вы получаете ссылку на объект или переменную, заимствуя ее. Это поведение можно рассматривать аналогично работе указателя, за исключением того, что для ссылки нельзя выполнять побитовые или арифметические операции и передавать или присваивать ее можно только раз, без возможности последующего манипулирования. Ссылки обозначают путем добавления к спецификатору типа символа `&`, и они также могут содержать время жизни, обозначаемое одинарной кавычкой `(')`. Время жизни указывается после `&` и необязательного идентификатора, например так: `&'a String`. Ссылки могут быть неизменяемыми (по умолчанию `&str`) или изменяемыми (`&mut String`). Перепишем нашу функцию реверсирования, на этот раз передав ей ввод по ссылке (листинг 4.4).

#### Листинг 4.4. Реверсирование строки, передача по ссылке

```
fn reverse(s: &str) -> String {
    let mut v = Vec::from_iter(s.chars());
    v.reverse();
    String::from_iter(v.iter())
}
```

Легко заметить, что между этими двумя функциями есть всего одно различие: аргумент `s: String` был заменен на `s: &str`. При тестировании кода можно поступить немного по-другому:

```
assert_eq!("abcdefg", reverse("gfedcba"));
```

Обратите внимание: вместо того чтобы создавать строку с помощью `String::from()`, можно передать статическую строку (например, `&'static str`). Это хороший подход, отличающийся большей эргономичностью. Если мы решим так сделать, то можем вызвать функцию реверсирования следующим образом:

```
assert_eq!(
    "race car", reverse(&String::from("rac ecar")) ←
);
```

Заемствование строки дает `&str`,  
так как `String` для возврата `&str`  
и `&mut str` реализует трейты `Borrow`  
и `BorrowMut` соответственно

А что, если мы решим обновить строку на месте (`in place`)? Это будет чуть сложнее, поскольку не получится легко выполнить подобающее (без копирования) реверсирование. Можно эмулировать это поведение, как показано в листинге 4.5.

Код обеспечивает удовлетворительное быстродействие за счет дополнительных издержек памяти.

#### Листинг 4.5. Реверсирование строки на месте (в каком-то роде)

```
fn reverse_inplace(s: &mut String) {
    let mut v = Vec::from_iter(s.chars());
    v.reverse();
    s.clear();
    v.into_iter().for_each(|c| s.push(c));
}
```

Протестировать этот код можно так:

```
let mut abcdefg = String::from("gfedcba");
reverse_inplace(&mut abcdefg);
assert_eq!("abcdefg", abcdefg);
```

#### Почему в Rust невозможно изменять строки на месте

Вы могли заметить, что в Rust сложно работать со строками на месте. Объясняется просто: в этом языке строки всегда представлены в формате UTF-8, то есть их символы могут содержать несколько байтов или состоять из кластеров графем в стандарте Unicode.

*Графема* — минимальная письменная единица, которой может быть стандартный символ (например, буква a), символ со знаком ударения, например é, или эмодзи. Когда речь идет о строках и символах, мы склонны считать, что один символ на экране равен одному байту, что верно только для символов, относящихся строго к стандарту ASCII.

Кластеры графем могут содержать несколько символов Unicode и несколько байтов, поэтому корректная их обработка усложняется, в связи с чем стандартная библиотека Rust не поддерживает их непосредственную обработку. Вместо этого нужно использовать крейт `unicode-segmentation` (<https://crates.io/crates/unicode-segmentation>).

Если вам необходимо обновить строку на месте путем изменения ее байтов, то у вас есть два варианта:

- ◆ использовать функцию `std::mem::take` для получения доступа к байтам строки и непосредственного управления содержимым буфера;
- ◆ использовать небезопасный метод, такой как `String::as_mut_vec()` или `str::as_bytes_mut()`, возвращающий внутренние байты строки.

Первый способ предпочтительнее, поскольку не требует небезопасного кода. Но в любом случае нужно будет подумать, как безопасно обработать символы UTF-8. Если вы попытаетесь манипулировать байтами строки напрямую, то можете получить неожиданные результаты.

### 4.2.3. Когда передавать по значению, а когда — по ссылке

Поначалу может быть неясно, когда стоит передавать аргумент по ссылке, а когда — по значению через перемещение, поэтому я дам некоторые подсказки, которые помогут вам определиться с выбором. Как и в любом деле, лучшим учителем будет практика. Со временем вы научитесь разбирать, какой паттерн и в какой ситуации окажется более уместным. Если вы уже имеете некоторый опыт или являетесь знатоком Rust (либо языка с похожей семантикой), разница в выборе может быть для вас очевидной. Но все же есть смысл озвучить ее, чтобы вы могли сформировать новые ментальные модели.

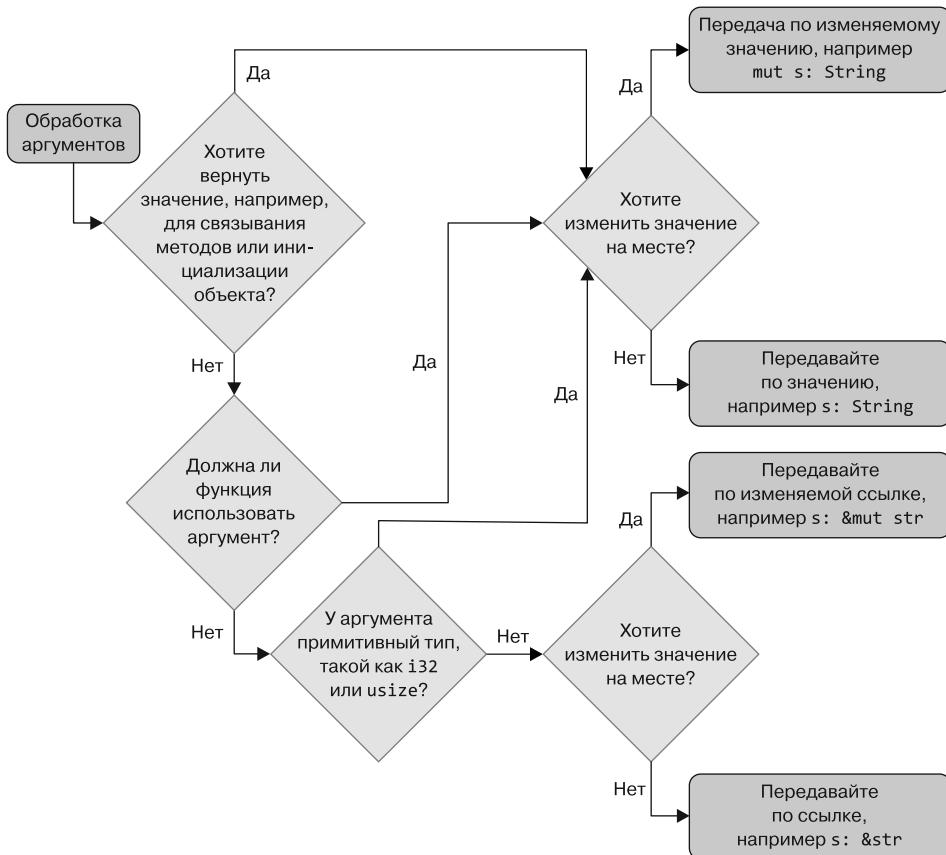
Ситуация с выбором варианта передачи осложняется еще и тем, что методы объектов обычно получают в качестве аргумента `self`, следуя всем тем же правилам: `self` можно передать по значению (путем перемещения) или по ссылке (без перемещения). А это, как вы неоднократно заметите в ходе работы с книгой, может создать интересные паттерны, являющиеся в некотором роде уникальными для Rust. Для начала взглянем на разные способы обработки аргументов (табл. 4.1).

**Таблица 4.1.** Способы передачи аргументов

Аргумент передается по	Префикс	Перемещается?	Владение	Базовый случай
Ссылке	&	Нет	Сохраняется за вызывающим	Вызываемый требует временного доступа к значению
Изменяемой ссылке	<code>&amp;mut</code>	Нет	Сохраняется за вызывающим	Вызываемому нужно изменить значение, не получая владение им
Значению	—	Да	Передается вызываемому	Вызываемому нужно получить владение значением
Изменяемому значению	<code>mut</code>	Да	Передается вызываемому	Вызываемому нужно и получить владение, и изменить значение

Чаще всего требуется передавать аргументы по ссылке. Как правило, вы не будете передавать их таким образом только в двух случаях: при использовании примитивных типов (таких как `i32` и `usize`), а также когда нужно передать владение значением вызываемому коду и, возможно, вернуть то же значение вместе с новым объектом либо отдельно. Но здесь нужно хорошенько подумать о том, почему вы хотите передать владение. Планируете ли вы изменить значение? Если да, то имеется ли причина, по которой вы не можете использовать ссылку (например, во избежание копирования или связывания методов в цепочку)?

Чтобы упростить для вас эту задачу, я создал простую блок-схему, показанную на рис. 4.3. Вы можете использовать ее в качестве руководства по передаче аргументов в большинстве случаев, пока не выработаете достаточное чутье в этом процессе.



**Рис. 4.3.** Принятие решения об обработке аргументов

### 4.3. Конструкторы

Строго говоря, в Rust нет формальной концепции конструкторов, как, например, в C++, C# и Java. Здесь *конструктор* — это просто паттерн проектирования. В нем вы создаете метод, обычно называемый `new()`, который получает любое количество аргументов инициализации и возвращает новый объект сразу же после его создания. И хотя в Rust нет ключевого слова `new`, можете рассматривать этот метод как эквивалент `new` в других языках. Но я специально повторюсь:

в Rust нет такого понятия, как конструктор. Поэтому любые встреченные в этом языке конструкторы являются строго условными паттернами, а не специальными методами, как в C++, C# и Java.

В листинге 4.6 показан простой конструктор, создающий контейнер для моделирования пиццы с топпингом. Обратите внимание, что этот конструктор не дает возможности добавлять топпинги (пока).

#### Листинг 4.6. Моделирование пиццы

```
#[derive(Debug, Clone)]
pub struct Pizza {
    toppings: Vec<String>,
}

impl Pizza {
    pub fn new() -> Self { ←
        Self { toppings: vec![] } ←
    }
}
```

Этот конструктор не получает аргументов и возвращает Self (пустую Pizza)

Создадим пустую Pizza:

```
let pizza = Pizza::new();
println!("pizza={:?}", pizza);
```

При выполнении предыдущего кода мы получим такой вывод:

```
pizza=Pizza { toppings: [] }
```

В простых случаях вам наверняка потребуется инициализировать объекты со значениями, возможно полученными из аргументов конструктора. В Rust `new()` обычно не получает аргументы и возвращает пустой объект, как в случае `Vec::new()`, когда возвращается пустой вектор.

Ничто не запрещает добавлять аргументы инициализации с помощью `new()`, но обычно вместо этого реализуют трейт `From`, когда нужно создать новый объект из другого объекта. Этот подход имеет смысл только при отображении 1:1 (например, когда `String::from(...)` создает новую строку). Теперь перепишем конструктор из листинга 4.6, чтобы можно было инициализировать топпинги для пиццы (листинг 4.7).

#### Листинг 4.7. Доработанный конструктор пиццы

```
impl Pizza {
    pub fn new(toppings: Vec<String>) -> Self { ←
        Self { toppings } ←
    }
}
```

Наш конструктор получает Vec с топпингами и перемещает их в созданную пиццу

Имена аргумента конструктора и члена Pizza совпадают, поэтому можно сократить toppings: toppings до toppings

Проверим наш новый конструктор:

```
let pizza = Pizza::new(vec![
    String::from("tomato sauce"),
    String::from("mushrooms"),
    String::from("mozzarella"),
    String::from("pepperoni"),
]);
println!("pizza={:#?}", pizza);
```

Тестируя код новой пиццы, которая теперь стала намного вкуснее, мы получим следующий вывод:

```
pizza=Pizza {
    toppings: [
        "tomato sauce",
        "mushrooms",
        "mozzarella",
        "pepperoni",
    ],
}
```

Поскольку Rust не поддерживает перегрузку функций, можно создать только один метод `new()`. Поэтому хорошоенько подумайте, чего вы хотите от функции. В большинстве случаев она должна обеспечивать минимальное необходимое поведение, например возвращать новый пустой объект (как в случае `Vec::new()`) с минимумом необходимых аргументов. Кроме того, некоторые люди создают дополнительные конструкторы, которые начинаются с `new_` и получают дополнительные аргументы. К примеру, `Vec::new_in(alloc: A)` (доступен только вочной версии Rust) получает необязательный аллокатор памяти и возвращает пустой `Vec`, использующий этот аллокатор.

**ПРИМЕЧАНИЕ** Если набор аргументов инициализации усложняется, то стоит использовать паттерн «Строитель», о котором мы будем говорить в главе 5.

## 4.4. Видимость членов объекта и доступ к ним

В Rust по умолчанию для всего используется закрытый режим видимости. При желании вы можете делать сущности открытыми, используя ключевое слово `pub`. В зависимости от контекста открытая видимость может означать несколько разные вещи. Здесь мы будем обсуждать члены объектов, и в этом случае добавление `pub` означает, что к ним можно обращаться или изменять их напрямую. Вернемся к примеру с пиццей из подраздела 4.3.1 и на этот раз сделаем топпинги открытыми для доступа извне текущего модуля (листинг 4.8).

### Листинг 4.8. Пицца с `pub`-топпингами

```
#[derive(Debug, Clone)]
pub struct Pizza {
    pub toppings: Vec<String>, ←
}
```

Обратите внимание  
на спецификатор  
видимости `pub`

По сути, этот код позволяет рассматривать `toppings` как простую переменную и делать следующее:

```
let mut pub_pizza = Pizza {
    toppings: vec![String::from("sauce"), String::from("cheese")],
};

// Удалять последний топпинг.
pub_pizza.toppings.remove(1);
println!("pub_pizza={:?}", pub_pizza);
```

Если мы выполним этот код, то получим такой вывод:

```
pub_pizza=Pizza { toppings: ["sauce"] }
```

Когда может потребоваться подобное решение? Оно подойдет, только когда у вас есть контейнеры данных без методов и их единственная цель — хранить данные. Чаще всего вам нужно контролировать доступ к членам с помощью *аксессоров* (методов, получающих приватные члены) и изменять члены с помощью *мутаторов* (методов, позволяющих это делать). Аксессоры и мутаторы зачастую называются *геттерами* и *сеттерами*, хотя в Rust важно понимать различие между установкой значения (как при перемещении) и его изменением на месте.

**СОВЕТ** Эти методы подразумевают постоянный набор шаблонного кода, но инструменты наподобие rust-analyzer упрощают генерацию геттеров и сеттеров для каждого члена. В другой моей книге, *Code Like a Pro in Rust* (<https://www.manning.com/books/code-like-a-pro-in-rust>), которую я уже несколько раз упоминал выше, есть раздел, посвященный rust-analyzer, но информация по теме генераторов доступна и в соответствующей документации по адресу <https://mng.bz/5lV8>.

Пора снова обновить код пиццы, переведя `toppings` обратно в закрытый режим (мы не хотим, чтобы они были открытыми), а также добавив аксессор, мутатор и сеттер (листинг 4.9).

#### Листинг 4.9. Обеспечение доступа к `toppings` пиццы

```
impl Pizza {
    pub fn toppings(&self) -> &[String] { ← | Аксессор (геттер) возвращает
        self.toppings.as_ref()                         | через вектора топпингов
    }

    pub fn toppings_mut(&mut self) -> &mut Vec<String> { ← | Мутатор возвращает
        &mut self.toppings                           | изменяемую ссылку
    }                                                 | на вектор топпингов

    pub fn set_toppings(&mut self, toppings: Vec<String>) { ← | Сеттер получает новый
        self.toppings = toppings;                     | вектор по значению
    }                                                 | и заменяет им
}                                                 | существующий
```

В этом примере каждый метод получает ссылку на `self` и изменяемые методы получают изменяемую ссылку. Обратите внимание, что вместо прямой ссылки

я возвращаю из `Vec` срез. Возврат данных в виде `Vec` и в виде среза по своей сути равнозначны, но простой возврат среза более идиоматичен, так как срезы обычно используются для выражения неизменяемых непрерывных последовательностей (например, наименьшего общего знаменателя).

Можно было также немного изменить `set_toppings()`, чтобы он возвращал (или перемещал) существующие `toppings`, заменяя текущие. Это изменение можно назвать, например, `replace_toppings()` (листинг 4.10).

#### Листинг 4.10. Обеспечение метода для замены `toppings`

```
impl Pizza {
    pub fn replace_toppings(
        &mut self,
        toppings: Vec<String>,
    ) -> Vec<String> {
        std::mem::replace(&mut self.toppings, toppings)
    }
}
```

Этот код использует `std::mem::replace()`, позволяя нам заменить существующие `toppings`. Для этого мы делаем два перемещения: одним убираем текущие `toppings`, а вторым возвращаем на их место предыдущие. Такой подход исключает клонирование и дублирование, обеспечивая некоторую оптимизацию.

## 4.5. Обработка ошибок

Это может удивить, но обрабатывать ошибки в Rust не так уж сложно. Обычно для этого активно используют структуру `Result`, имеющую особую поддержку оператора `?`, что мы пронаблюдаем чуть позже.

Прежде чем переходить к конкретным фрагментам кода, я озвучу два аспекта обработки ошибок, такие как их вывод (например, в функции, которая может вернуть ошибку) и обработка результата (что делать в случае ошибки).

Для вывода ошибок обычно используются простые структуры или перечисления, которые содержат необходимые метаданные (тип ошибки, сопутствующие сообщения и т. д.). В стандартной библиотеке доступны несколько типов ошибок (например, `std::io::Error`), но зачастую они просто добавляются в пользовательские типы (например, в виде вариантов перечислений) или возвращаются непосредственно без изменений. Создать собственный тип ошибки не сложнее, чем определить структуру или перечисление. После этого можно возвращать данный тип в `Result`. Кроме того, в стандартной библиотеке есть специальный трейт `std::error::Error`, который можно реализовывать для собственных типов ошибок, но это не обязательно. На практике для пользовательских типов ошибок его реализуют редко.

Обработка ошибок обычно сопряжена с использованием одновременно двух подходов: явной обработки каждого случая с помощью сопоставления с шаблоном (или иной управляющей логики) либо передачи ошибок вызывающему коду. Во втором случае иногда можно обойтись использованием оператора `?`. Сделать это несложно: нужно добавить его после любого вызова функции, возвращающей `Result` или `Option`, и он будет разворачивать результат при преждевременном завершении функции ошибкой или возвращением `None` (в случае `Option`). Недостаток оператора `?` в том, что он может иметь отложенное действие, а вам иногда нужно обрабатывать ошибки и предпринимать соответствующие действия явно. При его использовании вам наверняка придется реализовывать для типов ошибок трейт `From`, создавая еще одно место, куда можно добавить логику обработки ошибок.

Напишем функцию, которая будет считывать из файла `n` строку и возвращать ее. Как вы увидите, эта функция может дать сбой по нескольким причинам, поэтому потребуется обработать каждый такой случай. В листинге 4.11 показана наша первая попытка создания функции.

#### Листинг 4.11. Считывание `n` строки из файла

```
use std::path::Path;
#[derive(Debug)]
pub enum Error {
    Io(std::io::Error),
    BadLineArgument(usize),
}
```

Наш тип ошибки является перечислением, которое может содержать одно из двух возможных значений

```
impl From<std::io::Error> for Error {
    fn from(error: std::io::Error) -> Self {
        Self::Io(error)
    }
}

fn read_nth_line(path: &Path, n: usize) -> Result<String, Error> {
    use std::fs::File;
    use std::io::{BufRead, BufferedReader};
    let file = File::open(path)?;
```

Error::Io содержит `std::io::Error`, которую мы передаем вызывающему коду

Error::BadLineArgument — это ошибка, которую мы возвращаем в случае недействительного номера строки

Реализуем `From`, чтобы можно было преобразовать `std::io::Error` в собственный тип ошибки `Error`

```
    let mut reader_lines = BufferedReader::new(file).lines();
    reader_lines
        .nth(n - 1)
        .map(|result| result.map_err(|err| err.into()))
        .unwrap_or_else(|| Err(Error::BadLineArgument(n)))
```

Здесь используем оператор `?` для получения дескриптора файла

Обратите внимание, что нам нужно вычесть из `n` единицу, так как первой считываемой строкой файла является нулевая

Метод `nth()` возвращает `Option<Result<String, std::io::Error>>`, поэтому нужно преобразовать внутреннюю ошибку в наш тип

BufReader обеспечивает для нашего дескриптора файла буферизованный модуль чтения, а трейт `BufRead` предоставляет метод `lines()`, позволяющий перебирать каждую строку файла

В завершение, если вернется `None`, мы выполняем чтение за пределами конца файла

В нашей функции `read_nth_line()` используется трейт `std::io::BufRead`, предоставляющий несколько удобных возможностей, в том числе метод `lines()`, который возвращает итератор, перебирающий каждую строку файла. Теперь протестируем нашу функцию:

```
let path = Path::new("Cargo.toml");
println!(
    "The 4th line from Cargo.toml reads: {}",
    read_nth_line(path, 4)?
);
```

При выполнении этого кода мы получим такой вывод:

```
The 4th line from Cargo.toml reads: edition = "2021"
```

В строке, где мы вычитаем единицу из `n` при вызове `nth()`, есть небольшой баг. Если `n` равно `0`, то возникает переполнение, так что этот момент нужно обработать. Обратите внимание: если программа компилируется в режиме релиза, то переполнение замечено не будет, поскольку в Rust проверка целочисленного переполнения выполняется только при сборке кода в режиме отладки. В остальных случаях Rust ведет себя аналогично C.

Обработать этот случай можно по-разному. Мы будем проверять значение `n` и преждевременно завершим функцию с ошибкой, если это значение меньше 1 (листинг 4.12).

#### Листинг 4.12. Считывание nth строки из файла

```
fn read_nth_line(path: &Path, n: usize) -> Result<String, Error> {
    if n < 1 {
        return Err(Error::BadLineArgument(0));
    }
    use std::fs::File;
    use std::io::{BufRead, BufReader};
    let file = File::open(path)?;

    let mut reader_lines = BufReader::new(file).lines();
    reader_lines
        .nth(n - 1)
        .map(|result| result.map_err(|err| err.into()))
        .unwrap_or_else(|| Err(Error::BadLineArgument(n)))
}
```

Далее нужно написать для функции модульные тесты, чтобы убедиться в ее корректной работе. Их реализация показана в листинге 4.13.

Как видите, обрабатывать ошибки в Rust несложно. В большинстве случаев для вашей библиотеки или приложения нужно создать тип ошибки, который будет включать в себя все возможные сбои, а во многих случаях достаточно просто возвращать непосредственно саму ошибку.

**Листинг 4.13. Модульные тесты для считывания из файла nth строки**

```

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_can_read_cargotoml() {
        let third_line = read_nth_line(Path::new("Cargo.toml"), 3)
            .expect("unable to read third line from Cargo.toml");
        assert_eq!("version = \"0.1.0\"", third_line);
    }

    #[test]
    fn test_not_a_file() {
        let err = read_nth_line(Path::new("not-a-file"), 1)
            .expect_err("file should not exist");
        assert!(matches!(err, Error::Io(_)));
    }

    #[test]
    fn test_bad_arg_0() {
        let err = read_nth_line(Path::new("Cargo.toml"), 0)
            .expect_err("0th line is invalid");
        assert!(matches!(err, Error::BadLineArgument(0)));
    }

    #[test]
    fn test_bad_arg_too_large() {
        let err = read_nth_line(Path::new("Cargo.toml"), 500)
            .expect_err("500th line is invalid");
        assert!(matches!(err, Error::BadLineArgument(500)));
    }
}

```

Попытка считывания файла not-a-file, который не существует, в связи с чем возвращается ошибка ввода/вывода

Проверяем возвращенную ошибку, используя matches!, которая позволяет предоставить шаблон для сопоставления и возвращает логическое значение, подтверждающее ошибку

Здесь мы проверяем, вызовет ли ошибку передача в п нулевого значения

Файл Cargo.toml проекта содержит всего восемь строк, так что значение 500 явно выходит за его пределы и должно вести к ошибке

Здесь мы проверяем возвращенную ошибку, используя сопоставление с шаблоном при конкретном значении (0)

Снова проверяем, соответствуют ли возвращенные совпадения ожидаемому значению

**4.6. Глобальное состояние**

Каждый разработчик так или иначе иногда сталкивается с обработкой глобального состояния. И есть веские причины стараться избегать этого. Глобальное состояние может создавать условия гонки, несет риск повреждения данных, в нем слабо разделены обязанности, вдобавок оно чревато и множеством других проблем. Тем не менее, как бы вы ни старались избегать его, рано или поздно возникнет ситуация, когда без глобального состояния не обойтись. В этом разделе я опишу несколько стратегий его обработки в Rust, которые влекут за собой как сложности, так и преимущества, обусловленные моделями владения и распределения памяти.

Глобальное состояние иногда реализуется с помощью паттерна «Одиночка», который некоторые разработчики считают антипаттерном. Мы еще вернемся к этой теме в главе 10, но здесь я скажу лишь, что при использовании глобального состояния и этого паттерна нужно быть очень осмотрительными.

Для начала мы поговорим о глобальных переменных в Rust. Этот язык позволяет использовать лишь два их вида: `static` и `const`. В обоих случаях значение

переменной должно определяться при компиляции. Иными словами, нельзя выполнять инициализацию в среде выполнения с неопределенными глобальными переменными. Можно определять изменяемые статические переменные, чтобы потом корректировать их значения при выполнении, но этот подход считается небезопасным и требует использования ключевого слова `unsafe`. Кроме того, для обеспечения потокобезопасного доступа, исключающего состояние гонки, статические переменные также должны быть синхронизированы (`Sync`). При этом для статических переменных запрещено выделение памяти (нельзя использовать что-либо выделяющее память в куче), и метод `drop()` из `Drop` при завершении никогда не вызывается.

Из-за всех этих ограничений для глобального состояния часто используют особую форму отложенной (just-in-time, JIT) инициализации. Эта задача упрощается из-за наличия нескольких крейтов, но прежде, чем знакомить вас с ними, я покажу, как можно выполнить реализацию вручную.

Мы не можем создать статический вектор слов, так как память под `Vec` и `String` выделяется в куче. Поэтому следующий код не скомпилируется.

```
static POPULAR_BABY_NAMES_2021: Vec<String> = vec![
    String::from("Olivia"),
    String::from("Liam"),
    String::from("Emma"),
    String::from("Noah"),
];
```

Попытка его компиляции приведет к длинному списку ошибок:

```
error[E0010]: allocations are not allowed in statics
--> src/main.rs:1:47
 |
1 |     static POPULAR_BABY_NAMES_2021: Vec<String> = vec![^
 |
2 |         String::from("Olivia"),
3 |         String::from("Liam"),
4 |         String::from("Emma"),
5 |         String::from("Noah"),
6 |     ];
|_|     ^ allocation not allowed in statics
|
= note: this error originates in the macro `vec` (in Nightly builds,
run with -Z macro-backtrace for more info)
```

```
error[E0015]: cannot call non-const fn `<String as From<&str>>::from` in statics
--> src/main.rs:2:5
 |
2 |     String::from("Olivia"),
|     ^^^^^^^^^^^^^^^^^^^^^^
```

```
= note: calls in statics are limited to constant functions, tuple
structs and tuple variants
= note: consider wrapping this expression in `Lazy::new(|| ...)`
from the `once_cell` crate: https://crates.io/crates/once\_cell
```

Вы могли заметить предложение компилятора использовать `once_cell`, что мы скоро и сделаем. Но для начала посмотрим, можно ли добиться успеха в этом подходе, не применяя крейты.

Для создания статической глобальной переменной нужно использовать макрос `std::thread_local!`, предоставляющий локальное для потока хранилище, которое является `Sync` (то есть потокобезопасно). Это хранилище позволяет хранить данные, локальные для текущего потока, но также обеспечивает глобальную доступность памяти.

Для безопасного предоставления доступа к внутренним данным нам нужно использовать указатель с подсчетом ссылок `Arc` и `Mutex`. Наконец, `Vec<String>` необходимо завернуть в `Option`, так как нельзя инициализировать `Vec` или `String` на этапе компиляции. В этом случае указатель, который мы используем для доступа к данным, является локальным для текущего потока, а сами эти данные — глобальными, поэтому у нас получается код, показанный в листинге 4.14.

#### **Листинг 4.14. Объявление глобальной статической переменной локально в потоке**

```
thread_local! {
    static POPULAR_BABY_NAMES_2021: Arc<Mutex<Option<Vec<String>>>> =
        Arc::new(Mutex::new(None));
}
```

Нужно инициализировать `Vec` с какими-нибудь данными. Где-то в коде, например в `main()`, для этого необходимо сделать следующее (листинг 4.15).

#### **Листинг 4.15. Инициализация глобальной статической переменной локально в потоке**

```
let arc = POPULAR_BABY_NAMES_2021.with(|arc| arc.clone());
let mut inner = arc.lock().expect("unable to lock mutex");
*inner = Some(vec![
    String::from("Olivia"),
    String::from("Liam"),
    String::from("Emma"),
    String::from("Noah"),
]);
```

Но исправлять таким образом нашу ситуацию нежелательно. Кроме того, корректная инициализация глобального состояния требует особой внимательности. Необходимо сделать это в подходящее время до того, как другой код сможет попытаться получить доступ к инициализируемым значениям.

На практике так обрабатывать глобальное состояние не стоит. Вместо этого я предлагаю использовать один из нескольких крейтов, обеспечивающих нужное поведение с помощью удобного API (табл. 4.2).

**Таблица 4.2.** Обобщенная информация о крейтах для реализации глобального состояния

Крейт	Репозиторий	Скачиваний на март 2024 года	Описание
lazy-static.rs	<a href="https://mng.bz/oegy">https://mng.bz/oegy</a>	215 759 981	Макрос для объявления статических переменных с отложенной инициализацией
once_cell	<a href="https://github.com/matklad/once_cell">https://github.com/matklad/once_cell</a>	213 996 727	Предоставляет два новых типа, которые можно использовать для инициализации глобального состояния
static_init	<a href="https://gitlab.com/okannen/static_init">https://gitlab.com/okannen/static_init</a>	3 391 550	Предоставляет возможность создания глобальных переменных, обладающих большей эффективностью и дополнительной функциональностью, в том числе допускающих отбрасывание данных

Далее мы реализуем пример из листингов 4.14 и 4.15, используя крейты из табл. 4.2.

#### 4.6.1. Крейт *lazy-static.rs*

Этот крейт — самое популярное средство решения проблемы с глобальным состоянием в Rust (на момент написания книги). Его API основан на простом макросе, который для определения глобальных переменных использует синтаксис `static ref` с возможностью инициализации путем замыкания. С помощью этого крейта можно инициализировать глобальное состояние. Пример использования `lazy-static.rs` показан в листинге 4.16.

##### Листинг 4.16. Инициализация списка популярных имен новорожденных с помощью крейта *lazy-static.rs*

```
use lazy_static::lazy_static;

lazy_static! {
    static ref POPULAR_BABY_NAMES_2020: Vec<String> = {
        vec![
            String::from("Olivia"),
            String::from("Liam"),
```

```

        String::from("Emma"),
        String::from("Noah"),
    ]
};

}

```

Если вам нужна возможность изменения инициализируемых данных, то используйте `Mutex<Vec<String>>` или `RwLock<Vec<String>>`, но в данном примере мы будем рассматривать данные как неизменяемые. Теперь выполним этот код.

```
println!("popular baby names of 2020: {:?}", *POPULAR_BABY_NAMES_2020);
```

Обратите внимание, что для доступа к значению переменной достаточно разыменовать ее с помощью оператора `*`, так как `lazy-static.rs` предоставляет трейт `Deref`. При выполнении предыдущего кода мы получим такой вывод:

```
popular baby names of 2020: ["Olivia", "Liam", "Emma", "Noah"]
```

#### **4.6.2. Крейт once\_cell**

Крейт быстро набирает популярность. В сравнении с `lazy-static.rs` он имеет более обобщенный API для обработки глобального состояния. По этой причине я советую использовать в новых проектах именно `once_cell`. Если же вы уже работаете с ним или просто лучше разбираетесь в нем, то он станет прекрасным решением.

Теперь реализуем тот же список имен, но используя `once_cell`. Листинг 4.17 имеет приятный и лаконичный API.

##### **Листинг 4.17. Инициализация списка популярных имен новорожденных с помощью крейта once\_cell**

```

use once_cell::sync::Lazy;

static POPULAR_BABY_NAMES_2019: Lazy<Vec<String>> = Lazy::new(|| {
    vec![
        String::from("Olivia"),
        String::from("Liam"),
        String::from("Emma"),
        String::from("Noah"),
    ]
});

```

API `once_cell::sync::Lazy` предоставляет трейт `Deref`, позволяющий обращаться к значениям с помощью оператора `*`:

```
println!("popular baby names of 2019: {:?}", *POPULAR_BABY_NAMES_2019);
```

Как и в случае крейта `lazy-static.rs`, чтобы обеспечить изменяемость данных, можно обернуть их с помощью `Mutex` или `RwLock`.

### 4.6.3. Крейт static\_init

Ну и последним мы разберем `static_init`, имеющий несколько интересных возможностей и обеспечивающий прекрасное быстродействие (листинг 4.18).

#### Листинг 4.18. Инициализация списка популярных имен новорожденных с помощью крейта static\_init

```
use static_init::dynamic;

#[dynamic]
static POPULAR_BABY_NAMES_2018: Vec<String> = vec![
    String::from("Эмма"),
    String::from("Лиам"),
    String::from("Оливия"),
    String::from("Ной"),
];
```

Чтобы обеспечить возможность изменения, нужно добавить ключевое слово `mut` (например, `static mut POPULAR_BABY_NAMES_2018 ...`). В крейте `static_init`, как и в `lazy-static.rs` или `once_cell`, тоже есть трейт `Deref`. Это позволяет обратиться к значению так:

```
println!("popular baby names of 2018: {:?}", *POPULAR_BABY_NAMES_2018);
```

### 4.6.4. std::cell::OnceCell

Отмечу, что в стандартной библиотеке Rust (для версии Rust 1.70) есть `std::cell::OnceCell` и `std::sync::OnceLock`, которые частично решают проблему статической инициализации, но без удобства отложенной инициализации на глобальном уровне. Для использования этой возможности есть экспериментальный API `std::cell::LazyCell`, но в стабильном Rust он на момент написания книги отсутствует. Использование `std::cell::OnceCell` примерно равноценно использованию макроса `thread_local!`, о чём я говорил в этой главе ранее.

Вы можете создать глобальный экземпляр `std::cell::OnceCell`, но не можете инициализировать его значение в глобальной области видимости внутри одного выражения. Если вы предпочитаете избегать использования крейтов для этой задачи, то такое решение может стать приемлемым компромиссом. Основной недостаток разделения объявления и инициализации заключается в том, что такой подход снижает ясность кода и может привести к дублированию кода инициализации или потенциальному состоянию гонки, если к этому коду будет вести несколько путей.

Для полноты картины мы реализуем то же самое поведение, используя `std::cell::OnceCell`, но наша инициализация должна происходить внутри функции, а не в глобальном контексте. В листинге 4.19 код просто помещается в `main()`.

**Листинг 4.19. Использование std::cell::OnceCell**

```
let popular_baby_names_2017: OnceCell<Vec<String>> = OnceCell::new();
popular_baby_names_2017.get_or_init(|| {
    vec![
        String::from("Emma"),
        String::from("Liam"),
        String::from("Olivia"),
        String::from("Noah"),
    ]
});
```

**Резюме**

- RAII используется в Rust повсеместно и отлично работает вкупе с семантикой перемещения, позволяя безопасно обрабатывать владение, освобождение ресурсов и синхронизацию.
- RAII можно использовать для создания структур данных и контейнеров, безопасно управляющих ресурсами и выполняющих очистку, реализуя трейт `Drop`.
- Аргументы для вызова функции можно передавать по значению или по ссылке. Первый вариант передачи позволяет реализовать в Rust уникальные паттерны.
- Аргументы, передаваемые по значению, перемещаются из контекста вызывающего кода в контекст вызываемого и могут возвращаться от вызываемого вызывающему.
- Члены объектов по умолчанию являются закрытыми. В отличие от обработки открытых членов, для доступа к ним или изменения их значений обычно нужно писать методы. Исключением является случай, когда структуры используются строго в качестве контейнеров данных и прямой доступ предпочтительнее.
- Функции, в которых могут возникать ошибки, должны возвращать `Result`, и обычно для этого создаются специальные типы ошибок, которые содержат всю необходимую информацию.
- С помощью оператора `?` можно сохранять чистоту кода, не обрабатывая явно каждый ошибочный случай.
- Реализуя для типов ошибок трейт `From`, мы можем изящно обрабатывать различные случаи ошибок.
- Обработка глобального состояния в Rust сопряжена со многими нюансами, но есть крейты, которые упрощают эту задачу. Например, крейт `once_cell` предоставляет удобный и лаконичный API для отложенной инициализации и глобального состояния.

# Паттерны проектирования: расширяем функционал

## В этой главе

- ✓ Метапрограммирование с помощью макросов.
- ✓ Реализация паттерна «Строитель» на Rust.
- ✓ Создание гибких интерфейсов.
- ✓ Реализация паттернов «Наблюдатель», «Команда» и «Новый тип».

В главах 2 и 3 вы познакомились с фундаментальными компонентами Rust: дженериками, трейтами, сопоставлением с шаблоном и возможностями функционального программирования. В текущей главе мы продолжим изучать эти темы и перейдем непосредственно к освоению паттернов проектирования.

На основе пройденного материала мы можем начать создавать более конкретные паттерны в соответствии с идиомами языка Rust. И хотя мы не будем рассматривать все возможные паттерны, я представлю тщательно отобранные примеры, демонстрирующие основы, необходимые для создания на Rust любого паттерна.

Если дженерики, трейты, сопоставление с шаблоном и замыкания являются сырыми ингредиентами, то паттерны из этой главы выступают как архетипы практически любого другого паттерна, совмещающего эти возможности. Но прежде, чем переходить к их непосредственному рассмотрению, мы разберем макросы, которые сами по себе являются паттернами, но зачастую используются в более сложных структурах.

Знакомство с макросами на данном этапе может показаться странным, однако эта тема важна, поскольку позволяет изучить остальные темы. Без базового понимания макросов сделать это будет сложно. Кроме того, мы используем их в главе 8, чтобы показать, как они взаимодействуют, сочетаются и расширяют возможности кода.

Из пяти разбираемых в этой главе паттернов четыре встречаются в разных языках программирования, библиотеках и SDK довольно часто. Это паттерны «Строитель», «Гибкий интерфейс», «Наблюдатель» и «Команда». Последним мы разберем паттерн «Новый тип», который относится больше к Rust. Все эти паттерны заслуживают свою популярность, так как предоставляют понятные, полезные и широко применяемые абстракции для многих распространенных задач программирования. И даже если вы никогда не будете реализовывать эти паттерны, то после знакомства с ними как минимум будете видеть их повсюду.

## **5.1. Метапрограммирование с помощью макросов**

*Макросы* — это инструменты для метапрограммирования, обычно использующие препроцессор. Они позволяют расширять функциональность языка программирования. *Метапрограммирование* — это процесс использования кода для генерации кода, а *препроцессинг* — процесс выполнения кода (или макроса) до его компиляции.

Макросы зачастую передаются в качестве предметно-ориентированного языка (domain-specific language, DSL) для генерации кода перед выполнением компиляции. Кроме того, с помощью макросов можно расширять возможности других языков, используя их в качестве пользовательского этапа предварительной обработки (препроцессинга), который иногда встречается в тех языках, которые по умолчанию не поддерживают эту функциональность.

Тем не менее многие языки, в том числе C и C++, имеют простую, но весьма полезную систему макросов. Более сложные макросы предоставляет Lisp. Кроме того, они есть в Elixir, Erlang, Scala и OCaml.

В сравнении со своими аналогами в C и C++ макросы в Rust являются чуть более сложными. В C и C++ они используют текстовую подстановку и предоставляют довольно скромные возможности в плане проверки типов, сопоставления параметров и даже обработки областей видимости.

Если же говорить о макросах Rust, то они отличаются от своих «коллег» типо-безопасностью, то есть являются более безопасными и удобными в работе по сравнению с другими аналогичными системами. Кроме того, компилятор Rust предоставляет качественную помощь в виде информативных сообщений об ошибках, хотя в случае особо сложных макросов могут возникать затруднения. Помимо проверки типов, макросы Rust выполняют проверку чистоты, чтобы

присутствующие в них переменные и идентификаторы не конфликтовали с переменными и идентификаторами в вызывающем коде.

Макросы могут прекрасно дополнять любую кодовую базу, в частности содержащую громоздкие и повторяющиеся конструкции. Как и любые другие инструменты, макросы можно использовать некорректно, в том числе для маскировки некачественного кода.

И еще одно: на данный момент в Rust есть две системы макросов — *декларативные макросы*, доступные по умолчанию, и *процедурные макросы*, активируемые с помощью флага. Процедурные являются куда более сложными, но предлагают гораздо больше возможностей и гибкости. О них мы будем говорить в главе 8, а здесь обсудим декларативные.

### 5.1.1. Базовый декларативный макрос в Rust

Разберем простой макрос. В Rust его можно узнать по символу `!`, идущему за ключевым словом. Вызов макроса похож на вызов функции с дополнительным `!` перед аргументами. Вы наверняка встречали в коде выражение `vec![ ]`. Это макрос. Вот еще пара популярных примеров: `println!` и `dbg!`. При вызове макроса символ `!` в конце его имени обязателен, так как он сообщает компилятору (и любому, кто читает код), что вы используете макрос, а не стандартную функцию. Определение макроса начинается с `macro_rules!`, сопровождаемого его именем:

```
macro_rules! noop_macro {
    () => {};
}
```

Этот макрос ничего не делает. Его можно вызвать с помощью `noop_macro!()`. Возможно, вы заметили, что его тело несколько похоже на оператор `match` — поскольку это он и есть. После `!` можно выполнять сопоставление с чем угодно, включая различные виды скобок. В качестве них допустимо использовать `()`, `{}` или `[ ]`, но сами по себе скобки обязательны. Макросы выполняются на этапе компиляции, поэтому их содержимое представляет не результат исполнения кода, а сам фактически исполняемый код. Иными словами, в макросах Rust можно выполнять код на основе сопоставления с различными конструкциями, также известными как *фрагменты кода*. Вот пример:

```
macro_rules! print_what_it_is {
    () => {
        println!("A macro with no arguments")
    };
    ($e:expr) => {
        println!("A macro with an expression")
    };
    ($s:stmt) => {
        println!("A macro with a statement")
    };
}
```

Срабатывает при отсутствии аргументов, например `print_what_it_is!()`

Срабатывает при одном аргументе, если это выражение наподобие `print_what_it_is!(...)`

Срабатывает при одном аргументе, если это оператор сродни `print_what_it_is!(...;)`

В макросе выше есть три правила, с которыми производится сопоставление: одно выполняется при отсутствии аргументов, второе — при передаче выражений и третье — при передаче операторов. Аргументы для последних двух доступны в переменных `$e` и `$s` соответственно. Можем вызвать наш макрос так:

```
print_what_it_is!();
print_what_it_is!(());
print_what_it_is!();
```

При выполнении этого кода мы получим такой вывод:

```
A macro with no arguments
A macro with an expression
A macro with a statement
```

В Rust фрагментом может выступать любая конструкция кода при условии ее синтаксической корректности. Кроме того, сопоставление можно производить с несколькими аргументами:

```
macro_rules! print_what_it_is {
    // ... опущено для краткости ...
    ($e:expr, $s:expr) => {
        println!("An expression followed by a statement")
    };
}
```

Если вызвать этот макрос как `print_what_it_is!( {}, ; )`, то он выдаст `An expression followed by a statement`. Если вызвать его с недопустимым аргументом (не соответствующим ни одному из правил), то возникнет ошибка компиляции. Вызов `print_what_it_is!` с двумя операторами (`print_what_it_is!( ; , ; )`) приведет к следующей ошибке:

```
error: no rules expected the token ` `,
--> src/main.rs:27:24
   |
5 |     macro_rules! print_what_it_is {
   |     ----- when calling this macro
...
27 |         print_what_it_is!( ; , ; ); // ошибка!
   |                     ^ no rules expected this token in macro call
```

Можно написать для этого паттерна сопоставление:

```
macro_rules! print_what_it_is {
    // ... опущено для краткости ...
    ($e:stmt, $s:stmt) => {
        println!("Two back-to-back statements")
    };
}
```

Теоретически в качестве аргументов макроса можно использовать любую комбинацию операторов и паттернов. Но я советую хорошенько подумать, прежде чем писать такой длинный макрос, поскольку большое количество аргументов

может сбить с толку вызывающий код. Как минимум важно иметь представление о различных возможных сценариях, чтобы знать, как поступать, встретив их в чужом коде.

### 5.1.2. Когда использовать макросы

Макросы в Rust очень похожи на функции, поэтому логично спросить: «Зачем использовать макрос вместо функции?» На это есть убедительные причины. Первая заключается в том, что макросы позволяют перегружать аргументы. Еще одна причина — макросы поддерживают вариативность, то есть им можно передавать переменное количество аргументов с помощью разделителя. К прочим случаям применения макросов относится пользовательское логирование данных (например, с помощью крейта `log`; <https://crates.io/crates/log>) и создание мини-DSL (например, с помощью крейта `lazy_static`; [https://crates.io/crates/lazy\\_static](https://crates.io/crates/lazy_static)). Предположим, вы хотите написать собственную версию макроса `println!`. Как вы могли заметить, он получает  $N + 1$  аргументов, то есть является вариативным. Первый аргумент — это строка формата, которая может содержать подставляемые переменные. Следующие же  $N$  аргументов — это значения, которые подставляются. Напишем собственный макрос, который обернет `println!`:

```
macro_rules! special_println {
    ($($arg:tt)*) => {
        println!{$($arg)*}
    };
}
```

Макрос `special_println!` можно вызвать точно так же, как `println!`. Чтобы создать предыдущий пример, я просто скопировал определение `println!`. Теперь разберем спецификацию аргументов, представленную как `$($arg:tt)*`.

- Идентификатор `$arg` — это именованный идентификатор для аргументов, совпадающих с данным правилом.
- Мы делаем сопоставление с `tt` (сокращение от *token trees* — «деревья токенов»). Дерево может содержать один идентификатор, последовательность идентификаторов или последовательность деревьев токенов (которые, в свою очередь, тоже могут содержать идентификаторы, так как являются рекурсивными). Собственно, именно из-за их рекурсивной природы эти структуры и называют *деревьями*.
- Правило для сопоставления находится в скобках, как в `$(...)`, указывая на тот факт, что сопоставление с ним можно выполнять неоднократно. Но нам нужно указать, сколько именно раз оно должно производиться (подробнее — в следующем пункте).
- Последним идет символ звездочки (\*). Он сообщает компилятору, что эти аргументы могут повторяться любое количество раз. В Rust используются те же операторы, что и в регулярных выражениях: `+` для одного или несколь-

ких сопоставлений, \* для любого их количества и ? для одного сопоставления или их отсутствия.

- Дерево токенов может быть последовательностью, поэтому нам не нужно добавлять пунктуацию, так как мы передаем его целиком.

Разворачивание или транскрибирование аргументов выполняется с помощью `$(#arg)*`, которую мы просто передаем в `println!`. Обратите внимание: одни макросы могут вызывать другие, и это также означает, что вы можете выполнять с их помощью рекурсию.

Слегка расширим функциональность нашего макроса. Предположим, нам нужно добавить ко всем вызовам `special println!` некий префикс, как часто делается в системах логирования:

```
macro_rules! special_println {
    ($($arg:tt)*) => {
        println!("Printed specially: {}", $($arg)*)
    };
}
```

Здесь мы передаем  
все аргументы `println!`  
в виде второго аргумента.  
В предыдущем же примере мы  
передавали их в виде первого

Неплохо! Теперь, если вызвать этот код как `special println!("hello world!")`, он выдаст "Printed specially: hello world!".

Но в нашем макросе есть проблема: в своей текущей форме он получает лишь один аргумент, что не несет особой пользы. Дело в том, что в качестве первого аргумента для нашего вызова `println!` мы жестко прописали спецификатор формата `{}`. В результате макрос ожидает и принимает только один параметр.

Чтобы заставить наш макрос получать любое количество аргументов (как `println!`), мы обернем эти аргументы с помощью `format!` — специального макроса, который корректно обрабатывает строковую интерполяцию и переменное количество аргументов. Определение макроса `format!` есть в стандартной библиотеке Rust (<https://doc.rust-lang.org/std/macro.format.html>). Оно во многом похоже на наш код, только вызывает специальный макрос `format_args!` и функцию `std::fmt::format()` из стандартной библиотеки (листинг 5.1).

### Листинг 5.1. Определение макроса `format!` из стандартной библиотеки Rust

```
macro_rules! format {
    ($($arg:tt)*) => {{
        let res = $crate::fmt::format(
            $crate::export::format_args!($($arg)*));
        res
    }}
}
```

Если мы заглянем чуть глубже, то увидим, что `format_args!` — особый встроенный макрос (<https://mng.bz/ngaV>). Реализует его компилятор, так что здесь мы остановимся. (Чтобы разобраться в теме более обстоятельно, нужно изучать

исходный код компилятора Rust.) В листинге 5.2 показано определение из стандартной библиотеки Rust.

### Листинг 5.2. Определение макроса `format_args!` из стандартной библиотеки Rust

```
macro_rules! format_args {
    ($fmt:expr) => {{ /* относится к компилятору */ }};
    ($fmt:expr, $($args:tt)*) => {{ /* относится к компилятору */ }};
}
```

Идем дальше. Теперь мы обновим макрос `special println!`, чтобы он мог использовать `format!` для вычисления аргументов до вызова `println!`:

```
macro_rules! special println {
    ($($arg:tt)*) => {
        println!("Printed specially: {}", format!($($arg)*)) ◀
    };
}
```

До передачи в `println!` аргументы передаются в `format!`, чтобы можно было также представить их в виде доступной для форматирования строки

Теперь можно вызвать этот макрос как `special println!("with an argument of {}", 5)`, на что мы получим вывод "Printed specially: with an argument of 5". Для отладки макроса нужно включить функцию трассировки (доступна только вочной сборке), добавив следующий атрибут:

```
#![feature(trace_macros)]
```

**СОВЕТ** Переключиться на ночную сборку можно путем выполнения `rustup default nightly` или переопределения цепочки инструментов текущего проекта командой `rustup override set nightly`. Кроме того, можно сопроводить вызов `cargo` аргументом `+nightly` для запуска конкретного крейта через ночную сборку, например: `cargo +nightly build`. Наконец, можно создать в корне проекта файл `rust-toolchain.toml` со следующим содержимым: `toolchain.channel = "nightly"`.

Следующий код приведет к выводу сообщений компилятора, отражающих результат разворачивания макроса. Чтобы использовать трассировку, нужно включить ее для конкретных вызовов с помощью `trace_macros!`:

```
trace_macros!(true);
special println!("hello world!");
trace_macros!(false);
```

Теперь при компиляции кода мы получим следующий вывод:

```
note: trace_macro
--> src/main.rs:84:5
|
84 |     special println!("hello world!");
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```

= note: expanding `special.println! { "hello world!" }`  

= note: to `println! ("Printed specially: {}", format! ("hello world!"))`  

= note: expanding `println! { "Printed specially: {}", format! ("hello world!") }`  

= note: to `{
    $crate :: io ::  

        print($crate :: format_args_nl!
            ("Printed specially: {}", format! ("hello world!")));
}`  

= note: expanding `format! { "hello world!" }`  

= note: to `{  

    let res = $crate :: fmt ::  

        format($crate :: export :: format_args! ("hello world!")); res
}`

```

В качестве альтернативы можно использовать `cargo expand` для вывода развернутого макроса. Это удобно, когда вы планируете работать только со стабильной версией Rust. При этом для использования возможностейочной версии всегда можно протестировать работу крейта в ней с помощью `cargo` и аргумента `+nightly`.

**СОВЕТ** Если вы никогда не использовали `cargo-expand`, то можете установить его, используя команду `cargo install cargo-expand`.

Далее для демонстрации других возможностей мы напишем еще один макрос. Он будет получать любое количество идентификаторов и выводить их значения в виде `name=value`. Такой подход может пригодиться, например, при отладке. Имейте в виду, что для этого уже существует `dbg!`, но мы в качестве упражнения напишем собственный макрос. Его определение будет выглядеть так:

```

macro_rules! var_print {
    ($($v:ident),*) => {
        println!(
            concat!($stringify!($v), "={:?} "), *),
            $($v), *
        );
    };
}

```

Принимает список идентификаторов,  
разделенных запятыми

Превращает каждый аргумент в строку и конкатенирует их  
в качестве первого аргумента для `println!`, также добавляя весь  
список оставшихся аргументов в виде второго аргумента

Этот макрос более сложный, так что разберем его по частям.

- Макрос выполняет сопоставление со списком разделенных запятыми идентификаторов, о чем говорит `$( $v:ident ),*`.
- В макросе есть два этапа разворачивания `$v`. Первое разворачивание позволяет создать первый аргумент для вызова `println!`, а второе служит для передачи оставшихся аргументов.
- Первый аргумент для `println!` — это строка формата, которая должна содержать каждую переданную в макрос переменную в виде `name=value`.
- Макрос `stringify!` преобразует каждый токен в строку.
- Макрос `concat!` конкатенирует полученные строки.

- Первое разворачивание, обозначенное как `$(stringify!($v), "={:?} ")*;`, конкатенирует все преобразованные в строки аргументы с помощью `"=:??"`.
- Второе разворачивание аргументов — это `$( $v ), *.`. Оно передается в `println!` в качестве второго аргумента.
- Обратите внимание: при сопоставлении знаки препинания `(,)` убираются, поэтому нужно их вернуть, используя `,` \* при разворачивании аргументов.

Протестируем получившийся макрос:

```
let counter = 7;
let gauge = core::f64::consts::PI;
let name = "Peter";
var_print!(counter, gauge, name);
```

При его выполнении мы получим такой вывод:

```
counter=7 gauge=3.141592653589793 name="Peter"
```

Проанализируем развернутый вид этого макроса с помощью `cargo expand`:

```
let counter = 7;
let gauge = 3.14;
let name = "Peter";
{
    ::std::io::_print(::core::fmt::Arguments::new_v1(
        &["counter=", " gauge=", " name=", "\n"],
        &[
            ::core::fmt::ArgumentV1::new_debug(&counter),
            ::core::fmt::ArgumentV1::new_debug(&gauge),
            ::core::fmt::ArgumentV1::new_debug(&name),
        ],
    )));
};
```

**ПРИМЕЧАНИЕ** Вы могли заметить, что наш макрос `var_print!` похож на `dbg!` из стандартной библиотеки, хотя у `dbg!` есть дополнительные возможности. Рекомендую более подробно изучить `dbg!`, чтобы иметь о нем более полное представление.

Обратите внимание: наш код дополнительно разворачивается с помощью команды `println!`, которая разбивает аргументы формата из одной строки в отдельные строки для каждого. Этот процесс обрабатывается внутренне компилятором.

### 5.1.3. Использование макросов для написания мини-DSL

Как я намекнул в подразделе 5.1.2, макросы в Rust можно использовать для создания небольшого предметно-ориентированного языка (DSL). При этом DSL не обязательно должен быть миниатюрным. Это вполне может быть сложный

язык. Однако в случае его создания на основе макросов лучше придерживаться более простого решения.

Хорошим примером использования макросов для создания DSL является крейт `lazy_static` (описанный в главе 4). Взгляните на определение макроса (листинг 5.3).

### Листинг 5.3. Определение макроса из крейта `lazy_static`

```
macro_rules! lazy_static {
    ($(#[$attr:meta])* static ref $N:ident : $T:ty = $e:expr; $($t:tt)*) => {
        // Используем () для явного перенаправления информации
        // о закрытых элементах
        __lazy_static_internal!($(#[$attr])* () static ref $N : $T = $e; $($t)*);
    };
    ($(#[$attr:meta])* pub static ref $N:ident : $T:ty = $e:expr;
    $($t:tt)*) => {
        __lazy_static_internal!($(#[$attr])* (pub) static ref $N : $T
    = $e; $($t)*);
    };
    ($(#[$attr:meta])* pub ($($vis:tt)+) static ref $N:ident : $T:ty =
    $e:expr; $($t:tt)*) => {
        __lazy_static_internal!($(#[$attr])* (pub $($vis)+)) static ref
    $N : $T = $e; $($t)*);
    };
    () ()
}
```

Поначалу этот макрос выглядит сложным, но по факту он весьма прост. В нем производится сопоставление лишь с двумя возможными паттернами следующего вида:

- `static ref NAME: TYPE = EXPR;`
- `pub static ref NAME: TYPE = EXPR;`

Сопоставление с `$(#[$attr:meta])*` в начале каждого паттерна позволяет при желании добавить атрибуты, а `($t:tt)*` в конце делает макрос рекурсивным путем добавления всего, что идет за первым сопоставлением в переменной `$t`.

Детали реализации, такие как генерация кода, обрабатываются макросом `lazy_static_internal`. С помощью рекурсии он разворачивает завершающую часть макроса, содержащуюся в переменной `$t`, которая следует за ; и активирует очередной этап рекурсии.

Последнее правило для сопоставления, `() => ()`, обеспечивает способ завершения рекурсии, когда совпадений больше не обнаруживается. В противном случае возникла бы ошибка, так как в очередном рекурсивном сопоставлении заключительное выражение давало бы сбой.

### 5.1.4. Использование макросов для соблюдения принципа «Не повторяйся»

Еще один типичный случай применения декларативных макросов — определение структур или блоков кода, содержащих много повторений с минимумом вариативности. Иногда нам нужно создать реализации для множества компонентов, которые совпадают во всем, кроме имени или каких-то других свойств. Обычно такие макросы используются в закрытом режиме видимости. Экспортировать их вы вряд ли захотите.

Предположим, что мы хотим создать структуру для каждой из сотен пород собак. Вместо того чтобы определять каждую такую структуру отдельно, можно использовать макрос:

```
macro_rules! dog_struct {
    ($breed:ident) => {
        struct $breed {
            name: String,
            age: i32,
            breed: String, ← Название пород можно
        }                         сохранять внутри структур
        impl $breed {
            fn new(name: &str, age: i32) -> Self {
                Self {
                    name: name.into(),
                    age,
                    breed: stringify!($breed).into(), ← Преобразуем название
                }                         в строку и сохраняем
            }
        }
    };
}

dog_struct!(Labrador);
dog_struct!(Golden);
dog_struct!(Poodle);
```

Выполнив `cargo expand`, мы увидим вывод макроя `dog_struct!`:

```
struct Labrador {
    name: String,
    age: i32,
    breed: String,
}
impl Labrador {
    fn new(name: &str, age: i32) -> Self {
        Self {
            name: name.into(),
            age,
            breed: "Labrador".into(),
        }
    }
}
```

```

struct Golden {
    name: String,
    age: i32,
    breed: String,
}
impl Golden {
    fn new(name: &str, age: i32) -> Self {
        Self {
            name: name.into(),
            age,
            breed: "Golden".into(),
        }
    }
}
struct Poodle {
    name: String,
    age: i32,
    breed: String,
}
impl Poodle {
    fn new(name: &str, age: i32) -> Self {
        Self {
            name: name.into(),
            age,
            breed: "Poodle".into(),
        }
    }
}

```

Если мы хотим реализовать на Rust функциональность отражения (reflection), то единственным способом будут макросы. Мы не можем изменять код в среде выполнения, но с помощью макроса можем эмулировать его создание на этапе компиляции. Добавим в наши структуры трейт для определения пород собак:

```

trait Dog {
    fn name(&self) -> &String;
    fn age(&self) -> i32;
    fn breed(&self) -> &String;
}

```

Трейт Dog обеспечивает аксессоры для членов структур пород, а также позволяет определять собак, используя ограничения трейтов. Обновим определение макроса, чтобы использовать созданный трейт:

```

macro_rules! dog_struct {
    ($breed:ident) => {
        struct $breed {
            name: String,
            age: i32,
            breed: String,
        }
        impl $breed {
            fn new(name: &str, age: i32) -> Self {
                Self {
                    name: name.into(),

```

```
        age,
        breed: stringify!($breed).into(),
    }
}
impl Dog for $breed {
    fn name(&self) -> &String {
        &self.name
    }
    fn age(&self) -> i32 {
        self.age
    }
    fn breed(&self) -> &String {
        &self.breed
    }
}
};
```

Протестируем это отражение:

```
let peter = Poodle::new("Peter", 7);
println!(
    "{} is a {} of age {}",
    peter.name(),
    peter.breed(),
    peter.age()
);
}
```

Вывод этого макроса: "Peter is a Poodle of age 7".

При правильном использовании декларативные макросы в Rust весьма эффективны. В случае же задач различной сложности вам доступны процедурные макросы (будем разбирать их в главе 6). Декларативные хоть и дают широкие возможности, но все же несколько ограничены.

Решение об использовании макросов определяется личными предпочтениями и стилем написания кода. Рекомендуется использовать их обдуманно и только там, где они имеют очевидное преимущество перед альтернативными решениями. Если у вас есть повторяющийся, громоздкий или уязвимый для ошибок код, то он может стать отличным кандидатом для применения макроса. Но использовать макросы нужно, только когда повторяется значительная часть кода со всего несколькими значениями, блоками, операторами или переменными, которые должны подставляться. Если же ваш код прост, прозрачен и понятен, то лучше оставить его без макросов.

**СОВЕТ** Более углубленно тема макросов в Rust вместе с различными способами их использования освещается в книге Сэма Ван Овермайера (Sam Van Overmeire) *Write Powerful Rust Macros* (Manning, 2024; <https://www.manning.com/books/write-powerful-rust-macros>). Справочная документация по макросам в языке Rust доступна по ссылке <https://mng.bz/v80m>.

## 5.2. Необязательные аргументы функций

Необязательные аргументы функций поддерживаются многими языками, но в Rust такой возможности нет. Они позволяют указывать предустановленные значения аргументов в определении функции или (в случае языков наподобие C++ и Java) делают возможной ее перегрузку. Перегрузка функции — один из способов выражения необязательных аргументов, позволяющий компилятору создавать отдельные функции с одинаковым именем, различающиеся лишь количеством или типом аргументов. И необязательные аргументы, и перегрузка функций — формы синтаксического сахара.

Такие аргументы весьма удобны, поскольку позволяют программисту обеспечивать большую гибкость кода, вызывающего функцию. Они особенно полезны, когда вам нужно добавить новые аргументы в функцию, сохранив ее обратную совместимость.

При всем при этом необязательные аргументы не лишены проблем. Если действовать их чересчур часто, это может привести к проблемам дизайна. Кроме того, они способствуют тому, чтобы разработчики использовали существующие функции, а не создавали новые, что может делать API менее понятным. Наконец, чрезмерное использование перегрузок функций усложняет понимание происходящего при вызове конкретной функции, особенно если API со временем меняется.

### 5.2.1. Необязательные аргументы в Python

Чтобы вы могли получить более полное представление о необязательных аргументах, предлагаю взглянуть на их пример в другом популярном языке: Python. Здесь такие аргументы выглядят, как показано ниже: функция `func` получает два аргумента, каждый из которых имеет значение по умолчанию:

```
def func(optional_bool=True, optional_int=11):
    # ... тело функции ...
```

В Python необязательные аргументы очень просты и сжаты. Предустановленные значения можно указывать прямо в определении функции на всеобщее обозрение, не внося лишней двусмысленности. Python даже позволяет указывать каждый аргумент по имени, а не просто по расположению. Вызовем эту функцию, чтобы указать второй аргумент:

```
func(optional_int=1024)
```

В Python необязательные аргументы работают прекрасно, но в Rust используется иной подход, с избеганием подобного стиля программирования в целях сохранения совместимости с библиотеками С.

### 5.2.2. Необязательные аргументы в C++

C++ позволяет использовать необязательные аргументы путем перегрузки функций. То есть в этом языке можно прописывать несколько определений функций с разными аргументами, и эти функции смогут подставлять значения по умолчанию для любых отсутствующих аргументов. Пример этого паттерна с тремя перегруженными функциями будет выглядеть так:

```
void func() {           ← Вызывает func() с предустановленными
    func(true, 11);   значениями
}
void func(optional_bool: bool) { ← Вызывает func() с первым
    func(optional_bool, 11);   предустановленным значением
}
void func(optional_bool: bool, optional_int: int) {
    // ... тело функции ...
}
```

C++ решает эту задачу, искажая имена функций, что делает их несовместимыми с библиотеками на основе С. Из C++ вызвать код С легко, а вот вызывать код C++ из С не стоит.

### 5.2.3. Необязательные аргументы в Rust, вернее, их отсутствие

Явное отсутствие необязательных аргументов или перегрузки в Rust — сознательное решение разработчиков, принятое ими отчасти ради совместимости с C, а отчасти во избежание критических замечаний, о которых говорилось выше. Тем не менее в Rust эту функциональность можно до определенной степени эмулировать. Для этого есть три варианта:

- расширение с помощью трейтов;
- использование макросов для сопоставления с аргументами на этапе компиляции;
- обертывание аргументов с помощью `Option`.

Мы обсудим первый вариант.

### 5.2.4. Эмуляция необязательных аргументов с помощью трейтов

Для начала покажу, что можно использовать два трейта с конфликтующими именами методов:

```
struct Container {
    name: String,
}
```

```

trait First {
    fn name(&self) {}
}
trait Second {
    fn name(&self) {}
}
impl First for Container {
    fn name(&self) {}
}
impl Second for Container {
    fn name(&self) {}
}

```

Здесь два трейта, которые различаются только названиями. Оба реализованы для структуры `Container`. Пока все выглядит прекрасно, но что произойдет, если вызвать `name()`? Попробуем:

```

let container = Container {
    name: "Henry".into(),
};
container.name();

```

Компиляция этого кода приведет к ошибке:

```

error[E0034]: multiple applicable items in scope
--> src/main.rs:25:15
   |
25 |     container.name();
   |     ^^^^^ multiple `name` found
   |
note: candidate #1 is defined in an impl of the trait `First` for the type
`Container`
--> src/main.rs:14:5
   |
14 |     fn name(&self) {}
   |     ^^^^^^^^^^^^^^
note: candidate #2 is defined in an impl of the trait `Second` for the type
`Container`
--> src/main.rs:18:5
   |
18 |     fn name(&self) {}
   |     ^^^^^^^^^^^^^^
help: disambiguate the associated function for candidate #1
--> 25 |     First::name(&container);
   |     ~~~~~~
help: disambiguate the associated function for candidate #2
--> 25 |     Second::name(&container);
   |     ~~~~~~

```

И этот вывод абсолютно обоснован. Невозможно исключить двусмысленность вызова функции, и компилятор дает на этот счет полезные подсказки.

А что произойдет, если методы трейтов будут иметь разные сигнатуры? Добавим аргумент в трейт `Second` (параметр `bool`):

```
trait First {
    fn name(&self) {}
}

trait Second {
    fn name(&self, _: bool) {}
}

impl First for Container {
    fn name(&self) {}
}

impl Second for Container {
    fn name(&self, _: bool) {}
}
```

Этот код вроде работает, но при компиляции вы получите ту же ошибку. Попробуем иначе. Можно использовать ограничения трейтов, определив две эти функции так:

```
fn get_name_from_first<T: First>(t: &T) {
    t.name() ← Вызывает name() из First, которая получает только &self
}

fn get_name_from_second<T: Second>(t: &T) {
    t.name(true) ← Вызывает name() из Second, которая получает &self и bool
}
```

Протестируем:

```
let container = Container {
    name: "Henry".into(),
};

get_name_from_first(&container);
get_name_from_second(&container);
```

Компилятор это устроит. Таким образом, мы узнали, что можем использовать ограничения трейтов, чтобы указать компилятору, какие методы нужно использовать в зависимости от контекста. Даже когда у нас есть несколько конфликтующих трейтов, компилятор игнорирует те, которые не указаны в ограничениях. Если у нас есть обобщенная функция и мы пытаемся вызвать какой-либо метод для обобщенного параметра, то компилятор будет ругаться:

```
fn get_name<T>(t: &T) {
    t.name()
}
```

Этот код дает ошибку:

```
error[E0599]: no method named `name` found for reference `&T` in the current
scope
--> src/main.rs:29:7
 |
29 |     t.name()
|     ^^^^ method not found in `&T`
```

```

= help: items from traits can only be used if the type parameter is
  bounded by the trait
help: the following traits define an item `name`, perhaps you need to
restrict type parameter `T` with one of them:
|
28 | fn get_name<T: First>(t: &T) {
      ~~~~~
28 | fn get_name<T: Second>(t: &T) {
      ~~~~~

```

В данном примере интересно то, что компилятор удачно угадывает наши намерения. Понимая это, можно начать смотреть на необязательные аргументы несколько иначе. Нам известно следующее:

- имена функций и методов не должны совпадать, даже если содержат разные аргументы;
- трейты можно реализовывать с использованием в типе конфликтующих методов;
- если использовать дженерики, то можно указывать ограничения трейтов, чтобы исключить двусмысленность конфликтующих методов.

Таким образом, мы можем закладывать в программу ожидание некой функциональности, которую предоставляют трейты. В Rust сделать это несложно, поскольку можно добавить ограничения трейтов в любую функцию. При этом в Rust обычно лучше получать обобщенные параметры функций, за исключением базовых типов наподобие `String` и численных значений.

### 5.3. Паттерн «Строитель»

«Строитель» — один из оригинальных паттернов, описанных в работе «Банды четырех». Он стал невероятно популярным в сфере проектирования ПО и, не считая итераторов, является одним из самых долгоживущих паттернов, описанных в культовой книге. Кроме того, «Строитель» можно рассмотреть как вариант каррирования. *Каррирование* — это способ преобразования функции, получающей несколько аргументов, в набор функций, получающих по одному аргументу каждая.

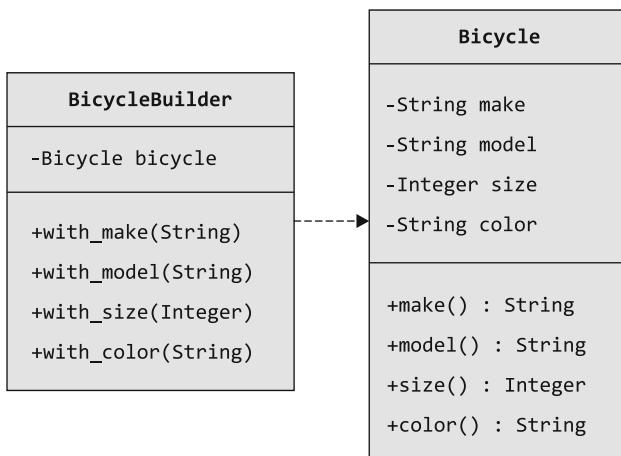
Я большой поклонник паттерна «Строитель» и считаю его настолько полезным, что посвятил ему целый раздел текущей главы. Реализовать строитель в Rust не слишком сложно, но чуть позже мы разберем пример, который поможет связать воедино все, что вы усвоили до этого момента.

К использованию этого паттерна вас могут подтолкнуть разные причины, такие как инкапсуляция, удобство, разделение обязанностей, эргономика и безопасность. В частности, в Rust мы обычно не хотим раскрывать структуры напрямую, и, как я говорил в подразделе 5.2.4, данный язык не поддерживает необязательные аргументы. Поэтому вместо того, чтобы использовать конструкторы со множеством аргументов, можно обрабатывать более сложные случаи с помощью строителей.

Тем не менее этот паттерн не лишен проблем, поскольку привносит дополнительный уровень сложности. Так что понимание, когда его использование уместно, — скорее искусство, нежели наука.

### 5.3.1. Реализация паттерна «Строитель»

Напишем простой вид строителя для велосипеда, который мы хотим смоделировать. На рис. 5.1 показано, какие связи мы будем моделировать.



**Рис. 5.1.** UML-диаграмма паттерна «Строитель»

Приступим к реализации (листинг 5.4).

#### Листинг 5.4. Код паттерна «Строитель»

```

#[derive(Debug)]
struct Bicycle {
    make: String,
    model: String,
    size: i32,
    color: String,
}

impl Bicycle { ←
    fn make(&self) -> &String {
        &self.make
    }
    fn model(&self) -> &String {
        &self.model
    }
    fn size(&self) -> i32 {
        &self.size
    }
}
  
```

Предоставляем аксессоры  
для структуры `Bicycle`

```

fn color(&self) -> &String {
    &self.color
}

struct BicycleBuilder {
    bicycle: Bicycle,
}

impl BicycleBuilder {
    fn new() -> Self {
        Self {
            bicycle: Bicycle {
                make: String::new(),
                model: String::new(),
                size: 0,
                color: String::new(),
            },
        }
    }

    fn with_make(&mut self, make: &str) {
        self.bicycle.make = make.into()
    }

    fn with_model(&mut self, model: &str) {
        self.bicycle.model = model.into()
    }

    fn with_size(&mut self, size: i32) {
        self.bicycle.size = size
    }

    fn with_color(&mut self, color: &str) {
        self.bicycle.color = color.into()
    }

    fn build(self) -> Bicycle {
        self.bicycle
    }
}

```

Структура BicycleBuilder содержит данные о велосипеде

Построение BicycleBuilder приведет к инициализации Bicycle со значениями по умолчанию

Для каждого свойства Bicycle создадим функцию, присваивающую значение (типа сеттера)

Вызов build() заберет владение строителем и вернет извлеченный из него Bicycle

Наша реализация соответствует базовому определению строителя. Теперь протестируем ее:

```

let mut bicycle_builder = BicycleBuilder::new();
bicycle_builder.with_make("Huffy");
bicycle_builder.with_model("Radio");
bicycle_builder.with_size(46);
bicycle_builder.with_color("red");
let bicycle = bicycle_builder.build();
println!("My new bike: {:?}", bicycle);

```

При выполнении этого кода мы получим такой вывод:

```

My new bike: Bicycle {
    make: "Huffy",
    model: "Radio",
    size: 46,
    color: "red",
}

```

### 5.3.2. Расширение строителя трейтами

Нашу реализацию можно улучшить. Начнем с создания трейта `Builder`:

```
trait Builder<T> {
    fn new() -> Self;
    fn build(self) -> T;
}
```

Немного перестроим код для `BicycleBuilder`, чтобы реализовать этот новый трейт:

```
impl Builder<Bicycle> for BicycleBuilder {
    fn new() -> Self {
        Self {
            bicycle: Bicycle {
                make: String::new(),
                model: String::new(),
                size: 0,
                color: String::new(),
            },
        }
    }
    fn build(self) -> Bicycle {
        self.bicycle
    }
}
```

Попутно нужно добавить в `Bicycle` трейт, который обеспечит для нас экземпляр строителя:

```
trait Buildable<Target, B: Builder<Target>> {
    fn builder() -> B;
}
```

Далее реализуем трейт `Buildable` для `Bicycle`:

```
impl Buildable<Bicycle, BicycleBuilder> for Bicycle {
    fn builder() -> BicycleBuilder {
        BicycleBuilder::new()
    }
}
```

Теперь можно получать новый экземпляр строителя прямо из `Bicycle`:

```
let mut bicycle_builder = Bicycle::builder();
bicycle_builder.with_make("Huffy");
bicycle_builder.with_model("Radio");
bicycle_builder.with_size(46);
bicycle_builder.with_color("red");
let bicycle = bicycle_builder.build();
println!("My new bike: {:?}", bicycle);
```

Наш код постепенно обретает форму, более свойственную Rust.

### 5.3.3. Расширение строителя макросами

Если взглянуть на методы `with_...()` в нашем строителе, то они кажутся лишними. Иногда нам нужно специализировать эти функции, но чаще лучше написать простой макрос. Использовать макрос для объемного или повторяющегося кода полезно, так как помогает избежать опечаток. Попробуем этот подход, заменив упомянутые методы макросами (листинг 5.5).

#### Листинг 5.5. Добавление в BicycleBuilder макросов `with_str!` и `with!`

```
macro_rules! with_str {
    ($name:ident, $func:ident) => {
        fn $func(&mut self, $name: &str) {
            self.bicycle.$name = $name.into()
        }
    };
}

macro_rules! with {
    ($name:ident, $func:ident, $type:ty) => {
        fn $func(&mut self, $name: $type) {
            self.bicycle.$name = $name
        }
    };
}

impl BicycleBuilder {
    with_str!(make, with_make);
    with_str!(model, with_model);
    with!(size, with_size, i32);
    with_str!(color, with_color);
}
```

`with_str!` получает два идентификатора:  
имена своего члена и функции

Отображаемая функция  
присваивает аргумент  
непосредственно члену,  
вызывая `into()` (из трейта `Into`)

`Макрос with! аналогичен, только он  
также получает аргумент типа`

В листинге 5.5 прописаны два макроса: `with_str!` и `with!`. Макрос `with_str!` используется для строковых полей, так как мы в целях удобства хотим получать `&str`, но сохранять это поле в виде `String`. Макрос `with!` получает параметр типа, и мы предполагаем, что его значение передается путем перемещения. Один макрос можно использовать, чтобы сделать тип необязательным, но так код получается более понятным.

**СОВЕТ** Небольшие одноразовые макросы вроде тех, что показаны в нашем примере, достаточно распространены. Вы можете избежать набора лишнего кода и ошибок, разделив типичные части на небольшие макросы, которые можно использовать повторно.

На данном этапе мы свой строитель уже особо не улучшим. Можно сделать его чуть более обобщенным, но пользы от этого будет немного.

Мы не затронули еще один момент — видимость. Нам стоит раскрыть наши типы, трейты, аксессоры и методы строителя. Сделать это можно, используя ключевое

слово `pub` для трейта `Buildable`, `Bicycle` и `BicycleBuilder`. Для начала обновим трейт `Buildable` и структуру `Bicycle` (листинг 5.6).

### Листинг 5.6. Делаем публичными `Bicycle` и `Buildable`

```
pub trait Buildable<Target, B: Builder<Target>> { ← Трейт Buildable
    fn builder() -> B;
}

#[derive(Debug)]
pub struct Bicycle { ← Структура Bicycle стала открытой
    make: String,
    model: String,
    size: i32,
    color: String,
}
impl Buildable<Bicycle, BicycleBuilder> for Bicycle {
    fn builder() -> BicycleBuilder {
        BicycleBuilder::new()
    }
}
```

Теперь сделаем открытым трейт `Builder` и `BicycleBuilder` (листинг 5.7).

### Листинг 5.7. Делаем публичными `Builder` и `BicycleBuilder`

```
pub trait Builder<T> { ← Трейт Builder стал открытым
    fn new() -> Self;
    fn build(self) -> T;
}

pub struct BicycleBuilder { ← Структура BicycleBuilder стала открытой
    bicycle: Bicycle,
}
impl Builder<Bicycle> for BicycleBuilder {
    fn new() -> Self {
        Self {
            bicycle: Bicycle {
                make: String::new(),
                model: String::new(),
                size: 0,
                color: String::new(),
            },
        }
    }
    fn build(self) -> Bicycle {
        self.bicycle
    }
}
```

Внесем в код еще одну доработку, добавив макросы для аксессоров. Итоговая форма нашего строителя показана в листинге 5.8.

### Листинг 5.8. Итоговые Bicycle и BicycleBuilder с макросами

```

macro_rules! accessor {
    ($name:ident, &$ret:ty) => {
        pub fn $name(&$self) -> &$ret {
            &$self.$name
        }
    };
    ($name:ident, $ret:ty) => {
        pub fn $name(&$self) -> $ret {
            self.$name
        }
    };
}

impl Bicycle {
    accessor!(make, &String);
    accessor!(model, &String);
    accessor!(size, i32);
    accessor!(color, &String);
}

macro_rules! with_str {
    ($name:ident, $func:ident) => {
        pub fn $func(&mut self, $name: &str) {
            self.bicycle.$name = $name.into()
        }
    };
}

macro_rules! with {
    ($name:ident, $func:ident, $type:ty) => {
        pub fn $func(&mut self, $name: $type) {
            self.bicycle.$name = $name
        }
    };
}

impl BicycleBuilder {
    with_str!(make, with_make);
    with_str!(model, with_model);
    with!(size, with_size, i32);
    with_str!(color, with_color);
}

```

**ПРИМЕЧАНИЕ** Несмотря на то что реализация этих паттернов является интересным способом изучения языка и его возможностей, многое из этой функциональности уже реализовано в различных крейтах. К примеру, крейт derive\_builder ([https://crates.io/crates/derive\\_builder](https://crates.io/crates/derive_builder)) предоставляет способ создания строителей с помощью атрибута #[derive]. Конечно, полезно уметь реализовывать эти паттерны самостоятельно, но также нужно знать, когда можно использовать существующие решения (такие как derive\_builder), чтобы экономить время и силы. В частности, одним из богатых на функциональность и проверенных крейтов является вышеупомянутый derive\_builder.

## 5.4. Паттерн «Гибкий интерфейс»

«Гибкий интерфейс» (*fluent interface*, называют также *текучим*) основывается на паттерне «Строитель». Его основные черты — использование цепочек методов. Цепочки методов — это практика объединения вызовов функций, позволяющего выполнять операцию до тех пор, пока кто-нибудь ее не завершит (обычно путем вызова метода).

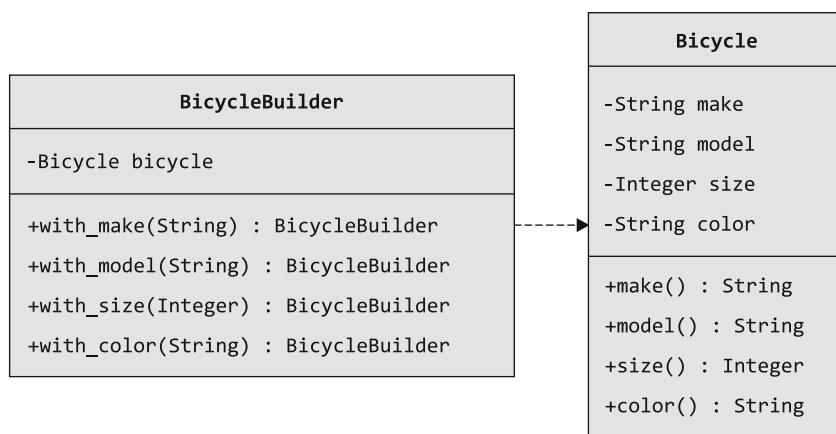
Мы уже видели хороший пример этого паттерна в Rust — это был трейт `Iterator`. Для реализации цепочки методов нужно возвращать из каждого вызова метода тип, который будет вести к следующему звену цепи. Вот сигнатура метода `map()` в трейте `Iterator`:

```
fn map<B, F>(self, f: F) -> Map<Self, F> where
    F: FnMut(Self::Item) -> B { ... }
```

Здесь возвращаемый тип — это `Map`, который является еще одним итератором. Можно вызвать `map()` еще раз, на что вернется очередной `Map` и т. д. В теории таким образом можно связывать функции бесконечно.

### 5.4.1. Гибкий строитель

Для наглядной демонстрации вернемся к примеру строителя из предыдущего раздела. Мы обновим операции присваивания методов, чтобы они возвращали строитель. Обновленная версия диаграммы на языке UML (Unified Modeling Language) показана на рис. 5.2, где при каждом присваивании метода возвращается новый строитель.



**Рис. 5.2.** UML-диаграмма для паттерна «Гибкий строитель»

Поскольку мы использовали макросы, то осталось лишь обновить их для реализации этого изменения:

```

macro_rules! with_str {
    ($name:ident, $func:ident) => {
        pub fn $func(self, $name: &str) -> Self {
            Self {
                bicycle: Bicycle {
                    $name: $name.into(),
                    ..self.bicycle
                },
            }
        }
    };
}

macro_rules! with {
    ($name:ident, $func:ident, $type:ty) => {
        pub fn $func(self, $name: $type) -> Self {
            Self {
                bicycle: Bicycle {
                    $name,
                    ..self.bicycle
                },
            }
        }
    };
}

```

В развернутом виде код нашего строителя выглядит так:

```

impl BicycleBuilder {
    pub fn with_make(self, make: &str) -> Self {
        Self {
            bicycle: Bicycle {
                make: make.into(),
                ..self.bicycle
            },
        }
    }
    pub fn with_model(self, model: &str) -> Self {
        Self {
            bicycle: Bicycle {
                model: model.into(),
                ..self.bicycle
            },
        }
    }
    pub fn with_size(self, size: i32) -> Self {
        Self {
            bicycle: Bicycle {
                size,
                ..self.bicycle
            },
        }
    }
    pub fn with_color(self, color: &str) -> Self {
        Self {

```

```

        bicycle: Bicycle {
            color: color.into(),
            ..self.bicycle
        },
    }
}
}

```

Отлично! Обратите внимание на пару нюансов этого кода.

- Методы присваивания получают `self`, а не `&mut self`. Иными словами, каждый вызов метода присваивания потребляет (consume) предыдущий строитель.
- Вместо того чтобы копировать или возвращать старый строитель и внутреннюю структуру, мы создадим новый с новой структурой.
- Мы используем синтаксис распаковки `(..)` для инициализации структуры `Bicycle` с обновленным полем.

### Инициализация структур с помощью синтаксиса распаковки

Если вы не встречали применение синтаксиса распаковки для инициализации структур, то ничего страшного. Эта нотация позволяет инициализировать структуры со значениями существующей структуры, одновременно обновляя конкретные поля. В этой операции используется перемещение, поэтому при инициализации новой структуры она поглощает существующую. Такая нотация является синтаксическим сахаром, облегчающим обработку структур со множеством полей.

Одним из удобных побочных эффектов синтаксиса распаковки является то, что он позволяет менять поля в структуре, даже если они неизменяемы. Но работает это только при владении структурой. Наша структура `Bicycle` демонстрирует этот принцип:

```

let bicycle1 = Bicycle {
    make: "Rivendell".into(),
    model: "A. Homer Hilsen".into(),
    size: 51,
    color: "red".into(),
};
println!("{}?", bicycle1);
let bicycle2 = Bicycle {
    size: 58,
    ..bicycle1
};
println!("{}?", bicycle2);
// println!("{}?", bicycle1);

```

← Создаем новый экземпляр структуры Bicycle с указанием всех полей

← Создаем новый экземпляр той же структуры, но с изменением размера поля

← Нельзя использовать `bicycle1` после использования синтаксиса распаковки, так как эта переменная перемещается в новую структуру

При выполнении предыдущего кода мы получим такой вывод:

```

Bicycle { make: "Rivendell", model: "A. Homer Hilsen", size: 51,
color: "red" }
Bicycle { make: "Rivendell", model: "A. Homer Hilsen", size: 58,
color: "red" }

```

Во время присваивания используется перемещение, поэтому нельзя использовать распаковку со ссылками. Попытка скомпилировать следующий код приведет к ошибке:

```
let bicycle = Bicycle {
    make: "Rivendell".into(),
    model: "A. Homer Hilsen".into(),
    size: 51,
    color: "red".into(),
};

let bicycle = Bicycle {
    size: 58,
    ..&bicycle
};
```

Компилятор выдает ошибку  
с сообщением «несоответствие типов,  
ожидался Bicycle, получен &Bicycle»

## 5.4.2. Тестирование гибкого строителя

Теперь обновим тестовый код, чтобы использовать новый гибкий интерфейс:

```
let bicycle = Bicycle::builder()
    .with_make("Trek")
    .with_model("Madone")
    .with_size(52)
    .with_color("purple")
    .build();
println!("{:?}", bicycle);
```

Прекрасно! Выглядит намного лучше старого.

## 5.5. Паттерн «Наблюдатель»

Паттерн «Наблюдатель» — один из паттернов, описанных в книге «Банды четырех». Он имеет много вариаций и довольно популярен. Его используют для того, чтобы дать объектам возможность видеть изменения в других объектах. Наблюдатель часто необходим в системах, выполняющих обработку событий, например в сетевых службах.

### 5.5.1. А почему не обратные вызовы?

Прежде чем углубляться в изучение паттерна «Наблюдатель», нужно обсудить обратные вызовы. Некоторые языки (в частности, JavaScript) активно используют обратные вызовы, что может вести к так называемому *аду обратных вызовов* (callback hell). Это ситуация, когда обратные вызовы глубоко вложены друг в друга и становится довольно сложно понять код. Кто-то даже разработал сайт <http://callbackhell.com>, чтобы описать эту проблему и предложить какие-то решения.

Обратные вызовы часто используются в функциональных языках внутри *функций высшего порядка*. Таковой является функция, которая получает другую функцию в качестве параметра или возвращает другую функцию. К примеру,

итераторы используют обратные вызовы с помощью функций наподобие `map()`. Вот простейшая форма обратного вызова в Rust:

```
fn callback_fn<F>(f: F)
where
    F: Fn() -> (),
{
    f();
}

fn main() {
    let my_callback = || println!("I have been called back");
    callback_fn(my_callback); ← Обратный вызов происходит в callback_fn()
}
```

Здесь ничего не происходит. Наша функция не вызывалась — мы ее только объявили

Здесь для обратного вызова я использую замыкание (как обычно происходит в JavaScript), но можно было просто передать стандартную функцию. Подобные простые случаи вполне нормальны, но, когда у вас внутри одних обратных вызовов есть другие, содержащие еще один уровень обратных вызовов, уследить за логикой выполнения становится сложно.

Несмотря на то что обратные вызовы не являются проблемой сами по себе, паттерн «Наблюдатель» обеспечивает более слабое зацепление, упрощает закрепление и открепление наблюдателей (эквивалент обратным вызовам) и позволяет реализовывать отношение по типу «многие к одному» вместо «один к одному». В более общем смысле этот паттерн можно использовать, когда у нас есть код (субъект), который должен уведомлять о событиях другой код, не создавая зависимости от наблюдателей. То есть субъект может уведомлять наблюдатели, и для этого ему не нужно их знать.

Еще одна проблема с обратными вызовами заключается в том, что они не позволяют отделять состояние от передаваемой в них функции. Обязательно нужно привязывать состояние к обратному вызову, используя замыкание или глобальные переменные.

### 5.5.2. Реализация наблюдателя

Реализовать наблюдатель можно по-разному, и каждый способ будет сопряжен с определенными компромиссами. В этом подразделе я покажу достаточно гибкий пример, чтобы вы могли менять детали его реализации под различные ситуации. На рис. 5.3 схематично показано, как именно мы реализуем этот паттерн.

Начнем с реализации двух трейтов: `Observer` и `Observable`. Первый мы используем для объектов, которым нужно наблюдать за другими объектами, а второй,

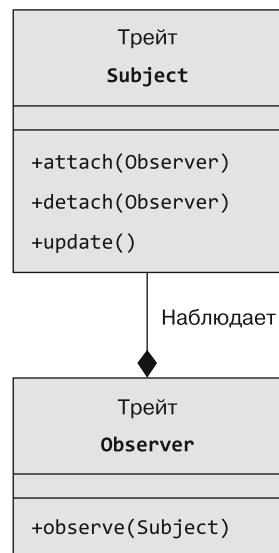


Рис. 5.3. UML-диаграмма паттерна «Наблюдатель»

напротив, будет реализовываться теми объектами, которые должны быть видимы другим. В листинге 5.9 показан трейт `Observer`.

### Листинг 5.9. Трейт `Observer`

```
pub trait Observer {
    type Subject; ← Используем для субъекта
    fn observe(&self, subject: &Self::Subject); ← ассоциированный тип
} ← Метод observe() вызывается
      субъектом, когда происходит
      обновление
```

В отношении паттерна «Наблюдатель» я использую термин «*наблюдать*», а не «*уведомлять*» (в соответствии с оригинальным паттерном проектирования). Теперь рассмотрим листинг 5.10, демонстрирующий трейт `Observable`.

### Листинг 5.10. Трейт `Observable`

```
pub trait Observable {
    type Observer; ← Для наблюдателя используется ассоциированный тип
    fn update(&self);
    fn attach(&mut self, observer: Self::Observer);
    fn detach(&mut self, observer: Self::Observer);
}
```

Трейт `Observable` обеспечивает методы для нашего субъекта и соответствует оригинальному паттерну проектирования. Мы не делаем никаких допущений относительно типа наблюдателя или субъекта, что делает наш паттерн более гибким. Далее нужно создать субъект и реализовать для него трейт `Observable` (листинг 5.11).

### Листинг 5.11. Реализация трейта `Observable` для `Subject`

```
pub struct Subject {
    observers: Vec<Weak<dyn Observer<Subject = Self>>>, ← Мы сохраняем слабые указатели на объекты
                                                Observer, где субъектом выступает Self
}
impl Observable for Subject { ← Наблюдатель необходимо предоставить
    type Observer = Arc<dyn Observer<Subject = Self>>; ← в виде Arc, чтобы повысить гибкость
    fn update(&self) { ← и обеспечить общее владение
        self.observers
            .iter()
            .flat_map(|o| o.upgrade())
            .for_each(|o| o.observe(self)); ← self.observers содержит
                                              слабые ссылки,
                                              которые здесь нужно
                                              модифицировать
                                              (upgrade). Поскольку
                                              upgrade() при выполнении
                                              в отношении Weak
                                              возвращает option,
                                              то flat_map()
                                              разворачивает и удаляет
                                              случаи None
    }
    fn attach(&mut self, observer: Self::Observer) { ← В завершение вызываем
        self.observers.push(Arc::downgrade(&observer)); ← observe() для каждого
                                                               действительного
                                                               наблюдателя
    }
    fn detach(&mut self, observer: Self::Observer) { ← При добавлении нового
        self.observers
            .retain(|f| {
                !f.ptr_eq(&Arc::downgrade(&observer))
            });
    } ← Для поиска совпадающего объекта нужно использовать ptr_eq(). Vec::retain()
          отбирает все объекты, которые соответствуют переданному в этот метод указателю
}
```

Чтобы предоставить дополнительную гибкость, я решил потребовать, чтобы наблюдатели передавались в виде `Arc<dyn Observer>`. Во-первых, это позволит сохранять указатели в качестве слабых. То есть при их выходе из области видимости мы сможем игнорировать их, не сохраняя объект в действующем виде. Вдобавок при использовании `Arc` возможно общее владение (то есть мы не хотим, чтобы наш субъект получал исключительное владение над наблюдателями). Наблюдатель определяется в трейте как соответствующий тип, поэтому можно легко изменить его с `Arc` на какой-нибудь другой, повторно использовав те же трейты.

Далее добавим нашему субъекту состояние, чтобы можно было его тестировать, а также обеспечим аксессор и добавим метод `new()`. В листинге 5.12 показан обновленный код.

### Листинг 5.12. Добавление state и new() к Subject

```
pub struct Subject {
    observers: Vec<Weak<dyn Observer<Subject = Self>>>,
    state: String,
}

impl Subject {
    pub fn new(state: &str) -> Self {
        Self {
            observers: vec![],
            state: state.into(),
        }
    }

    pub fn state(&self) -> &str {
        self.state.as_ref()
    }
}
```

Далее создадим наблюдатель и реализуем для него трейт `Observer` (листинг 5.13).

### Листинг 5.13. Создание наблюдателя

```
struct MyObserver {
    name: String, ← | Даем наблюдателю имя, чтобы
}                                     | можно было его идентифицировать

impl MyObserver {
    fn new(name: &str) -> Arc<Self> { ← | Метод new() будет вместо Self
        Arc::new(Self { name: name.into() })
    }
}

impl Observer for MyObserver {           | Тип нашего субъекта Subject;
    type Subject = Subject;              | он определен в листинге 5.12
    fn observe(&self, subject: &Self::Subject) { ← |
        println!(                                | При вызове observe() выводит
            "observed subject with state={:{}?} in {}",
            subject.state
        )
    }
}
```

```
        subject.state(),
        self.name
    );
}
```

В завершение протестируем наш наблюдатель (листинг 5.14).

### Листинг 5.14. Тестирование наблюдателя

```
let mut subject = Subject::new("some subject state");
```

```
let observer1 = MyObserver::new("observer1");
let observer2 = MyObserver::new("observer2");
```

`subject.attach(observer1.clone());` |  
`subject.attach(observer2.clone());`

Нам нужно клонировать указатель;  
в противном случае он при передаче  
по значению выйдет из области видимости

// ... что-то происходит ...  
subject.update();

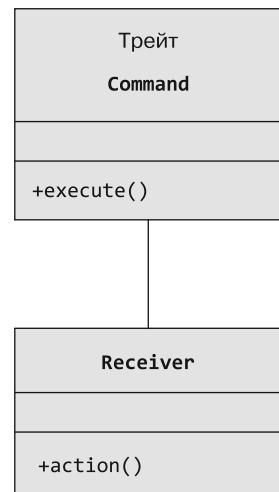
Обычно при любом изменении состояния субъекта мы бы для активации наблюдателей вызывали update() внутри него, но в текущем примере вызываем его здесь

При выполнении этого кода мы получим такой вывод:

observed subject with state="some subject state" in observer1  
observed subject with state="some subject state" in observer2

## 5.6. Паттерн «Команда»

*Паттерн «Команда» хранит состояние или инструкции в одной структуре и позволяет изменить их позже. Он довольно популярен, но не имеет хорошей спецификации и немного устарел. Но мне хочется раскрыть тему паттернов максимально полно, поэтому я опишу простой пример его реализации на Rust.*



**Рис. 5.4.** UML-диаграмма паттерна «Команда»

В листинге 5.15 показано определение трейта `Command`.

### Листинг 5.15. Определение трейта `Command`

```
trait Command {
    fn execute(&self) -> Result<(), Error>;
}
```

Обратите внимание: я сделал так, чтобы трейт `Command` возвращал результат, что делает возможным простую обработку ошибок (как мы увидим в подразделе 5.6.2). Трейт `Command` — центральный элемент паттерна «Команда», но, чтобы понять этот паттерн, нам нужно собрать все элементы воедино. Реализуем поддержку получателя, который является любым объектом, где может выполняться команда в соответствии с определением конкретной реализации трейта `Command`.

### 5.6.2. Реализация паттерна «Команда»

Для этого примера мы создадим два объекта команд, работающих с дескрипторами файлов: команду для чтения файла и команду для записи в него. Получателем будет выступать дескриптор файла. Первой определим команду `ReadFile` (листинг 5.16).

### Листинг 5.16. Реализация команды `ReadFile`

```
struct ReadFile {
    receiver: File, ←
}

impl ReadFile {
    fn new(receiver: File) -> Box<Self> { ←
        Box::new(Self { receiver }) ←
    }
}

impl Command for ReadFile {
    fn execute(&self) -> Result<(), Error> { ←
        println!("Reading from start of file");
        let mut reader = BufReader::new(&self.receiver); ←
        reader.seek(std::io::SeekFrom::Start(0))?; ←
        for (count, line) in reader.lines().enumerate() { ←
            println!("{}: {}", count + 1, line?); ←
        }
        Ok(())
    }
}
```

receiver — это получатель (или цель) команды

Мы возвращаем объект в кучу, чтобы можно было использовать объекты трейтов позднее. На данном этапе не обязательно возвращать Box, но так можно сделать код чище и сообщить вызывающему, как предполагается его использовать

Мы используем буферизованный ридер, чтобы получить несколько приятных возможностей наподобие итератора, перебирающего строки файла

Мы всегда перемещаемся в начало файла перед его повторным считыванием. Для изящной обработки ошибок ввода/вывода используем оператор ?

Выводится каждая строка, и если при чтении файла возникнет ошибка, то мы вернем ее с помощью оператора ?

Мы индексируем строки, чтобы вместе с содержимым выводить их номер

Заметьте, что мы реализуем для `ReaderFile` метод `new()` `ReadFile`, который получает дескриптор файла и возвращает выделенный в куче объект `ReadFile`. Как мы увидим в листинге 5.18, это очень важный процесс. Далее определим команду `WriteFile` (листинг 5.17).

### Листинг 5.17. Реализация команды WriteFile

```
struct WriteFile {
    content: String, ← Эта команда включает в себя поле content, представляющее
    receiver: File, ← содержимое, которое мы хотим записать в файл
}
impl WriteFile {
    fn new(content: String, receiver: File) -> Box<Self> {
        Box::new(Self { content, receiver }) ← receiver — это дескриптор файла,
    }                                                 к которому применяется команда
}
impl Command for WriteFile {
    fn execute(&self) -> Result<(), Error> {
        println!(" Writing new content to file");
        let mut writer = self.receiver.try_clone()?;
        writer.write_all(self.content.as_bytes())?; ← Как и в ReadFile, здесь мы возвращаем
        writer.flush()?; ← объект, выделенный в куче
    }
    Ok(())
}
```

Для записи в файл нам нужен изменяемый объект, и самый простой способ получить его — клонировать дескриптор файла

Преобразуем строку UTF-8 в сырье байты и записываем их в текущую позицию курсора в файле

Сбрасываем буферизованные данные на диск, чтобы гарантировать запись всех файлов. При этом обрабатываем ошибки с помощью оператора ?

Команда `WriteFile` во многом похожа на `ReadFile`, за исключением того, что также имеет в качестве аргумента содержимое, которое мы хотим добавить в файл. Обратите внимание: в данной реализации предполагается, что `WriteFile` может легко делать запись в текущую позицию файлового дескриптора, но в более надежном ее варианте перед записью будет всегда происходить перемещение в конец файла (этую реализацию можно назвать `AppendFile`). Попробуйте внести это изменение самостоятельно.

Далее нужно реализовать клиент для этого паттерна. Добавим его в метод `main()` (листинг 5.18).

### Листинг 5.18. Реализация клиента для паттерна «Команда»

```
use std::fs::File;
use std::io::{BufRead, BufReader, Error, Seek, Write};

fn main() -> Result<(), Error> {
    let file = File::options() ← Используем std::fs::File из стандартной библиотеки,
    .read(true)           чтобы открыть файл в режиме чтения/записи и если его нет,
    .write(true)          то создать, а если есть, то открыть его в режиме дополнения
    .create(true)
    .append(true)
    .open("file.txt")?;
}
```

```

    Используем объекты трейта с помощью
    Box<dyn Command>, что позволяет
    добавлять любые команды,
    реализующие трейт Command
    ↓
let commands: Vec<Box<dyn Command>> = vec![
    ReadFile::new(file.try_clone()?), ←
    WriteFile::new(
        "file content\n".into(), file.try_clone()?) ←
    ), ←
    ReadFile::new(file.try_clone()?), ←
];
    ↓
for command in commands { ←
    command.execute()?;
}
    ↓
Ok(())
}

При создании каждой команды
мы клонируем дескриптор
файла и передаем его в нее
    ↓
Заметьте, что в конце
содержимого файла нужно
добавить символ перевода
строки \n, чтобы в случае
текстового файла получить
отдельные строки
    ↓
Перебираем каждый объект
команды и вызываем его
метод execute(), используя
оператор ? для обработки
ошибок

```

**ПРИМЕЧАНИЕ** Мы ввели понятие трейт-объектов (как в случае использования dyn Trait) в главе 2, когда разбирали тему трейтов.

Теперь можно протестировать код с помощью команды cargo run, которая (при условии, что file.txt не существует) выдаст следующее:

```

Reading from start of file
Writing new content to file
Reading from start of file
 1: file content

```

Данный пример имеет состояние, поэтому повторное выполнение кода будет давать разные результаты. Объясняется это тем, что файл при предыдущем выполнении был изменен. При повторном выполнении мы получим следующее:

```

Reading from start of file
 1: file content
Writing new content to file
Reading from start of file
 1: file content
 2: file content

```

Паттерн «Команда» может быть намного сложнее, поскольку порой используется для операций с сохранением состояния, например тех, которые могут выполнять прямое и обратное действие, как в системе undo/redo, позволяющей применять и отменять изменения. Чтобы этот подход работал, команды должны быть идемпотентными и отслеживать необходимое состояние при прямом и обратном выполнении. В предыдущем примере команда WriteFile не идемпотентна, так как выполняет операцию добавления и не производит каждый раз перемещение в конец файла. Чтобы сделать ее идемпотентной, нужно каждый раз выполнять перемещение к началу файла и переписывать все его содержимое.

## 5.7. Паттерн «Новый тип»

Паттерн «Новый тип» представляет собой расширение *структур кортежа* (особые структуры в Rust, действующие как кортежи), которое с помощью системы типов предоставляет информацию о типах или производит обработку данных. «Новый тип» полезен, когда сами данные достаточно локализованы с помощью основного или простейшего типа, например `String` или `i32`. Но вам нужно избегать добавления излишней инкапсуляции или косвенности поверх базового типа, делая возможным прямой доступ через кортеж.

«Новый тип» можно рассматривать как легковесный паттерн для предоставления дополнительного контекста или информации поверх кортежей с сохранением их удобства и простоты. Помимо этого, он используется для обеспечения возможности типобезопасного преобразования типов данных. Этот паттерн может показаться обманчиво простым, хотя на самом деле он бывает весьма удобен для работы с системой типов Rust.

**ПРИМЕЧАНИЕ** Во вступлении к текущей главе я отметил, что «Новый тип» относится больше к Rust, хотя по факту ничто не мешает применять тот же принцип в других языках программирования. Я лишь имел в виду, что этот паттерн, насколько мне известно, был создан в сообществе Rust.

Чтобы продемонстрировать применение паттерна «Новый тип», мы создадим типы `BitCount` и `ByteCount`, которые используем для хранения количества битов и байтов. Нам известно, что 1 байт содержит 8 бит, значит, можно определить методы для простого и безопасного преобразования между этими типами. Для начала напишем структуры кортежа, каждая из которых будет иметь тип `u32`:

```
#[derive(Debug)]
struct BitCount(u32);
#[derive(Debug)]
struct ByteCount(u32);
```

Этот код представляет простейший пример паттерна «Новый тип». Протестируем его:

```
let bits = BitCount(8);
let bytes = ByteCount(12);
dbg!(&bits);
dbg!(&bytes);
```

А таким будет вывод:

```
[src/main.rs:9] &bits = BitCount(
    8,
)
[src/main.rs:10] &bytes = ByteCount(
    12,
)
```

Далее нужно выполнить преобразование между количеством битов и байтов. Определим для этого два метода:

```
impl BitCount {
    fn to_bytes(&self) -> ByteCount {
        ByteCount(self.0 / 8) ←
    }
}

impl ByteCount {
    fn to_bits(&self) -> BitCount {
        BitCount(self.0 * 8)
    }
}
```

Может возвращать неожиданный результат, если число битов не делится ровно на 8

### Идиомы именования методов преобразования:

#### `as_...()`, `to_...()` и `into()`

Вы могли заметить, что в случае преобразования типов для именования методов используются три распространенные идиомы: добавление перед методами `as_`, `to_` и `into()`. И хотя разработчики не соблюдают эти условности слепо, так поступают многие библиотеки (особенно стандартная библиотека Rust):

- ◆ `as_...()` — для низкозатратных преобразований наподобие `as_ref()` из трейта `AsRef`. Получение ссылки — относительно дешевая операция, которую в некоторых случаях может оптимизировать компилятор;
- ◆ `to_...()` — для более затратных преобразований наподобие `to_string()` из `ToString`. Императив `to` подразумевает, что нужно проделать какую-то работу, например выделить память, создать новые объекты, выполнить преобразование или скопировать данные;
- ◆ `into()` — преобразования с использованием `into()` (посредством трейта `From`). Эти преобразования обычно сопряжены с высокими затратами и часто содержат операции выделения, копирования или клонирования.

Заметное исключение — использование `borrow()` из трейта `Borrow`, который действует аналогично `as_ref()` из `AsRef`, только возвращает не простую ссылку, а объект-ссылку (паттерн, о котором мы поговорим в главе 7), то есть `Ref<', T>` вместо `&T`. Например, `std::cell::RefCell` предоставляет `borrow()`, а не `as_ref()` из-за дополнительных издержек, возникающих в результате заимствования в среде выполнения.

Проверим правильность работы наших преобразований:

```
dbg!(bits.to_bytes());
dbg!(bytes.to_bits());
```

При выполнении этого кода мы получим вывод, в котором создается новый объект:

```
[src/main.rs:24] bits.to_bytes() = ByteCount(
    1,
)
[src/main.rs:25] bytes.to_bits() = BitCount(
    96,
)
```

Если очень захочется, то можно продолжать выполнять преобразование из битов в байты и обратно:

```
dbg!(bits.to_bytes().to_bits());
dbg!(bytes.to_bits().to_bytes());
```

Вот вывод этого кода:

```
[src/main.rs:27:5] bits.to_bytes().to_bits() = BitCount(
    8,
)
[src/main.rs:28:5] bytes.to_bits().to_bytes() = ByteCount(
    12,
)
```

Доступ к внутреннему значению нового типа происходит с помощью простого синтаксиса кортежа, так как, по сути, новые типы — это кортежи:

```
dbg!(bits.0);
dbg!(bytes.0);
```

При выполнении предыдущего кода мы получим следующее:

```
src/main.rs:30:5] bits.0 = 8
[src/main.rs:31:5] bytes.0 = 12
```

Преобразование между единицами измерения — например, битами и байтами, градусами Цельсия и Фаренгейта, метрами и футами — типичный случай использования паттерна «Новый тип», так как он позволяет кодировать логику конвертации в одном месте и обеспечивает корректность этой операции. Обратите внимание: если преобразование содержит операцию без потерь, например вычисления с плавающей запятой, то при каждом его выполнении точность будет снижаться. Поэтому рекомендуется сохранить исходное значение для последующих преобразований.

«Новый тип» удобен, не требует много шаблонного кода и очень легко воспринимается другими разработчиками. По своей сути этот паттерн представляет именованные кортежи с одним или несколькими методами, необходимыми для преобразования между родственными типами.

## Резюме

- Макросы в Rust обеспечивают один из способов метапрограммирования. С их помощью можно генерировать код, избавляя себя от лишнего набора кода и сокращая количество возможных ошибок, которые могут возникнуть при необходимости генерации или создания повторяющегося кода.
- Основные языковые паттерны Rust (дженерики и трейты) можно использовать для создания сложных паттернов, таких как «Строитель» и «Гибкий интерфейс».
- Паттерн «Строитель» показывает, как эффективно использовать инкапсулированные данные и разделять обязанности.
- Паттерн «Гибкий интерфейс» предоставляет удобный способ обработки цепочек операций и преобразования между типами.
- Паттерн «Наблюдатель» является альтернативой обратным вызовам, представляя более чистую абстракцию ценой небольшого объема шаблонного кода. В простых случаях обратных вызовов может быть вполне достаточно.
- Паттерн «Команда» предоставляет способ абстрагировать выполнение команды от ее цели (или получателя), а также порядок и тайминг выполнения.
- Паттерн «Новый тип» подразумевает обертывание других типов кортежем, выполняемое для кодирования дополнительной информации о типе или обеспечения безопасного преобразования данных. Кандидаты на применение этого паттерна — фундаментальные типы, такие как `String`, или примитивы наподобие `i32`. «Новый тип» позволяет легко преобразовывать друг в друга похожие, но при этом разные типы.

# 6

## Проектирование библиотеки

### В этой главе

- ✓ Правила создания хорошей библиотеки.
- ✓ Оценка эргономичности библиотеки Rust.

Мы добрались до середины книги. И теперь слегка отвлечемся на тему, которая зачастую вызывает споры, и поговорим о том, что подразумевается под хорошим дизайном библиотеки. Никто не спорит, что хороший дизайн лучше плохого, но люди часто расходятся в трактовке *хорошего*. Видение того, что считать хорошим, а что плохим, со временем меняется, и при разработке дизайна это тоже важно учитывать.

Грамотная разработка программного обеспечения строится на соблюдении нескольких универсальных правил. Они определяются тенденциями времени, контекстом, доступностью, качеством имеющихся инструментов и тем, как с учетом всего этого работает интерфейс, связывающий человека и компьютерную программу. Этот интерфейс представляет API вашей библиотеки и выступает важнейшей частью ее дизайна.

В текущей главе вы познакомитесь с идеями, процессами и методами, которые нужно учитывать при проектировании библиотеки, чтобы она получилась удобной в работе, подходящей для решения различных задач и при этом защищенной

от возможных ошибок пользователя. В качестве ориентира мы используем пример, который разбирали ранее.

Текущая глава написана для тех, кто заинтересован в публикации собственных библиотек в качестве проектов с открытым исходным кодом, SDK или API для внутреннего использования. Кроме того, она будет полезна тем, кто хочет более глубоко освоить проектирование библиотек и научиться создавать их в качестве легких в использовании и сопровождении, расширяемых инструментов.

Прежде чем переходить к конкретному примеру, мы рассмотрим ряд задач, с которыми сталкиваемся в роли создателей и своего рода хранителей программных библиотек.

## 6.1. Обдумайте дизайн библиотеки

Проектирование библиотеки — да и любого приложения — всегда сопряжено с компромиссами. Их можно представить в виде скользящей шкалы. Каждый выбор, который мы совершаем при разработке, связан с поиском оптимального баланса между различными вариантами. При этом он может быть двоичным, скалярным, а также основываться на трех, четырех и более компромиссах. Где-то на шкале компромиссов есть точка удачного баланса.

Пример двоичного выбора — ситуация, когда для реализации функциональности нужно либо добавить зависимость, либо написать решение самостоятельно. Скалярный выбор подразумевает поиск баланса между как минимум двумя опциями: например, свободой конфигурации и использованием штатных настроек (то есть либо сделать настраиваемым все или какие-то части, либо исключить возможность настройки полностью).

В большинстве практических задач программирования основное ограничение состоит в том, как обеспечить необходимые функции за минимальное время без ущерба для качества. В рамках трех измерений — скорости, завершенности и качества — оптимизация одних функций может потребовать обеднения других (например, придется пожертвовать скоростью в пользу качества или отказаться от каких-то функций, чтобы ускорить вывод продукта на рынок).

Когда дело касается проектирования API библиотеки, можно вдохновляться принципами Мари Кондо, постоянно задавая себе вопрос: «Вызывает ли это радость?» Работа с библиотекой должна быть удобной для будущих пользователей, поэтому необходимо узнать их предпочтения. Зачастую для этого достаточно просто попользоваться библиотекой, сравнить ее интерфейс с похожими решениями и сделать так, чтобы этот интерфейс и предлагаемые ею паттерны соответствовали ожиданиям людей. Нужно исключать элементы интерфейсов, которые не удовлетворяют запросам пользователей.

## **6.2. Делайте что-то одно и делайте это качественно и правильно**

Будучи ответственными участниками экосистемы Rust, мы хотим создавать библиотеки, которые будут следовать принципам этого языка и его этосу. Многие крейты посвящены реализации небольших наборов задач с акцентом на качестве. И мы хотим, чтобы наши крейты могли успешно сочетаться с другими. Нам не нужно привлекать лишние зависимости и в случае их использования надо проследить, чтобы все работало исправно. Иногда мы делаем зависимости или функциональность необязательными за счет использования специальных флагов, но излишнее их количество может вносить путаницу и усложнять использование библиотеки. Попытка одновременно достичь всех этих целей подразумевает непростой поиск баланса, который значительно затрудняется по мере усложнения библиотеки.

Поэтому качественно делать что-то одно — это способ, который позволяет обеспечить не только корректность работы библиотеки, но и удобство ее тестирования, сопровождения и использования.

Правильная работа библиотеки важнее производительности и завершенности. Добившись этого, вы гарантируете, что ваша библиотека будет надежным и прогнозируемым образом выполнять заявленные задачи (то есть соответствовать спецификациям или документации). Сделать библиотеку корректно работающей сложно, если пытаться реализовывать сразу слишком много аспектов или не учитывать принципы языка и контекст.

Обеспечение правильности работы библиотеки — сложная тема, которую не опишешь в одной главе. Для реализации этой задачи можно использовать инструменты наподобие тестирования на основе свойств, фаззинга и формальной верификации, которая, к слову, является самым сложным вариантом. Большинству из нас вряд ли когда-либо понадобится ее выполнять, но полезно знать, что такая возможность есть. Тестирование на основе свойств и фаззинг более доступны, и их можно активно использовать для обеспечения корректности работы библиотеки.

## **6.3. Избегайте излишнего абстрагирования**

В процессе разработки библиотеки нужно решать, что именно важно представить в публичных интерфейсах. В большинстве случаев мы начинаем с предоставления минимального набора типов, методов, трейтов и функций, обеспечивающих необходимый минимум функциональности. Нам не нужно слишком активно использовать абстракции или инкапсуляцию, в частности, в отношении сырых данных. Вместо этого нужно дать будущим пользователям библиотеки возможность обрабатывать данные удобным для них способом.

Мы реализуем распространенные трейты (`Debug`, `Clone` и т. д.), чтобы упростить работу, но нам не обязательно следовать принципу «все пригодится» и добавлять всевозможные трейты просто потому, что мы можем это делать.

Недостаток излишнего абстрагирования заключается в том, что это затрудняет использование библиотеки, препятствуя доступу к ней и ее применению. Особенno это актуально, когда реализуемые абстракции не соответствуют идиоме языка или предметной области, отличаясь от того, что пользователи ожидают найти в библиотеке. Если абстракции будут излишне сложными, то могут сделать вашу библиотеку несовместимой с другими. И это станет проблемой, если вы предполагаете ее использование в различных контекстах.

Есть старая шутка: Микеланджело спросили, как он создал статую Давида, на что скульптор ответил: «Я лишь отсек все лишнее». Тот же принцип касается и проектирования библиотек. Нужно просто отсекать любые абстракции, которые не являются необходимыми, пока не найдется простейшее и наиболее элегантное решение поставленной задачи.

## 6.4. Страйтесь использовать простые типы

Один из способов сделать так, чтобы к библиотеке имели доступ большое количество приложений, — как можно чаще использовать простые типы. Введение их новых видов и пользовательских данных означает, что любой, кто захочет задействовать вашу библиотеку, будет вынужден дополнительно выполнять преобразование между своими структурами данных и вашими.

В идеале вы можете использовать типы из стандартной библиотеки. Если же вам потребуется ввести новые, то нужно будет обеспечить для них удобную конвертацию, предоставив все необходимые механизмы преобразования (например, реализовав `From`). Кроме того, необходимость конвертации может привести к некоторому снижению производительности, что нежелательно для пользователей.

Стандартной библиотеки и коллекций Rust (в том числе `Vec`, `HashMap` и `HashSet`) вполне достаточно для решения большинства задач, и по возможности следует стараться применять их везде. Но вы можете пойти дальше и начать принимать в качестве ввода для функций срезы или итераторы, дополнительно повышая гибкость библиотеки.

Рассмотрим пример с библиотекой, принимающей на входе интерфейс `Vec`. Он не слишком гибкий, так как передавать можно только `Vec`:

```
fn do_something_with_vec<T>(v: &Vec<T>) {  
    // ...  
}
```

Следующий интерфейс уже более гибкий, поскольку здесь можно передавать `Vec`, массив и любой другой тип в срез:

```
fn do_something_with_slice<T>(v: &[T]) {  
    // ...  
}
```

Срез может уступать по гибкости итератору, но он куда более гибкий, чем `Vec`.

## 6.5. Пользуйтесь инструментами

Инструменты наподобие Clippy и rustfmt могут повысить степень соответствия идиомам и условным соглашениям Rust. Например, стиль Rust подразумевает использование верблюжьего регистра для типов, змеиного для переменных и функций-членов класса, верхнего для констант и т. д. Clippy предоставляет средства проверки кода (линты) на соответствие всем этим соглашениям. Это один из самых мощных инструментов, позволяющих обеспечить соответствие кода идиомам Rust.

Clippy и rustfmt преимущественно относятся к идиомам, поэтому будут не слишком полезны при проектировании, создании архитектуры или обеспечении корректности работы библиотеки. Но при этом с их помощью можно избежать типичных сложностей, а также сделать так, чтобы код был относительно читабельным и понятным.

Встраивание этих инструментов в редактор и процесс непрерывной интеграции/развертывания (continuous integration/continuous delivery, CI/CD) позволяет обеспечить соответствие кода стандартам в длительной перспективе. Изменять код после его написания куда затратнее, чем писать его корректно сразу, так что эти инструменты стоят вашего внимания, особенно если учсть, что они бесплатны, удобны в использовании и просты в интеграции.

## 6.6. Хорошие художники копируют; великие — воруют (из стандартной библиотеки)

Когда вы не уверены, какие соглашения соблюдать, используйте популярные крейты Rust в качестве опорных данных, которые можно проанализировать и понять, что сработает, а что — нет. Используя популярные крейты как ориентир, часто можно избежать неудачных решений при проектировании. Однако данное утверждение не означает, что для образца подойдет любой крейт, который стал популярным.

Если вам нужен ориентир, то стандартная библиотека Rust выступает эталоном идиоматического кода. Она прекрасно документирована, протестирована

и грамотно структурирована. Вы можете ознакомиться с исходным кодом и историей обсуждений в репозитории Rust, чтобы понять, почему его разработчики приняли те или иные решения.

Официальная документация стандартной библиотеки ссылается напрямую на исходный код, который помогает изучить внутренние механизмы работы. Rust и его стандартная библиотека имеют двойную лицензию: Apache 2.0 и Массачусетского технологического института (MIT), поэтому в большинстве случаев вы вполне можете использовать примеры из исходного кода в своих проектах в качестве опорных точек.

## **6.7. Документируйте каждый нюанс и приводите примеры**

Документирование библиотеки — один из важнейших этапов ее создания. Не следует рассматривать его как завершающий штрих. Напротив, документацию вместе с примерами кода нужно составлять в течение всего процесса написания библиотеки.

Примерами иногда пренебрегают, хотя они являются важной частью справочных материалов. Как правило, пользователь библиотеки начинает именно с копирования примеров из документации, изменения их под свои нужды. Уверен, что читатели, которые имеют опыт использования библиотеки, тоже так делали и сейчас кивают в знак согласия.

## **6.8. Не ломайте код пользователя**

Следует всегда стремиться сохранять обратную совместимость. При публикации крейтов нужно использовать семантическое версионирование, указывая потребителям на совместимость версий. Если мы хотим опубликовать крейт, то следует иметь в виду, что последующее сопровождение библиотеки — непрерывный процесс, требующий гибкости по части внедрения новых функций и паттернов и параллельного исключения устаревших.

Обратная совместимость — настолько важная черта библиотеки, что ее нужно стараться сохранять изо всех сил. Лучше немного пожертвовать качеством API, чем нарушить работу кода пользователя. Если нарушения совместимости неизбежны, то необходимо предоставить пользователям возможность перейти на новую версию. Кроме того, следует четко описать изменения как в документации, так и в примечаниях к релизу.

При внесении в библиотеку изменений (как нарушающих обратную совместимость, так и нет) помните, что многие пользователи не утружддают себя чтением примечаний, логов изменений или документации. Люди просто обновляют

зависимости и ожидают, что все будет работать. Так что поддержка обратной совместимости избавит от множества проблем как пользователей библиотеки, так и ее создателей.

## 6.9. Помните о состоянии

Один из критически важных этапов проектирования библиотеки — продумывание обработки ее состояния пользователями. Здесь не следует создавать глобальные переменные, а также использовать изменяемые статические значения и синглтоны.

В большинстве случаев благодаря грамотному дизайну библиотеки ее пользователи имеют возможность создания экземпляров контекста, в котором она работает. Пользователи при необходимости передают этот контекст куда требуется, где он выступает в качестве точки доступа к функциональности библиотеки. И это хороший паттерн, поскольку он позволяет создавать несколько экземпляров библиотеки, а также упрощает ее тестирование.

Идеальная библиотека может не иметь состояния, но в реальности часто нужен способ управления им, и в таких случаях следует предоставлять пользователям соответствующий механизм. При этом также необходимо прояснить этот процесс и его последствия. Если состояние должно быть постоянным или его нужно сохранять, то надо предоставить способ, позволяющий сериализовать и десериализовать это состояние.

К примерам состояния, требующего обработки, относятся: конфигурация, пулы подключений, кэши, счетчики и аккумуляторы. У библиотеки наверняка будет точка входа, получающая объект контекста, который будет передаваться внутри нее разным функциям. За создание объекта контекста будет отвечать фабричная функция или паттерн «Строитель», а сам этот объект будет обрабатывать состояние библиотеки. Разберем следующий пример:

```
let ctx = MyLibrary::new()           ← Создает новый экземпляр гипотетической библиотеки
    .with_option(true)
    .with_param(3.14)
    .with_setting(249295)           ← Создает объект контекста
    .build();                      ← с различными параметрами
ctx.do_operation();                ← Использует объект контекста для выполнения операции
let inner_module = MyLibrary::InnerModule::new(ctx);          ← Создает экземпляр
                                                               ← внутреннего модуля,
                                                               ← требующего объект
                                                               ← контекста
inner_module.do_an_inner_operation(); ← Использует внутренний
                                       ← модуль для выполнения
                                       ← операции
```

Этот пример представляет простой паттерн для управления состоянием библиотеки. Здесь пользователь с помощью интерфейса «Строитель» создает объект контекста, а также реализует обработку этого объекта и его передачу различным функциям библиотеки. В свою очередь, библиотеке не нужно ни с кем делиться

информацией объекта контекста, и пользователь при желании может создавать несколько экземпляров этой библиотеки с разной конфигурацией. Если доступ к объекту контекста потребуется другому модулю внутри библиотеки, то пользователь может передать ему данный объект, как происходит с внутренним модулем в предыдущем примере.

## 6.10. Помните об эстетичности

Первое впечатление важно, и эстетичность вашей библиотеки очень влияет на ее восприятие пользователями. Под эстетичностью в данном случае понимается не только внешний вид библиотеки, но и ощущения от пользования ею. Если она удобна в работе, понятна и не вызывает сложностей в отладке, то и восприниматься будет как более эстетичная, чем та библиотека, которая не обладает этими качествами.

На эстетичность библиотеки влияет множество факторов, таких как именование типов, функций и переменных, структура кода, документация, примеры и дизайн в целом. Правильная организованность, хорошая документация и простота использования делают библиотеку более эстетичной.

При проектировании библиотеки нужно учитывать эстетичность кода, документации и примеров. Используйте единообразные соглашения об именовании, организуйте код логично, а также предоставьте понятную, лаконичную, грамматически корректную документацию без ошибок. Эту задачу существенно упрощают инструменты документирования, которые сами генерируют красивую документацию. Пишите примеры, которые будут демонстрировать простые и понятные способы использования библиотеки. Учитывайте удобство работы с ней и стремитесь сделать этот опыт максимально приятным для пользователя.

## 6.11. Оценка эргономичности библиотеки Rust

Пора связать воедино некоторые из пройденных тем книги и создать библиотеку на основе фрагмента кода, который приводился в главе 3. Это станет отличным упражнением. Вы можете многому научиться, просто задокументировав код и протестировав его с позиции конечных пользователей библиотеки. Кроме того, я считаю, что для создания качественного кода полезно вникать в его детали, представляя себя в роли пользователя. В процессе написания библиотек вы научитесь инкапсуляции, разделению обязанностей и поймете, как создавать продуманные интерфейсы.

Вы можете расстроиться, если ожидали найти в этой книге полный список рекомендаций о том, что нужно, а чего нельзя делать при создании библиотек. Я не могу предоставить такой список, зато могу помочь вам обрести навыки, которые понадобятся для создания качественного кода.

### 6.11.1. Вернемся к связанным спискам

В качестве основы библиотеки мы используем пример со связанным списком из главы 3. Начнем с того, что создадим ее командой `cargo new --lib linkedlist`. Затем скопируем код из главы 3 в файл `src/lib.rs`. Далее создадим в библиотеке интеграционный тест. Для этого создадим файл `tests/integration_test.rs` и заполним его кодом из предыдущего теста:

```
#[test]
fn test_linkedlist() {
    use linkedlist::LinkedList;
    let mut linked_list = LinkedList::new("first item"); ←
    // ... опущено для краткости ...
}
```

Ошибка: `LinkedList`  
является закрытым

Мы используем интеграционный тест, а не модульный, так как хотим протестировать библиотеку извне крейта. В своем текущем состоянии код теста (который мы скопировали напрямую из старого кода) компилироваться не будет, поскольку мы не учли видимость. Компилятор выдаст следующую ошибку:

```
error[E0603]: struct `LinkedList` is private
--> tests/integration_test.rs:3:21
   |
3 |     use linkedlist::LinkedList;
   |     ^^^^^^^^^^ private struct
   |
note: the struct `LinkedList` is defined here
--> /Users/brenden/dev/idiomatic-rust-book/c06/
linkedlist/src/lib.rs:22:1
|
22 | struct LinkedList<T> {
   | ^^^^^^^^^^^^^^^^^^
```

Это сообщение об ошибке вполне понятно. Исправим проблему с видимостью, добавив `pub` в каждый метод блока `impl<T> LinkedList<T> { ... }` и саму структуру `LinkedList`. Имейте в виду, что отдельные поля структуры по-прежнему остаются закрытыми, поскольку в Rust по умолчанию все является таковым. Теперь при очередной попытке компиляции мы получим еще больше ошибок. Вот первая:

```
error[E0446]: private type `Iter<'_, T>` in public interface
--> src/lib.rs:41:5
   |
41 |     pub fn iter(&self) -> Iter<T> {
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^ can't leak private type
...
62 | struct Iter<'a, T> {
   | ----- `Iter<'_, T>` declared as private
```

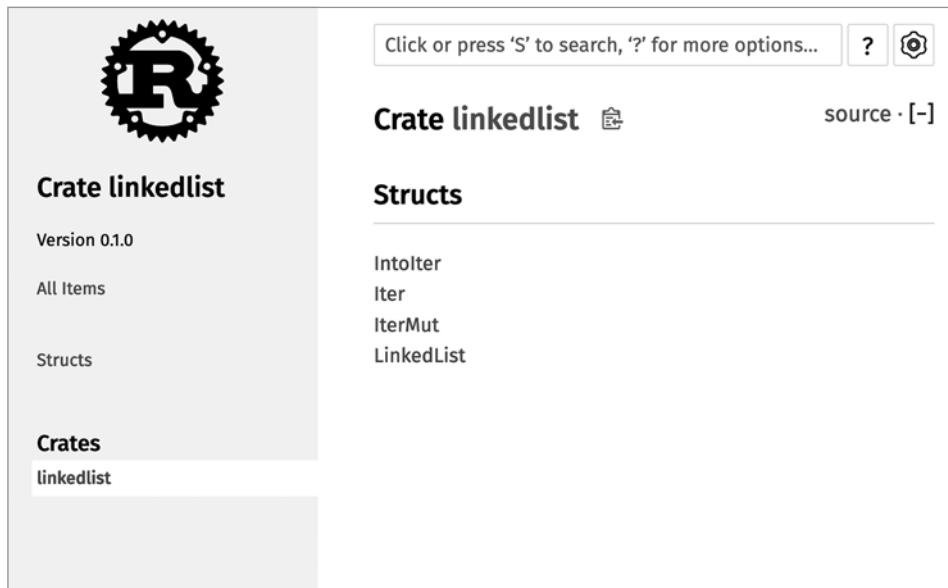
Ах да, мы забыли сделать открытыми итераторы. Исправим это, добавив `pub` в три структуры итераторов: `Iter`, `IterMut` и `IntoIter`.

После внесения этих изменений код благополучно скомпилируется и тест успешно выполнится. Нам пришлось исправить проблемы с видимостью его элементов, чтобы превратить код в полноценную библиотеку.

### 6.11.2. Улучшение дизайна API с помощью `rustdoc`

Далее мы проанализируем API нашей библиотеки. Лучшим способом сделать это будет генерация документации с помощью `rustdoc`. Любой, кто будет использовать нашу библиотеку, наверняка потратит много времени на изучение документации, поэтому очень важно сделать ее качественной.

Документацию мы сгенерируем командой `cargo doc`, которая поместит итоговые HTML-файлы в каталог `target/doc` крейта. Откроем главный файл `linkedlist/index.html`. Никакую документацию мы еще не писали, поэтому перед нами откроется пустая страница, на которой будут перечислены структуры, отмеченные ключевым словом `pub` (рис. 6.1).



**Рис. 6.1.** Пустая документация для нашего крейта `linkedlist`

Если мы щелкнем на ссылке структуры `LinkedList`, то увидим страницу, показанную на рис. 6.2.

Click or press 'S' to search, '?' for more options... ? ⚙

## Struct linkedlist::LinkedList

source · [-]

```
pub struct LinkedList<T> { /* private fields */ }
```

### Implementations

- `[+] impl<T> LinkedList<T>` source
- `pub fn new() -> Self` source
- `pub fn append(&mut self, t: T)` source
- `pub fn iter(&self) -> Iter<'_, T> ⓘ` source
- `pub fn iter_mut(&mut self) -> IterMut<'_, T> ⓘ` source
- `pub fn into_iter(self) -> IntoIter<T> ⓘ` source

### Trait Implementations

- `[+] impl<T: Clone> Clone for LinkedList<T>` source
- `[+] fn clone(&self) -> Self` source
 

Returns a copy of the value. Read more
- `[+] fn clone_from(&mut self, source: &Self)` source
 

Performs copy-assignment from `source`. Read more
- `[+] impl<T: Debug> Debug for LinkedList<T>` source
- `[+] fn fmt(&self, fmt: &mut Formatter<'_>) -> Result` source

Рис. 6.2. Пустая документация структуры LinkedList

Обратите внимание: мы еще никак не документировали код, но у нас уже есть весьма полезное его описание. Просто благодаря тому, что мы перечислили структуры и методы (при условии их правильного именования), пользователь сможет многое понять о том, что делает наша библиотека и как она работает. Это окажется особенно актуальным, если мы выбрали удачные имена для объектов, методов и трейтов. Но каким бы самоочевидным, по нашему мнению,

ни был функционал библиотеки, нам все равно нужно написать дополнительную документацию.

Первым делом нужно описать сам крейт, чтобы любой, кто обратится к документации, знал, с чего начать. Задокументируем крейт, добавив в файл `lib.rs` внешнюю документацию. В Rust внешняя документация предоставляется в виде комментариев, начинающихся с `//!`, а внутренняя — в комментариях, начинающихся с `///`. Внешняя относится к внешней области документируемого файла, а внутренняя — к элементу, следующему за комментарием.

Сначала мы добавим общее описание крейта и общий пример его использования. Дополним код таким фрагментом, разместив его в начале файла `src/lib.rs`:

```
///! # linkedlist crate
//!
//! Этот крейт предоставляет простую реализацию связанного списка.
//!
//! Он служит в качестве учебного примера для книги [_Rust Advanced
//! Techniques_](https://www.manning.com/books/idiomatic-rust).
//!
//! ## Пример использования
//!
//! ````rust
//! use linkedlist::LinkedList;
//!
//! let mut animals = LinkedList::new();
//! animals.append("chicken");
//! animals.append("ostrich");
//! animals.append("antelope");
//! animals.append("axolotl");
//! animals.append("okapi");
//! ````
```

На рис. 6.3 показано, как будет выглядеть сгенерированная документация на уровне крейта.

Неплохо! Документация начинает напоминать реальный крейт.

**СОВЕТ** Работая с документацией, используйте команду `cargo watch -x doc`, которая позволяет автоматически генерировать документацию в случае внесения изменений. Установить `cargo-watch` можно командой `cargo install cargo-watch`.

Теперь, когда у нас есть документация и рабочий пример, можно протестировать ее. Любой фрагмент кода в документации является еще и интеграционным тестом. Если мы выполним `cargo test`, то увидим, что наш пример стал тестом (обозначен как `Doc-tests linkedlist`):

```
$ cargo test
    Finished test [unoptimized + debuginfo] target(s) in 0.00s
      Running unit tests src/lib.rs
        (target/debug/deps/linkedlist-2e0286b0918288ae)

running 0 tests
```

```

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
    Running tests/integration_test.rs
      (target/debug/deps/integration_test-c95f81c9911957c8)

running 1 test
test test_linkedlist ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

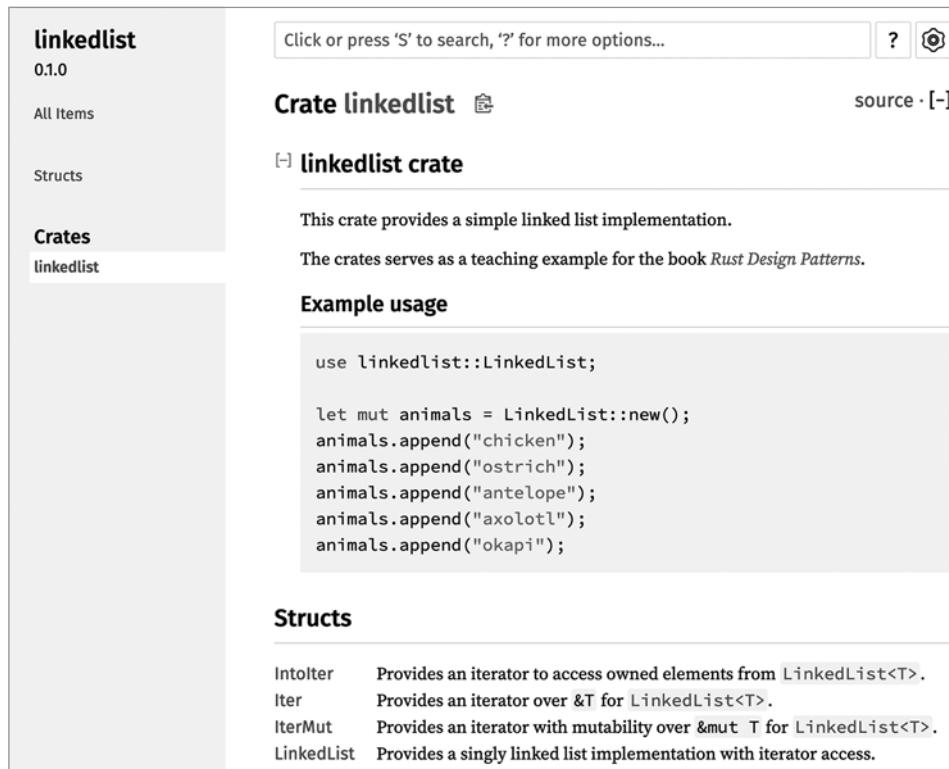
Doc-tests linkedlist

running 1 test
test src/lib.rs - (line 10) ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.26s

```

Обратите внимание, что нам не нужно писать в примерах функцию `main()`. Базовую предварительную обработку выполняет `rustdoc`, который обертывает код в `fn main() { ... }` и на ходу создает код для выполнения через `cargo test`.



**Рис. 6.3.** Крейт `linkedlist` с общей документацией

Теперь поговорим о нашем API. Когда мы писали этот код, мы не задумывались о том, как его могут использовать другие люди. Первое, что вызывает вопросы, — метод `new()` в `Linked-List`, который кажется здесь неуместным. Почему `new()` получает любые параметры? Думаю, нам следует эмулировать поведение других коллекций в Rust, например `Vec`. В документации `Vec::new()` говорится:

*Создает новый пустой Vec<T>.*

*Этот вектор не будет выделять память, пока ему не передадут элементы.*

Чтобы соблюсти согласованность, нам следует использовать тот же подход, что и для `Vec`. Обновим код, изменив `new()` так, чтобы он возвращал пустой `LinkedList`. В добавок попутно задокументируем `LinkedList` (листинг 6.1).

### Листинг 6.1. LinkedList с документацией

```
/// Предоставляет реализацию односвязного списка
/// с доступом через итератор
pub struct LinkedList<T> {
    head: Option<ListItemPtr<T>>,
}

impl<T> LinkedList<T> {
    /// Создает новый пустой [`LinkedList<T>`].
    pub fn new() -> Self {
        Self { head: None }
    }
    /// Добавляет элемент в конец списка. Если список пуст,
    /// элемент становится в нем первым.
    pub fn append(&mut self, t: T) {
        match &self.head {
            Some(head) => {
                let mut next = head.clone();
                while next.as_ref().borrow().next.is_some() {
                    let n = next.as_ref().borrow()
                        .next.as_ref().unwrap().clone();
                    next = n;
                }
                next.as_ref().borrow_mut().next =
                    Some(Rc::new(RefCell::new(ListItem::new(t))));
            }
            None => {
                self.head = Some(Rc::new(RefCell::new(ListItem::new(t))));
            }
        }
    }
    /// Возвращает итератор списка
    pub fn iter(&self) -> Iter<T> {
        Iter {
            next: self.head.clone(),
            data: None,
        }
    }
}
```

```

        phantom: PhantomData,
    }
}
/// Возвращает итератор списка с возможностью изменения
pub fn iter_mut(&mut self) -> IterMut<T> {
    IterMut {
        next: self.head.clone(),
        data: None,
        phantom: PhantomData,
    }
}
/// Потребляет этот список и возвращает итератор по его значениям
pub fn into_iter(self) -> IntoIter<T> {
    IntoIter {
        next: self.head.clone(),
    }
}
}
}

```

Обратите внимание, что мы также обновили структуру `LinkedList`, сделав `head` необязательным. Это изменение позволяет иметь пустой экземпляр, поскольку прежняя версия предполагала, что у нас всегда есть элемент `head`. На рис. 6.4 показана обновленная документация `LinkedList`.

Получилась неплохая документация. Можно значительно расширить функциональность нашего типа коллекции, но пока сосредоточимся на самом важном. На ум приходят два дополнения: вывод содержимого списка и клонирование списка. Обе эти возможности не так просты, как может показаться. Можно использовать `#[derive(Clone, Debug)]`, который вполне справится с этими задачами, но такое решение будет неидеальным. Далее мы разберем эти два процесса по отдельности.

Если мы хотим реализовать для нашего связанного списка функцию `Clone`, то нужно подумать о том, что означает клонирование связанного списка. Скорее всего, вызывая `clone()` для списка, человек будет ожидать клонирования его структуры и содержимого, а не только структуры. Иными словами, мы не хотим, чтобы в новую структуру копировались только указатели, так как они продолжат указывать на те же самые данные.

Исправить `Clone` можно двумя способами: переписать `LinkedList`, чтобы он не использовал `Rc<RefCell<T>`, или вместо использования `#[derive(Clone)]` предоставить собственную реализацию. Но мы хотим продолжить использовать `Rc<RefCell<T>`, так как это облегчит нам работу, если позже мы решим добавить в список дополнительную функциональность. Значит, реализуем `Clone` сами. Вот определение трейта `Clone`:

```

pub trait Clone {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) { ... }
}

```

The screenshot shows the Rustdoc interface for the `Struct linkedlist::LinkedList`. The left sidebar contains a logo and sections for `LinkedList`, `Methods` (with items `append`, `into_iter`, `iter`, `iter_mut`, `new`), `Trait Implementations` (with items `Clone`, `Debug`, `Intoliterator`, `Intoliterator`, `Intoliterator`), and `Auto Trait Implementations` (with items `!RefUnwindSafe`, `!Send`, `!Sync`, `Unpin`, `!UnwindSafe`). The main content area has a search bar at the top with placeholder text "Click or press 'S' to search, '?' for more options...". Below it, the title `Struct linkedlist::LinkedList` is shown with a copy icon and a source link. A code block displays the `pub struct LinkedList<T>` definition. A note below it says "[–] Provides a singly linked list implementation with iterator access." Under the heading `Implementations`, there are two entries: `impl<T> LinkedList<T>` (source) and `pub fn new() -> Self` (source). The `new()` method is described as "Constructs a new, empty `LinkedList<T>`". Below this, the `append` method is listed with the description "Appends an element to the end of the list. If the list is empty, the element becomes the first element of the list." The `iter` method is described as "Returns an iterator over the list." The `iter_mut` method is described as "Returns an iterator over the list that allows mutation." The `into_iter` method is described as "Consumes this list returning an iterator over its values."

**Рис. 6.4.** Задокументированный `LinkedList`

Весьма лаконично. Если мы взглянем на документацию `Clone` более внимательно, то найдем в ней следующее утверждение для метода `clone_from()`:

*а.clone\_from(&b) равнозначен а = b.clone() по функциональности, но его можно переопределить для повторного использования ресурсов а во избежание лишних операций выделения.*

Важно иметь в виду это утверждение, так как наверняка будет легче реализовать `clone_from()`, чем `clone()`. Можно вызвать `clone_from()` из нашей реализации `clone()`:

```
Обратите внимание на ограничение трейта в T. Мы предоставляем
Clone только для типов, которые реализуют Clone
impl<T: Clone> Clone for LinkedList<T> {
    fn clone(&self) -> Self {
        Создает новый список
        let mut cloned = Self::new();
        cloned.clone_from(self);
        cloned
    }
    fn clone_from(&mut self, source: &Self) {
        self.head = None;
        source.iter().for_each(|item| {
            self.append(item.clone());
        });
    }
}
```

Мы клонируем элементы из старого списка, `self`, в новый список через вызов `clone_from()`, который определен ниже

Заключительное выражение возвращает новый список

Установка начала (`head`) списка на `None`, по сути, сбрасывает его

Мы используем итератор для клонирования каждого значения списка и добавления этих значений в целевой список, то есть `self`

Этот код упрощает задачу и в качестве дополнительного бонуса позволяет соблюсти принцип «Не повторяйся», поэтому любые изменения в `clone_from()` отражаются через `clone()`.

### 6.11.3. Улучшение связанного списка с помощью дополнительных тестов

Мы добавили несколько новых функций, значит, пора протестировать код. Обновим интеграционные тесты, чтобы проверить каждую функцию отдельно. Начнем с листинга 6.2, в котором тестируется метод `iter()` в `LinkedList`.

#### Листинг 6.2. Тестирование `iter()` в `LinkedList`

```
#[test]
fn test_linkedlist_iter() {
    use linkedlist::LinkedList;
    let test_data =
        vec!["first", "second", "third", "fourth", "fifth and last"];

    let mut linked_list = LinkedList::new();
    test_data
        .iter()
        .for_each(|s| linked_list.append(s.to_string()));

    assert_eq!(
        test_data,
        linked_list
            .iter()
            .map(|s| s.as_str())
            .collect::<Vec<&str>>());
}
```

Добавляем в список тестов `String`, несмотря на то что у нас есть `Vec<&str>`

Используем `assert_eq!`, чтобы сравниваемые типы обязательно совпадали. Вместо того чтобы преобразовывать `Vec<&str>` в `Vec<String>`, мы с помощью `collect()` достаем временный `Vec<&str>` из нашего связанного списка

В следующем коде тестируется метод изменяемого итератора `iter_mut()` из `LinkedList` (листинг 6.3).

#### Листинг 6.3. Тестирование `iter_mut()` из `LinkedList`

```
#[test]
fn test_linkedlist_iter_mut() {
    use linkedlist::LinkedList;
    let test_data =
        vec!["first", "second", "third", "fourth", "fifth and last"];

    let mut linked_list = LinkedList::new();
    test_data
        .iter()
        .for_each(|s| linked_list.append(s.to_string()));

    assert_eq!(
        test_data,
        linked_list
            .iter_mut()
            .map(|s| s.as_str())
            .collect::<Vec<&str>>()
    );
}
```

Следующий код тестирует метод `into_iter()` из `LinkedList` (листинг 6.4).

#### Листинг 6.4. Тестирование `into_iter()` из `LinkedList`

```
#[test]
fn test_linkedlist_into_iter() {
    use linkedlist::LinkedList;
    let test_data =
        vec!["first", "second", "third", "fourth", "fifth and last"];

    let mut linked_list = LinkedList::new();
    test_data
        .iter()
        .for_each(|s| linked_list.append(s.to_string()));

    assert_eq!(
        test_data
            .iter()
            .map(|s| s.to_string())
            .collect::<Vec<String>>(),
        linked_list.into_iter().collect::<Vec<String>>() ←
    );
}
```

Для проверки `into_iter()` преобразуем тестовые данные в `Vec<String>`, а не наоборот

Следующий код тестирует нашу реализацию трейта `Clone` (листинг 6.5).

#### Листинг 6.5. Тестирование `Clone` для `LinkedList`

```
#[test]
fn test_linkedList_cloned() {
    use linkedlist::LinkedList;
    let test_data =
        vec!["first", "second", "third", "fourth", "fifth and last"];

    let mut linked_list = LinkedList::new();
    test_data
        .iter()
        .for_each(|s| linked_list.append(s.to_string()));

    let cloned_list = linked_list.clone();

    linked_list`  

    .into_iter() ← | Чтобы протестируировать операцию клонирования, мы используем into_iter(),  

    .zip(cloned_list.into_iter()) ← так как она возвращает внутреннее занятое значение, что мы и хотим проверить

    .for_each(|(left, right)| { ← | Проверяет значения изначального
        assert_eq!(left, right); ← и клонированного списков на соответствие
        assert!(!std::ptr::eq(&left, &right)); ←
    });
}  

} ← | Проверяет, относятся ли изначальные и клонированные значения к разным областям  

     памяти. Эта проверка в некотором смысле излишня, так как две переменные не могут  

     указывать на одни и те же занятые объекты в области. Но мы ее все равно выполним
```

Все тесты проходят успешно, значит, можно двигаться дальше.

#### 6.11.4. Упрощение отладки библиотеки

Теперь поговорим о трейте `Debug`. Просто интереса ради посмотрим, что произойдет, если использовать `#[derive(Debug)]` и вывести список наших тестовых данных через `dbg!(linked_list)`. Вывод будет выглядеть так:

```
[tests/integration_test.rs:20] linked_list = LinkedList {
    head: Some(
        RefCell {
            value: ListItem {
                data: RefCell {
                    value: "first",
                },
                next: Some(
                    RefCell {
                        value: ListItem {
                            data: RefCell {
                                value: "second",
                            },
                        },
                    ),
                ),
            },
        },
    ),
}
```

Увы, этот результат нам ничем не поможет. Если кто-то попробует использовать наш связанный список, то такой вывод его только запутает, особенно если в нем есть глубоко вложенные структуры. Этот код нельзя использовать в его текущем виде. Предлагаю взглянуть на трейт `Debug`, чтобы понять, как его реализовать:

```
pub trait Debug {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}
```

В Debug нас интересует `Formatter`. Это инструмент Rust, который отвечает за упорядочение вывода, беря на себя основную часть его форматирования. Он позволяет легко форматировать отладочную информацию списков с помощью `debug_list()`.

**ПРИМЕЧАНИЕ** Полное описание инструмента Formatter вы найдете по адресу <https://mng.bz/67VG>.

Теперь реализуем трейт `Debu`, используя `Formatter::debug_list()`:

```
impl<T: Debug> Debug for LinkedList<T> {
    fn fmt(&self, fmt: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        fmt.debug_list().entries(self.iter()).finish()
    }
}
```

При использовании обновленной реализации `Debug` вывод тестов командой `dbg!()` получится намного качественнее:

```
[tests/integration_test.rs:20] linked_list = [
    "first",
    "second",
    "third",
    "fourth",
    "fifth and last",
]
```

В завершение взглянем на документацию для созданных нами итераторов. Мы не писали никакой документации для `Iter`, `IterMut` и `IntoIter`. Но если мы посмотрим на сгенерированное описание, то увидим, что `Iterator` предоставил

нам много функций, которые уже задокументированы. Для полноты картины дадим краткое описание к каждому итератору (рис. 6.5).

The screenshot shows the Rust documentation for the `linkedlist::Iter` struct. At the top, there's a search bar with placeholder text "Click or press 'S' to search, '?' for more options..." and a help icon. Below the search bar, the title `Struct linkedlist::Iter` is displayed, along with a "source" link and a copy icon.

**Iter**

**Trait Implementations**

- Iterator

**Auto Trait Implementations**

- !RefUnwindSafe
- !Send
- !Sync
- Unpin
- !UnwindSafe

**Blanket Implementations**

- Any
- Borrow<T>
- BorrowMut<T>
- From<T>
- Into<U>
- Intoliterator
- TryFrom<U>
- TryInto<U>

**Struct linkedlist::Iter**

source · [-]

```
pub struct Iter<'a, T> { /* private fields */ }
```

[–] Provides an iterator over &T for `LinkedList<T>`.

### Trait Implementations

[–] `impl<'a, T> Iterator for Iter<'a, T>` source

[–] `type Item = &'a T`

The type of the elements being iterated over.

[–] `fn next(&mut self) -> Option<Self::Item>` source

Advances the iterator and returns the next value. Read more

[–] `fn next_chunk<const N: usize>(&mut self) -> Result<[Self::Item; N], IntoIter<Self::Item, N>>` source

This is a nightly-only experimental API. (`iter_next_chunk`)

Advances the iterator and returns an array containing the next N values. Read more

[–] `fn size_hint(&self) -> (usize, Option<usize>)` 1.0.0 · source

Returns the bounds on the remaining length of the iterator. Read more

[–] `fn count(self) -> usize` 1.0.0 · source

**Рис. 6.5.** Документация для итераторов

Вот теперь у нас получился прекрасный крейт! Наша библиотека соответствует идиомам Rust, о чём говорят качественная документация, предоставленные реализации ключевых трейтов и отображенный API `vec`. Те, кто уже знаком с `Vec`, смогут без проблем использовать нашу коллекцию, так как мы следовали существующим в языке паттернам.

## Резюме

- Разработать грамотный дизайн библиотеки сложно. Для этого нужно тщательно обдумывать все нюансы и при необходимости вносить доработки.
- Библиотеки нужно разрабатывать с учетом интересов пользователей. Мы должны стремиться сделать их простыми в использовании и понятными.
- Фокусируйтесь на чем-то одном и реализуйте это качественно. Библиотеку не нужно перегружать множеством обязанностей, она должна решать одну конкретную задачу.
- Крайне важно сосредоточиться на корректности работы библиотеки и использовать для ее обеспечения такие инструменты, как тестирование свойств и фаззинг.
- Нужно избегать излишних абстракций и при любой возможности использовать базовые типы.
- Инструменты наподобие Clippy и rustfmt помогают обеспечить идиоматичность кода и сделать его понятным.
- В качестве полезных ориентиров для проектирования библиотеки рекомендуется анализировать популярные крейты.
- Сопровождение библиотеки — непрерывный процесс, особенно если вы планируете ее опубликовать. В ходе этого процесса может потребоваться исправлять баги, добавлять функциональность, а также обновлять крейт при появлении в Rust новых возможностей. При этом очень важно сохранять обратную совместимость, а также использовать семантическое версионирование.
- При проектировании библиотеки нужно обращать особое внимание на особенности работы с ее API, оценивая этот процесс с точки зрения пользователя. Предоставляя хорошую документацию с примерами и сравнивая наши API с популярными, можно создавать отличные библиотеки, прикладывая минимум усилий.

## Часть III

# *Более сложные паттерны*

Надеюсь, что к этому моменту вы уже успели поэкспериментировать с паттернами из предыдущих глав и усвоить их принципы. В этой части вы познакомитесь с паттернами посложнее, и для полноценного их освоения может потребоваться больше практики и времени. Если я все сделал правильно, то усвоение предыдущего материала далось вам легко. Если же это не так, то не волнуйтесь: терпение и труд все перерут.

Кроме того, вы могли заметить, что некоторые из описанных паттернов более специализированы и не всегда широко применимы. Но знать и понимать их очень полезно, так как вы наверняка встретите их в других программах. Чем больше времени вы проводите, практикуясь в программировании на Rust, тем более полезными будут эти паттерны для вашей работы.

# *Использование трейтов, дженериков и структур для особых задач*

---

## **В этой главе**

- ✓ Постоянные дженерики.
- ✓ Применение трейтов к внешним типам.
- ✓ Расширение типов специальными трейтами.
- ✓ Реализация `blanket`-трайтов.
- ✓ Использование трейтов-маркеров для разметки типов атрибутами.
- ✓ Создание тегов с помощью структур.
- ✓ Предоставление доступа к внутренним данным с помощью объектов-ссылок.

В предыдущих главах вы познакомились с некоторыми сложными приемами работы в Rust. Теперь мы более подробно обсудим некоторые из этих тем, а также рассмотрим ряд сложных паттернов проектирования. Они будут полезны вам во многих случаях, но использовать эти паттерны вы будете наверняка реже, так как реализовывать их непросто и сценарии, для которых они подходят, встречаются нечасто.

Если использовать аналогию, то паттерны, о которых мы говорили выше, можно сравнить с привычными бытовыми инструментами: молотком, плоскогубцами,

отверткой, электродрелью и т. д., которые помогают решать широкий спектр задач. Что же касается паттернов, которые мы изучим в этой главе, то они предназначены для более специализированных задач. Можно сравнить их со столярными инструментами: циркулярным и токарным станком, рубанком, ленточной пилой и др. Эти паттерны не так полезны в рядовом программировании на Rust, но их стоит изучить, чтобы иметь возможность использовать при необходимости.

## 7.1. Постоянные дженерики

*Постоянные дженерики* в Rust — удобная модификация дженериков, позволяющая использовать постоянные значения обобщенно. Они решают давнюю проблему многих языков программирования, которая возникает, когда вы хотите добавить поле в структуру, зависящую от постоянного значения (например, от длины массива). Это постоянное значение известно только в момент создания экземпляра, поэтому при отсутствии постоянных дженериков единственный способ использовать такое значение — создать версию структуры для каждого желаемого размера массива, что и делают многие библиотеки.

Постоянные дженерики можно использовать везде, где есть примитивная константа и обобщенный параметр, например для определения размера массива. Для этого подойдет целочисленный примитив любого размера, например `i32`, `u32` или `usize`. Кроме того, можно использовать типы `char` и `bool` (которые при компиляции на большинстве платформ равнозначны `u8`). Но значения с плавающей запятой при этом не допускаются.

Чтобы понять, что такое постоянные дженерики, нужно разобрать, какую проблему они решают. Предположим, у нас есть обобщенная структура с массивом байтов, которую мы назовем буфером:

```
struct Buffer {
    buf: [u8; 256],
}
```

Буфер содержит 256 байт. А что, если мы хотим сделать его обобщенным, чтобы он мог хранить любой тип, а не только байты? Сделаем это:

```
struct Buffer<T> {
    buf: [T; 256],
}
```

Вуаля! Ничего сложного. Теперь буфер может хранить 256 любых элементов. Но подождите... А если мы захотим сделать длину массива произвольной? Иными словами, нам нужно сделать его длину переменной в момент создания экземпляра. Один из способов — использовать `Vec`, размер которого можно менять в среде выполнения. Но может возникнуть одна проблема. Для использования `Vec` требуется выделять память в куче (в то время как мы можем аллокировать

простой массив на стеке), к тому же возможны лишние издержки, такие как копирование значений вместо их перемещения.

Если нам известно, что длина массива в течение жизни буфера меняться не будет (как это обычно бывает), то можно использовать обобщенный *постоянный* (`const`) параметр. Введем параметр `LENGTH`, используя постоянные дженерики:

```
#[derive(Debug)]
struct Buffer<T, const LENGTH: usize> {
    buf: [T; LENGTH],
}
```

Теперь в структуре есть два обобщенных параметра: тип элементов массива и его длина. Параметр `LENGTH` можно рассматривать как любой другой обобщенный параметр, за исключением того, что он представляет постоянное значение, а не тип, в связи с чем возможны некоторые интересные побочные эффекты. Например, при инстанцировании он создает новый отдельный тип, что может пригодиться, когда мы хотим использовать систему типов Rust в отношении массивов произвольной длины. Мы можем предоставить реализации трейтов для конкретных постоянных значений, чтобы избежать целого класса ошибок, таких как несоответствие переданной длины массива ожидаемой. Можно специализироваться на конкретных формах этой структуры, например используя трейт `From` для массива `[u8; 256]`:

```
impl From<[u8; 256]> for Buffer<u8, 256> {
    fn from(buf: [u8; 256]) -> Self {
        Buffer { buf }
    }
}
```

Эта реализация позволяет создать `Buffer` из массива типа `[u8; 256]` (и никакого другого) путем перемещения массива в структуру. Но это не слишком practicalный подход. Лучше реализовать обобщенный трейт `From` и использовать специализированные формы при необходимости:

```
impl<T, const LENGTH: usize> From<[T; LENGTH]> for Buffer<T, LENGTH> {
    fn from(buf: [T; LENGTH]) -> Self {
        Buffer { buf }
    }
}
```

Данный код позволяет перемещать массив произвольного типа и длины в `Buffer`. Это весьма полезный подход, особенно если код создан для работы с `Buffer`, а не с сырьими массивами. Теперь протестируем полученный буфер:

```
let buf = Buffer::from([0, 1, 2, 3]); ←
```

Обратите внимание, что здесь не нужно указывать параметр длины со значением 4. Компилятор выведет эту величину сам

При выполнении этого кода мы получим такой вывод:

```
[src/main.rs:14] &buf = Buffer {
    buf: [
        0,
        1,
        2,
        3,
    ],
}
```

Постоянные дженерики упрощают создание пользовательских типов на основе массивов с фиксированной длиной, что может значительно сократить объем шаблонного кода.

## **7.2. Реализация трейтов для типов из других крейтов**

Когда вы начнете работать с трейтами, они могут вам настолько понравиться, что вы станете писать их для всего подряд. Это, безусловно, станет интересным опытом, но так будет лишь до тех пор, пока вы не столкнетесь с известным ограничением их дизайна: нельзя реализовать трейт для типов вне вашего крейта.

У этого ограничения есть причина: если бы трейты можно было реализовывать для любого типа, то вы бы быстро столкнулись со множеством конфликтующих реализаций для одного и того же типа. Ситуация может дополнительно усложниться ввиду того, что разные крейты развиваются с разной скоростью, и их реализаций становятся все менее совместимыми. Компилятор может применить эвристику для выбора подходящей реализации, но такой подход всегда будет вызывать путаницу и сложности в понимании. В связи с этим разработчики Rust решили исключить эту возможность в принципе.

Однако не спешите огорчаться. В Rust есть ряд возможностей, которые позволяют реализовывать аналогичное поведение, не вызывая конфликтов.

### **7.2.1. Структуры-обертки**

Чтобы получить возможность задействовать трейты для внешних типов и продолжить использовать функциональность этих типов, нужно совместить два паттерна: структуры-обертки и трейт `Deref`. *Структура-обертка* — это структура, которая обертывает другой тип. В своей простейшей форме она содержит только одно поле оборачиваемого типа. После ее создания можно будет реализовать для этой структуры любой трейт.

Используя оберточные структуры вместе с трейтом `Deref`, можно реализовывать трейты для типов из внешних крейтов, обходя ограничение языка и делая

так, чтобы объект вел себя как его субъект. Увидеть это можно на примере обертывания `Vec`:

```
struct WrappedVec<T>(Vec<T>); ← По форме структура равнозначна кортежу
```

**ПРИМЕЧАНИЕ** Структура-кортеж, например `WrappedVec`, по сути, равнозначна обычному кортежу, за исключением того, что здесь мы определяем новый тип с именем и можем писать блоки `impl`, как в любой другой структуре.

Совсем не сложно. Но если мы попытаемся использовать `WrappedVec` как `Vec`, то у нас ничего не получится:

```
let wrapped_vec = WrappedVec(vec![1, 2, 3]);
wrapped_vec.iter().for_each(|v| println!("{}", v)); ← При вызове iter() возникает ошибка: «метод не найден во WrappedVec<{integer}>»
```

Этот код не работает по очевидной причине, ведь мы не реализовали `iter()`. Но мы не хотим заново реализовывать все методы, которые предоставляет `Vec`, поэтому нужно получить к ним доступ из нашей обертки.

### 7.2.2. Использование трейта `Deref` для развертывания обернутой структуры

Чтобы добиться от структур-оберток корректной работы, нужно реализовать для `WrappedVec` трейт `Deref`. Когда мы это сделаем, компилятор будет автоматически разыменовывать обертку при обнаружении вызова несуществующих методов. Этот прием называется *приведение разыменовыванием* (`Deref coercion`), но слишком часто использовать его не стоит. Реализуется же `Deref` весьма легко:

```
impl<T> Deref for WrappedVec<T> {
    type Target = Vec<T>; ← Мы хотим автоматически разыменовывать обертку в целевой тип
    fn deref(&self) -> &Self::Target {
        &self.0 ← Элемент .0 в self указывает на первый элемент в структуре-кортеже.
    }
} ← Ни один элемент кортежа имени не имеет
```

Теперь можно вызвать все методы из `Vec`, например `iter()`. Но здесь есть несколько ограничений. Одно из них — невозможность использовать методы, которые получают `self` по значению, например `into_iter()`. Для этого потребуется реализовать метод `into_iter()`:

```
impl<T> WrappedVec<T> {
    fn into_iter(self) -> IntoIter<T> {
        self.0.into_iter()
    }
}
```

Чтобы вызывать методы `Vec`, получающие `&mut self`, нужно реализовать трейт `DerefMut`, который почти идентичен `Deref`. Теперь напишем короткий тест для нашего обернутого вектора:

```
let wrapped_vec = WrappedVec(vec![1, 2, 3]);
wrapped_vec.iter().for_each(|v| println!("{}", v));
wrapped_vec.into_iter().for_each(|v| println!("{}", v));
```

В нашем WrappedVec нет методов-итераторов, но их можно вызвать из Vec, как в случае с обычным вектором

При выполнении этого кода мы получим такой вывод:

```
1
2
3
1
2
3
```

### 7.3. Трейты-расширения

Трейты-расширения — это трейты, которые добавляют функциональность типам и трейтам, находящимся вне крейта, в котором они определены. Пример их использования — расширение функциональности типов из стандартной библиотеки, например добавление метода в основной тип Vec. Трейты-расширения обычно подчиняются соглашению об именовании, в соответствии с которым нужно использовать постфикс Ext. Эти трейты можно встретить в крейтах, которые предоставляют функции для вышестоящих крейтов или стандартной библиотеки.

Чтобы вы могли их увидеть, мы расширим Vec, добавив в него новый трейт, ReverseExt, в котором создадим метод reversed(), возвращающий реверсированную копию вектора. Вот определение нашего трейта:

```
pub trait ReverseExt<T> {
    fn reversed(&self) -> Vec<T>;
```

Метод reversed()  
возвращает Vec<T>

Чтобы не усложнять, в этом примере мы возвращаем Vec<T>. Для улучшения этого интерфейса можете добавить в возвращаемый тип контейнера второй обобщенный параметр по аналогии с тем, как реализованы методы collect() и collect\_into() в std::iter::Iterator.

На практике можно написать библиотеку, которая будет экспорттировать этот трейт с одной или несколькими реализациями, доступными для импорта и применения в других местах. Для использования трейтов-расширений не обязательно писать библиотеку. Кроме того, их можно использовать внутри крейта или приложения, не экспорттируя. Реализуем ReverseExt для Vec:

```
impl<T> ReverseExt<T> for Vec<T>
where
    T: Clone,
```

Добавляем в T границу трейта Clone,  
чтобы можно было клонировать  
любой элемент в Vec

```
{ fn reversed(&self) -> Vec<T> {
    self.iter().rev().cloned().collect() }
```

Для реверсирования вектора просто  
получаем итератор, выполняем  
реверсирование с помощью rev(),  
клонируем каждый элемент и переносим  
результат в новый Vec

Протестируем этот код:

```
let forward = vec![1, 2, 3];
let reversed = forward.reversed();
dbg!(&forward);
dbg!(&reversed);
```

А так выглядит ожидаемый вывод:

```
[src/main.rs:17] &forward = [
    1,
    2,
    3,
]
[src/main.rs:18] &reversed = [
    3,
    2,
    1,
]
```

Еще один способ использования трейтов-расширений заключается в их применении к другому трейту, а не к типу. Продолжая предыдущий пример, можно добавить в `std::iter::DoubleEndedIterator` метод `to_reversed()`:

```
Используем супертрейт (будем говорить о них позже)
для ограничения области действия нашего трейта, чтобы
тот применялся только к DoubleEndedIterator
```

```
pub trait DoubleEndedIteratorExt: DoubleEndedIterator { ←
    fn to_reversed<'a, T>(self) -> Vec<T>
    where
        T: 'a + Clone,
        Self: Sized + Iterator<Item = &'a T>; ← Нужно потребовать реализацию границы
                                                    трейта Clone для целевого типа T
```

```
Тип и время жизни
элемента итератора
должны соответствовать
типу и времени жизни T,
вдобавок итератор должен
реализовывать Sized
```

```
impl<I: DoubleEndedIterator> DoubleEndedIteratorExt for I {
    fn to_reversed<'a, T>(self) -> Vec<T>
    where
        T: 'a + Clone,
        Self: Sized + Iterator<Item = &'a T>, ← Почти идентично предыдущей версии,
        {                                         только здесь нет вызова iter()
            self.rev().cloned().collect()       ←
        }
}
```

Протестируем этот расширяющий трейт:

```
let other_reversed = forward.iter().to_reversed();
dbg!(&other_reversed);
```

Получим тот же ожидаемый вывод:

```
[src/main.rs:38] &other_reversed = [
    3,
    2,
    1,
]
```

Один из приятных эффектов применения расширяющего трейта к другому трейту (а не к типу) заключается в том, что это позволяет использовать трейт для любого типа, реализующего `DoubleEndedIterator`, а именно `Vec`, срезов, `std::collections::LinkedList` и пр.

## 7.4. Blanket-трейты

В Rust мы иногда имеем дело с особыми обобщенными трейтами. Особыми в том смысле, что они применяются практически к любому типу. Для этих трейтов могут потребоваться `blanket`-реализации. *Blanket-реализация трейта*, в отличие от конкретной реализации, оперирует обобщенными параметрами. Вдобавок можно создавать частичные `blanket`-реализации, которые для одних параметров будут специализированными, а для других — обобщенными.

С помощью `blanket`-трейтов можно быстро и легко реализовать трейт для всех типов, удовлетворяющих установленному нами критерию. Этот критерий устанавливается в виде ограничений трейтов. То есть реализация `blanket`-трейта будет применяться к любому типу, реализующему указанные в ограничении трейты.

Например, некоторые трейты из стандартной библиотеки Rust предоставляют `blanket`-реализации, которые нередко зависят от других трейтов или типов. В качестве примера можно привести `ToString` со следующей `blanket`-реализацией:

```
impl<T: Display> ToString for T {
    // ...
}
```

Эта реализация, взятая из стандартной библиотеки Rust, требует, чтобы для `T` был реализован `Display`. Для любого типа, который предоставляет `Display`, `ToString` предлагается автоматически (то есть можно вызывать метод `to_string()`).

Создать `blanket`-реализацию относительно легко. Нужно просто использовать обобщенные параметры для всего целевого типа или его части. При желании мы можем создать `blanket`-трейт для всех типов нашего крейта:

```
trait Blanket {}  
impl<T> Blanket for T {} ← Реализует Blanket для всех типов крейта
```

Этот пример в его текущем виде не особо полезен, но его код корректен. `Blanket`-реализации могут пригодиться, когда мы применяем их к конкретным типам или привязываем к другому трейту с помощью ограничений. Иногда нам нужно использовать `blanket`-трейты в качестве маркеров, и об этом мы поговорим в разделе 7.5. Еще один случай — применение бланкет-трейтов для совмещения нескольких других трейтов в один.

`Blanket`-трейты полезны для разработчиков библиотек, которые хотят предоставить пользователям различную функциональность, не реализуя все возможные комбинации типов. Используя пример с `Buffer` из раздела 7.1, мы можем предоставить `blanket`-трейт для выполнения преобразования из `Vec<T>` в `Buffer` (листинг 7.1).

**Листинг 7.1. Реализация blanket-трейта с постоянными дженериками**

```
impl<T: Default + Copy, const LENGTH: usize> From<Vec<T>>
for Buffer<T, LENGTH>
{
    fn from(v: Vec<T>) -> Self {
        assert_eq!(LENGTH, v.len()); ←
        let mut ret = Self {
            buf: [T::default(); LENGTH],
        };
        ret.buf.copy_from_slice(&v); ←
        ret
    }
}
```

Размер Vec должен совпадать с объявленным параметром LENGTH

copy\_from\_slice() внутренне использует memcpy()  
и требует, чтобы источник копирования и цель имели одинаковую длину

Этот код при условии наличия Vec предоставляет blanket-реализацию для трейта `From`, дополняющего `Buffer` любого типа или размера. Он позволяет конвертировать `Vec` в `Buffer`, используя `into()` или `from()`. Этот код также объединяет `Default` и `Copy`, которые являются довольно распространенными, поэтому можно быть уверенными, что они будут доступны для большинства типов. Протестируем наш blanket-трейт:

```
let group_of_seven = vec![
    "Canada",
    "France",
    "Germany",
    "Italy",
    "Japan",
    "United Kingdom",
    "United States",
    "European Union", ←
];
```

Нужно указать 8 в качестве длины целевого буфера, так как на этапе компиляции длина вектора неизвестна; вектор имеет переменную длину

```
let g7_buf: Buffer<&str, 8> = Buffer::from(group_of_seven); ←
dbg!(&g7_buf);
```

**ПРИМЕЧАНИЕ** Вы заметили, что в списке Group of Seven восемь элементов? Это не ошибка. По причинам, описание которых выходит за рамки данной книги, European Union в перечисление не входит.

При выполнении этого кода мы получим такой вывод:

```
[src/main.rs:34] &g7_buf = Buffer {
    buf: [
        "Canada",
        "France",
        "Germany",
        "Italy",
        "Japan",
        "United Kingdom",
        "United States",
        "European Union",
    ],
}
```

Применение blanket-трейтов в ходе разработки библиотек повышает удобство их использования в дальнейшем. Однако не нужно стремиться предоставить максимально обобщенную или любую возможную реализацию. Напротив, следует сосредоточиться на обработке наиболее типичных случаев, как мы сделали, предоставив `From` для `Vec`.

## 7.5. Трейты-маркеры

Когда вы освоитесь с трейтами, вы начнете замечать в других проектах Rust использование *трейтов-маркеров*. Это абстракции, которые отмечают функции или атрибуты типа, но какое-либо поведение при этом могут не предоставлять. (Трейты-маркеры зачастую опознаются по отсутствию в них методов.) Для них нет особого случая применения. Такие трейты могут быть полезными в совершенно разных контекстах.

Отличие трейтов-маркеров от обычных трейтов заключается в том, что первые не обязательно предоставляют какое-либо поведение. Например, `Sync` и `Send` — трейты-маркеры, но ни один из них сам по себе не предоставляет никакой функциональности. Это особые случаи, так как эти трейты-маркеры не получится реализовать без использования `unsafe`. Сделать это безопасно может только компилятор.

Один из видов трейтов-маркеров предоставляет blanket-реализацию, которая совмещает в себе другие трейты. Если нам нужно кратко показать, например, что конкретный тип реализует указанный набор трейтов, то можно обозначить его соответствующим образом. Рассмотрим трейт в листинге 7.2.

### Листинг 7.2. Полноценный трейт-маркер

```
#[derive(
    Clone, Copy, Debug, Default, Eq, Hash, Ord, PartialEq, PartialOrd,
)]
struct KitchenSink; ← Пустая структура, для которой мы выводим все
                     возможные трейты из стандартной библиотеки

trait FullFeatured {} ← Пустой трейт-маркер
impl<T> FullFeatured for T where ← Blanket-реализация трейта-маркера
    T: Clone
        + Copy
        + std::fmt::Debug
        + Default
        + Eq
        + std::hash::Hash
        + Ord
        + PartialEq
        + PartialOrd
    {
}
```

Этот код создает пустой трейт-маркер `FullFeatured`. После этого можно создать blanket-реализацию для любого типа, отвечающего ограничениям,

представленным в виде списка всех допустимых для вывода (derive) трейтов. В этом примере единичная структура `KitchenSink` намеренно пуста, но мы вывели все возможные трейты (из доступных в стандартной библиотеке), добавив атрибут `##[derive(...)]`. Теперь можно использовать этот трейт-маркер везде, где мы захотим убедиться, что эти функции были реализованы и что нам не пришлось перечислять их каждый раз:

```
#[derive(Debug)]
struct Container<T: FullFeatured> { ← | Устанавливает для T ограничение
    t: T,
}
```

Здесь создается тип контейнера, который содержит один элемент. Мы ограничили этот элемент возможностью использовать только те типы, которые предоставляют трейт `FullFeatured`. Явно мы этот трейт не реализовывали, так как опираемся на `blanket`-реализацию. Теперь протестируем этот код:

```
let container = Container { t: KitchenSink {} };
println!("{:?}", container);
```

Вот вывод:

```
Container { t: KitchenSink }
```

Трейты-маркеры не обязательно должны быть пусты, хотя зачастую именно так и есть. Вы можете рассматривать трейты, содержащие методы, как трейты-маркеры, но смешивание этих понятий будет запутывать других людей. В качестве общего правила трейты-маркеры должны быть пустыми (не содержать методов или типов).

## Супертрейты

Здесь будет уместно разобрать *супертрейты*, которые устанавливают трейты, состоящие из других трейтов. Мы уже это делали в примере трейта `FullFeatured`.

Супертрейты можно использовать, когда нужно объединить трейты, создав из них один. Такой подход способен упростить код в другом месте, например уменьшить количество отдельных трейтов, необходимых для указания ограничений. Ограничения трейтов порой становятся довольно сложными, и с помощью супертрейтов можно собрать воедино список требуемых трейтов.

Чтобы создать супертрейт, нужно написать простой и по аналогии с указанием ограничений указать список трейтов, которые от него зависят. Супертрейт-маркер, совмещающий `Clone` и `Debug`, будет выглядеть так:

```
trait CloneAndDebug: Clone + Debug {}
```

Разница между использованием супертрейтов и предоставлением `blanket`-реализации с ограничениями трейтов (как мы делали с `FullFeatured`) состоит в том, что первые предоставляют чуть меньше возможностей (из-за строгости компилятора) и менее

удобны. С помощью супертрейтов нельзя вывести трейт `CloneAndDebug`, если наш тип не будет реализовывать `Clone` и `Debug`. Использование же `blanket`-реализации позволяет делать особые исключения для конкретных типов. Мы по-прежнему можем вывести `FullFeatured` для любого типа, но компилятор не будет ни к чему нас принуждать, как в случае с супертрейтами.

При выборе между супертрейтами и явными реализациями с использованием ограничений трейтов, как в случае с `FullFeatured`, следует отдавать предпочтение первым, если вам нужен лишь псевдоним для набора существующих трейтов. Кроме того, супертрейты позволяют предоставлять дефолтные реализации методов трейтов, использующих зависимые трейты.

Обновим трейт `CloneAndDebug`, чтобы он выводил клонированную копию себя самого и возвращал ее:

```
trait CloneAndDebug: Clone + Debug {
    fn clone_and_dbg(&self) -> Self {
        let r = self.clone();
        dbg!(&r);
        r
    }
}
```

## **7.6. Использование структур в качестве тегов**

Иногда для тегирования или разметки обобщенных типов (тех, что имеют обобщенные параметры) используются структуры. Называется этот подход *тегирование с помощью структур* (struct tagging). Он позволяет использовать пустые структуры (также называемые *единичными*, unit struct) для тегирования обобщенного типа. Делается это путем добавления тега в виде неиспользуемого параметра. Сам такой тег не имеет состояния и может никогда не инстанцироваться.

Аналогично трейтам-маркерам используемые для тегирования структуры обычно пусты. Они применяются для определения состояния внутри самой системы типов. Хитрость в том, что, несмотря на использование абстракции, предназначеннной для хранения состояния (в данном случае структуры), мы не сохраняем в структуре состояние среды выполнения. Вместо этого мы предполагаем использование структуры в качестве параметра обобщенного типа.

Как и в случае трейтов-маркеров, мы задействуем одну из фундаментальных абстракций Rust в некотором смысле «перпендикулярно» ее основной цели. Тем не менее это позволяет нам реализовывать паттерны этапа компиляции типобезопасным способом. Если сравнивать с C++, то этот подход представляет некую форму *метапрограммирования на основе шаблонов*, как в библиотеке Boost MPL (<https://mng.bz/oevN>).

Тегирование структур можно использовать, когда нужно выполнить вычисления на этапе компиляции без привлечения макросов. Этот подход чуть более сложен, но дает преимущество в виде типобезопасности и проверки компилятором. Если вы пишете библиотеку, то можете создавать интерфейсы, которые будут проверяться на корректность во время компиляции, а не при выполнении, что позволит повысить надежность программного обеспечения. Чтобы вы могли увидеть, как используется тегирование структур, смоделируем лампочку, которая будет иметь два состояния: включена и выключена (On и Off) (листинг 7.3).

### Листинг 7.3. Моделирование лампочки с помощью тегирования структур

```
struct LightBulb<T> {
    phantom: PhantomData<T>, ← Структура для моделирования лампочки
}                                     с параметром типа, отражающим ее состояние
                                         ↓
struct On; ← Единичная структура-тег, представляющая
                                         ↓
struct Off; ← включенную лампочку
                                         ↓
                                         Единичная структура-тег, представляющая
                                         ↓
                                         выключенную лампочку
```

Используя `let bulb = LightBulb<Off> { ... }`, мы создадим экземпляр лампочки, представляющий ее выключенное состояние. Такая абстракция пригодится, когда нужно будет поддерживать синхронность состояния ПО с внешним состоянием, например при управлении внешним устройством (той же лампочкой) программным способом. Моделирование с использованием типов вместо переменных позволяет задействовать компилятор для проверки валидности состояний и переходов между ними, о чем я подробнее расскажу чуть позже.

Пока что наш код в порядке, но нам еще нужно создать для состояния лампочки трейт-маркер и добавить ограничение трейта. Вдобавок следует присвоить T более описательное имя (листинг 7.4).

### Листинг 7.4. Добавление трейта в модель лампочки

```
trait BulbState {} ← Добавили к состоянию лампочки трейт-маркер
                     ↓
struct LightBulb<State: BulbState> { ← Установили в LightBulb ограничение трейта
    phantom: PhantomData<State>,   ↓
}                                     для State, обозначив его как тип, который
                                         ↓
                                         должен содержать трейт BulbState
                                         ↓
struct On {}
struct Off {}

impl BulbState for On {} ← Реализуем трейт-маркер BulbState
                           ↓
impl BulbState for Off {}           для состояний On и Off
```

Этот паттерн будет особенно полезен, если мы начнем использовать состояние типа для создания методов. Предположим, что хотим перевести лампочку между двумя состояниями, то есть из On в Off и обратно (листинг 7.5).

**Листинг 7.5. Добавление переходов между состояниями**

```

impl LightBulb<On> {
    fn turn_off(self) -> LightBulb<Off> { ←
        LightBulb::<Off>::default() ← Создаем конкретную реализацию
    }                                     для лампочки в состоянии On
    fn state(&self) -> &str { ←
        "on"                                Определяем метод turn_off(),
    }                                     который получает текущую лампочку
                                            и возвращает новую в состоянии Off
}

impl LightBulb<Off> {
    fn turn_on(self) -> LightBulb<On> { ←
        LightBulb::<On>::default() ← Для удобства добавили метод,
    }                                     возвращающий имя этого состояния
    fn state(&self) -> &str {
        "off"
    }
}

```

Обратите внимание: в этом примере и `turn_off()`, и `turn_on()` получают занятый `self`, который потребляет текущую `LightBulb` и возвращает новую. В обобщенных структурах менять параметр типа нельзя, поэтому их нужно создавать и уничтожать. Ну а теперь протестируем результат:

```

let lightbulb = LightBulb::<Off>::default();
println!("Bulb is {}", lightbulb.state());
let lightbulb = lightbulb.turn_on();
println!("Bulb is {}", lightbulb.state());
let lightbulb = lightbulb.turn_off();
println!("Bulb is {}", lightbulb.state());

```

Вот вывод:

```

Bulb is off
Bulb is on
Bulb is off

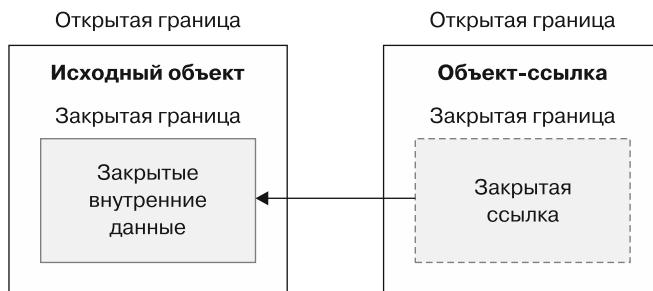
```

Отлично! Этот паттерн имеет большое преимущество: он обеспечивает проверку со стороны системы типов. Его можно использовать для создания типобезопасного конечного автомата, о чем пойдет речь в главе 8.

## 7.7. Объекты-ссылки

*Объекты-ссылки* предоставляют ссылку на внутренние данные. С помощью таких объектов можно реализовывать частичное заимствование внутренних данных, не давая открытого доступа к ним. Иными словами, можно завернуть закрытые внутренние данные в открытый объект-ссылку, избежав создания абстракций, допускающих утечки, или раскрытия внутренних данных. Обычно объекты-ссылки содержат в своем имени постфикс `Ref`, который определяет их как несущие ссылку.

На рис. 7.1 показано, как объекты-ссылки сохраняют границы открытого и закрытого доступа, в то же время предоставляя ссылочные данные (частично или полностью) внутри объекта.



**Рис. 7.1.** Объекты-ссылки

С помощью объектов-ссылок мы даем потребителям нашего API возможность обмениваться данными по ссылкам, не раскрывая внутренние структуры или детали реализации. Как правило, такие объекты передаются в наш API через интерфейсы, оперирующие этими данными, что в некоторых случаях позволяет избежать излишнего копирования. Подобные объекты-ссылки предназначены для использования только вместе со своим родительским API.

Предположим, мы хотим создать две структуры: `Student` и `StudentList`. `StudentList` является публичной и содержит `Vec`, а `Student` закрытая, поскольку эти данными мы делиться не хотим. Базовое определение объекта выглядит так:

```

#[derive(Debug)]
struct Student {
    name: String,
    id: u32,
}

#[derive(Debug)]
pub struct StudentList {
    students: Vec<Student>,
}

```

А теперь предположим, что хотим спроектировать код, получающий ссылки на отдельных студентов из списка, но не хотим предоставлять прямой доступ к внутренней информации. У нас могут быть методы, которые работают с объектами-ссылками и выполняют операции, не имея прямого доступа к данным. Создадим открытый объект-ссылку:

```

#[derive(Debug)]
pub struct StudentRef<'a> { ←
    student: &'a Student,
}

```

Параметр времени жизни '`'a` позволяет хранить эту ссылку на протяжении существования объекта `Student`

На данном этапе у нас есть базовый объект-ссылка `StudentRef`. Протестируем его:

```
let student = Student {
    name: "Walter".into(),
    id: 582,
};

let student_ref = StudentRef { student: &student };
dbg!(&student);
dbg!(student_ref);
```

Вывод выглядит так:

```
[src/main.rs:59] &student = Student {
    name: "Walter",
    id: 582,
}
[src/main.rs:60] student_ref = StudentRef {
    student: Student {
        name: "Walter",
        id: 582,
    },
}
```

Этот пример работает ожидаемым образом, но нужно сделать его чуть более реалистичным. Для начала добавим в объект `Student` конструктор и аксессоры (листинг 7.6).

### Листинг 7.6. Student с конструктором и аксессорами

```
#[derive(Debug)]
struct Student {
    name: String,
    id: u32,
}

impl Student {
    fn new(name: String, id: u32) -> Self {
        Self { name, id }
    }

    fn name(&self) -> &str {
        self.name.as_ref()
    }

    fn id(&self) -> u32 {
        self.id
    }
}
```

Теперь нам нужен способ получения ссылки из `Student`. Создадим метод `to_ref()` (листинг 7.7).

**Листинг 7.7. Реализация Student::to\_ref() для получения ссылки**

```
impl<'a> Student { ← Обратите внимание на параметр времени жизни 'a
    fn to_ref(&'a self) -> StudentRef<'a> {
        StudentRef::new(self) ←
    }
}
```

Мы еще не создали метод StudentRef::new()

Тот же параметр 'a используется для получателя метода self и возвращаемой StudentRef

Теперь добавим конструктор, получающий список кортежей. Кроме того, нужно предоставить доступ к данным об отдельных студентах, указанным в StudentList. Будет удобно находить их по ID или имени, так что добавим соответствующие методы (листинг 7.8).

**Листинг 7.8. StudentList с конструктором и методами для поиска**

```
#[derive(Debug)]
pub struct StudentList {
    students: Vec<Student>,
}

impl StudentList {
    pub fn new(students: &[(&str, u32)]) -> Self { ←
        Self {
            students: students
                .iter()
                .map(|(name, id)| {
                    Student::new((*name).into(), *id) ←
                })
                .collect(),
        }
    }
}

impl<'a> StudentList { ←
    fn find<F: Fn(&&Student) -> bool>(
        &'a self,
        pred: F,
    ) -> Option<StudentRef<'a>> { ←
        self.students.iter()
            .find(pred) ←
            .map(Student::to_ref) ←
    }
    pub fn find_student_by_id(&'a self, id: u32) -> Option<StudentRef<'a>> {
        self.find(|s| s.id() == id) ←
    }
    pub fn find_student_by_name(
        &'a self,
        name: &str,
    ) -> Option<StudentRef<'a>> { ←
        self.find(|s| s.name() == name) ←
    }
}
```

Получаем срез кортежей для инициализации списка

Каждый кортеж содержит данные о студенте

Обратите внимание на параметр времени жизни 'a

Параметры 'a в self и StudentRef должны совпадать

Iterator::find() прекращает перебор, когда предикат возвращает true

Отображаем Some(student) в StudentRef, используя метод Student::to\_ref()

Оба метода вызывают закрытый метод find(), передавая замыкания с почти идентичными реализациями, различающимися только параметром поиска

Обратите внимание, что StudentList::find\_student\_by\_id() и StudentList::find\_student\_by\_name() почти идентичны. Они различаются лишь параметрами id

и `name`, которые мы реорганизовали в закрытый метод `find()`, получающий предикат в виде замыкания. Протестируем получившийся код:

```
let student_list =
    StudentList::new(&[("Lyle", 621), ("Anna", 286)]);

dbg!(&student_list);
dbg!(student_list.find_student_by_id(621));
dbg!(student_list.find_student_by_name("Anna"));
```

При его выполнении мы получим такой вывод:

```
[src/main.rs:84] &student_list = StudentList {
    students: [
        Student {
            name: "Lyle",
            id: 621,
        },
        Student {
            name: "Anna",
            id: 286,
        },
    ],
}
[src/main.rs:85] student_list.find_student_by_id(621) = Some(
    StudentRef {
        student: Student {
            name: "Lyle",
            id: 621,
        },
    },
)
[src/main.rs:86] student_list.find_student_by_name("Anna") = Some(
    StudentRef {
        student: Student {
            name: "Anna",
            id: 286,
        },
    },
)
```

Пока все выглядит гладко. Закончим с нашим `StudentRef`, добавив конструктор (листинг 7.9).

#### Листинг 7.9. `StudentRef` с конструктором

```
#[derive(Debug)]
pub struct StudentRef<'a> {
    student: &'a Student,
}

impl<'a> StudentRef<'a> {
    fn new(student: &'a Student) -> Self {
        Self { student }
    }
}
```

В завершение создадим открытую функцию, работающую с закрытыми данными. Для этого используем `StudentRef`, не раскрывая внутренние данные объекта `Student` вызывающему коду. Кроме того, реализуем трейт `PartialEq` для проверки равенства идентификаторов студентов:

```
impl<'a> PartialEq for StudentRef<'a> {
    fn eq(&self, other: &Self) -> bool {
        self.student.id() == other.student.id()
    }
}
```

Протестируем `PartialEq`:

```
let student_ref_621 = student_list.find_student_by_id(621).unwrap();
let student_ref_286 = student_list.find_student_by_id(286).unwrap();
dbg!(student_ref_286 == student_ref_621);
dbg!(student_ref_286 != student_ref_621);
```

Вывод будет выглядеть так:

```
[src/main.rs:99] student_ref_286 == student_ref_621 = false
[src/main.rs:100] student_ref_286 != student_ref_621 = true
```

Напоследок добавлю, что можно создавать изменяемые объекты-ссылки, но эту задачу я оставил вам в качестве упражнения. Они почти такие же, только обычно в их имени есть постфикс `MutRef`. В случае их использования потребуется добавлять ключевое слово `mut` ко всем ссылкам, чтобы исключить ошибки заимствования (используя при необходимости `&mut` и `&'a mut`).

## Резюме

- Постоянные дженерики позволяют использовать постоянные значения в качестве параметров типов. Это дает возможность применять массивы любой фиксированной длины.
- Нельзя реализовать трейт для типов вне крейта, но можно обойти это ограничение, используя оберточные структуры с трейтами `Deref` и `DerefMut`.
- Трейты-расширения дополняют или изменяют поведение внешних типов либо трейтов, например, из стандартной библиотеки.
- Можно реализовывать трейт автоматически для любой комбинации типов, используя обобщенные реализации, известные как `blanket`-трейты.
- Трейты-маркеры позволяют обозначать типы, имеющие конкретную функциональность или атрибуты, например совмещающие несколько других трейтов.
- Можно тегировать обобщенные типы, используя пустую (то есть единичную) структуру в качестве тега.
- Объекты-ссылки предоставляют доступ к закрытым внутренним данным, не требуя передачи владения ими или раскрытия внутренних закрытых объектов.

# Конечные автоматы, корутины, макросы и прелюдии

---

## В этой главе

- ✓ Создание конечных автоматов с помощью трейтов.
- ✓ Использование корутин для создания функций с возможностью приостановки.
- ✓ Реализация процедурных макросов.
- ✓ Повышение удобства использования крейтов с помощью прелюдий.

В этой главе мы продолжим рассматривать некоторые темы из главы 7 и будем во многом оперировать понятиями, описанными в книге ранее. Начнем с обсуждения конечных автоматов и корутин. После этого перейдем к процедурным макросам и в завершение обсудим прелюдии — популярный библиотечный паттерн в Rust.

Конечные автоматы — надежный способ моделирования систем с состоянием. Вы увидите, что на Rust создавать их типобезопасным способом на удивление просто.

Корутины — экспериментальная функциональность, которая заслуживает внимания, поскольку в ближайшем будущем начнет играть в Rust важную роль. Если вы имели опыт работы с генераторами в Python, то корутины Rust покажутся вам знакомыми.

Процедурные макросы — продвинутая функциональность Rust, которая позволяет генерировать код на этапе компиляции. Ну а прелюдии — это коллекции типов, функций и макросов, предназначенные для импорта в код. Предоставление прелюдий позволяет сделать работу с библиотеками более удобной для пользователей.

Начнем с конечных автоматов. Эта тема всегда была мне интересна, и я пользовался этими элементами много раз. Но особенно мне понравилось то, что создавать их в Rust очень просто — не нужно использовать дополнительные крейты и библиотеки. Реализуя системы с состоянием в Rust, я по мере необходимости создаю множество конечных автоматов.

## 8.1. Конечный автомат на основе трейтов

В предыдущих главах вы познакомились с трейтами и дженериками, и теперь можно начать создавать поверх системы типов Rust интересные абстракции. Одна из таких полезных абстракций связана с построением конечных автоматов. *Конечный автомат* обычно состоит из списка состояний и набора переходов между ними. Мы можем определять столько состояний и переходов, сколько захотим, но выполнять можно только валидные переходы. Такие правила диктует нам система типов Rust.

В главе 7 эти правила были показаны на примере с лампочкой, которая имела два состояния: включена и выключена. Здесь же мы углубимся в эту тему и смоделируем сеанс учетной записи пользователя с помощью конечного автомата (рис. 8.1). Предположим, что у нас может быть анонимный либо аутентифицированный пользователь.

В зависимости от состояния сеанса пользователю доступны различные действия, например возможность изменять настройки учетной записи. Наш сеанс, помимо ряда дополнительных свойств, будет содержать ID, отображающийся в состояние на стороне пользователя (например, через куки) и ID сеанса в базе данных. Один и тот же код можно использовать как на стороне клиента, так и на сервере. В листинге 8.1 показаны структуры, которые мы создадим.

### Листинг 8.1. Моделирование состояния сеанса с помощью трейтов и тегирования структур

```
pub trait SessionState {}

#[derive(Debug, Default)]
pub struct Session<State: SessionState = Initial> {
    session_id: Uuid,
    props: HashMap<String, String>, ←
    phantom: PhantomData<State>, ←
}
```

Устанавливаем базовое  
состояние сеанса на Initial

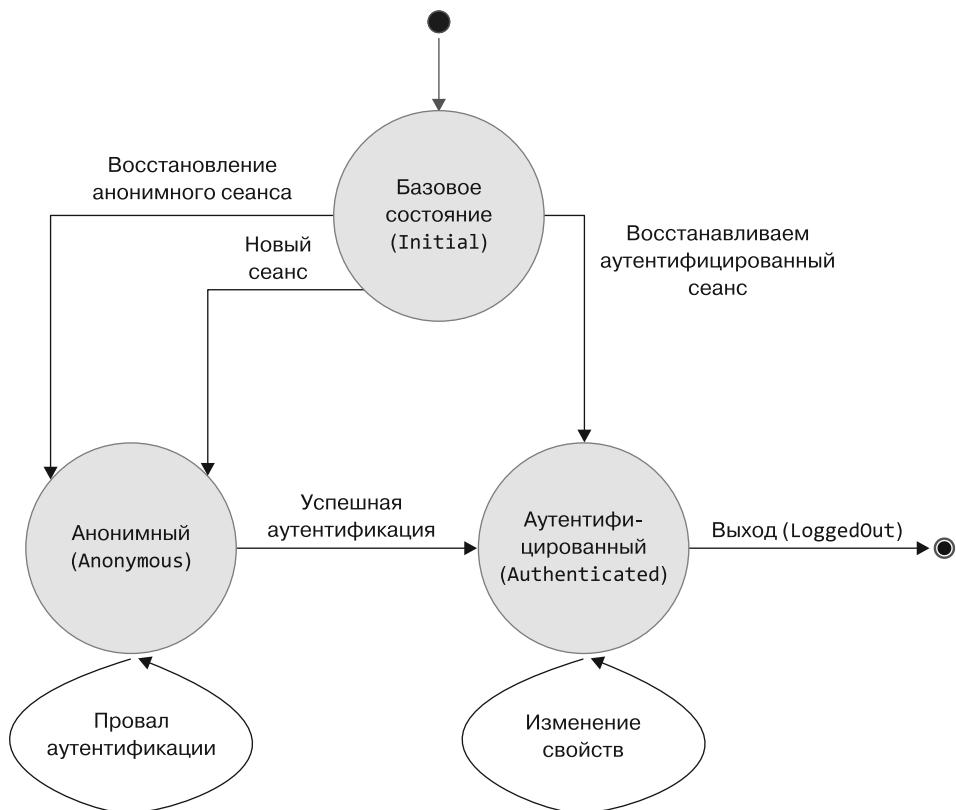
Мы будем вести HashMap произвольных свойств,  
которую можно хранить в базе данных

```

#[derive(Debug, Default)]
pub struct Initial;
#[derive(Debug, Default)]
pub struct Anonymous;
#[derive(Debug, Default)]
pub struct Authenticated;
#[derive(Debug, Default)]
pub struct LoggedOut;

impl SessionState for Initial {}
impl SessionState for Anonymous {}
impl SessionState for Authenticated {}
impl SessionState for LoggedOut {}

```



**Рис. 8.1.** Моделирование сеанса пользователя с помощью конечного автомата

В этом листинге определяются четыре состояния сеанса: `Initial`, `Anonymous`, `Authenticated` и `LoggedOut`. На рис. 8.1 показана связь между ними. Мы добавили в структуру `Session` поле `session_id`, где будет храниться универсальный

уникальный идентификатор (universally unique identifier, UUID), предоставляемый крейтом `uuid`. Теперь приступим к добавлению методов. Начнем с листинга 8.2.

### Листинг 8.2. Обработка базового (Initial) состояния сеанса

```
#[derive(Debug)]
pub enum ResumeResult { Invalid, Anonymous(Session<Anonymous>),
    Authenticated(Session<Authenticated>),
}

impl Session<Initial> {
    // Возвращает новый сеанс, по умолчанию с состоянием anonymous.
    pub fn new() -> Session<Anonymous> {
        Session::<Anonymous> {
            session_id: Uuid::new_v4(),
            props: HashMap::new(),
            phantom: PhantomData,
        }
    }

    /// Возвращает результат восстановления этого сеанса на основе имеющегося
    ID.
    pub fn resume_from(session_id: Uuid) -> ResumeResult {
        ResumeResult::Authenticated(
            Session::<Authenticated> {
                session_id,
                props: HashMap::new(),
                phantom: PhantomData,
            })
    }
}
```

Перечисление, представляющее результат изначального перехода между состояниями

Эти методы, аналогично сеансу с базовым состоянием, ограничены `Session<Initial>`

// Возвращает новый сеанс, по умолчанию с состоянием anonymous.

Предоставляем метод `new()` для нового анонимного сеанса

Возвращает из текущего сеанса `ResumeResult` в `resume`

Здесь нужно сверить `session_id` с базой данных и вернуть соответствующий результат. В этом примере в целях тестирования мы вернем новый сеанс с аутентификацией

С помощью этого кода можно создать новый анонимный сеанс или восстановить существующий аутентифицированный. На практике операция восстановления будет подразумевать поиск по базе данных и сверку с ID сеанса, но мы опустим эти этапы. Теперь перейдем к листингу 8.3.

### Листинг 8.3. Добавление перехода для анонимного сеанса

```
impl Session<Anonymous> {
    pub fn authenticate(
        self,
        username: &str,
        password: &str,
    ) -> Result<Session<Authenticated>, Session<Anonymous>> {
        // ...
        if !username.is_empty()
            && !password.is_empty() {
            Ok(Session::<Authenticated> {
                session_id: self.session_id,
                props: HashMap::new(),
            })
        } else {
            Err(Session::<Anonymous> {
                session_id: self.session_id,
                props: HashMap::new(),
            })
        }
    }
}
```

Эти методы ограничены экземплярами `Session<Anonymous>`

Возвращаем `Result` с успехом или провалом, и он потребляет `self`

Симулируем проверку учетных данных на предмет их отсутствия

Здесь должна выполняться аутентификация, но в данном примере мы симулируем этот процесс. Мы используем в качестве типа ошибки `Session<Anonymous>`. Это позволяет указать, что аутентификация провалилась и сеанс по-прежнему находится в анонимном состоянии

```
        phantom: PhantomData,
    })
} else {
    Err(self)
}
}
```

И в завершение — листинг 8.4.

**Листинг 8.4. Добавление переходов для аутентифицированного сеанса**

```
impl Session<Authenticated> { ← Эти методы ограничены экземплярами Session<Authenticated>
    pub fn update_property(&mut self,
                          key: &str,
                          value: &str) {
        if let Some(prop) = self.props.get_mut(key) {
            *prop = value.to_string();
        } else {
            self.props.insert(key.to_string(), value.to_string());
        }
        // ... ← | Здесь можно производить фактическое обновление свойств (например,
        | запись в базу данных), а также обрабатывать ошибки и пограничные
        | случаи, но в данном случае мы симулируем этот пример
    }
}

pub fn logout(self) -> Session<LoggedOut> { ← Вызов logout() ведет
    // ... ← | к потреблению сеанса и возврату
    | его в состоянии LoggedOut
    Session {
        session_id: Uuid::nil(),
        props: HashMap::new(),
        phantom: PhantomData,
    }
}
```

Теперь у нас есть прекрасный конечный автомат для обработки сеансов. Протестируем его:

```
let session = Session::new();
println!("{:?}", session);
if let Ok(mut session) =
    session.authenticate("username", "password")
{
    session.update_property("key", "value");
    println!("{:?}", session);
    let session = session.logout();
    println!("{:?}", session);
}
```

При выполнении этого кода мы получим такой вывод:

```
Session { session_id: f0981fc3-3761-407f-b037-8759535acf87, props: {}, phantom: PhantomData }
Session { session_id: f0981fc3-3761-407f-b037-8759535acf87, props: {"some.preference.bool": "true"}, phantom: PhantomData }
Session { session_id: 00000000-0000-0000-0000-000000000000, props: {}, phantom: PhantomData }
```

Неплохо! Это очень мощная абстракция, и, выполняя моделирование с помощью конечных автоматов, можно создавать достаточно надежные системы. И хотя конечные автоматы не панацея, они могут значительно упрощать понимание сложных систем с различными состояниями. Можно легко и быстро создать конечный автомат в сочетании с системой типов Rust, не используя дополнительные библиотеки.

## 8.2. Корутины

Корутины в Rust служат для приостановки выполнения функций. С их помощью можно создать замыкание, которое будет возвращать данные вызывающему коду по двум отдельным путям: через операцию `yield` (промежуточная выдача) и в виде конечного результата. Кроме того, можно приостанавливать или завершать корутину после выдачи (`yield`), что позволяет при необходимости не дожидаться окончания ее выполнения. Корутины в Rust покажутся знакомыми тем, кто уже работал с генераторами в Python. Корутины в Rust пока являются экспериментальной функциональностью и доступны только вочной версии, но они заслуживают внимания, так как являются очень важными и обладают мощным потенциалом.

### Об истоках корутин

Корутины в некотором смысле определяются как функции, выполнение которых можно приостанавливать и возобновлять. Сегодня внимание к ним снова возрастает, но изначально они были придуманы еще Мелвином Конвеем (Melvin Conway) (закон Конвея). Конвей разработал и закрепил термин «корутина» в далеком 1958 году. Примерно в то же время Дж. Эрдинн (J. Erdwinn) и Дж. Мернер (J. Merner) изучали аналогичную идею, но их работа *Bilateral Linkage*, в которой были описаны достигнутые ими результаты, так и не вышла. В 1963 году Конвей более развернуто объяснил принцип корутин в статье *Design of a Separable Transition-Diagram Compiler*, опубликованной в журнале Communications of the ACM.

Возрождение популярности корутин могло стать следствием их использования в реализации генераторов на языке Python (появились в Python 2.5 в 2006 году), а также применения в языке Go и др. В последнее время аналогичные реализации были добавлены во многие популярные языки программирования, такие как C++20, C# 2.0, Ruby Fibers и PHP 5.5.

Благодаря корутинам можно использовать конкурентность, не применяя потоки, обратные вызовы или межпроцессное взаимодействие. С их помощью можно создавать сложные потоки управления, такие как кооперативная многозадачность и циклы событий.

Внутренне корутины реализуются компилятором Rust на основе простого конечного автомата. При их использовании возникают минимальные издержки,

которые заключаются лишь в одном перечислении для отслеживания текущего состояния корутины.

**ПРИМЕЧАНИЕ** Информацию о развитии функциональности корутин в Rust можно найти в Rust Unstable Book по адресу <https://mng.bz/ngnv>.

Корутины можно использовать для решения разных задач, но один из вариантов их применения — это создание итераторов потоков данных. Корутины Rust предназначены для расширения функциональности `async/await`. Их также можно задействовать в качестве строительных блоков при создании систем, использующих переключение контекста или мультиплексирование, например при сетевом программировании и в «зеленых» потоках. Реализация корутин в Rust определена в трейте `std::ops::Coroutine` (листинг 8.5).

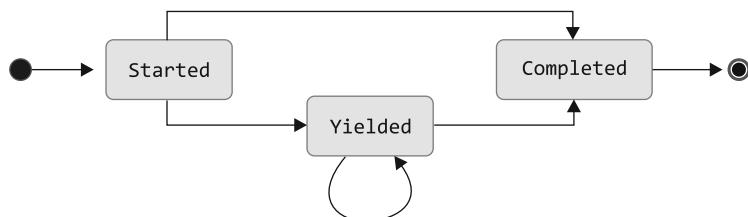
### Листинг 8.5. Определение трейта `std::ops::Coroutine` из стандартной библиотеки Rust

```
pub trait Coroutine<R = ()> { ← R — это аргументы замыкания,
    type Yield; ← Self::Yield — это тип выдачи, которым будет unit(), если не установлено иное
    type Return; ← Self::Return определяет возвращаемый тип замыкания

    // Необходимый метод
    fn resume( ← Корутина необходимо
        self: Pin<&mut Self>, ← закреплять в памяти
        arg: R
    ) -> CoroutineState<Self::Yield, Self::Return>;
}
```

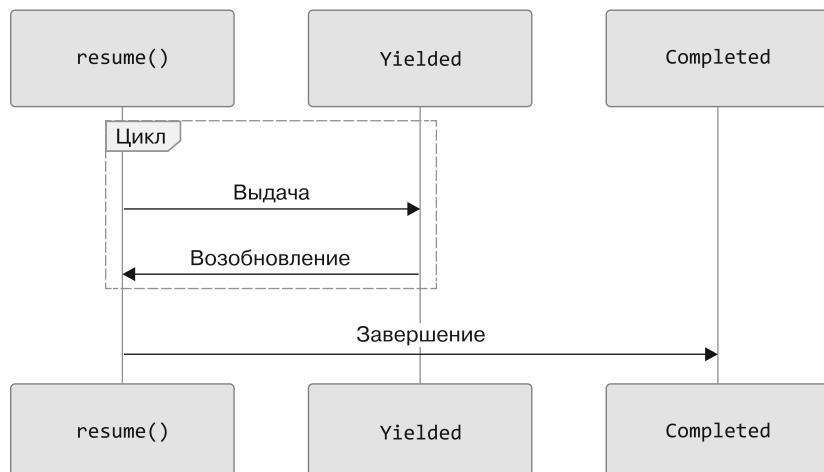
Трейт корутины не обязательно реализовывать явно. Когда вы создаете замыкание с инструкцией `yield`, компилятор делает это за вас. А при возникновении более сложных сценариев, которые будут представлены в листингах 8.6 и 8.7, нужно будет разобраться, как корутины реализуются с помощью трейта `Coroutines`.

На рис. 8.2 показан конечный автомат для корутины. Здесь мы видим, что при запуске первым вызовом `resume()` корутина может продолжать выдавать значения бесконечно, пока не завершит свое выполнение, в случае чего перейдет в состояние `Completed` и прекратит выдачу.



**Рис. 8.2.** Внутренний конечный автомат корутины

Корутина начинается с состояния `Started` и после первого вызова `resume()` переходит к `Yielded` или `Completed`. Если она выдает значение, то переходит в состояние `Yielded` и может продолжать выдавать их бесконечно (рис. 8.3). Когда корутина завершает выполнение, то переходит в состояние `Completed` и больше значений не выдает. Корутина может возобновлять выполнение любое количество раз (как цикл), но после завершения возобновить ее будет уже нельзя.



**Рис. 8.3.** Блок-схема механизма работы корутины

Теперь разберем базовый синтаксис для создания корутины. Для этого достаточно написать замыкание с инструкцией `yield` и применить к нему атрибут `#[Coroutine]`. Все это показано в листинге 8.6.

#### Листинг 8.6. Базовая корутина на Rust

```

#![feature(coroutines,
          coroutine_trait,
          stmt_expr_attributes)]
use core::f64::consts::PI;
use std::ops::{Coroutine, CoroutineState};
use std::pin::Pin;
fn main() {
    let mut yield_pi = #[coroutine]
    || {
        yield PI;
        "Coroutine complete!"
    };
}

```

Coroutines — это доступная только в ночной версии и пока нестабильная функциональность, для активации которой необходимо установить специальный флаг (feature gate). Вдобавок для этого нужно включить атрибуты выражений

Корутина определяется через создание замыкания

К замыканию необходимо применить атрибут `#[coroutine]`

Замыкание должно содержать оператор `yield`. При этом может использоваться несколько таких операторов, но их типы должны совпадать

Кроме того, корутины имеют тип возвращаемого значения, отличающийся от типа, выдаваемого при остановке. Мы можем опустить явный возврат, так как корутина сама по себе является оператором (возвращает итоговое выражение)

```

loop {
    match Pin::new(&mut yield_pi).resume(()) {
        CoroutineState::Yielded(val) => {
            dbg!(&val);           Значение можно выдавать
                                   любое количество раз
        }
        CoroutineState::Complete(val) => {
            dbg!(&val);
            break;
        }
    }
}

```

Когда корутина возвращает результат из своего замыкания, она завершается

Корутины не выполняются, пока не будут возобновлены, и их необходимо закреплять. Закрепление исключает перемещение корутины в памяти в процессе выполнения

Наша базовая корутина выдает число пи, после чего возвращает строку. Выдачу она делает при первом вызове `resume()`, а при втором — возвращает результат и входит в состояние `Complete`. Выдача значения — необязательная операция. Вдобавок можно использовать оператор `yield` без аргумента, что будет равнозначно выдаче `unit()`. При выполнении кода выше мы получим такой вывод:

```
[src/main.rs:15:17] &val = 3.141592653589793
[src/main.rs:18:17] &val = "Coroutine complete!"
```

**ПРИМЕЧАНИЕ** На момент написания книги итератор нужно реализовывать в самой корутине. Но в будущих версиях Rust для трейта `Coroutine` может быть представлена `blanket`-реализация `Iterator`.

Чтобы вы могли увидеть более интересный пример, мы реализуем поверх корутины трейт `Iterator`. Это позволит, помимо прочих возможностей итератора, использовать синтаксис цикла `for`. Для наглядности прочтем файл `Cargo.toml` из проекта, над которым работаем. Определим структуру `CargoTomlReader` (листинг 8.7).

### Листинг 8.7. Реализация `Iterator` поверх корутины

```

struct CargoTomlReader {
    coroutine:
        Pin<Box<dyn Coroutine<
            Yield = (usize, String),
            Return = ()
        >>>,
}

```

Мы используем трейт-объект внутри зафиксированной (Pin) области памяти в куче (Box), указывая типы для Yield и Return

```

impl CargoTomlReader {
    fn new() -> io::Result<Self> {
        let file = File::open("Cargo.toml")?;
        let mut reader = BufReader::new(file);
        let mut line_number: usize = 0;
    }
    let coroutine = Box::pin(
        #[coroutine]
        move || loop {

```

Замыкание нашей корутины состоит из цикла, который будет выдавать каждую строку файла, и мы перемещаем в это замыкание перехваченные значения

Отслеживаем нумерацию строк

Замыкание должно иметь атрибут #[coroutine]

```

let mut line = String::new();
line_number += 1;
match reader.read_line(&mut line) {
    Ok(0) => return,
    Ok(_) => yield (line_number, line),
    _ => return,
}
};

let coroutine = Box::pin(
#[coroutine]
move || loop {
    let mut line = String::new();
    line_number += 1;
    match reader.read_line(&mut line) {
        Ok(0) => return,
        Ok(_) => yield (line_number, line),
        _ => return,
    }
},
);
Ok(Self { coroutine })
}

impl Iterator for CargoTomlReader {
    type Item = (usize, String);
    fn next(&mut self) -> Option<Self::Item> {
        match self.coroutine.as_mut().resume() {
            CoroutineState::Yielded(val) => Some(val),
            CoroutineState::Complete(_) => None,
        }
    }
}

```

При каждом проходе цикла номер строки инкрементируется

Если строка BufReader::read\_line возвращает 0, значит, мы достигли конца файла — завершаем выполнение и возвращаем результат

Все остальные ошибочные случаи приводят к завершению корутины

Передаем в корутину unit() в качестве стартового аргумента

Протестируем нашу `CargoTomlReader`, используя следующий код, чтобы вывести каждую пронумерованную строку:

```

let cargo_reader = CargoTomlReader::new()?;
for (line_number, line) in cargo_reader {
    println!("{}: {}", line_number, line);
}

```

Мы получим такой вывод:

```

1: [package]
2: name = "coroutines" 3: version = "0.1.0"
4: edition = "2021"
5:
6: # See more keys and their definitions at https://doc.rust-lang.org/cargo/
   reference/manifest.html
7:
8: [dependencies]

```

Этот пример демонстрирует несколько ключевых моментов использования корутин совместно с итератором. В частности, поскольку вам нужно фиксировать замыкание корутины в памяти, то относительно легко это сделать с помощью `Pin<Box<T>>`. Любое состояние, которое необходимо внутри корутины, можно инициализировать в начале замыкания или же использовать `move` для перемещения в это замыкание перехваченных переменных, как я сделал в листинге 8.7.

Корутины — прекрасное нововведение, но им еще предстоит доработка. Будьте внимательны, полагаясь на этот API, так как он все еще нестабилен. Нельзя с уверенностью сказать, когда эта функциональность стабилизируется, но вы можете смело экспериментировать с ней и давать обратную связь разработчикам Rust. Естественно, если вы вообще готовы работать с нестабильной функцией.

## **8.3. Процедурные макросы**

*Процедурные макросы* — передовой механизм макросов в Rust, который позволяет выполнять метапрограммирование любой сложности, создавая всевозможные расширения языка. Мы уже использовали процедурные макросы ранее в книге, но их реализацию не разбирали.

Они применяются во многих крейтах, и чаще всего, как мы неоднократно видели, в виде атрибута `#[derive(...)]`. Процедурные макросы — очень обширная тема, заслуживающая отдельной книги, поэтому в данном разделе я затрону лишь основные моменты.

Создание процедурного макроса сопряжено с написанием библиотеки, которая будет экспортировать один или несколько макросов и использовать крейт `proc_macro` для его реализации. Крейт `proc_macro` является частью Rust и широко используется в экосистеме языка. Определить процедурный макрос в двоичном крейте не получится. Для этого используется отдельный библиотечный крейт, хотя вы можете добавить его в свой проект в качестве элемента рабочего пространства. Процедурные макросы бывают трех видов:

- в стиле функций, похожие на декларативные макросы, например `my_functionlike_macro!();`;
- производные, к примеру `#[derive(MyDerivableMacro)];`
- атрибуты, такие как `#[MyAttribute]`.

Универсальных правил о том, какую форму и когда использовать, нет, но я дам по каждой из них комментарий.

- Процедурный макрос в стиле функции. Может быть представлен в форме `macro!()`, `macro!{}` или `macro![ ]`. Применим в любой части кода и обычно рассматривается как функция или блок кода.

- Производный макрос. Выражается в виде `#[derive(...)]` и может использоваться только с объявлением структуры либо перечисления. Но такие макросы позволяют внедрять после них код.
- Макрос-атрибут. Выражается в виде `#[MyAttribute]` и может использоваться для внедрения кода практически везде, но требует прикрепления к существующему элементу. Макросы-атрибуты обладают одной особенностью — позволяют предоставлять для атрибута аргументы.

Определение процедурных макросов подразумевает предоставление кода на Rust, возвращающего синтаксис на Rust. Имеется в виду, что определение вашего макроса будет написано на Rust и будет создавать код на этом языке. Для реализации таких макросов используется крейт `proc_macro`. При этом готовый макрос аналогично любым другим будет вычисляться на этапе компиляции.

Теперь разберем простой пример, в котором создадим библиотеку с помощью следующего кода:

```
use proc_macro::TokenStream;
#[proc_macro]
pub fn say_hello_world(_item: TokenStream) -> TokenStream {
    println!("hello world").parse().unwrap()
}
```

Этот атрибут указывает, что следующая за ним функция является процедурным макросом

Реализации процедурных макросов — это функции, которые получают `TokenStream` и взамен возвращают `TokenStream`

`parse()` из `FromStr` будет выполнять парсинг этой строки в `TokenStream`

Этот код работает с необработанными потоками токенов. На практике вы не станете писать процедурный макрос таким образом. Для этого вы используете более высокоуровневые библиотеки, о которых мы поговорим чуть позже. Кроме того, нам нужно обновить `Cargo.toml`, указав, что это крейт `proc_macro`. Он умеет экспорттировать только процедурные макросы, но вы можете добавлять и другие крейты в качестве зависимостей:

```
[lib]
proc-macro = true
```

Теперь протестируем наш код:

```
use hello_world::say_hello_world;
say_hello_world!();
```

При выполнении этого кода мы получим `hello world`. Как я уже говорил, вы вряд ли станете работать напрямую с `TokenStream`. Вместо этого для написания процедурного макроса стоит использовать две важные библиотеки: `syn` и `quote`. Крейт `syn` предоставляет библиотеку парсинга, упрощающую работу с исходным кодом, а `quote` ускоряет генерацию Rust-кода.

Теперь я приведу более реалистичный пример процедурного макроса, чтобы вы могли понять, как все это работает вместе. Мы создадим собственный

производный макрос, который будет выводить имя структуры, к которой он прикреплен. Этот макрос представляет форму отражения, и мы используем именно его, так как он удобно крепится к объявлению структуры или перечисления.

Для начала определим трейт, который должен находиться в отдельном крейте, поскольку библиотека процедурных макросов может экспорттировать только такие макросы. Вот этот трейт (листинг 8.8).

### Листинг 8.8. Трейт для вывода имени структуры

```
pub trait PrintName {
    fn name() -> &'static str;
    fn print_name() {
        println!("{}", Self::name());
    }
}
```

Для реализации трейта `PrintName` нужно определить метод `name()`, после чего можно будет вызвать `print_name()`, чтобы вывести имя того, для чего он реализуется. Теперь напишем макрос (листинг 8.9).

### Листинг 8.9. Реализация производного макроса `PrintName`

```
#[proc_macro_derive(PrintName)]
pub fn print_name(input: TokenStream) -> TokenStream {
    let input = parse_macro_input!(input as DeriveInput); ←
    let generics = add_trait_bounds(input.generics);
    let (impl_generics, type_generics, where_clause) = ←
        generics.split_for_impl(); ←
    let name = input.ident; let expanded = quote! { ←
        impl #impl_generics print_name::PrintName for #name #type_generics ←
            #where_clause { ←
                fn name() -> &'static str { ←
                    stringify!(#name) ←
                } ←
            } ←
            Преобразуем вывод ←
            quote в поток токенов, ←
            предоставляемый крейтом quote ←
    }; ←
    TokenStream::from(expanded) ←
}

fn add_trait_bounds(mut generics: Generics) -> Generics {
    for param in &mut generics.params {
        if let GenericParam::Type(ref mut type_param) = *param {
            type_param.bounds.push( ←
                parse_quote!(print_name::PrintName) ←
            );
        }
    }
    generics
}
```

Annotations for Listing 8.9:

- Преобразуем поток входных токенов в синтаксическое дерево, используя `parse_macro_input!` из крейта `syn`**: Points to the first line of the macro implementation.
- Добавляем необходимые ограничения трейтов, только если есть обобщенные параметры**: Points to the `add_trait_bounds` call.
- Разделяем обобщенные условия на части: `impl_generics`, `type_generics` и `where_clause`**: Points to the `generics.split_for_impl()` call.
- «Цитируем» фактическую реализацию трейта со всеми необходимыми параметрами, включая ограничения. `quote!`, предоставляемый крейтом `quote`, преобразует синтаксис Rust в `TokenTree`**: Points to the `quote!` block within the `impl` block.
- Превращаем в строку имя типа, к которому применяем производный макрос. `#name` отражает значение переменной `name` в «цитируемом» блоке**: Points to the `#name` placeholder in the `impl` block.
- Добавляем ограничения трейта `PrintName` ко всем общим параметрам для целевого типа**: Points to the `parse_quote!` call within the `add_trait_bounds` function.

В этом листинге мы специально добавили ограничения трейтов, но они не обязательны для работы производного макроса. Теперь объединим все и проверим наш код, используя небольшой интеграционный тест:

```
use print_name::PrintName;
use print_name_derive::PrintName;

#[test]
fn test_derive() {
    #[derive(PrintName)]
    struct MyStruct;

    assert_eq!(MyStruct::name(), "MyStruct");
    MyStruct::print_name();
}
```

Выполним `cargo expand --test test_derive` из каталога `print_name_derive`, чтобы проверить вывод нашего макроса:

```
fn test_derive() {
    struct MyStruct;
    impl print_name::PrintName for MyStruct {
        fn name() -> &'static str {
            "MyStruct"
        }
    }
    // ... опущено для краткости ...
}
```

Прекрасно! Но можно создать куда более сложную реализацию процедурных макросов, особенно при обработке параметров атрибутов или отдельных атрибутов полей.

**СОВЕТ** Помимо ознакомления с фрагментом, представленным в книге, изучите документацию к `syn`, доступную по ссылке <https://docs.rs/syn/latest/syn>, и официальную документацию Rust, которую можно найти по адресу <https://mng.bz/v8gx>. Это позволит вам лучше понять реализацию процедурных макросов. Кроме того, издательство Manning Publications выпустило замечательную книгу *Write Powerful Rust Macros* Сэма Ван Овермайера (Sam Van Overmeire) (<https://www.manning.com/books/write-powerful-rust-macros>). Реальные же примеры применения процедурных макросов ищите в крейте `rocket`, где они активно используются в реализации его веб-фреймворка для Rust (<https://crates.io/crates/rocket>).

И напоследок добавлю, что процедурные макросы создают много сложностей. Они невероятно эффективны, но у этого достоинства есть и обратная сторона. Такие макросы сложно отлаживать, и они негигиеничны, то есть могут засорять пространства имён, в которых используются, или конфликтовать с ними. Процедурный макрос просто выводит код, который внедряется перед компиляцией, поэтому разработчику нужно быть внимательным, чтобы не создать конфликтов и не засорить пространство имён.

## 8.4. Прелюдии

Последней темой этой главы станут *прелюдии* — коллекции полезных типов, функций и макросов, предназначенные для импорта в ваш код. Наличие прелюдий при написании библиотек облегчает пользователям работу с этими библиотеками.

Некоторые прелюдии, которые есть в самом Rust, импортируются автоматически, например, из стандартной библиотеки. Но я буду говорить конкретно о добавлении прелюдий в крейты, а не об использовании тех, которые предоставляет Rust.

**СОВЕТ** Подробнее о прелюдиях в Rust — в документации по ссылке <https://mng.bz/4JdB>.

Одна из причин использования прелюдий при написании библиотек состоит в том, что бывает сложно понять, какие символы нужно импортировать. Если забыть импортировать, например, трейт, то код может отказаться компилироваться или нужная функциональность будет отсутствовать. Выяснить при этом причину проблемы бывает трудно. Прелюдии реализуются с помощью реэкспорта, то есть обратного экспорта символов, полученных из другого модуля или крейта.

Но прежде, чем переходить к реализации прелюдий, поговорим о `use`. К этому моменту вы уже встречали в книге импорты наподобие такого:

```
use std::cell::RefCell;
use std::marker::PhantomData;
use std::rc::Rc;
```

По факту мы можем реэкспортировать с помощью оператора `use` что угодно, добавив ключевое слово `pub`, как делаем в случае любого другого типа или функции:

```
pub use std::cell::RefCell;
pub use std::marker::PhantomData;
pub use std::rc::Rc;
```

При обратном экспорте с помощью `pub use ...`; символы, импортируемые `use`, могут поступать извне модуля, хотя делать так с типами из стандартной библиотеки мы не будем. Кроме того, важно помнить, что для импорта всех символов, экспортируемых любым модулем, можно использовать вместе с этими символами синтаксис подстановки (\*):

```
use mylib::*;


```

Этот синтаксис приведет к импорту всего, что экспортируется из верхнего модуля крейта `mylib`. В крейтах многих библиотек есть явный модуль прелюдии (обычно с именем `prelude`), и импортировать его можно так:

```
use mylib::prelude::*;


```

Использование отдельного модуля прелюдии — один из способов избежать засорения пространства имен. Разберем пример с реализацией трейта прелюдии

на случай, если вы решите сделать это в своей библиотеке. Предположим, у вас есть крейт со следующей структурой:

```
$ tree
.
├── Cargo.lock
├── Cargo.toml
└── src
    ├── a.rs
    ├── b.rs
    └── lib.rs

1 directory, 5 files
```

В этой небольшой библиотеке есть модули `a` и `b`. Для начала взглянем на листинг 8.10.

#### Листинг 8.10. Код `lib.rs` без прелюдий

```
pub mod a; ← pub mod означает, что эти модули
pub mod b;   доступны публично (вне крейта)
```

```
pub struct TopLevelStruct {}
```

Внутри `a.rs` и `b.rs` мы создали пустые публичные структуры: `InnerA` и `InnerB` соответственно. Находясь вне крейта, можно импортировать эти структуры из `a` и `b` следующим кодом.

```
use mylib::a::InnerA;
use mylib::b::InnerB;

// InnerA и InnerB теперь находятся внутри области видимости
```

Но прелюдию мы еще не создали. Прелюдией будет один модуль для всего крейта, экспортирующий все самые полезные структуры (в данном случае `TopLevelStruct`, `InnerA` и `InnerB`). Создадим модуль с именем `prelude`. Вот структура нового крейта:

```
$ tree
.
├── Cargo.lock
├── Cargo.toml
└── src
    ├── a.rs
    ├── b.rs
    ├── lib.rs
    └── prelude.rs

1 directory, 6 files
```

Заполните модуль `prelude` кодом из листинга 8.11.

#### Листинг 8.11. Код `prelude.rs`

```
pub use crate::a::InnerA;
pub use crate::b::InnerB;
pub use crate::TopLevelStruct;
```

Этот код реэкспортирует все структуры из вашего крейта в одно место. Чтобы добавить в крейт `prelude.rs`, нужно добавить `pub mod prelude;` в `lib.rs`. Впоследствии пользователь крейта сможет импортировать все три эти структуры с помощью синтаксиса подстановки:

```
use mylib::prelude::*;

// InnerA, InnerB и TopLevelStruct теперь в области видимости
```

С операторами `use` можно использовать псевдонимы, а также организовывать реэкспорт. При желании вы можете реэкспортировать `TopLevelStruct` из прелюдии под другим именем:

```
pub use crate::TopLevelStruct as AltStruct;
```

**ПРЕДУПРЕЖДЕНИЕ** Рекомендую с осторожностью использовать эту возможность. Она полезна в основном тогда, когда вы хотите использовать в качестве символа другое внутреннее имя вместо внешнего.

Для реэкспорта символов не обязательно предоставлять отдельный модуль прелюдии именно таким образом. Просто в Rust это распространенный паттерн. Символы можно реэкспортировать из любого публичного модуля, в том числе из тех, которые находятся вне вашего крейта. Разработчики часто экспортируют зависимости из сторонних крейтов, чтобы упростить пользователям работу с их крейтами.

Будьте внимательны: прелюдии удобны, но могут запутывать, так как скрывают часть сложности за счет загрязнения пространства имен. Не злоупотребляйте ими.

Если вы только начинаете изучать Rust и хотите начать публиковать собственные библиотеки, то при использовании прелюдий соблюдайте умеренность. По мере накопления опыта вы научитесь понимать, где они действительно нужны. В качестве общего правила имейте в виду, что они требуются только в том случае, если ваш крейт предоставляет в рамках своей ключевой функциональности множество трейтов.

## **Резюме**

- Совмещая полученные знания о дженериках и трейтах, можно создавать поверх системы типов Rust абстракции наподобие конечных автоматов.
- Корутины — экспериментальная функциональность Rust, похожая на генераторы Python. Она предоставляет альтернативный способ выражения функций с возможностью приостановки для выдачи данных (`yield`).
- Процедурные макросы позволяют реализовывать расширения языка и заниматься метапрограммированием, значительно превосходя возможности декларативных макросов.
- Чтобы сделать библиотеки более удобными для пользователя, можно при их создании предоставить прелюдии, экспортав наиболее полезные части библиотеки в один модуль.

## *Часть IV*

# *Предотвращение проблем и создание надежного ПО*

В заключительной части книги мы будем меньше говорить о том, какие паттерны желательно использовать, и сосредоточимся на тех, которых лучше избегать. Иногда стоит немного пожертвовать производительностью или эффективностью потребления памяти, чтобы создать программное обеспечение, в котором все правильно, которое легко понимать и поддерживать.

К счастью, в случае Rust в большинстве сценариев производительностью жертвовать не нужно. Некоторые специалисты утверждают, что у этого языка нет реальных конкурентов в плане безопасности и быстродействия, поэтому разработчикам на Rust лишь изредка приходится жертвовать скоростью ради обеспечения корректности работы ПО.

Вы могли заметить, что труднее всего отлаживать именно мудреный код (который также часто становится причиной багов). Поэтому в последней части книги мы будем говорить о том, как избегать избыточно сложных или непонятных паттернов.

# 9

## Неизменяемость

### В этой главе

- ✓ Преимущества неизменяемости.
- ✓ Принцип неизменяемых данных и механизм их работы в Rust.
- ✓ Обеспечение неизменяемости с помощью трейтов.
- ✓ Крейты, предоставляющие неизменяемые структуры данных.

*Неизменяемость (иммутабельность)* — мощный принцип, который помогает создавать более качественное программное обеспечение. В контексте написания ПО неизменяемость означает, что после объявления и присваивания значения его нельзя изменять. В свою очередь, *изменяемость (мутабельность)* подразумевает, что значение может быть изменено после его объявления.

В Rust неизменяемость — важный, но при этом один из самых нелюбимых и недооцененных паттернов. Я же считаю его настолько ценным, что решил посвятить ему отдельную главу, хотя на деле он заслуживает целой книги. К сожалению, в одной главе не получится раскрыть эту тему так, как мне хотелось бы, но эта глава послужит прекрасной отправной точкой для дальнейшего изучения данного паттерна.

В Rust все объявленные переменные по умолчанию являются неизменяемыми и изменяемыми их нужно делать явно. Однако в более сложных структурах данных обработку изменяемости и неизменяемости нужно продумывать еще тщательнее. В некоторых языках понятие неизменяемости возведено в крайность и какие-либо изменения не допускаются в принципе. Но в Rust (к лучшему

или худшему) это решение остается за разработчиками. Во многих языках программирования и библиотеках неизменяемость присутствует в качестве одной из главных особенностей, но в Rust используется более прагматичный подход: разработчики могут сами решать, где и когда использовать этот механизм.

В текущей главе я расскажу о преимуществах отказа от изменяемых данных, объясню, какие проблемы могут возникнуть при попытке использовать структуры данных в изменяемой форме, проанализирую подход Rust к этому принципу и покажу, как с помощью возможностей этого языка сделать неизменяемыми практически любые стандартные данные. В завершение я опишу ряд крейтов, предоставляющих неизменяемые структуры данных, в том числе некоторые оптимизации.

## 9.1. Преимущества неизменяемости

Если вы не работали с языками или библиотеками, ориентированными на неизменяемость, то этот принцип может показаться вам чуждым. Разработчики часто поначалу воспринимают его скептически, но будет полезно познакомиться с ним поближе. Чтобы вам было проще понять этот принцип, я перечислю категории задач, которые он помогает решить, и покажу, как именно он это делает. Большинство программных ошибок относятся к одной из следующих категорий (список неполный).

- *Логические ошибки* — ошибки, недопонимание или упущения в коде, которые ведут к его некорректному поведению. Пример — бизнес-логика в программе, которая вычисляет налоги на покупку и ошибочно применяет не ту ставку.
- *Состояние гонки* — баги, возникающие, когда совместно используемые данные недостаточно синхронизированы. Такая ситуация может привести к повреждению данных, взаимным блокировкам и другим проблемам. Состояние гонки чаще всего возникает из-за одновременного доступа к общим изменяемым данным, например, когда несколько потоков пытаются изменить одни и те же данные одновременно или не в том порядке.
- *Неожиданные побочные эффекты* — непреднамеренные изменения в состоянии программы, которые возникают при выполнении функции или метода, ведущем к неожиданному поведению или появлению трудноуловимых багов. Побочные эффекты зачастую вызываются функциями, которые изменяют свои аргументы или глобальное состояние либо же зависят от него. Любые операции, связанные с вводом/выводом, такие как чтение из файла или запись в него, тоже подвержены побочным эффектам.
- *Проблемы с безопасностью памяти* — баги, возникающие при попытке программы обратиться к той области памяти, к которой она обращаться не должна. К примерам относятся попытки обратиться к памяти, которая уже освобождена, выходит за границы структуры данных или к которой у программы нет прав доступа. Такие баги могут вести к сбоям, повреждению данных и, что самое страшное, к уязвимостям в безопасности.

Неизменяемость во многих случаях помогает решить все эти проблемы. Если говорить о логических ошибках, то применение этого принципа упрощает понимание того, как в программе изменяются данные. Когда данные неизменяемы, то вы можете быть уверены, что они не изменятся неожиданно, а значит, вам будет проще понимать и прогнозировать поведение кода.

Состояние гонки возникает только в программах, имеющих совместно используемое общее состояние. Неизменяемость в этом случае выступает гарантом спокойствия, так как исключает возможность изменения совместно используемых данных. Общее состояние все равно может потребоваться, но если это состояние никогда не меняется, то и волноваться о возникновении гонки не придется.

Побочные эффекты возникают, только когда данные изменяются. Если мы пишем функцию, не имеющую побочных эффектов, то называем ее *чистой*. Такие функции более понятны, легче тестируются, и их удобнее использовать повторно. Когда мы пишем код без побочных эффектов, то обычно называем его *чисто функциональным*.

**ПРИМЕЧАНИЕ** Если вы не знакомы с функциональным программированием, то оно поначалу может показаться сложным. Но, как и во многих других случаях, практика все упрощает. Когда вы начнете писать чисто функциональный код, вы удивитесь тому, как вообще раньше обходились без него.

Чистые функции обладают одним полезным свойством, а именно являются *ссыльно прозрачными* (*referentially transparent*). Это означает, что вы можете заменить вызов функции результатом выполнения этой функции, и на поведение программы это не повлияет. Иначе взглянуть на этот принцип можно так: для любого конкретного набора входных данных вывод функции всегда будет одинаковым. Ссыльная прозрачность позволяет производить оптимизации и рефакторинг, который был бы невозможен в случае нечистых функций. Разумеется, намного проще анализировать (и тестировать) код, зная, что функция при одинаковом наборе входных данных всегда будет возвращать одинаковый результат. Если ПО будет полностью лишено побочных эффектов, то вся программа может стать детерминированной. И это важная особенность, которая дает множество преимуществ при тестировании и отладке.

Наконец, неизменяемость помогает предотвратить проблемы с потреблением памяти. Нередко бывает, что такие проблемы возникают при появлении неожиданных изменений, например, в граничных случаях. И даже когда мы пишем тесты для всех известных нам случаев, могут остаться такие, которые мы не предвидим. В подобных ситуациях помогают такие стратегии, как тестирование свойств и фаззинг, но они не могут охватить все возможные граничные сценарии.

Если объединить все эти проблемы, то становится очевидно, почему неизменяемость особенно полезна в системах с реализацией параллельности или конкурентности, где зачастую их нужно решать одновременно. Некоторые языки

программирования, библиотеки и фреймворки внедрили неизменяемость в качестве основного принципа. Самые яркие примеры — Erlang, Elixir, Haskell, Clojure и Elm. Эти языки известны своей надежностью и понятным кодом. Вдобавок они достаточно популярны в областях, где надежность играет первостепенную роль, например в телекоммуникациях, финансовой сфере, здравоохранении, а также в аэрокосмической отрасли.

Использование неизменяемости поддерживают и некоторые известные библиотеки и фреймворки, в том числе Redux, популярная библиотека для управления состоянием в JavaScript и TypeScript. Основанная на принципах функционального программирования и неизменяемости, Redux известна своей надежностью и понятностью. React, очень популярная библиотека для создания интерфейсов, также поддерживает неизменяемость и в последних версиях делает акцент на написании чисто функциональных компонентов. В свою очередь, популярные библиотеки Immutable.js и Lodash для JavaScript и TypeScript тоже предоставляют инструменты для работы с неизменяемыми данными.

Все эти примеры оказали свое влияние на проектирование Rust и некоторых популярных библиотек для него. Если вы сталкивались с этими языками ранее, то можете замечать некоторые их сходства с Rust.

## **9.2. Почему неизменяемость не панацея**

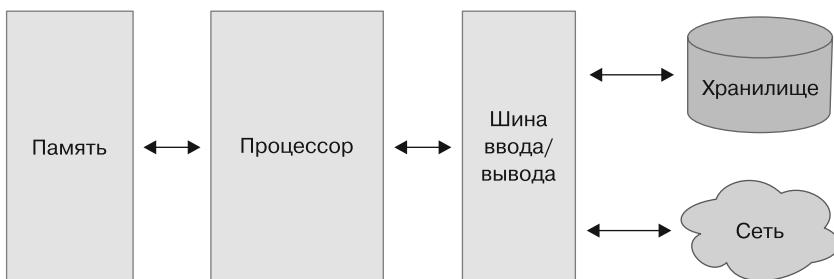
Неизменяемость не дается даром. За нее надо платить. И самой очевидной платой является то, что для изменения данных приходится их часто повторять, а это не только требует затрат ресурсов памяти и процессорного времени, но и усложняет код. Неизменяемость затратна еще и в том смысле, что больше времени приходится тратить на предварительное обдумывание структуры данных и программ.

Rust стремится минимизировать все эти сопутствующие затраты, предоставляя разработчикам возможность при необходимости использовать изменяемые данные. Тем не менее в его стандартной библиотеке никакие паттерны неизменяемости в строгой форме не заложены. Основные структуры данных Rust (такие как `Vec`) похожи на векторы C++ или массивы C, где изначально предполагается использование именно изменяемых структур данных. Этим Rust отличается от языков, в которых изменения запрещены, таких как Erlang, Elixir, Haskell, Clojure и Elm.

Некоторые языки, например Scala, предоставляют некий компромиссный вариант, позволяя выбирать между использованием изменяемых и неизменяемых структур данных, хотя в Rust по умолчанию предоставляются только изменяемые структуры. Это решение, намеренно или нет, было принято в отношении стандартной библиотеки, чтобы облегчить работу специалистов, приходящих из языков наподобие C и C++, а также чтобы сделать процесс написания производительного кода более удобным.

### 9.3. Как воспринимать неизменяемые данные

Понятие неизменяемости в ее общей форме в некотором смысле противоречит нашему пониманию данных и, в частности, механизму их обработки компьютерами. Практически все вычислительные устройства, начиная с миниатюрных смартфонов и заканчивая масштабными суперкомпьютерами, созданы для обработки изменяемых данных и основаны на архитектуре фон Неймана (von Neumann) (рис. 9.1).



**Рис. 9.1.** Архитектура фон Неймана

Эта архитектура лежит в основе практически всех современных компьютеров. Она состоит из процессора, который сохраняет данные и инструкции программ на одном запоминающем устройстве. В результате память, где данные сохраняются для быстрого доступа, является ограниченной и по своей природе изменяемой. Это отличает ее от прочих систем хранения, таких как диски или магнитная лента. Ввиду таких неизбежных ограничений памяти мы вынуждены повторно использовать одну и ту же память для разных целей, из чего и проистекает принцип изменяемых данных.

Если бы у нас была гипотетическая система с возможностью бесконечного расширения, куда можно было бы только добавлять данные (например, бесконечная магнитная лента, аналогичная машине Тьюринга), то мы могли бы создать полностью неизменяемую структуру. Представьте теоретический компьютер, использующий в качестве памяти бесконечную ленту, на которую можно записывать данные только раз. В такой модели можно будет считывать данные с ленты и записывать их на нее, но без возможности изменения. Естественно, реальные компьютеры работают не так.

В связи с этим неизменяемость в контексте программирования — лишь абстракция, существующая только на относительно высоком уровне. Она не заложена в аппаратную архитектуру или операционную систему. Это принцип, который привносится языком программирования, библиотеками и фреймворками, которые мы используем.

В случае Rust модуль проверки заимствований помогает отслеживать изменяемые и неизменяемые части программы. Если же речь идет о масштабном

программном обеспечении, то именно разработчикам нужно решать, как лучше обрабатывать данные. И это верно даже для языков, которые неизменяемы по своей природе, поскольку любой программе для того, чтобы быть полезной, нужно взаимодействовать с внешним миром, а он изменяется, конечен и наполнен состояниями.

Чтобы лучше понять принцип неизменяемых данных, нужно рассматривать их в разрезе владения, заимствования и того, какие меры вы готовы принять ради обеспечения неизменности. Думаю, большинство разработчиков найдут правильный баланс, взаимодействуя с языком и экспериментируя с различными паттернами и подходами.

Помимо вышесказанного, важно помнить, что неизменяемость не категоричный принцип. Вы можете сделать одни части программы изменяемыми или чисто функциональными, а другие — нет. Вы можете проявить строгость в отношении наиболее критических компонентов и, напротив, дать большую свободу в случаях, когда высокая производительность дороже корректности.

Именно вам решать, как лучше для вашей программы, и понимание этого будет вырабатываться с опытом. Да, не каждому захочется ждать, но такова реальность при наработке любого навыка, будь то игра на музыкальном инструменте или работа с компьютерами и программным обеспечением. Это нужно просто принять.

## **9.4. Принцип неизменяемости в Rust**

В Rust все объявленные переменные по умолчанию неизменяемы. Единственное исключение возникает при использовании ключевого слова `unsafe` для обхода защитных механизмов языка. Если вы хотите сделать переменную изменяемой, то вам потребуется объявить ее таковой с помощью ключевого слова `mut`. Причем этот принцип работает каскадно, то есть данные, объявленные в неизменяемой структуре, также являются неизменяемыми. Это называется *наследственной изменяемостью*.

Исключение из правила изменяемости возникает при использовании изменяемых контейнеров `Cell`, `RefCell` и `OnceCell`, к которым можно получить общий доступ. Эти контейнеры допускают *внутреннюю изменяемость*, которая позволяет изменять данные внутри неизменяемого контейнера. И эта возможность не только оказывает важное влияние, но и позволяет создавать мощные паттерны. Но здесь важно понимать сопутствующие компромиссы. Использование изменяемых контейнеров с возможностью общего доступа имеет одно свойство: в некоторых обстоятельствах они допускают скрытые изменения, а это не всегда желательно, хотя в большинстве ситуаций проблемой не является.

Единственное практическое различие между наследуемой и внутренней изменяемостью заключается в том, что первая обеспечивается компилятором, а вторая накладывается в среде выполнения. В качестве общего правила можно рассматривать значения внутри `Cell`, `RefCell` или `OnceCell` как *потенциально изменяемые*,

даже если сам контейнер иммутабелен. Иными словами, эти контейнеры дают возможность изменять их внутренние данные, но не требуют для этого изменений.

И `Cell` в этом плане более прямолинеен, нежели `RefCell` и `OnceCell`, так как позволяет вносить изменения только путем замены. Нельзя изменить значение в `Cell` напрямую, но можно его заменить. Это тонкий, но важный нюанс, и он является одной из причин, по которым `Cell` считается более безопасным при реализации внутренней изменяемости. `RefCell` и `OnceCell` позволяют получать мутабельную ссылку на хранящиеся в них данные.

В стандартной библиотеке Rust нет структур данных, предполагающих неизменяемость. Обычно это стандартные структуры, ведущие себя вполне ожидаемым образом. `Vec` и `HashMap` предоставляют различные методы для мутабельного доступа и не предлагают особых возможностей для иммутабельной работы с данными, кроме реализации трейта `Clone`. В будущем положение дел может измениться, ну а пока нужно работать с тем, что есть. Учитывая, что в Rust особое внимание уделяется производительности, зачастую более эффективной оказывается работа с изменяемыми структурами данных. Основной паттерн для реализации неизменяемости в языках наподобие Rust, которые не предоставляют эту возможность в качестве основной, состоит из двух шагов:

- после объявления и присваивания значения его не нужно изменять на месте;
- чтобы изменить значение, вы его копируете и затем меняете уже копию (что, по сути, все равно является изменением, но уже нового значения, а не оригинала).

В языках, где неизменяемость заложена в основе, процесс копирования и изменения выполняется абстрагированно, но, по существу, операция аналогична. Мы можем использовать абстракции для скрытия деталей этого процесса, и некоторые из таких приемов мы разберем в разделах 9.6–9.8.

## 9.5. Анализ основ неизменяемости в Rust

Давайте проанализируем основы неизменяемости в Rust на примерах кода. Этот процесс может показаться тривиальным, однако будет полезно разбирать сопутствующие вопросы по ходу освещения темы, чтобы вы могли понимать причины тех или иных наших действий.

Неизменяемая (иммутабельная) операция обычно подразумевает присваивание результата вычислений новому значению. Предположим, нам нужно инкрементировать значение переменной `x` иммутабельно, для чего мы объявим новую переменную `y` с помощью `let y = x + 1`. В случае изменяемой операции значение `x` менялось бы напрямую через `x += 1`. В нашем примере `x += 1` представляет более короткий и эффективный способ инкрементирования `x`. Но он при этом более уязвим к ошибкам и затрудняет понимание происходящего в больших кодовых базах или сложных сценариях.

Кроме того, Rust позволяет затенять переменную, то есть объявлять новую и присваивать ей значение, используя имя существующей. Это распространенный паттерн в Rust, который часто используется для преобразования изменяемой переменной в неизменяемую. В листинге 9.1 показано, как можно выполнить такое затенение.

### Листинг 9.1. Основы неизменяемости в Rust

```
let x = 1;
dbg!(x);
let y = x + 1; // y = 2
dbg!(y);
// x += 1; ← Если эту строку раскомментировать,
// то возникнет ошибка компиляции
// ошибка: нельзя дважды присвоить значение неизменяемой переменной `x`
```

```
let mut x = x; // x = 1 ← Затеняет x изменяемой
x += 1; // x = 2
dbg!(x);
```

Вот вывод этого кода:

```
[src/main.rs:14] x = 1
[src/main.rs:16] y = 2
[src/main.rs:22] x = 2
```

Углубляясь в тему, важно отметить, что семантика изменяемости в Rust также применяется в вызовах функций, но с одним небольшим различием. Занятые (owned) значения можно переводить из неизменяемых в изменяемые при перемещении, но вызывающий функцию код к этому никак не причастен (листинг 9.2).

### Листинг 9.2. Изменяемость в вызовах функций

```
fn mutability(
    a: i32, // Неизменяется ← Переменная a перемещается
    mut b: i32, // Изменяется ← в функцию и является неизменяемой
) {
    // a += 1; // ошибка: нельзя дважды присвоить значение
    // неизменяемой переменной `a`
    b += 1;

    dbg!(a);
    dbg!(b);
}
```

Функцию `mutability()` можно вызвать так:

```
let a = 1;
let b = 2; ← Переменная b неизменяется, но перемещается
mutability(a, b); в функцию как изменяемая
```

Вывод будет таким:

```
[src/main.rs:8] a = 1
[src/main.rs:9] b = 3
```

Обратите внимание: в этом примере мы превращаем переменную `b` из неизменяемой в изменяемую, когда передаем ее в функцию `mutability()`. Для этого мы просто применяем к аргументу ключевое слово `mut`. В некоторых случаях такое действие может запутать вызывающий функцию код, но поскольку владение передается (`move`) функции, то данная функция может делать со значением все, что захочет. Несмотря на то что мы изменяем аспект изменяемости `b`, это не влияет на исходную переменную `b` в области действия вызывающего кода, так что данный паттерн не представляет угрозы.

Невозможно поправить изменяемость ссылки так же, как это делается с занятymi значениями при вызове функции. Ссылки были заимствованы, а менять аспект изменяемости у заимствованных данных нельзя (в этом весь смысл модуля проверки заимствования Rust).

**ПРИМЕЧАНИЕ** Запомните еще один важный момент, связанный с передачей аргументов по значению: в случае доступности трейта `Copy` компилятор будет его использовать. `Copy` отличается от `Clone` тем, что является трейтом-маркером, указывающим компилятору, что тип можно продублировать путем копирования его элементов в памяти. При этом трейт `Clone` предоставляет метод для явного копирования значения. Суть в том, что любое значение, реализующее `Copy`, при передаче в функцию или присваивании новой переменной будет копироваться, а не переноситься. В большинстве случаев это касается только примитивных типов наподобие целых чисел, чисел с плавающей запятой и логических значений, но может применяться и к кортежам, массивам и структурам, которые содержат только типы `Copy`.

Для реализации внутренней изменяемости также можно использовать `RefCell` (листинг 9.3).

### Листинг 9.3. Использование `RefCell` для обеспечения внутренней изменяемости

```
let immutable_string =
    String::from("This string cannot be changed");           | Объявляем
// immutable_string.push_str("... or can it?"); // ошибка: нельзя заимствовать
`immutable_string` as mutable, as it is not declared as mutable ←
dbg!(&immutable_string);                                     | Если эту строку раскомментировать,
                                                               | то возникнет ошибка компиляции
```

```
let not_so_immutable_string = RefCell::from(immutable_string);
not_so_immutable_string
    .borrow_mut()
    .push_str("... or can it?"); ←                                | Теперь можно изменять
                                                               | строку внутри RefCell
dbg!(&not_so_immutable_string);
```

Теперь можно изменять строку внутри `RefCell`

Создаем `RefCell` и перемещаем в нее неизменяемую строку.  
Обратите внимание, что `RefCell` не объявляется как изменяемая

При выполнении этого кода мы получим такой вывод:

```
[src/main.rs:32] &immutable_string = "This string cannot be changed"
[src/main.rs:38] &not_so_immutable_string = RefCell {
    value: "This string cannot be changed... or can it?",
}
```

Как видите, совместно используемые изменяемые контейнеры в Rust предоставляют возможности для обхода правил изменяемости. Здесь важно помнить, что `RefCell` владеет своими данными. В примере выше мы переместили строку в `RefCell`, для чего потребовался простой вызов функции, позволяющий изменять критерий изменяемости любого занятого значения.

## **9.6. Обеспечение неизменяемости с помощью трейтов**

Мы разобрали некоторые преимущества и недостатки неизменяемости, но еще нужно понять, как применять ее на практике. В стандартной библиотеке Rust для этого есть несколько инструментов. В текущем разделе мы обсудим трейт `std::borrow::ToOwned`, который выступает основой паттерна, позволяющего сделать неизменяемым почти все что угодно.

При работе с неизменяемыми данными нужно избегать создания их лишних копий. Для этого мы используем ссылки, позволяющие заимствовать эти данные. Вы могли встретить код на Rust, похожий на этот:

```
let s = "A static string".to_owned();
```

В этом коде для преобразования `&str` в `String` используется трейт `ToOwned`. Rust предоставляет blanket-реализацию для любого типа `T`, где `T` реализует `Clone`. То есть `ToOwned` можно рассматривать как обобщение `Clone` для ссылок или срезов. Реализация `ToOwned::to_owned()` просто вызывает `Clone::clone()`, а для `[T]` возвращает `Vec` с клонированными элементами. В листинге 9.4 показано определение трейта `ToOwned`.

### **Листинг 9.4. Определение `ToOwned` из стандартной библиотеки Rust**

```
pub trait ToOwned {
    type Owned: Borrow<Self>;

    // Необходимый метод
    fn to_owned(&self) -> Self::Owned;

    // Предоставляемый метод
    fn clone_into(&self, target: &mut Self::Owned) { ... }
}
```

Располагая этим определением, мы без труда сможем использовать неизменяемые данные повсюду. Будет достаточно предоставить реализацию `Clone`, после чего использовать `ToOwned`, чтобы преобразовать данные в занятое значение, когда нам нужно будет их изменить. Этот процесс может показаться немного громоздким, но можно повысить его эргономичность с помощью типа `Cow`.

## 9.7. Использование типа `Cow` для обеспечения неизменяемости

Тип `Cow` — это умный указатель, реализующий паттерн клонирования при записи. Сам по себе он реализуется как перечисление, в связи с чем его содержимое должно реализовывать трейт `ToOwned`. Вы наверняка слышали о механизме копирования при записи, и `Cow` следует тому же принципу, откладывая клонирование до момента необходимости. Этот тип можно использовать в качестве контейнера для данных, которые нужно сделать неизменяемыми. Для этого следует производить операции изменения только над клонированными данными, но не исходными. Если все сделать правильно, то этот подход не исключит изменяемость полностью, но защитит разработчика от необходимости изменять исходные данные или использовать изменяемые ссылки. В листинге 9.5 показано определение типа `Cow`.

### Листинг 9.5. Определение `Cow` из стандартной библиотеки Rust

```
pub enum Cow<'a, B>
where
    B: 'a + ToOwned + ?Sized,
{
    Borrowed(&'a B),
    Owned(<B as ToOwned>::Owned),
}
```

Обратите внимание на некоторые нюансы этого типа.

- `Cow` — обобщение для типа `B` и его ссылок с временем жизни `'a`. При этом `B` должен реализовывать трейт `ToOwned`.
- `Cow` — перечисление с двумя вариантами: `Borrowed` и `Owned`. Он похож на `Option`, но более специализированный.
- `Cow` можно использовать для обертывания любой ссылки на тип, реализующий `Clone` (а значит, и `ToOwned`), после чего получить занятое значение, когда потребуется его изменить.
- `Cow` реализует трейт `Deref`, который позволяет рассматривать его как ссылку на содержащиеся в нем данные.

В листинге 9.6 показан простой пример использования типа Cow.

### Листинг 9.6. Простой случай применения Cow

```
use std::borrow::Cow;

let cow_say_what = Cow::from("The cow goes moo");
dbg!(&cow_say_what);

let cows_dont_say_what =
    cow_say_what
        .clone()           | Можно изменять клонированные данные,
        .to_mut()          | не влияя на исходные. Но обратите внимание,
        .replace("moo", "toot"); | что для получения изменяемой ссылки нужно
dbg!(&cow_say_what);   | клонировать Cow и затем вызвать to_mut()
dbg!(&cows_dont_say_what); | Исходные данные по-прежнему неизменямы
                           | Клонированные данные
                           | были изменены
```

Обратите внимание, что для получения нового Cow все так же нужно вызывать `clone()`, направленный на сам умный указатель, а не содержащиеся в нем данные. Затем мы вызываем `to_mut()`, чтобы получить изменяемую ссылку на внутренние данные после их клонирования. При выполнении этого кода мы получим такой вывод:

```
[src/main.rs:5:5] &cow_say_what = "The cow goes moo"
[src/main.rs:9:5] &cow_say_what = "The cow goes moo"
[src/main.rs:10:5] &cows_dont_say_what = "The cow goes toot"
```

Теперь доработаем пример, чтобы приблизить его к реальному случаю использования. Напишем функцию, выполняющую нечто похожее, то есть возвращающую новый объект (листинг 9.7).

### Листинг 9.7. Доработка использования типа Cow

```
fn loud_moo<'a>(mut cow: Cow<'a, str>) | Функция получает занятый Cow
                                             | и возвращает Cow
    -> Cow<'a, str> {
        if cow.contains("moo") {
            Cow::from(cow.to_mut().replace("moo", "MOO"))
        } else {
            cow           | Если Cow содержит
        }               | moo, то мы изменяем
    }               | его и заменяем на MOO
}               | Если Cow не содержит moo,
                  | мы возвращаем исходный Cow
```

Функцию `loud_moo()` можно вызвать так:

```
let cow_say_what = Cow::from("The cow goes moo");
let yelling_cows = loud_moo(cow_say_what.clone());
dbg!(&cow_say_what);
dbg!(&yelling_cows);
```

Вывод этого кода выглядит так:

```
[src/main.rs:21:5] &cow_say_what = "The cow goes moo"
[src/main.rs:22:5] &yelling_cows = "The cow goes MOO"
```

Если мы используем `Cow`, то нам не обязательно раскрывать эту деталь реализации в публичном API, поэтому мы завернем данные в структуру. Поместим в нее `Cow` и предоставим метод, позволяющий изменять внутренние данные, не раскрывая сам этот тип (листинг 9.8).

### Листинг 9.8. Обертывание Cow в структуру

```
#[derive(Debug, Clone)]           ← Выводим Clone, чтобы иметь возможность клонировать
struct CowList<'a> {           CowList, в том числе внутренний список cow
    cows: Cow<'a, [String]>,   ← Используем Cow для обертывания вектора
}                                строк, а также времени жизни 'a

impl<'a> CowList<'a> {
    fn add_cow(&self, cow: &str) -> Self { ← Предоставляем метод для добавления Cow
        let mut new_cows = self.clone(); ← в список, возвращая новый CowList
        new_cows.cows.to_mut().push(   ← Сначала клонируем CowList, чтобы
            cow.to_string()          иметь возможность его изменять
        );
        new_cows               ← Изменяем внутренний Cow,
    }                           ← Возвращаем           вызывая to_mut() и затем push()
}                                новый CowList

impl Default for CowList<'_> {
    fn default() -> Self {
        CowList {
            cows: Cow::from(vec![]),
        }
    }
}
```

Теперь протестируем код:

```
let list_of_cows = CowList::default()
    .add_cow("Bessie")
    .add_cow("Daisy")
    .add_cow("Moo");
dbg!(&list_of_cows);
let list_of_cows_plus_one = list_of_cows.add_cow("Penelope");
dbg!(&list_of_cows);           ← Исходный CowList по-прежнему неизменяем,
dbg!(&list_of_cows_plus_one);  на что указывает его повторный вывод
```

Вывод этого кода выглядит так:

```
[src/main.rs:49:5] &list_of_cows = CowList {
    cows: [
        "Bessie",
        "Daisy",
        "Moo",
    ],
}
```

```
[src/main.rs:52:5] &list_of_cows = CowList {
    cows: [
        "Bessie",
        "Daisy",
        "Moo",
    ],
}
[src/main.rs:53:5] &list_of_cows_plus_one = CowList {
    cows: [
        "Bessie",
        "Daisy",
        "Moo",
        "Penelope",
    ],
}
```

В качестве альтернативной реализации можно поместить каждый Cow в Vec:

```
#[derive(Debug, Clone)]
struct CowVec<'a> {
    cows: Vec<Cow<'a, str>>,
}
```

Преимуществом этого метода (в противоположность использованию Vec внутри Cow) является то, что каждый элемент вектора можно клонировать отложенно. Такой подход называется *раскрытием структуры* (structural sharing). Он более эффективен при наличии множества копий одинаковых элементов, особенно когда среди них есть крупные.

Как видите, тип Cow нисколько не сложен, но его можно скрыть от публичного API, чтобы слегка упростить использование этого API. Обратите внимание: в примере выше при изменении Cowlist мы никогда не изменяем его исходный экземпляр, а всегда возвращаем новый CowList.

Тип Cow можно применять практически везде, где нужно обусловить использование неизменяемых данных. Но для этого надо понять, что он собой представляет и как работает. Если вы с ним еще не встречались, то он может показаться странной абстракцией, особенно когда вы можете вызывать clone() напрямую или изменять данные. Если вы считаете, что это довольно неудобный подход для работы с данными, то вы не одиноки. Так что следующий раздел будет посвящен структурам данных, которые облегчают реализацию механизма неизменяемости.

## **9.8. Получение неизменяемых структур данных с помощью крейтов**

В этом разделе вы познакомитесь с крейтами, которые предоставляют неизменяемые структуры данных, что позволяет разработчикам воспользоваться преимуществами этой концепции, не создавая собственные решения. Речь пойдет о двух крейтах:

- `im` — предоставляет списки, множества и ассоциативные массивы (<https://crates.io/crates/im>);

- rpds (Rust persistent data structures — постоянные структуры данных Rust) — предоставляет списки, множества, очереди и ассоциативные массивы (<https://crates.io/crates/rpds>).

Оба крейта содержат структуры, оптимизированные под использование в программах и библиотеках, ориентированных на неизменяемость. Но при этом их API также дают возможность использовать эти структуры изменяемым образом. Хотя оптимизированы они все же под неизменяемость.

### 9.8.1. Использование крейта `im`

Крейт `im` — наиболее популярная библиотека неизменяемых структур данных. Количество его скачиваний с <https://crates.io> уже превысило 18 миллионов, вдобавок он используется во многих других крейтах и проектах.

Этот крейт предоставляет структуру `Vector`, аналог `Vec` из стандартной библиотеки Rust. Кроме того, в нем есть упорядоченные и неупорядоченные множества и хеш-таблицы, ориентированные на неизменяемость. Далее мы используем `im` для создания `Vector` с помощью макроса `vector!` и добавим в этот вектор элементы (листинг 9.9).

#### Листинг 9.9. Использование `im` для создания `Vector`

```
use im::vector;

let shopping_list =
    vector!["milk", "bread", "butter", "cheese", "eggs"];
    ↑ Нужно клонировать
    исходный вектор

let mut updated_shopping_list = shopping_list.clone(); ←
updated_shopping_list.push_back("grapes"); ←
    ↑ Заметьте, что мы изменяем вектор
    как стандартный и добавляем в него
    элемент с возможностью изменения,
    используя push_back()
```

Стоит отметить, что `Vector` из `im` можно использовать изменяемым образом. Нет никакой нужды каждый раз делать копию вектора. Крейт `im` лишь предполагает использование структур данных изменяемым образом, но строго этого не требует. Помните, что предоставленная реализация `Clone` оптимизирована под случаи с неизменяемостью, поэтому можно свободно клонировать `Vector`, не беспокоясь о производительности. При выполнении приведенного выше кода мы получим такой вывод:

```
[src/main.rs:10:5] &shopping_list = [
    "milk",
    "bread",
    "butter",
    "cheese",
    "eggs",
]
```

```
[src/main.rs:11:5] &updated_shopping_list = [
    "milk",
    "bread",
    "butter",
    "cheese",
    "eggs",
    "grapes",
]
```

В добавок к структурам данных наподобие множеств и ассоциативных массивов (`map`) `im` предоставляет возможность использования итераторов на основе библиотеки `Rayon` (для параллельной итерации), поддержку `Serde`, а также возможность тестирования свойств с помощью `PropTest` и `QuickCheck`. Подробнее — в документации `im` по адресу <https://docs.rs/im/latest/im>.

### 9.8.2. Использование крейта `rpds`

Крейт `rpds` аналогичен `im`, но предоставляет несколько дополнительных структур данных, таких как очереди и стеки. И хотя `rpds` не так популярен, как `im` (около 6 миллионов скачиваний с <https://crates.io>), это весьма полезная библиотека, которая регулярно обслуживается.

В отличие от `im` крейт `rpds` предоставляет API для обеспечения неизменяемости данных напрямую (конкретно методы, возвращающие новую структуру), хотя в нем есть и API для работы с изменяемыми данными на случай, если вы захотите избежать клонирования. Далее мы используем `rpds` для создания `Vector` и добавления в него элементов (листинг 9.10).

#### Листинг 9.10. Использование `rpds` для создания `Vector`

```
use rpds::Vector;

let streets = Vector::new()
    .push_back("Elm Street")
    .push_back("Maple Street")
    .push_back("Oak Street");

let updated_streets = streets.push_back("Pine Street");

dbg!(&streets);
dbg!(&updated_streets);
```

Обратите внимание: в случае `rpds` каждый вызов `push_back()` возвращает новый `Vector`, поэтому вам не нужно клонировать его явно. В добавок он предоставляет `push_back_mut()`, позволяя изменять вектор на месте. При выполнении данного кода мы получим такой вывод:

```
[src/main.rs:11:5] &streets = Vector {
    root: Leaf(
        [
            "Elm Street",

```

```

        "Maple Street",
        "Oak Street",
    ],
),
bits: 5,
length: 3,
}
[src/main.rs:12:5] &updated_streets = Vector {
    root: Leaf(
        [
            "Elm Street",
            "Maple Street",
            "Oak Street",
            "Pine Street",
        ],
),
bits: 5,
length: 4,
}

```

Кроме того, крейт `rpds` обеспечивает поддержку Serde и макросов для инициализации структур данных. Подробности вы найдете в документации по адресу <https://docs.rs/rpds/latest/rpds>.

## Резюме

- Неизменяемость (иммутабельность) — мощная абстракция для написания надежного программного обеспечения.
- Неизменяемость помогает избежать логических ошибок, состояний гонки, нежелательных побочных эффектов и проблем с безопасным доступом к памяти.
- Неизменяемость можно совмещать с паттернами функционального программирования, такими как чистые функции и ссылочная прозрачность, делая код более надежным, понятным и удобным в тестировании.
- Rust всегда проводит различие между изменяемыми и неизменяемыми значениями, за чем следует модуль проверки заимствований. Благодаря принципу наследуемой изменяемости можно легко определять, является ли значение изменяемым.
- В Rust есть ряд инструментов, помогающих работать с неизменяемыми данными, — например, трейт `ToOwned` и тип `Cow`.
- Крейты `im` и `rpds` предоставляют структуры данных, оптимизированные под неизменяемость, которые можно использовать в качестве элементов программ и библиотек, работающих с неизменяемыми данными.

# 10

## *Антипаттерны*

---

### **В этой главе**

- ✓ Что такое антипаттерны программирования.
- ✓ Основные антипаттерны в Rust.
- ✓ Когда неоднозначные паттерны можно использовать, а когда их лучше избегать.

*Антипаттерны* — это практики программирования, которые считаются вредными в определенных контекстах или всегда. Зачастую они становятся результатом недопонимания языка или недостатка опыта работы с конкретным стеком технологий. В этой главе вы узнаете, что такое антипаттерн, и познакомитесь с некоторыми распространенными антипаттернами в Rust. Кроме того, мы обсудим, когда можно использовать, а когда лучше избегать конкретных паттернов, и какие случаи могут стать исключением.

Описанные в этой главе правила не являются жесткими. Всегда есть исключения. Но важно понимать логику, стоящую за правилами, чтобы знать, когда их можно нарушить. По мере развития Rust эти правила могут меняться, так что если ваша цель — писать максимально эффективный код на этом языке, то вам нужно следить за появлением новых рекомендуемых практик.

## 10.1. Что такое антипаттерн

Термин «антипаттерн» достаточно условный. Разработчики часто используют его в уничтожительном смысле, описывая некую практику, которую они не любят. В конечном итоге определение антипаттерна зависит от личного мнения и предпочтений специалиста. Однако в некоторых случаях та или иная практика оказывается бесспорно плохой, например, когда она небезопасна, неэффективна или усложняет сопровождение кода. Такие случаи возникают в результате комбинации неудачного дизайна, желания сохранить обратную совместимость и постоянно меняющегося ландшафта приемлемой структуры программного обеспечения.

Разработчики на С провели практическое исследование на тему развития практик проектирования. Этот язык, пожалуй, можно назвать самым влиятельным языком в истории программирования. Несмотря на его повсеместность, С — один из худших языков с точки зрения безопасности и простоты использования, особенно в сфере системного программирования. Даже опытные эксперты легко могут допустить в нем такие ошибки, которые будет сложно обнаружить и исправить.

Некоторые утверждают, что С по современным стандартам объективно плох, и здесь сложно не согласиться. Я провел немало часов за исправлением багов кода на С, которые легко допустить, но сложно обнаружить ввиду самой структуры этого языка. Написание кода на С может вызывать ностальгию, но делать это каждый день не хочется. Тем не менее этот язык и по сей день остается лучшим вариантом для создания многих приложений, особенно при программировании для встраиваемых систем или когда единственной удачной альтернативой является ассемблер.

В свою очередь, при создании Rust акцент ставился на том, чтобы избежать неудачных программных решений, имеющихся в языках наподобие С (не стрелять себе в ногу). Кроме того, разработчики Rust постарались лишить С его основного аргумента — высокой скорости выполнения, сделав Rust таким же быстрым, если не быстрее. Во многих тестах Rust превосходит С по чистой скорости и делает это, не используя небезопасный код.

Но даже Rust стал жертвой собственного успеха, так как его популярность затруднила внесение существенных изменений. Трудно обосновывать и вносить корректировки, нарушающие обратную совместимость, поскольку затраты на переписывание существующего кода настолько велики, что люди избегают обновления (проблема, преследовавшая С и другие языки не один десяток лет).

Но сравнивать Rust с С не совсем справедливо, поскольку его создание стало возможным только благодаря значительным улучшениям в инфраструктуре компилятора (а именно, проекта LLVM; <https://llvm.org>) и лучшему пониманию самих принципов проектирования языка программирования в сравнении с тем, которое имели инженеры в 1970-х. То, что мы называем антипаттерном сегодня,

в те годы могло считаться рекомендуемой практикой. В Rust хорошо то, что его компилятор очень многое проделывает за вас, если только вы избегаете применения блоков `unsafe`. Но то же самое не скажешь о языках наподобие C, где компилятор представляет собой довольно ограниченный инструмент и основная часть обязанностей по оптимизации кода лежит на программисте. (Хотя, по сути, и Clang, и GCC неплохо справляются с оптимизацией кода на C.)

Однако, несмотря на все вышеизложенное, тема антипаттернов заслуживает изучения, и я надеюсь, что по окончании этой главы вы получите более глубокое представление как о самом Rust, так и о его ограничениях. Ну и возвращаясь к вопросу «Что такое антипаттерн?», я определию его как любой паттерн, который не по душе.

## 10.2. Использование `unsafe`

Источник всех антипаттернов в Rust — неправильное использование опасного ключевого слова `unsafe`. Его приходится применять для реализации многих задач в этом языке, но в неумелых руках оно очень просто может стать причиной серьезных багов. Использование `unsafe` можно рассматривать как обходной путь, который позволяет выполнять операции, нарушающие правила Rust, например работать с сырьими указателями, вызывать функции C и обращаться к ресурсам вне выделенной области памяти программы или изменять их. В большинстве случаев это ключевое слово не нужно и следует избегать его применения.

Тем не менее почти невозможно работать с Rust, вообще не используя небезопасный код (`unsafe`), как минимум косвенно, поскольку в самой стандартной библиотеке такой код встречается повсеместно. Найти его можно, например, в реализациях `Box`, `Vec` и `String`. Кроме того, выделение и освобождение памяти, системные вызовы ОС и прочие низкоуровневые операции также являются небезопасными. Многие примеры использования небезопасного кода в стандартной библиотеке являются либо оптимизациями, либо необходимыми операциями, которые выполнить безопасно невозможно (интерфейс внешних функций (foreign function interface, FFI), системные вызовы и т. д.). К примеру, реализация метода `Vec::insert()` содержит блок `unsafe` (листинг 10.1). Этот блок позволяет оптимизировать вставку элемента в вектор.

### Листинг 10.1. `Vec::insert()` из стандартной библиотеки Rust

```
pub fn insert(&mut self, index: usize, element: T) {
    #[cold]
    #[cfg_attr(not(feature = "panic_immediate_abort"), inline(never))]
    #[track_caller]
    fn assert_failed(index: usize, len: usize) -> ! {
        panic!("insertion index (is {index}) should be <= len (is {len})");
    }

    let len = self.len();
```

```

// Место под новый элемент.
if len == self.buf.capacity() {
    self.reserve(1);
}

unsafe { ← Начало блока unsafe
    // Безотказная операция.
    // Место для размещения нового значения.
    {
        let p = self.as_mut_ptr().add(index); ← Получаем изменяемый указатель на буфер Vec, используя as_mut_ptr() и арифметику указателей. Вызов as_mut_ptr() безопасен, но вызов add() в возвращаемом указателе — нет
        if index < len {
            // Сдвиг для освобождения места. (Дублирование указанного в index элемента в двух новых последовательных точках.)
            ptr::copy(p, p.add(1), len - index); ← Сдвигаем элементы, используя арифметику указателей с помощью ptr::copy(), чтобы освободить место под новый
        } else if index == len {
            // Никакие элементы сдвигать не нужно.
        } else {
            assert_failed(index, len);
        }
        // Записываем элемент, переписывая первую копию элемента из index.
        ptr::write(p, element); ← Записываем новый элемент в Vec, используя ptr::write()
    }
    self.set_len(len + 1);
}
}

```

Вы можете поинтересоваться, почему метод `insert()` реализуется с блоком `unsafe`. Если коротко, то небезопасная версия вставки намного быстрее безопасной, поэтому авторы стандартной библиотеки пошли на этот компромисс ради быстродействия. В частности, `ptr::copy()` равнозначна `memmove()` в C, которую можно оптимизировать с помощью SIMD (single instruction, multiple data — один поток команд, множественный поток данных) и прочих низкоуровневых инструкций при условии, что копируемые области памяти не будут пересекаться. Код выше написан как безотказный, то есть он не вызывает паники или неопределенного поведения, несмотря на свою техническую небезопасность. В листинге 10.1 автор стандартной библиотеки старательно задокументировал использование `unsafe`, оставив комментарии о его использовании (что является хорошей практикой).

### 10.2.1. Что делает `unsafe`

Теперь разберем, почему небезопасный код необходим в Rust и когда он может потребоваться. Ключевое слово `unsafe` позволяет:

- разыменовывать сырье указатели;
- вызывать небезопасные функции или методы;
- обращаться к изменяемым статическим переменным и изменять их;
- реализовывать небезопасные трейты;
- обращаться к полям типов `union`, предоставляемых для совместимости с C.

На практике чаще всего `unsafe` используется при работе с библиотеками С или другим кодом на основе FFI. Если вы, например, хотите интегрировать библиотеку Python, то должны использовать `unsafe` для вызова API Python/C. Но лучше для этого обратиться к фреймворку наподобие PyO3 (<https://github.com/PyO3/pyo3>), предоставляющему все необходимые привязки.

Так, библиотеки С известны тем, что в них можно использовать сырье указатели, вручную выделять и освобождать память и выполнять прочие действия, которые в Rust не приветствуются. Эти библиотеки полезны при работе с системными вызовами, которые необходимы для взаимодействия с ОС (чтение/запись файлов, создание процессов и управление ими, доступ к периферийным устройствам и т. д.).

Стандартная библиотека Rust предоставляет множество безопасных абстракций, избавляя вас от необходимости писать небезопасный код самим, но нужно иметь в виду, что при использовании этих абстракций по факту вы используете такой код. К примеру, модуль `std::ffi` предоставляет безопасные абстракции для FFI, такие как `CString`, `CStr`, `OsStr` и `OsString`. В свою очередь, модуль `std::fs` предоставляет безопасные абстракции для файлового ввода/вывода, такие как `File`, `DirEntry` и `Metadata`. А модуль `std::process` предоставляет абстракции для управления процессами, такие как `Command`, `ExitStatus` и `Stdio`.

Для арифметики указателей модуль `std::ptr` предоставляет абстракции, которые позволяют работать с указателями. Однако большинство ключевых методов небезопасны, поэтому требуют использования блоков `unsafe`.

Выделение и освобождение памяти тоже небезопасные операции, и в стандартной библиотеке есть безопасные абстракции для них, например `Box`, `Vec` и `String`. Внутренне `Box`, `Vec` и `String` используют трait `Rust Allocator`, который обертыывает `malloc()` и `free()` в UNIX-подобных системах, а также `HeapAlloc()` и `HeapFree()` в Windows. API этого аллокатора представляет набор небезопасных функций, которые являются частью небезопасного трейта `Allocator`, позволяющего выделять и освобождать память. С его помощью стандартная библиотека предоставляет безопасные абстракции для этих операций. На момент написания книги API аллокатора находится на экспериментальной стадии и доступен только вочной версии Rust.

### 10.2.2. Где можно использовать `unsafe`

Ключевое слово `unsafe` можно использовать следующим образом.

- Определить блок как небезопасный, указав `unsafe` и заключив сам блок в фигурные скобки: `unsafe { ... }`. Такой блок кода будет вычисляться как выражение, и его значением будет итоговое значение последнего выражения в блоке.
- Определить функцию как небезопасную, используя ключевое слово `unsafe`, например `unsafe fn foo() { ... }`. Небезопасную функцию можно вызывать только из блока `unsafe` или другой небезопасной функции.

- Определить трейт как небезопасный также с помощью ключевого слова `unsafe`, например `unsafe trait Foo { ... }`. Такой трейт может содержать безопасные и небезопасные методы, но вызывать его можно только из блока `unsafe` или функции. Любой трейт с одним или более небезопасными методами считается небезопасным.

В листинге 10.2 показана небезопасная функция, вызывающая `printf()` из стандартной библиотеки С.

#### Листинг 10.2. Определение unsafe функции для вызова printf()

```
unsafe fn unsafe_function() {
    libc::printf(
        "calling C's printf() within unsafe_function()\n\0".as_ptr() as *const i8,
    );
}
```

Протестируем эту функцию, вызвав `unsafe_function()` из блока `unsafe` (листинг 10.3).

#### Листинг 10.3. Вызов unsafe\_function() из блока unsafe

```
unsafe {
    unsafe_function();
}
```

При выполнении этого кода мы получим такой вывод:

```
calling C's printf() within unsafe_function()
```

Можно также определить небезопасный трейт (листинг 10.4).

#### Листинг 10.4. Определение небезопасного трейта

```
unsafe trait UnsafeTrait {
    fn safe_method(&self);
    unsafe fn unsafe_method(&self);
}

struct MyStruct;

unsafe impl UnsafeTrait for MyStruct {
    fn safe_method(&self) {
        println!("calling println!() within UnsafeTrait::safe_method()");
    }
    unsafe fn unsafe_method(&self) {
        libc::printf(
            "calling C's printf() within UnsafeTrait::unsafe_method()\n\0"
            .as_ptr() as *const i8,
        );
    }
}
```

Кроме того, в этом примере есть безопасный метод, `safe_method()`, который можно вызвать без использования блока `unsafe`. А вот метод `unsafe_method()` вызывается только из этого блока (листинг 10.5).

#### Листинг 10.5. Тестирование небезопасного трейта

```
let my_struct = MyStruct;
my_struct.safe_method();
unsafe {
    my_struct.unsafe_method();
}
```

Вывод выглядит так:

```
calling println!() within UnsafeTrait::safe_method()
calling C's printf() within UnsafeTrait::unsafe_method()
```

Вы могли заметить, что можно скрывать небезопасный код за безопасными абстракциями. Возможность это или ошибка – вопрос субъективный. На практике невозможно создать язык программирования наподобие Rust, который будет на 100 % лишен небезопасного кода. Поэтому решение его разработчиков допустить использование безопасного кода для сокрытия небезопасного является pragматичным.

С помощью атрибута `#![forbid(unsafe_code)]` можно сделать так, чтобы ваш крейт не содержал небезопасный код, но это не касается крейтов, которые вы добавляете в качестве зависимостей, или стандартной библиотеки Rust (листинг 10.6). Иными словами, даже при использовании `#![forbid(unsafe_code)]` вы вряд ли избежите использования небезопасного кода, просто написан он будет не вами.

#### Листинг 10.6. Использование `#![forbid(unsafe_code)]`

```
#![forbid(unsafe_code)]

fn main() {
    // unsafe { ←
    //     libc::printf("Hello, world!\n".as_ptr() as *const _);
    // }
    let mut fruits = vec!["apple", "banana", "cherry"];
    fruits.insert(0, "orange"); ←
}
```

Раскомментирование этой строки приведет к ошибке компиляции

Эта строка не приведет к ошибке компиляции, хоть и вызывает `Vec::insert()`, которая содержит небезопасный код

**ПРИМЕЧАНИЕ** На момент написания книги в Rust нет способа избежать небезопасного кода в зависимостях. Но можно использовать крейт cargo-geiger (<https://crates.io/crates/cargo-geiger>) для анализа объема этого кода в крейте и его зависимостях.

### 10.2.3. Когда следует использовать `unsafe`

Чаще всего небезопасный код используется:

- при работе с библиотеками С или другим кодом на базе FFI;
- создании системных вызовов, не имеющих безопасных абстракций в стандартной библиотеке;
- реализации безопасных абстракций поверх небезопасного кода;
- написании низкоуровневых оптимизаций, которые нельзя выразить безопасно.

Некоторые ярые приверженцы Rust категоричны в отношении небезопасного кода и считают, что его быть не должно. Более же pragматичный подход (сторонником которого являюсь и я) состоит в том, что `unsafe` следует избегать по возможности, но не бояться использовать при необходимости. Когда вам действительно нужен небезопасный код, надо проявить повышенную осторожность и проследить, чтобы он был корректным и не вызывал неопределенное поведение. Но сказать проще, чем сделать, и это одна из причин, по которым небезопасный код считается антипаттерном.

Когда его использования не избежать, можно снизить вероятность внесения критических багов за счет применения надежных инструментов, таких как тестирование свойств, фаззинг и статический анализатор. Сообщество Rust разработало серию руководств для использования `unsafe`, которые можно найти по адресу <https://rust-lang.github.io/unsafe-code-guidelines>.

### 10.2.4. Нужно ли беспокоиться насчет `unsafe`

Чаще всего беспокоиться по поводу небезопасного кода не стоит, особенно в отношении стандартной библиотеки Rust. Она хорошо протестирована и обслуживается. Команда Rust очень строго следит за тем, чтобы в этой библиотеке не возникло неопределенного поведения. Кроме того, библиотека спроектирована для предоставления безопасных абстракций поверх небезопасного кода, поэтому в большинстве случаев использовать `unsafe` вам не придется.

Я встречал случаи, когда небезопасного кода было не избежать, например при работе с абстракциями ОС, не все из которых проработаны в стандартной библиотеке. Одним из недостатков абстракций стандартной библиотеки является то, что они созданы как кросс-платформенные и обычно предоставляют лишь те возможности, которые есть на всех этих plataформах. В итоге вам может потребоваться небезопасный код для использования каких-то характерных для отдельной платформы возможностей, например Windows API или оптимизация. Я понял, что небезопасный код не так страшен, если хорошо разбираться в Rust, понимать принцип работы модуля проверки заимствований, а также знать лучшие практики для управления ресурсами в этом языке, например использовать прием «получение ресурса есть инициализация» (resource acquisition is initialization, RAII) и умные указатели.

### 10.3. Использование метода `unwrap()`

Неправильное использование метода `unwrap()` — распространенный антипаттерн в Rust, который разработчики часто применяют, когда ленятся обрабатывать значения `Option` или `Result`. Но можно относительно легко избежать применения `unwrap()`, заменив его одним или несколькими из следующих методов:

- `expect()` — аналогичен `unwrap()`, но позволяет указывать собственное сообщение об ошибке на случай, если значение окажется `None` или `Err` (для `Option` и `Result` соответственно). Этот метод полезен для отладки, но использовать его для обработки ошибок в работающем коде желательно только в том случае, когда ожидаемым поведением программы является выход. Использование `expect()` функционально равноценно использованию утверждения, такого как `assert!(value.is_some())`;
- `map()` — позволяет преобразовывать значение `Option` или `Result` с помощью замыкания. Если это значение `None` или `Err`, то замыкание не вызывается и метод возвращает `None` или `Err`;
- `and_then()` — позволяет связывать в цепочку значения `Option` или `Result`, избегая глубокого вкладывания операторов `match` и `if let`;
- `unwrap_or()` — позволяет указывать предустановленное значение для случаев `None` или `Err` и предотвращает панику.

Кроме того, оператор `?` позволяет передавать информацию об ошибке вверх по стеку вызовов и особенно полезен при работе со значениями `Result`.

Метод `unwrap()` не всегда является антипаттерном, но часто создает проблемный код и может указывать на то, что вы забыли об обработке ошибок или возможности возникновения значений `None`. Кроме того, его наличие говорит о том, что вы не продумываете поток управления программы или не учитываете возможность ее сбоя.

Бывают и исключения, например, когда вы уверены, что значение никогда не будет `None` или `Err`. В таких случаях намного лучше использовать `expect()` с пользовательским сообщением об ошибке, так как он предоставит больше информации, если значение будет `None` или `Err`.

### 10.4. Пренебрежение `Vec`

Динамический массив `Vec`, один из самых распространенных типов в Rust, прекрасно подходит для большинства случаев. Многие разработчики допускают оплошность, пытаясь вместо использования `Vec` оптимизировать код путем написания собственных структур данных или применения ассоциативных массивов, множеств, деревьев либо связанных списков.

По факту же `Vec` невероятно быстро справляется с многими задачами. Зачастую он обеспечивает лучшую производительность в контексте различных бенчмарков.

К примеру, `HashSet` или `HashMap` славятся исключительно высокой скоростью поиска, но когда вам нужно добавить в коллекцию новые элементы, `Vec` зачастую сделает это быстрее. То же касается `BTreeSet` и `BTreeMap`, которые прекрасно подходят для упорядоченных коллекций, но во многих задачах уступают по скорости `Vec`. В свою очередь, `LinkedList` зачастую медленнее `Vec`, а также менее эффективен по части потребления памяти. Для наглядной демонстрации я написал для `Vec`, `HashSet` и `LinkedList` простой набор бенчмарков, выполняющих следующие операции:

- добавление миллиона элементов в пустую коллекцию;
- поиск тысячи случайных значений в коллекции из миллиона уникальных элементов;
- удаление тысячи элементов из коллекции, содержащей миллион элементов.

В листинге 10.7 показан бенчмарк `append`. (Целиком он доступен в исходном коде, прилагающемся к книге.)

**Листинг 10.7.** Проверка эффективности `Vec`, `HashSet` и `LinkedList` при добавлении элементов

```
#[bench]
fn vec_append(b: &mut Bencher) {
    b.iter(|| {
        let mut nums: Vec<i32> = Vec::new();
        for n in 0..1_000_000 {
            nums.push(n);
        }
    });
}

#[bench]
fn list_append(b: &mut Bencher) {
    b.iter(|| {
        let mut nums: LinkedList<i32> = LinkedList::new();
        for n in 0..1_000_000 {
            nums.push_back(n);
        }
    });
}

#[bench]
fn set_append(b: &mut Bencher) {
    b.iter(|| {
        let mut nums: HashSet<i32> = HashSet::new();
        for n in 0..1_000_000 {
            nums.insert(n);
        }
    });
}
```

Выполнив все тесты, мы видим, что хоть `Vec` и не всегда оказывается самым быстрым, но очень достойно справляется во всех трех случаях. Результаты показаны в листинге 10.8.

**Листинг 10.8. Результаты бенчмарка для Vec, HashSet и LinkedList**

```
running 9 tests
test tests::list_append ... bench: 53,860,800 ns/iter (+/- 2,306,429)
test tests::list_find ... bench: 527,207 ns/iter (+/- 26,305)
test tests::list_remove ... bench: 61,830,454 ns/iter (+/- 1,462,953)
test tests::set_append ... bench: 23,774,245 ns/iter (+/- 549,095)
test tests::set_find ... bench: 11 ns/iter (+/- 0)
test tests::set_remove ... bench: 839,977 ns/iter (+/- 4,571)
test tests::vec_append ... bench: 2,095,262 ns/iter (+/- 146,611)
test tests::vec_find ... bench: 133,359 ns/iter (+/- 11,424)
test tests::vec_remove ... bench: 3,319,558 ns/iter (+/- 57,979)

test result: ok. 0 passed; 0 failed; 0 ignored; 9 measured; 0 filtered out;
finished in 136.97s
```

**ПРИМЕЧАНИЕ** Выполнение этих тестов на вашей машине даст иной результат. В них используется функциональность, доступная только в ночной версии, и выполнять их нужно командой cargo bench (а не cargo test).

`Vec` обходит `LinkedList` во всех бенчмарках, а `HashSet` лидирует в удалении и поиске элементов, но значительно уступает при их добавлении. Тип `Vec` более эффективно оперирует памятью в сравнении с конкурентами и во многих случаях более удобен.

В плане сложности эти результаты оказались близки к ожидаемым. В табл. 10.1 представлены показатели О-нотации и Θ-нотации для типичных операций, которые выполняются с использованием `Vec`, `HashSet` и `LinkedList`.

**ПРИМЕЧАНИЕ** Результаты моего анализа отличаются от указанных в документации Rust, так как в ней не проводится различие между средней сложностью и сложностью в самом тяжелом случае.

**Таблица 10.1.** Сводные данные оценки сложности по О-нотации и Θ-нотации при выполнении типичных операций с использованием `Vec`, `HashSet` и `LinkedList`

Структура	Добавление		Поиск		Удаление	
	Средняя	Худшая	Средняя	Худшая	Средняя	Худшая
<code>Vec</code>	$\Theta(1)$	$O(n)$	$\Theta(n)$	$O(n)$	$\Theta(n)$	$O(n)$
<code>HashSet</code>	$\Theta(1)$	$O(n)$	$\Theta(1)$	$O(n)$	$\Theta(1)$	$O(n)$
<code>LinkedList</code>	$\Theta(1)$	$O(1)$	$\Theta(n)$	$O(n)$	$\Theta(n)$	$O(n)$

В большинстве случаев операция `Vec` выделяется не слишком сильно, за исключением поиска по индексу, представляющего сложность  $O(1)$  (в таблице и бенчмарках отсутствует). Однако на практике при различных рабочих нагрузках он демонстрирует удивительно высокую среднюю производительность.

Как ни странно, `LinkedList` значительно медленнее справился с поочередной вставкой миллиона элементов, но это объясняется тем, что `LinkedList` нужно выделять память для каждого элемента, а `Vec` выделяет память частями. Быстро-действие `HashSet` в операциях добавления и вставки тоже оказалось невелико. Объясняется это все тем же выделением памяти и затратами на ребалансировку хеш-таблицы по мере ее роста.

### Инструменты бенчмарков в Rust

В Rust есть собственный инструмент бенчмарков, который позволяет оперативно составлять бенчмарки по принципу написания модульных тестов. На момент составления книги эта функция доступна только в ночной версии Rust.

С помощью атрибута `#![bench]` можно определить модульный тест, который будет проверять любую операцию. Этот инструмент выполняет тест несколько раз и выдает среднее время выполнения вместе со стандартным отклонением.

Стандартная библиотека Rust содержит объект `Bencher`, который предоставляет метод `iter()`, позволяющий измерять время выполнения блока кода. Этот метод получает замыкание, в котором можно разместить код, подлежащий тестированию. Любые настройки и очистку необходимо выполнять до и после вызова `Bencher::iter()`. Ниже показан минимальный бенчмарк:

```
#![feature(test)]

#[cfg(test)]
mod test {
    extern crate test;
    use test::Bencher;

    #[bench]
    fn hello_world_10_times(b: &mut Bencher) {
        b.iter(|| {
            for _ in 0..10 {
                println!("Hello, world!");
            }
        });
    }
}
```

Выполнение команды `cargo bench` приведет к компиляции кода в режиме релиза и выполнению бенчмарков. Эта команда получает аргументы аналогично команде `cargo test`, позволяя отфильтровывать бенчмарки по имени или выполнять только конкретные из них. При выполнении `cargo bench` библиотека тестирования Rust выполняет код внутри `Bencher::iter()` в соответствии со следующими правилами, обеспечивающими стабильный результат.

- ◆ Бенчмарк выполняется 50 раз, и по результатам вычисляется общая сводка.
- ◆ Выбросы из результатов удаляются (5 % самых быстрых и самых медленных результатов).
- ◆ Бенчмарк снова выполняется 50 раз с вычислением результатов.

- ◆ Если выполняется одно из следующих условий, то результаты возвращаются:
  - стандартное отклонение результатов меньше 100 мс;
  - бенчмарк выполнялся дольше 3 секунд.
- ◆ Если не выполняется ни одно из условий, то бенчмарк повторяется с шага 1.

Если вы хотите выполнить бенчмарк в стабильной версии Rust, то используйте крейт Criterion.rs (документация доступна по адресу <https://bheisler.github.io/criterion.rs/book>), который предоставляет обширную функциональность Criterion.rs — часть портированной на Rust библиотеки Haskell под именем Criterion.

Преимуществом Vec является то, что он представляет непрерывный блок памяти. Это делает его удобным для кэширования в большинстве современных процессоров и позволяет компилятору оптимизировать его операции на уровне инструкций. Локальность данных — ключевой фактор в производительности, особенно если учесть, что обращение к основной памяти (RAM) на компьютерах существенно медленнее обращения к кэшу процессора. К тому же благодаря тому, что Vec хранится в памяти в виде непрерывного блока, управление этим блоком упрощается. В этом случае довольно просто сдвигать элементы, не используя сложные алгоритмы. В большинстве случаев достаточно легко копировать содержимое памяти, что на современных ПК делается очень быстро.

По факту в некоторых случаях множество, ассоциативный массив, дерево или связанный список будут обходить по скорости вектор, но вам будет сложнее, чем вы думаете, оправдать использование этих типов. Vec оказывается хорошим решением для большинства рабочих задач и прекрасным для многих. Если сомневаетесь, то используйте Vec или хотя бы прогоните свой код через бенчмарк, прежде чем использовать более сложную структуру данных.

## 10.5. Излишнее клонирование

Некоторые приверженцы Rust сильно не восторг от метода `clone()`, и во многих случаях на то есть весомые причины. Я не являюсь злостным противником клонирования, но мне кажется, что метод `clone()` часто используется чрезмерно или неуместно.

Этот метод создает глубокую копию значения, и на своем опыте могу сказать, что Rust-программисты используют его в качестве костыля, стараясь избежать концепции заимствования. Но это ошибочный подход. Он может привести к проблемам производительности, лишнему загромождению памяти, а также вызвать баги.

Но вызов `clone()` не всегда является плохим решением. В главе 9 я предлагаю использовать этот метод в качестве простого способа реализации неизменяемых

структур данных. Если же вы вдруг решите применить `clone()` для обхода модуля проверки заимствований, то не спешите и еще раз подумайте о структуре вашего кода. Если же вы идете на это обдуманно, то никаких проблем с использованием `clone()` возникнуть не должно, особенно когда ваши решения основываются на данных и результатах бенчмарков.

## 10.6. Использование трейта Deref для эмуляции полиморфизма

Полиморфизм — это прием, который позволяет обрабатывать объекты разных типов так, будто они относятся к одному типу. В объектно-ориентированных языках использование данного приема реализуется через приемы субтиповизации или наследования, которые отсутствуют в Rust.

Иногда мы используем трейт `Deref`, чтобы упростить работу с объектами, позволяя компилятору самостоятельно выводить методы, которые нам нужно вызывать, через приведение `Deref`. В некотором смысле мы эмулируем полиморфизм, который вы могли видеть в других языках, таких как C++ и Java. И этот подход не обязательно плох, но может указывать на то, что вы выстраиваете структуру кода вразрез с идиомами Rust.

Трейт `Deref` (и его изменяемый напарник `DerefMut`) позволяет разыменовывать значение, используя оператор `*`, как в `*value`. Кроме того, компилятор неявно использует трейт `Deref`, обеспечивая возможность вызывать методы для значения, обернутого в умный указатель, например `Box`, `Rc` или `Arc`. Иными словами, если у нас есть `let value: Box<T> = Box::new(T);`, то мы можем вызывать для `value` методы так, будто это `T`, не разыменовывая его, — например, `value.method()`.

В главе 7 мы говорили о структурах и увидели, как применение трейта `Deref` позволяет рассматривать структуру-обертку так, будто она имеет тип того значения, которое обертывает. И это типичный случай использования `Deref`, как и в умных указателях, применяющих его аналогичным образом, но полиморфизмом он не является. Зачастую можно избежать использования `Deref` для эмуляции полиморфизма, задействовав трейты и дженерики или просто предоставив метод, при необходимости возвращающий внутреннее значение. В листинге 10.9 показано использование `Deref` в простом примере, где путем приведения возвращается первый член кортежа `Person`.

### Листинг 10.9. Демонстрация приведения с помощью трейта Deref

```
use std::ops::Deref;
struct Person(String, String, u32); ← Кортеж с именем,
                                         фамилией и возрастом
impl Deref for Person {
    type Target = String; ← Реализуем для Person трейт Deref, чтобы можно
                           было выполнять разыменовывание в String
```

```

fn deref(&self) -> &Self::Target {
    &self.0
}
}

fn main() {
    let ferris = Person("Ferris".to_string(), "Bueller".to_string(), 17);
    println!("Hello, {}!", *ferris);
    println!("The length of a person is {}", ferris.len());
}

```

В этом примере есть кортеж `Person`, который обертывает две строки (имя и фамилию) и возраст. Благодаря приведению с помощью `Deref` мы вызываем для `ferris` метод `len()` так, будто это `String`. В этом примере мы возвращаем имя человека, но для читателя может быть неочевидно, почему мы это делаем. Почему бы не вернуть имя из метода напрямую? При чтении этого кода может возникнуть легкое недоумение, так как он не следует идиомам Rust. Мы заставляем `Person` в определенном случае действовать как `String`, но причина этих наших действий неясна. При выполнении кода выше мы получим такой вывод:

```
Hello, Ferris!
The length of a person is 6
```

Можно было так же легко реализовать метод `first_name()`, возвращающий внутреннюю `String`, или даже предоставить метод `first_name_len()`, что было бы гораздо понятнее (хотя если возвращать строку, то ее будет достаточно, чтобы получить длину с помощью `ferris.first_name().len()`). Некоторое удобство обращения к имени не оправдывает ту неоднозначность, которую привносит использование `Deref`. Если мы хотим предоставить метод `first_name_len()`, то можем реализовать его так:

```

impl Person {
    fn first_name_len(&self) -> usize {
        self.0.len()
    }
}

```

Наглядный пример использования трейта `Deref` для эмуляции полиморфизма показан в листинге 10.10 и представляет один из вариантов эмуляции данной функциональности в Rust. В коде есть `Dog`, реализующий трейт `Animal`, и `Cat`, также реализующий этот трейт.

#### Листинг 10.10. Эмуляция полиморфизма с помощью трейт-объектов (хорошая практика)

```

trait Animal {
    fn speak(&self) -> &str;
    fn name(&self) -> &str;
}

```

```

struct Dog {
    name: String,
}
impl Dog {
    fn new(name: &str) -> Self {
        Self {
            name: name.to_string(),
        }
    }
}
impl Animal for Dog {
    fn speak(&self) -> &str {
        "Woof!"
    }
    fn name(&self) -> &str {
        &self.name
    }
}

struct Cat {
    name: String,
}
impl Cat {
    fn new(name: &str) -> Self {
        Self {
            name: name.to_string(),
        }
    }
}
impl Animal for Cat {
    fn speak(&self) -> &str {
        "Meow!"
    }
    fn name(&self) -> &str {
        &self.name
    }
}

```

Протестируем этот код, выполнив код ниже (листинг 10.11). В нем создадим `Box<dyn Animal>` и вызовем метод `speak()` для каждого `Animal` в векторе.

#### Листинг 10.11. Тестирование полиморфизма с помощью trait-объектов

```

fn main() {
    let dog = Box::new(Dog::new("Rusty"));
    let cat = Box::new(Cat::new("Misty"));

    let animals: Vec<Box<dyn Animal>> = vec![dog, cat];

    for animal in animals {
        println!("{} says {}", animal.name(), animal.speak());
    }
}

```

Вывод выглядит так:

```
Rusty says Woof!
Misty says Meow!
```

Перед вами пример идиоматического Rust. Мы использовали трейт-объект для создания вектора говорящих животных.

Теперь создадим нечто похожее, однако на сей раз используем для эмуляции полиморфизма трейт `Deref` (листинг 10.12). Мы построим структуру `Animal` со свойством `name` и будем рассматривать ее как суперкласс структур `Dog` и `Cat`, возвращая внутреннее `Animal` с помощью `Deref`.

### Листинг 10.12. Эмуляция полиморфизма с помощью трейта `Deref` (плохая практика)

```
use std::ops::Deref;

struct Animal {
    name: String,
}

impl Animal {
    fn new(name: &str) -> Animal {
        Animal { name: name.to_string() }
    }
    fn name(&self) -> &str {
        &self.name
    }
}

struct Dog(Animal);
impl Dog {
    fn new(name: &str) -> Self {
        Self(Animal::new(name))
    }
    fn speak(&self) -> &str {
        "Woof!"
    }
}
impl Deref for Dog {
    type Target = Animal;
    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

struct Cat(Animal);
impl Cat {
    fn new(name: &str) -> Self {
        Self(Animal::new(name))
    }
    fn speak(&self) -> &str {
```

```

        "Meow!"
    }
}
impl Deref for Cat {
    type Target = Animal;
    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

```

Теперь протестируем это решение, выполнив код из листинга 10.13.

#### Листинг 10.13. Тестирование реализации полиморфизма с помощью Deref

```

fn main() {
    let dog = Dog::new("Rusty");
    let cat = Cat::new("Misty");

    println!("{} says: {}", dog.name(), dog.speak());
    println!("{} says: {}", cat.name(), cat.speak());
}

```

Вывод выглядит так:

```
Rusty says: Woof!
Misty says: Meow!
```

Этот пример уже не является идиоматическим Rust. Здесь использование трейта `Deref` для эмуляции полиморфизма запутает любого, кто будет читать код. Структура `Animal` не является суперклассом `Dog` и `Cat`, и непонятно, почему мы должны рассматривать `Dog` и `Cat` как `Animal`.

Но и исключать полностью `Deref` не стоит. Главное, избегайте излишнего и неуместного его применения, особенно в контекстах, которые могут вводить в заблуждение. Если вы решите использовать этот трейт из желания эмулировать полиморфизм Java или C++, то рекомендую пересмотреть дизайн вашего кода.

## 10.7. Глобальные данные и синглтоны

В Rust нет собственной концепции глобальных данных или синглтонов (`singleton` — «одиночка»), и реализовывать их нужно самостоятельно. Так задумано изначально, поскольку глобальные данные и синглтоны часто считаются антипаттернами и могут вызывать различные проблемы, такие как сильная связанность, плохая тестируемость и сложность в понимании кода.

В Rust для создания глобальных данных или синглтонов можно использовать крейты наподобие `lazy_static`, но перед этим всегда нужно хорошо подумать. Во многих случаях можно избежать использования как глобальных данных, так и синглтонов, используя внедрение зависимостей или передавая данные функциям в виде аргументов.

Глобальные данные и синглтоны могут создавать проблемы, в частности, для библиотек: они усложняют понимание поведения последних и могут приводить к возникновению неожиданных ситуаций при использовании библиотеки в иных контекстах. Глобальные данные иногда становятся узким местом или источником взаимных блокировок в многопоточных программах, что может вести к утечкам памяти и создавать иные проблемы управления ресурсами.

В Rust есть `std::cell::OnceCell` и его потокобезопасный собрат `std::sync::OnceLock`, которые обеспечивают безопасный способ создания синглтонов. В качестве альтернативы ваша библиотека может предоставлять функцию, инициализирующую синглтон, а вы будете решать, как им управлять. Это хороший подход, позволяющий обеспечить гибкость и избежать проблем, связанных с глобальным состоянием и синглтонами.

## 10.8. Слишком много умных указателей

Умные указатели очень полезны, и конкретно в Rust они необходимы для реализации множества задач, в других языках являющихся тривиальными. Но умными указателями можно и злоупотребить, а также использовать их тип, не подходящий под конкретную задачу. В Rust три основных типа умных указателей.

- `Box` — умный указатель, обеспечивающий выделение и освобождение памяти в куче, а также позволяющий перемещать значения между различными областями видимости. Кроме того, он дает возможность хранить значения, размер которых неизвестен на этапе компиляции, в объектах фиксированного размера (например, `Sized`).
- `Rc` — умный указатель с подсчетом ссылок, допускающий множественное или совместное владение значением. Вдобавок он обеспечивает те же возможности, что и `Box`.
- `Arc` — атомарный умный указатель с подсчетом ссылок, который допускает наличие нескольких владельцев значения в разных потоках, предоставляя возможности `Rc` и `Box` потокобезопасным способом. При этом `Arc` не синхронизирует само значение. Его синхронизация касается только подсчета ссылок.

Указатель `Box` обычно используется, когда нужно выделить память в куче, но владение значением разделять нежелательно. `Rc` подходит, когда вам нужно обеспечить общее владение, но не разделять его между потоками. В свою очередь, `Arc` полезен в ситуациях, когда нужно разделить владение значением между разными потоками.

Помимо этого, `RefCell` и `Cell` обеспечивают внутреннюю изменяемость и часто используются совместно с `Rc` и `Arc`, которые дают возможность разделять владение значением, но не позволяют его изменять. В отличие от этих умных указателей `RefCell` и `Cell` позволяют изменять значение, но не дают возможности разделять владение им.

Иногда мы задействуем умные указатели, когда они нам не нужны, или злоупотребляем ими из соображений удобства либо потому, что они позволяют обойти модуль проверки заимствований. Бывает проще придерживаться чего-то внутри умного указателя `Rc` и клонировать его, чем продумывать механизм владения значением и его заимствования.

Еще один пример злоупотребления умными указателями — использование `Box` внутри `Vec`. И `Box`, и `Vec` выделяют память в куче для своего содержимого, так что этот подход может оказаться избыточным, вдобавок он требует двух операций выделения: для `Vec` и для содержащегося в нем `Box`. В листинге 10.14 показан пример тройного выделения: `String` внутри `Box`, который внутри `Vec`.

#### Листинг 10.14. Злоупотребление умными указателями

```
let mut string_box_vec: Vec<Box<String>> =
    vec![Box::new(
        String::from("unnecessarily boxed string"))
    }];
let mut string_vec: Vec<String> =
    vec![String::from("this is okay")]; ← Сохраняем String
                                            непосредственно в Vec
let boxed_string = string_box_vec.remove(0);
let normal_string = string_vec.remove(0); ← Удаляем выделенный в куче String
                                            из Vec, что равноценно удалению
                                            указателя на указатель в Vec
                                            Удаляет String из Vec, что равноценно
                                            удалению одного указателя
```

`String` эквивалентна имеющему длину указателю на память в куче, а `Box` — это указатель на некое выделенное в куче значение. Иногда мы используем `Box`, так как он позволяет перемещать значения между областями видимости, но в данном случае это будет излишним, поскольку `Vec` уже предоставляет эту функциональность. Можно без проблем поместить `String` в `Vec`, так как строки имеют переменную длину, а каждая сущность в `Vec` должна иметь фиксированный (и равный) размер.

Если вы используете умные указатели для обхода модуля проверки заимствований, то лучше пересмотрите структуру своего кода. Хорошим правилом будет стараться писать код без умных указателей и добавлять их только по необходимости.

## 10.9. Что дальше?

Когда вы дочитаете эту книгу, то самыми важными действиями по совершенствованию ваших навыков будут написание кода и применение полученных знаний. Практика всегда лучший учитель. Вы очень многое узнаете в процессе написания кода, получения обратной связи на него и чтения чужого кода. По мере дальнейшего обучения и погружения в Rust вы можете возвращаться к этой книге, открывая все новые нюансы. Если вы захотите прочесть другие учебники по Rust, то рекомендую книгу *Code Like a Pro in Rust* (<https://www.manning.com/>)

books/code-like-a-pro-in-rust), которую я уже неоднократно упоминал выше и которая послужила для меня вдохновением. Кроме того, еще один прекрасный источник знаний — официальная документация к языку, так что рекомендую ознакомиться и с ней. Вдобавок у Rust очень активное сообщество, в котором предлагается очень много ресурсов, в том числе канал в Reddit (<https://reddit.com/r/rust>), сервер в Discord (<https://discord.gg/rust-lang>) и форум пользователей (<https://users.rust-lang.org>).

И наконец, по Rust регулярно проводятся тематические семинары и конференции, которые позволяют включиться в жизнь сообщества и лично познакомиться со многими специалистами, работающими на этом языке. Я сам иногда посещаю семинар в Нью-Йорке и всегда рад поболтать о Rust и программировании в целом.

**СОВЕТ** На встречах в Нью-Йорке мы по традиции отвечаем на вопросы викторины Дэвида Толней (David Tolnay), посвященной Rust, что позволяет в увлекательной форме улучшать навыки и узнавать о некоторых необычных возможностях и синтаксисе языка. Викторину можно найти по ссылке <https://dtolnay.github.io/rust-quiz>.

## Резюме

- Антипаттерны — это практики программирования, которые считаются вредными в конкретных контекстах или всегда. Несмотря на то что их использование часто является делом вкуса, в некоторых случаях они объективно неуместны — например, когда это небезопасно, неэффективно или чревато трудностями в сопровождении кода.
- Ключевое слово `unsafe` — неотъемлемый элемент Rust, который иногда используется чрезмерно или неверно. Практически невозможно работать с Rust, не используя небезопасный код (как минимум косвенно), и, если он встречается вам в чужом коде или оказывается необходим в вашем, нужно тщательно его изучить. Никогда не используйте `unsafe` для обхода модуля проверки заимствований.
- Метод `unwrap()` — распространенный антипаттерн в Rust, который часто используется, когда разработчики ленятся обрабатывать значения `Option` или `Result`. Можно без особых сложностей избежать `unwrap()`, заменив его одним или несколькими следующими методами: `expect()`, `map()`, `and_then()`, `unwrap_or()` — и оператором `?`.
- `Vec` отлично справляется со множеством рабочих нагрузок и часто оказывается оптимальным выбором. Он нередко опережает такие структуры, как `HashSet`, `HashMap`, `BTreeSet`, `BTreeMap` и `LinkedList`, в целой серии бенчмарков, при этом отличаясь от них более эффективным использованием памяти.
- Метод `clone()` иногда применяется чрезмерно или неуместно. Это не всегда плохо, но может являться признаком неудачного кода, ведущего к проблемам с производительностью и лишнему загромождению памяти.

- Трейт `Deref` иногда используется для эмуляции полиморфизма, что в Rust может вводить в заблуждение. Вместо него следует использовать трейты или дженерики либо просто предоставлять метод, который будет при необходимости возвращать внутреннее значение.
- Использование глобальных данных и синглтонов часто считается антипаттерном, так как они могут вести к различным проблемам, таким как сильная связанность, плохая тестируемость и сложность понимания кода. В Rust для создания глобальных данных и синглтонов подходят крейты наподобие `lazy_static`, но такое решение необходимо тщательно обдумывать.
- Умные указатели невероятно полезны, но ими можно злоупотребить или использовать их тип, не подходящий для задачи. Если вы применяете умные указатели в качестве уловки для обхода модуля проверки заимствований, то вам следует пересмотреть структуру вашего кода.

# *Приложение.*

## *Установка Rust*

---

Чтобы получить от книги максимальную пользу, вам потребуется настроить набор инструментов Rust. Если вы только начинаете работать с этим языком, то нужно будет установить свежую версию этого набора, содержащего компилятор и стандартную библиотеку. В зависимости от используемой ОС также может потребоваться установить инструменты разработчика для компиляции и выполнения всех фрагментов кода, представленных в книге.

### ***П.1. Установка инструментов для работы с книгой***

Чтобы иметь возможность компилировать и запускать фрагменты кода из данной книги, установите необходимые зависимости.

#### ***П.1.1. Установка для macOS с помощью Homebrew***

```
$ brew install git
```

В macOS нужно установить пакет инструментов командной строки Xcode:

```
$ sudo xcode-select --install
```

#### ***П.1.2. Установка для систем Linux***

Чтобы установить инструменты для систем на базе Debian, выполните эту команду:

```
$ apt-get install git build-essential
```

Установка инструментов для систем на базе Red Hat выполняется с помощью этой команды:

```
$ yum install git make automake gcc gcc-c++
```

**СОВЕТ** Вы можете решить установить Clang вместо GCC, который демонстрирует более высокую скорость компиляции.

Для установки `rustup` в Linux и UNIX-совместимых системах, включая macOS, используйте следующую команду:

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

После этого убедитесь, что у вас установлены и стабильный (stable), и ночной (nightly) наборы инструментов:

```
$ rustup toolchain install stable nightly
...

```

### **П.1.3. Установка для Windows**

Если вы используете Windows, вам потребуется скачать последнюю копию `rustup` по ссылке <https://rustup.rs>. Предварительно собранные исполняемые файлы Windows для Clang скачайте с <https://releases.llvm.org/download.html>.

В качестве альтернативы можно использовать Windows Subsystem for Linux (WSL; <https://docs.microsoft.com/en-us/windows/wsl>) и выполнить инструкции из предыдущего подраздела, посвященного установке в Linux. Для многих пользователей этот подход станет самым простым решением, позволяющим работать с фрагментами кода.

## **П.2. Управление `rustc` и прочими компонентами Rust через `rustup`**

После установки `rustup` нужно будет установить компилятор Rust и сопутствующие инструменты. Как минимум советую установить стабильный и ночной каналы релизов Rust.

### **П.2.1. Установка `rustc` и других компонентов**

По умолчанию вам нужно установить наборы инструментов `stable` и `nightly`, но обычно лучше стараться работать именно со `stable`. Вот код для установки обоих этих наборов:

```
# Установит набор инструментов stable и сделает его используемым по умолчанию
$ rustup default stable
...
# Установит набор nightly
$ rustup toolchain install nightly
```

В примерах из книги используются `clippy` и `rustfmt`, которые можно установить с помощью `rustup`:

```
$ rustup component add clippy rustfmt
```

### **П.2.2. Переключение штатного набора инструментов с помощью rustup**

При работе в Rust может потребоваться часто переключаться между наборами инструментов `stable` и `nightly`. Это легко делать с помощью `rustup`:

```
# Переключение на набор инструментов stable  
$ rustup default stable  
# Переключение на набор инструментов nightly  
$ rustup default nightly
```

### **П.2.3. Обновление компонентов Rust**

Сохранять актуальность компонентов легко с помощью `rustup`. Для обновления всех установленных наборов инструментов и компонентов достаточно выполнить такую команду:

```
$ rustup update
```

В обычных обстоятельствах `update` нужно выполнять, только когда обновляется мажорная версия ПО. Иногда обновление требуется из-за проблем вочной сборке, но это бывает редко. Если ваш сетап работает, рекомендуется избегать частых обновлений (имеется в виду ежедневных), так как это легко может привести к проблемам.

**ПРИМЕЧАНИЕ** Чтобы обновить все компоненты и наборы инструментов Rust, их необходимо скачать, что в системах с ограниченной пропускной способностью сети может занять некоторое время.

# Read IT Club

**Комьюнити рецензентов  
и переводчиков ИТ-литературы**

Миссия участников клуба – обеспечить высокое качество профессиональной переводной литературы в России. «Книжные дебагеры» проверяют корректность терминологии и подписей на схемах и иллюстрациях, чтобы сделать книги более понятными русскоязычному читателю. Стать участником Read IT Club может любой ИТ-специалист, готовый поделиться опытом с сообществом.

