СТИВ КЛАБНИК, КЭРОЛ НИКОЛС

ПРОГРАММИРОВАНИЕ на RUST

ОФИЦИАЛЬНЫЙ ГАЙД

ОТ КОМАНДЫ РАЗРАБОТЧИКОВ RUST ИЗ MOZILLA FOUNDATION

Steve Klabnik, Carol Nichols

THE RUST PROGRAMMING LANGUAGE

by Steve Klabnik and Carol Nichols,
with contributions from
the Rust Community

no starch
press
San Francisco

СТИВ КЛАБНИК, КЭРОЛ НИКОЛС

ПРОГРАММИРОВАНИЕ на RUST

ОФИЦИАЛЬНЫЙ ГАЙД
ОТ КОМАНДЫ РАЗРАБОТЧИКОВ RUST ИЗ MOZILLA FOUNDATION

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2021

ББК 32.973.2-018.1
УДК 004.43
К47

Клабник Стив, Николс Кэрол

К47 Программирование на Rust. — СПб.: Питер, 2021. — 592 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1656-0

Официальный гайд по языку программирования Rust от команды разработчиков Rust из Mozilla Foundation. Добро пожаловать в мир Rust!

Этот официальный гид поможет вам создавать более быстрое и надежное программное обеспечение. Высокоуровневая эргономика и низкоуровневое управление часто противоречат друг другу, но Rust бросает вызов этому конфликту.

Авторы книги входят в команду разработчиков языка, а значит, вы получите всю информацию из первых рук — от установки языка до создания надежных и масштабируемых программ. От создания функций, выбора типов данных и привязки переменных вы перейдете к более сложным концепциям:

- Владение и заимствование, жизненный цикл и типажи.
- Гарантированная безопасность программ.
- Тестирование, обработка ошибок и эффективный рефакторинг.
- Обобщения, умные указатели, многопоточность, типажные объекты и сопоставления.
- Работа со встроенным менеджером пакетов Cargo для создания, тестирования, документирования кода и управления зависимостями.
- Продвинутые средства работы с Unsafe Rust.

Вы найдете множество примеров кода, а также три главы, посвященные созданию полноценных проектов для закрепления знаний: игры-угадайки, создание инструмента командной строки и многопоточного сервера.

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

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1718500440 англ.

ISBN 978-5-4461-1656-0

Краткое содержание

Оглавление

Вступление

Не сразу очевидно, но язык программирования Rust всецело строится на расширении возможностей: независимо от того, какой код вы пишете сейчас, Rust наделяет вас возможностями достичь большего, уверенно программировать в более широком спектре областей, чем раньше.

Возьмем, например, работу «на системном уровне», которая завязана на низкоуровневые детали управления памятью, представления данных и конкурентности. Традиционно на эту сферу программирования смотрят как на доступную только избранным, которые посвятили годы обучения, чтобы избежать ее печально известных ловушек. И даже те, кто работает в этой области, действуют весьма осторожно, чтобы не подвергать свой код уязвимостям, аварийным сбоям и повреждениям.

Язык Rust разрушает эти барьеры, устраняя старые ловушки и предоставляя дружественный, безупречный набор инструментов, который поможет вам на этом пути. Программисты, которым нужно погрузиться в более низкоуровневое управление, смогут сделать это с помощью языка Rust, не беря на себя привычный риск сбоев или дыр в безопасности и не изучая тонкости переменчивой цепочки инструментов. Более того, данный язык разработан так, чтобы естественным образом направлять вас к надежному коду, который является эффективным с точки зрения скорости и использования памяти.

Программисты, которые уже работают с низкоуровневым кодом, могут использовать Rust для повышения своих амбиций. Например, введение в Rust конкурентности является операцией с относительной невысокой степенью риска: компилятор будет отлавливать типичные ошибки за вас. И вы можете заняться более активной оптимизацией в коде и быть уверенным, что случайно не внесете сбои или уязвимости.

Но Rust не ограничивается программированием низкоуровневых систем. Он является выразительным и эргономичным настолько, что делает приятным написание CLI-приложений, веб-серверов и многих других видов кода — позже в книге вы найдете простые примеры того и другого. Работа с Rust позволяет накапливать навыки, применимые в одной области, и использовать их в другой сфере. Вы можете усвоить язык Rust, написав веб-приложение, а затем применить те же навыки в разработке кода для Raspberry Pi.

Эта книга всеобъемлюще охватывает потенциал языка Rust, наделяя его пользователей расширенными возможностями. Этот доступный текст был задуман для того, чтобы помочь вам повысить не только уровень знаний о языке Rust, но и ваши достижения и уверенность в себе как программиста в целом. Так что погружайтесь, готовьтесь учиться, и добро пожаловать в сообщество языка Rust!

Николас Мацакис и Аарон Турон

Предисловие

Авторы исходят из того, что вы используете язык Rust 1.31.0 или более позднюю версию с опцией `edition="2018"` во всех проектах Cargo.toml, где применяются идиомы Rust в редакции 2018 года. См. раздел «Установка» для получения информации об установке или обновлении Rust, а также приложение Д, где можно найти сведения о редакциях языка.

Редакция 2018 года языка Rust включает в себя ряд улучшений, которые делают Rust эргономичнее и проще в освоении. Настоящая версия книги содержит ряд изменений, отражающих эти улучшения:

- Глава 7 «Управление растущими проектами с помощью пакетов, упаковок и модулей» была в основном переписана. Система модулей и характер работы путей в редакции 2018 года были сделаны более согласованными.
- В главе 10 появились новые разделы, озаглавленные «Типажи в качестве параметров» и «Возвращение типов, реализующих типажи», которые объясняют новый синтаксис `impl Trait`.
- В главе 11 есть новый раздел под названием «Использование типа `Result<T, E>` в тестах», где показаны способы написания тестов, в которых используется оператор `?`.
- Раздел «Продвинутые сроки жизни» в главе 19 был удален, поскольку улучшения компилятора сделали конструкции в этом разделе еще более редкими.
- Предыдущее приложение Г «Макрокоманды» было расширено за счет процедурных макрокоманд и перенесено в раздел «Макрокоманды» главы 19.
- Приложение А «Ключевые слова» также объясняет новое языковое средство под названием «сырые идентификаторы», которое позволяет взаимодействовать коду, написанному в редакции 2015 года, с редакцией 2018 года.
- Приложение Г теперь называется «Полезные инструменты разработки», в нем рассказывается о недавно выпущенных инструментах, которые помогают вам писать код Rust.
- В книге мы исправили ряд мелких ошибок и неточных формулировок. Спасибо читателям, которые о них сообщили!

Обратите внимание, что любой код в первом варианте этой книги, который компилировался, будет продолжать компилироваться без опции `edition="2018"` в *Cargo.toml* проекта, даже после того как вы обновите используемую вами версию компилятора Rust. Яркий пример гарантии обратной совместимости Rust в действии!

Благодарности

Мы хотели бы поблагодарить всех, кто работал над Rust, за создание удивительного языка, о котором стоит написать книгу. Мы благодарны всем участникам сообщества языка Rust за их радушие и создание среды, достойной того, чтобы пригласить туда еще больше людей.

Мы особенно благодарны всем, кто читал первые редакции этой книги онлайн и давал отзывы, отчеты об ошибках и запросы на включение внесенных изменений. Особая благодарность Эдуарду-Михаю Буртеску и Алексу Крайтону за научное редактирование и Карен Рустад Тельва за обложку. Спасибо нашей команде из издательства No Starch — Биллу Поллоку, Лиз Чедвик и Джанель Людовиз — за то, что они усовершенствовали эту книгу и издали ее.

Стив хотел бы поблагодарить Кэрол за то, что она была замечательным соавтором. Без нее эта книга была бы гораздо менее качественной и написание заняло бы гораздо больше времени. Дополнительная благодарность Эшли Уильямс, которая оказала невероятную поддержку на всех этапах.

Кэрол хотела бы поблагодарить Стива за то, что он пробудил в ней интерес к Rust и дал возможность поработать над этой книгой. Она благодарна своей семье за постоянную любовь и поддержку, в особенности мужу Джейку Гулдингу и дочери Вивиан.

Об авторах

Стив Клабник возглавляет команду по документированию Rust и является одним из ключевых разработчиков языка. Часто выступает с лекциям и пишет много открытого исходного кода. Ранее работал над такими проектами, как Ruby и Ruby on Rails.

Кэрол Николс является членом команды разработчиков Rust Core и соучредителем Integer 32, LLC, первой в мире консалтинговой компании по разработке ПО, ориентированной на Rust. Николс является организатором конференции «Ржавый пояс» (Rust Belt) по языку Rust.

Введение

Добро пожаловать в язык программирования Rust! Он помогает писать более быстрое и надежное программное обеспечение. Высокоуровневая эргономика и низкоуровневое управление часто противоречат друг другу в дизайне языка программирования, но Rust бросает вызов этому конфликту. Обеспечивая сбалансированное сочетание мощных технических возможностей и великолепного опыта разработки программ, язык Rust дает возможность контролировать низкоуровневые детали (например использование памяти) без каких-либо хлопот, традиционно связанных с таким контролем.

Кому подойдет язык Rust

Rust идеально подходит многим людям по целому ряду причин. Давайте взглянем на несколько наиболее важных групп.

Команды разработчиков

Rust оказывается продуктивным инструментом для совместной работы больших команд разработчиков с различным уровнем знаний в области программирования. Низкоуровневый код подвержен множеству едва уловимых багов, которые в большинстве других языков можно обнаружить только путем тщательного тестирования и скрупулезного анализа кода опытными разработчиками. В Rust компилятор играет роль привратника, отказываясь компилировать код с такими неуловимыми багами, включая ошибки конкурентности. Работая бок о бок с компилятором, команда может сосредоточиться не на отлове багов, а на логике программы.

Rust также привносит современные инструменты разработчика в мир системного программирования:

- Включенный в комплект менеджер зависимостей и инструмент сборки Cargo делает добавление, компиляцию и управление зависимостями, безболезненными и согласованными во всей экосистеме Rust.
- Инструмент форматирования исходного кода Rustfmt обеспечивает согласованный стиль написания кода для всех разработчиков.
- Rust Language Server поддерживает интеграцию со средой разработки (IDE), обеспечивая автодополнение кода и построчные сообщения об ошибках.

Используя эти и другие инструменты в экосистеме Rust, разработчики могут продуктивно писать код на системном уровне.

Студенты

Rust предназначен студентам и тем, кто заинтересован в изучении системных понятий. Используя Rust, многие люди узнали о разработке операционных систем. Сообщество является очень гостеприимным и с удовольствием отвечает на вопросы студентов. Благодаря таким проектам, как эта книга, коллективы разработчиков на языке Rust хотят сделать системные понятия доступнее для большего числа людей, в особенности для новичков в программировании.

Компании

Сотни компаний, крупных и малых, используют Rust в производстве для различных задач. Эти задачи включают инструменты командной строки, веб-службы, инструменты DevOps, встраиваемые устройства, аудио- и видеоанализ и транскодирование, криптовалюты, биоинформатику, поисковые системы, приложения Интернета вещей, машинное обучение и даже основные компоненты веб-браузера Firefox.

Разработчики открытого исходного кода

Язык Rust предназначен людям, которые хотят развивать Rust, сообщество, инструменты разработчика и библиотеки. Мы хотели бы, чтобы вы внесли свой вклад в развитие языка.

Люди, ценящие скорость и стабильность

Язык Rust предназначен тем, кто жаждет скорости и стабильности в языке. Под скоростью мы подразумеваем скорость программ, которые вы можете создавать с помощью языка Rust, и скорость, с которой язык Rust позволяет вам их писать. Проверки компилятора языка Rust обеспечивают стабильность за счет добавления функциональности и рефакторинга. Это контрастирует с хрупким привычным кодом на языках, где подобных проверок нет, — и такое положение дел разработчики часто боятся изменять. Стремясь к нулевым по стоимости абстракциям, более высокоуровневым языковым средствам, которые компилируются в более низкоуровневый код так же быстро, как код, написанный вручную, язык Rust стремится сделать безопасный код в том числе и быстрым.

Разработчики Rust также надеются оказывать поддержку многим другим пользователям. Перечисленные здесь лица являются лишь частью наиболее заинтересованных в языке. В целом главная цель Rust — устранить компромиссы, которые программисты принимали на протяжении десятилетий, обеспечив безопасность и производительность, скорость и эргономику. Дайте Rust шанс и посмотрите, подходят ли вам его возможности.

Для кого эта книга

Мы предполагаем, что вы писали код на другом языке программирования, но не делаем никаких допущений относительно того, на каком именно. Мы постарались сделать этот материал доступным для тех, кто имеет широкий спектр навыков программирования. Мы не будем тратить время на разговоры о том, что такое программирование. Если вы в программировании абсолютный новичок, то для начала прочтите введение в программирование.

Как пользоваться этой книгой

В общем-то, авторы этой книги исходят из того, что вы читаете ее последовательно, от начала до конца. Последующие главы строятся на понятиях предыдущих глав, и в начальных главах мы можем не углубляться в детали по конкретной теме; обычно мы возвращаемся к этой теме в дальнейшем.

В этой книге вы найдете два типа глав: концептуальные и проектные. В концептуальных главах вы будете усваивать тот или иной аспект языка. Мы вместе будем создавать небольшие программы, применяя то, что вы уже усвоили. Главы 2, 12 и 20 посвящены разработке проектов, остальные главы — концептуальные.

В главе 1 рассказано, как установить Rust, написать программу «Hello, World!» и использовать пакетный менеджер и инструмент Cargo. Глава 2 представляет собой практическое введение в язык Rust. Здесь мы рассмотрим понятия с точки зрения высокоуровневого языка, а в последующих главах приведем дополнительные подробности. Если вы хотите сразу же приступить к практике, то сможете это сделать. Можно даже пропустить главу 3, в которой рассматриваются средства языка Rust, аналогичные средствам других языков программирования, сразу перейти к главе 4 и узнать о системе владения в Rust. Но если вы дотошны и предпочитаете разбирать каждую деталь, прежде чем переходить к следующей, то можете пропустить главу 2, перейти к главе 3, а затем вернуться к главе 2, когда захотите поработать над проектом. Так вы сможете применить знания, которые освоили.

В главе 5 обсуждаются структуры и методы, а в главе 6 рассматриваются перечисления, выражения `match` и конструкция управления потоком `if let`. Вы будете использовать структуры и перечисления для создания в языке Rust настраиваемых типов.

В главе 7 вы узнаете о системе модулей и правилах конфиденциальности для выстраивания организационной структуры вашего кода и его публичном интерфейсе программирования приложений (API). В главе 8 обсуждаются некоторые часто встречающиеся структуры сбора данных, обеспечиваемые стандартной библиотекой, такие как векторы, строки и хеш-отображения. В главе 9 изучаются философия и методы обработки ошибок.

В главе 10 мы погрузимся в обобщения, типажи и жизненные циклы, которые дают вам возможность определять код, применимый к нескольким типам. Глава 11 полностью посвящена тестированию, которое даже несмотря на гарантии безопасности языка Rust является необходимым для обеспечения правильной логики программы. В главе 12 мы построим собственную реализацию подмножества функциональности инструмента командной строки grep, которая ищет текст внутри файлов. Для этого мы воспользуемся многими понятиями, которые обсуждаются в предыдущих главах.

В главе 13 рассматриваются замыкания и итераторы — средства, которые восходят к функциональным языкам программирования. В главе 14 мы изучим Cargo подробнее и расскажем о лучших практических приемах обмена библиотеками с другими разработчиками. В главе 15 обсуждаются умные указатели, которые обеспечивает стандартная библиотека, и типажи, которые гарантируют их функциональность.

В главе 16 мы рассмотрим разные модели конкурентного программирования и поговорим о том, как Rust помогает вам безбоязненно программировать в множестве потоков исполнения. Глава 17 обращается к сопоставлению идиом Rust с принципами объектно-ориентированного программирования, с которыми вы, возможно, знакомы.

Глава 18 представляет собой справочный материал о паттернах и сопоставлении с паттернами, которые являются мощными способами выражения идей во всех программах на языке Rust. Глава 19 содержит ряд дополнительных тем, представляющих интерес, включая небезопасный код Rust, макрокоманды и другие сведения о типажах, типах, функциях и замыканиях.

В главе 20 мы осуществим проект, в котором выполним реализацию низкоуровневого многопоточного сервера!

Наконец, несколько приложений в конце книги содержат полезную информацию о языке в справочном формате. В приложении А приводятся ключевые слова языка Rust, в приложении Б рассказывается об операторах и символах языка Rust, в приложении В рассматриваются генерируемые типажи, предусмотренные стандартной библиотекой, в приложении Г приводятся некоторые полезные инструменты разработчика, а в приложении Д даются пояснения по поводу редакций языка Rust.

Просто невозможно прочитать эту книгу неправильно: если вы хотите пропустить что-то, пропускайте! В случае если вы почувствуете какую-то путаницу, то, возможно, вам придется вернуться к предыдущим главам. Короче, делайте все, что вам подходит.

Важная часть процесса усвоения языка Rust — научиться читать сообщения об ошибках, выводимые на экран компилятором: они будут направлять вас к рабочему коду. В связи с этим мы приведем много примеров, которые не компилируются вместе с сообщением об ошибке, которое компилятор покажет вам в каждой

ситуации. Знайте, что если вы введете и запустите выполнение случайного примера, то он может не скомпилироваться! Обязательно прочтите окружающий текст, чтобы увидеть, что пример, который вы пытаетесь выполнить, является неизбежно ошибочным. В большинстве ситуаций мы будем вести вас к правильной версии любого кода, который не компилируется.

Ресурсы

Эта книга распространяется по лицензии открытого исходного кода и текста. Если вы нашли ошибку, то, пожалуйста, не стесняйтесь добавить сообщение о проблеме или отправить запрос на включение внесенных изменений на GitHub по адресу https://github.com/rust-lang/book/. Более подробную информацию смотрите в CONTRIBUTING.md по адресу https://github.com/rust-lang/book/blob/master/CONTRIBUTING.md.

Исходный код примеров из этой книги, список опечаток и другая информация доступны по адресу https://www.nostarch.com/Rust2018/.

От издательства

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

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1 Начало работы

Давайте начнем наше «ржавое»[1] путешествие по языку Rust! Здесь есть чему поучиться, но любое путешествие нужно с чего-то начать. В этой главе мы обсудим следующее:

- Установка языка Rust в Linux, macOS и Windows.
- Написание программы, которая выводит `Hello, World!`.
- Использование `cargo`, пакетного менеджера и системы сборки языка Rust.

Установка

Первый шаг — это установка языка Rust. Мы скачаем Rust через `rustup`, инструмент командной строки для управления версиями языка Rust и связанными инструментами. Для скачивания вам понадобится подключение к интернету.

ПРИМЕЧАНИЕ

Если по какой-либо причине вы предпочитаете не использовать инструмент `rustup`, то, пожалуйста, обратитесь к веб-странице установки языка Rust по адресу https://www.rust-lang.org/tools/install/, чтобы ознакомиться с другими вариантами.

Следующие шаги позволяют установить последнюю стабильную версию компилятора языка Rust. Гарантии стабильности Rust обеспечивают, что все примеры из книги, которые компилируются, будут продолжать компилироваться с более новыми версиями языка Rust. Выходные данные могут немного отличаться от версии к версии, поскольку язык Rust часто улучшает сообщения об ошибках и предупреждения. Другими словами, любая новая стабильная версия языка Rust,

[1] Язык программирования Rust получил свое название от грибов семейства ржавчинные (англ. rust fungi; отсюда и перевод слова rusty как «ржавый»), а также от слова «robust» («надежный»). — *Здесь и далее примеч. пер.*

которую вы устанавливаете с помощью этих инструкций, должна работать с содержимым этой книги.

ОБОЗНАЧЕНИЯ КОМАНДНОЙ СТРОКИ

В этой главе и на протяжении всей книги мы покажем несколько команд, используемых в терминале. Все командные строки, которые вы должны вводить в терминале, начинаются с символа $. Вам не нужно вводить сам символ — он указывает на начало каждой команды. Строки, которые не начинаются с $, обычно показывают выходные данные предыдущей команды. В дополнение к этому, в примерах, относящихся к PowerShell, вместо $ будет использоваться символ >.

Установка инструмента rustup в Linux или macOS

Если вы используете Linux или macOS, то откройте терминал и введите следующую команду:

```
$ curl https://sh.rustup.rs -sSf | sh
```

Указанная команда скачивает скрипт и запускает установку инструмента `rustup`, который устанавливает последнюю стабильную версию языка Rust. Возможно, вам будет предложено ввести пароль. Если установка прошла успешно, то появится следующая строчка («Rust установлен. Отлично!»):

```
Rust is installed now. Great!
```

Если хотите, то можете самостоятельно скачать этот скрипт и проверить его перед запуском.

Скрипт установки автоматически добавит язык Rust в системный путь после следующего входа в систему. Если вместо перезагрузки терминала вы хотите сразу же начать использовать язык Rust, то в командной строке выполните следующую команду, чтобы добавить язык Rust в системный путь вручную:

```
$ source $HOME/.cargo/env
```

Как вариант, вы можете добавить следующую строку в свой профиль ~/.bash_profile:

```
$ export PATH="$HOME/.cargo/bin:$PATH"
```

В дополнение к этому вам понадобится какой-то редактор связей (линкер). Скорее всего, он уже установлен, но когда вы пытаетесь скомпилировать программу Rust и видите ошибки, которые говорят о том, что не получается исполнить редактор связей, это означает, что в вашей системе редактор связей не установлен и вам нужно установить его вручную. Компиляторы C, как правило, поставляются вместе с нужным редактором связей. Сверьтесь с документацией вашей платформы

и выясните, как устанавливать компилятор C. Кроме того, некоторые распространенные пакеты Rust зависят от кода C, и им требуется компилятор C. Поэтому, возможно, стоит установить его сейчас.

Установка инструмента rustup в Windows

В случае с Windows перейдите в раздел https://www.rust-lang.org/tools/install/ и следуйте инструкциям по установке языка Rust. В какой-то момент установки вы получите сообщение, объясняющее, что вам также понадобятся инструменты сборки C++ для Visual Studio 2013 или более поздней версии. Самый простой способ приобрести инструменты сборки — загрузить их для Visual Studio 2019 по адресу https://www.visualstudio.com/downloads/#build-tools-for-visual-studio-2019 в разделе «Другие инструменты и платформы».

В остальной части этой книги используются команды, которые работают как в интерпретаторе командной строки cmd.exe, так и в оболочке PowerShell. Если будут определенные различия, то мы объясним, какой из этих инструментов использовать.

Обновление и деинсталляция

После того как вы инсталлировали язык Rust инструментом `rustup`, его легко можно обновить до последней версии. Из командной строки выполните следующий ниже скрипт обновления:

```
$ rustup update
```

Для того чтобы деинсталлировать язык Rust и инструмент rustup, выполните следующий ниже скрипт деинсталляции из командной строки:

```
$ rustup self uninstall
```

Устранение неисправностей

Для того чтобы проверить правильность установки языка Rust, откройте оболочку и введите следующую команду:

```
$ rustc --version
```

Вы должны увидеть номер версии, хеш фиксации и дату фиксации последней стабильной версии, выпущенной в следующем формате:

```
rustc x.y.z (abcabcabc yyyy-mm-dd)
```

Если вы видите эту информацию, значит, вы успешно инсталлировали язык Rust! Если вы не видите эту информацию и находитесь в Windows, то проверьте, что язык Rust содержится в вашей системной переменной `%PATH%`. Если все

правильно, а Rust по-прежнему не работает, то вот несколько вариантов, где можно получить помощь. Самый простой — это канал `#beginners` на официальном веб-сайте языка Rust в мессенджере Discord по адресу https://discord.gg/rust-lang. Там вы можете пообщаться с другими растианами (растианин, на англ. *rustacean*, произносится как «растейшен», — это наше забавное самоназвание), которые вам непременно помогут. Другие замечательные ресурсы включают форум пользователей Users по адресу https://users.rust-lang.org/ и на веб-сайте Stack Overflow по адресу http://stackoverflow.com/questions/tagged/rust/.

Локальная документация

Установщик также содержит локальную копию документации, благодаря чему вы можете читать ее офлайн. Выполните команду `rustup doc`, и локальная копия документации откроется в браузере.

Всякий раз, когда тип или функция предусмотрены стандартной библиотекой и вы не уверены, что она делает или как ее использовать, то для ответов на эти вопросы используйте документацию об интерфейсе программирования приложений (API).

Здравствуй, Мир!

Теперь, когда вы установили язык Rust, давайте напишем первую программу Rust. Традиционно при изучении нового языка пишется небольшая программа, которая выводит на экране текст `Hello, World!`, и поэтому поступим так же.

ПРИМЕЧАНИЕ

Авторы книги исходят из базового знакомства с командной строкой. Язык Rust не предъявляет особых требований к редактированию или инструментам, а также к тому, где находится ваш код, поэтому, если вместо командной строки вы предпочитаете использовать интегрированную среду разработки (IDE), то, пожалуйста, используйте свою любимую IDE. Многие IDE теперь имеют некоторую степень поддержки языка Rust; для получения подробностей перепроверьте документацию интегрированной среды разработки. В последнее время команда разработчиков языка Rust сосредоточилась на обеспечении отличной поддержки IDE, и в этом направлении был достигнут значительный прогресс!

Создание каталога проектов

Вы начнете с создания каталога для хранения кода Rust. Неважно, где располагается код, но для упражнений и проектов из этой книги мы предлагаем создать каталог проектов projects в вашем домашнем каталоге и хранить там все проекты.

Откройте терминал и введите следующие команды, чтобы создать каталог projects и каталог для проекта Hello, world! в каталоге проектов.

Для Linux, macOS и оболочки PowerShell в Windows введите следующее:

```
$ mkdir ~/projects
$ cd ~/projects
$ mkdir hello_world
$ cd hello_world
```

Для интерпретатора командной строки `cmd` в Windows введите:

```
> mkdir "%USERPROFILE%\projects"
> cd /d "%USERPROFILE%\projects"
> mkdir hello_world
> cd hello_world
```

Написание и выполнение программы Rust

Далее создайте новый исходный файл и назовите его main.rs. Файлы Rust всегда заканчиваются расширением .rs. Если в имени файла вы используете более одного слова, то используйте нижнее подчеркивание, чтобы их отделить. Например, используйте hello_world.rs вместо helloworld.rs.

Теперь откройте файл main.rs, который вы только что создали, и введите код листинга 1.1.

Листинг 1.1. Программа, которая выводит Hello, World!

main.rs

```
fn main() {
    println!("Hello, World!");
}
```

Сохраните файл и вернитесь в окно вашего терминала. В Linux или macOS введите следующие команды для компиляции и выполнения файла:

```
$ rustc main.rs
$ ./main
Hello, World!
```

В Windows вместо команды ./main введите команду .\main.exe:

```
> rustc main.rs
> .\main.exe
Hello, World!
```

Независимо от вашей операционной системы в терминале должен быть выведен строковый литерал *Hello, World!*. Если вы не видите этих данных, то обратитесь к разделу «Устранение неполадок» для получения справки.

Если `Hello, World!` все-таки напечаталось, то примите наши поздравления! Вы официально написали программу на языке Rust. И значит, вы стали программистом на языке Rust — добро пожаловать!

Анатомия программы на языке Rust

Давайте подробно рассмотрим, что только что произошло в программе *Hello, World!*. Вот первый элемент пазла:

```
fn main() {

}
```

Эти строки кода определяют функцию на языке Rust. Функция `main` является особенной: это всегда первый код, который выполняется в каждой исполняемой программе Rust. Первая строка кода объявляет функцию с именем `main`, которая не имеет параметров и ничего не возвращает. Если бы имелись параметры, то они бы вошли внутрь скобок, `()`.

Также обратите внимание на то, что тело функции заключено в фигурные скобки, `{}`. В Rust они требуются вокруг всех тел функций. Правильно размещать открывающую фигурную скобку на той же строке кода, что и объявление функции, добавив один пробел между ними.

На момент написания этой книги инструмент автоматического форматирования под названием rustfmt находился на стадии разработки. Если в своих проектах Rust вы хотите придерживаться стандартного стиля, то rustfmt отформатирует код в определенном стиле. Коллектив Rust планирует в итоге включить этот инструмент в стандартный дистрибутив Rust, подобно rustc. Поэтому в зависимости от того, когда вы читаете эту книгу, он уже может быть установлен на вашем компьютере. Для получения более подробной информации ознакомьтесь с документацией онлайн.

Внутри функции `main` находится следующий код:

```
println!("Hello, World!");
```

Эта строка кода выполняет всю работу в этой маленькой программе: она выводит текст на экране. Здесь следует отметить четыре важные детали. Во-первых, стиль языка Rust предусматривает не табуляцию, а отступ из четырех пробелов.

Во-вторых, инструкция `println!` вызывает макрокоманду языка Rust. Если бы вместо этого она вызвала функцию, то мы бы ввели ее как `println` (без `!`). Мы обсудим макрокоманды языка Rust подробнее в главе 19. Пока же вам просто нужно знать: использование `!` означает, что вы вызываете макрокоманду вместо обычной функции.

В-третьих, вы видите строковый литерал `"Hello, world!"`. Мы передаем его в качестве аргумента макрокоманды `println!`, и этот строковый литерал выводится на экран.

В-четвертых, мы заканчиваем строку кода точкой с запятой (`;`). Она указывает на то, что это выражение закончено и готово начаться следующее. Большинство строк кода Rust заканчиваются точкой с запятой.

Компиляция и выполнение являются отдельными шагами

Вы выполнили только что созданную программу, поэтому давайте рассмотрим каждый шаг в этом процессе.

Перед выполнением программы Rust вам необходимо ее скомпилировать с помощью компилятора языка Rust, введя команду `rustc` и передав ей имя вашего исходного файла, как показано ниже:

```
$ rustc main.rs
```

Если у вас есть опыт работы с C или C++, то вы заметите, что команда похожа на `gcc` или `clang`. После успешной компиляции Rust выводит двоичный исполняемый файл.

В Linux, macOS и PowerShell в Windows вы увидите исполняемый файл, введя в командной строке команду `ls`. В Linux и macOS вы увидите два файла. С помощью PowerShell в Windows вы увидите те же три файла, что и при использовании интерпретатора командной строки cmd.

```
$ ls
main main.rs
```

С помощью интерпретатора командной строки cmd в Windows можно ввести следующее:

```
> dir /B %= опция /B говорит о том, что нужно показывать только имена файлов =%
main.exe
main.pdb
main.rs
```

Здесь вы видите файл исходного кода с расширением .rs, исполняемый файл (main.exe в Windows, но main на всех других платформах) и при использовании Windows файл, содержащий отладочную информацию с расширением .pdb. Таким образом, вы выполняете файл main, или main.exe, вот так:

```
$ ./main # или .\main.exe в Windows
```

Если бы программа main.rs была вашей программой *Hello, World!*, то в терминале было бы выведено `Hello, World!`.

Если вы лучше знакомы с динамическим языком, таким как Ruby, Python или JavaScript, то вы, возможно, не привыкли к компиляции и выполнению программы как к отдельным шагам. Rust — это язык с предварительным компилирова-

нием (Ahead-of-Time-компилированием), то есть вы можете компилировать программу и передавать исполняемый файл кому-то другому и он может выполнять его даже без установки языка Rust. Если же вы дадите кому-то файл .rb, .py или .js, то этот кто-то должен иметь установленную реализацию соответственно языка Ruby, Python или JavaScript. Но на этих языках для компиляции и выполнения программы вам потребуется только одна команда. В дизайне языков все является компромиссом.

Компиляция только с помощью rustc отлично подходит для простых программ, но по мере развития вашего проекта вам захочется управлять всеми вариантами и упрощать распространение вашего кода. Далее мы познакомим вас с инструментом Cargo, который поможет писать реальные программы языка Rust.

Здравствуй, Cargo!

Cargo — это система сборки и пакетный менеджер языка Rust. Большинство растиан используют этот инструмент для управления проектами Rust, потому что Cargo выполняет много работы за вас, в частности построение кода, скачивание библиотек, от которых зависит код, и построение этих библиотек. (Библиотеки, которые нужны коду, мы называем зависимостями.)

Самые простые программы Rust, как та, которую мы уже написали, не имеют никаких зависимостей. Поэтому, если бы мы построили проект *Hello, World!* с помощью пакетного менеджера Cargo, то он бы задействовал только ту часть Cargo, которая занимается построением вашего кода. При написании более сложных программ Rust вы будете добавлять зависимости, и если вы начнете проект с помощью Cargo, то добавлять зависимости будет намного проще.

Поскольку подавляющее большинство проектов Rust используют пакетный менеджер Cargo, в других главах этой книги мы исходим из того, что вы тоже используете Cargo. Если вы пользовались официальными установщиками, описанными в разделе «Установка», то Cargo устанавливается в комплекте с языком Rust. Если же вы инсталлировали Rust каким-то другим способом, то проверьте наличие установленного Cargo, введя в терминал следующее:

```
$ cargo --version
```

Если вы видите номер версии, то она у вас есть! Если же вы видите ошибку, например `command not found` («команда не найдена»), то обратитесь к документации вашего метода установки, чтобы выяснить, как установить Cargo отдельно.

Создание проекта с помощью Cargo

Давайте создадим новый проект с использованием Cargo и посмотрим, чем он отличается от нашего первоначального проекта *Hello, World!*. Вернитесь в каталог

проектов projects (или туда, где вы решили хранить свой код). Затем в любой операционной системе выполните следующие команды:

```
$ cargo new hello_cargo
$ cd hello_cargo
```

Первая команда создает новый каталог с именем *hello_cargo*. Мы назвали проект hello_cargo, и Cargo создает файлы в каталоге с тем же именем.

Перейдите в каталог *hello_cargo* и выведите список файлов. Вы увидите, что Cargo сгенерировал для нас два файла и одну папку: файл Cargo.toml и каталог src с файлом main.rs внутри. Он также инициализировал новый репозиторий Git вместе с файлом .gitignore.

ПРИМЕЧАНИЕ

Git — это распространенная система управления версиями. С помощью флага `--vcs` вы можете изменить команду `cargo new`, чтобы использовать другую систему управления версиями или вообще не использовать никакой системы. Выполните команду `cargo new --help`, чтобы увидеть имеющиеся варианты.

Откройте файл Cargo.toml в любом текстовом редакторе. Указанный файл должен быть похожим на код в листинге 1.2.

Листинг 1.2. Содержимое Cargo.toml, сгенерированное командой cargo new

Cargo.toml

```
[package]
name = "hello_cargo"
version = "0.1.0"
authors = ["Ваше имя <you@example.com>"]
edition = "2018"

[dependencies]
```

Этот файл имеет формат TOML («очевидный минимальный язык Тома», от англ. *Tom's Obvious, Minimal Language*), который является конфигурационным форматом Cargo.

Первая строка файла, `[package]`, является заголовком раздела, который указывает на то, что последующие инструкции настраивают пакет. По мере добавления информации в этот файл мы будем добавлять и другие разделы.

Следующие четыре строки задают информацию о конфигурации, необходимую для компиляции программы: название, версия, имя автора и используемая редакция языка Rust. Cargo получает информацию о вашем имени и электронной почте из вашей среды, поэтому, если эта информация неверная, исправьте ее сейчас и сохраните файл. Мы поговорим о ключе `edition` в приложении Д в конце книги.

Последняя строка, [dependencies], является началом раздела, в котором вы перечисляете все зависимости проекта. В языке Rust пакеты с исходным кодом называются упаковками (crate). Для этого проекта нам не нужны другие упаковки, но они нам понадобятся в первом проекте в главе 2, и поэтому мы будем использовать этот раздел зависимостей.

Теперь откройте файл src/main.rs и посмотрите:

src/main.rs

```
fn main() {
    println!("Hello, World!");
}
```

Cargo сгенерировал для вас программу *Hello, World!*, такую же, как мы написали в листинге 1.1! Пока что различия между нашим предыдущим проектом и проектом, который генерируется Cargo, заключаются в том, что Cargo поместил код в каталог src и что у нас есть конфигурационный файл Cargo.toml, который находится в верхнем каталоге.

Использование Cargo подразумевает, что ваши файлы с исходным кодом будут находиться в каталоге src. Верхнеуровневый каталог проекта предназначен только для файлов README, информации о лицензии, конфигурационных файлов и всего остального, что не связано с вашим кодом. Использование Cargo помогает вам организовывать проекты. Здесь есть место для всего, и все находится на своем месте.

Если вы начали проект, в котором не используется пакетный менеджер Cargo, как было в проекте *Hello, World!*, то вы можете сконвертировать его в проект, в котором будет использоваться пакетный менеджер Cargo. Переместите код проекта в каталог src и создайте соответствующий файл Cargo.toml.

Построение проекта Cargo и его выполнение

Теперь давайте посмотрим, чем отличается ситуация, когда мы создаем и выполняем программу *Hello, World!* с помощью Cargo. Из каталога hello_cargo постройте свой проект, введя следующую команду:

```
$ cargo build
    Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
     Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs
```

Эта команда создает исполняемый файл, правда, не в вашем текущем каталоге, а в target/debug/hello_cargo (или target\debug\hello_cargo.exe в Windows). Вы можете выполнить исполняемый файл с помощью следующей команды:

```
$ ./target/debug/hello_cargo # или .\target\debug\hello_cargo.exe в Windows
Hello, world!
```

Если все сделано правильно, то терминал должен вывести `Hello, World!`. Выполнение команды `cargo build` в первый раз также приводит к тому, что Cargo создает новый файл на верхнем уровне — Cargo.lock. Этот файл отслеживает точные версии зависимостей в проекте. Данный проект не имеет зависимостей, поэтому указанный файл немного разрежен. Вам никогда не придется изменять этот файл вручную, Cargo управляет его содержимым за вас.

Мы только что построили проект с помощью команды `cargo build` и выполнили его с помощью команды `./target/debug/hello_cargo`, но мы также можем использовать команду `cargo run` для компиляции кода и последующего выполнения результирующего исполняемого файла, и все это в одной команде:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/hello_cargo`
Hello, World!
```

Обратите внимание, что на этот раз мы не увидели данных, указывающих на то, что Cargo компилировал `hello_cargo`. Cargo выяснил, что файлы не изменились, поэтому он просто запустил двоичный файл. Если бы вы модифицировали исходный код, Cargo перестроил бы проект перед его выполнением, и тогда на выходе вы бы увидели вот это:

```
$ cargo run
    Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
     Finished dev [unoptimized + debuginfo] target(s) in 0.33 secs
      Running `target/debug/hello_cargo`
Hello, World!
```

Cargo также предоставляет команду `cargo check`. Эта команда быстро проверяет ваш код, чтобы убедиться в его компилируемости, но не создает исполняемый файл:

```
$ cargo check
    Checking hello_cargo v0.1.0 (file:///projects/hello_cargo)
     Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
```

Вы, наверное, захотите иметь исполняемый файл? Часто команда `cargo check` выполняется намного быстрее, чем команда `cargo build`, потому что она пропускает этап порождения исполняемого файла. Если вы непрерывно проверяете свою работу во время написания кода, то использование команды `cargo check` ускорит этот процесс! По этой причине многие растиане периодически выполняют команду `cargo check`, когда пишут программу, чтобы убедиться, что она компилируется. Затем, когда они готовы использовать исполняемый файл, они выполняют команду `cargo build`.

Давайте повторим то, что мы уже узнали о Cargo:

- Мы создаем проект, используя команды `cargo build` или `cargo check`.
- Мы создаем и выполняем проект в один прием, используя команду `cargo run`.
- Вместо сохранения результата построения в том же каталоге, что и наш код, Cargo сохраняет его в каталоге *target/debug*.

Дополнительным преимуществом использования пакетного менеджера Cargo является то, что команды одинаковы независимо от того, в какой операционной системе вы работаете. Таким образом, в этой книге мы больше не будем предоставлять инструкции конкретно для Linux и macOS по сравнению с Windows.

Сборка для релиза

Когда ваш проект будет окончательно готов к релизу, вы можете использовать команду `cargo build --release` для его компиляции с оптимизациями. Эта команда создаст исполняемый файл в *target/release* вместо *target/debug*. Оптимизации ускорят работу кода Rust, но их включение увеличивает время, необходимое для компиляции программы. Вот почему существуют два разных профиля: один для разработки, когда вы хотите перестраивать программу быстро и часто, а другой — для построения окончательной программы для пользователя, которая не будет перестраиваться повторно и будет работать как можно быстрее. Если вы проводите сравнительный анализ времени выполнения вашего кода, то обязательно выполните команду `cargo build --release` и сравните с исполняемым файлом в *target/release*.

Cargo как общепринятое средство

В случае с простыми проектами у Cargo нет какой-то большой ценности по сравнению с компилятором `rustc`. Но когда ваши программы усложнятся, он докажет свою ценность. Имея сложные проекты, состоящие из множества упаковок, гораздо проще передать обязанности по координированию сборки пакетному менеджеру Cargo.

Несмотря на то что проект `hello_cargo` прост, он теперь использует бóльшую часть реального инструментария, который вы будете применять в остальной части вашей Rust-овской карьеры. По сути дела, работая над любыми существующими проектами, вы можете использовать следующие команды, чтобы получать полную копию кода с помощью Git, переходить в каталог этого проекта и выполнять его построение:

```
$ git clone someurl.com/некийпроект
$ cd некийпроект
$ cargo build
```

Для получения дополнительной информации о пакетном менеджере Cargo ознакомьтесь с его документацией по адресу https://doc.rust-lang.org/cargo/.

Итоги

Неплохое начало путешествия по языку Rust! В этой главе вы узнали, как:

- Устанавливать последнюю стабильную версию языка Rust с помощью инструмента `rustup`.
- Обновлять язык Rust до более новой версии.
- Открывать локально установленную документацию.
- Написать и выполнить программу Hello, World!, используя компилятор `rustc` напрямую.
- Создавать и выполнять новый проект с использованием обозначений пакетного менеджера Cargo.

Сейчас самое подходящее время создать программу посерьезней, чтобы привыкнуть к чтению и написанию кода на языке Rust. И поэтому в главе 2 мы создадим программу игры-угадайки. Если вы предпочитаете начать с изучения того, как в Rust работают распространенные концепции программирования, то обратитесь к главе 3, а затем вернитесь к главе 2.

2 Программирование игры-угадайки

Давайте же сразу перейдем к языку Rust и разработаем вместе практический проект! Эта глава познакомит вас с несколькими распространенными понятиями Rust и покажет, как их использовать в реальной программе. Вы узнаете о `let`, `match`, методах, связанных функциях, использовании внешних упаковок и многом другом! В последующих главах мы изучим эти идеи подробнее. В этой главе вы будете заниматься основами языка.

Мы выполним реализацию классической задачи для начинающих программистов — игру-угадайку. Вот как она работает: программа сгенерирует случайное целое число от 1 до 100. Затем предложит игроку отгадать число и ввести его значение. После ввода числа-догадки программа сообщит, является ли оно слишком маленьким либо слишком большим. Если игрок правильно отгадает число, то игра покажет поздравительное сообщение и завершит работу.

Настройка нового проекта

Для настройки нового проекта перейдите в каталог проектов projects, созданный в главе 1, и сделайте новый проект с помощью Cargo:

```
$ cargo new guessing_game
$ cd guessing_game
```

Первая команда `cargo new` берет имя проекта (`guessing_game`) в качестве первого аргумента. Вторая команда меняет каталог нового проекта.

Взгляните на сгенерированный файл Cargo.toml:

Cargo.toml

```
[package]
name = "guessing_game"
```

```
version = "0.1.0"
authors = ["Ваше имя <you@example.com>"]
edition = "2018"

[dependencies]
```

Если сведения об авторе, полученные из вашей среды, неверны, то исправьте их в файле и сохраните его снова.

Как вы увидели в главе 1, команда `cargo new` генерирует программу «Hello, World!». Проверьте файл src/main.rs:

src/main.rs

```
fn main() {
    println!("Hello, World!");
}
```

Теперь давайте скомпилируем эту программу «Hello, World!» и выполним ее, используя команду `cargo run`:

```
$ cargo run
    Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
      Running `target/debug/guessing_game`
Hello, World!
```

Команда `run` бывает очень кстати, когда требуются быстрые итерации по проекту, как в этой игре, где нужно будет оперативно тестировать каждую итерацию перед переходом к следующей.

Снова откройте файл src/main.rs. Вы будете писать весь код в этот файл.

Обработка загаданного числа

В первой части программы игры на угадывание пользователю нужно будет ввести данные, программа обработает эти данные и проверит, что они в корректной форме. Вначале мы даем игроку ввести загаданное число. Наберите код из листинга 2.1 в файл src/main.rs.

Листинг 2.1. Код, который получает от пользователя загаданное число и выводит его

src/main.rs

```
use std::io;

fn main() {
    println!("Угадайте число!");

    println!("Пожалуйста, введите свою догадку.");
```

```
    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Не получилось прочитать строку");

    println!("Вы загадали: {}", guess);
}
```

Этот код содержит много информации, поэтому давайте посмотрим на каждую строку отдельно. Для получения данных от пользователя и затем распечатки результата нам нужно ввести библиотеку `io` (библиотеку ввода-вывода) в область видимости. Библиотека `io` берется из стандартной библиотеки (носящей название `std`):

```
use std::io;
```

По умолчанию Rust в прелюдии вводит в область видимости каждой программы всего несколько типов. Если тип, который вы хотите использовать, не находится в прелюдии, то вы должны ввести этот тип в область видимости с помощью инструкции `use`. Использование библиотеки `std::io` обеспечивает вас рядом полезных средств, включая способность принимать данные от пользователя.

Как вы видели в главе 1, функция `main` является точкой входа в программу:

```
fn main() {
```

Синтаксис `fn` объявляет новую функцию, круглые скобки `()` указывают на отсутствие параметров, а фигурная скобка `{` начинает тело функции.

Как вы также узнали из главы 1, `println!` является макрокомандой, которая выводит строковый литерал на экране:

```
println!("Угадайте число!");

println!("Пожалуйста, введите свою догадку.");
```

Этот код выводит подсказку с указанием названия игры и запрашивает у пользователя данные.

Хранение значений с помощью переменных

Далее мы создадим место для хранения введенных пользователем данных. Это делается так:

```
let mut guess = String::new();
```

Теперь программа становится интересной! В этой маленькой строке кода много чего происходит. Обратите внимание на инструкцию `let`, которая используется для создания переменной. Вот еще один пример:

```
let foo = bar;
```

Эта строка кода создает новую переменную с именем `foo` и привязывает ее к значению переменной `bar`. В Rust переменные по умолчанию являются неизменяемыми. Мы подробно обсудим это понятие в разделе «Переменные и изменяемость» (с. 61). В следующем примере показано, как использовать ключевое слово `mut` перед именем переменной, чтобы сделать переменную изменяемой:

```
let foo = 5; // неизменяемая
let mut bar = 5; // изменяемая
```

ПРИМЕЧАНИЕ

Синтаксис `//` начинает комментарий, который продолжается до конца строки кода. Все, что находится в комментариях, которые обсуждаются в главе 3 подробнее, язык Rust игнорирует.

Давайте вернемся к программе игры-угадайки. Теперь вы знаете, что `let mut guess` вводит в программу изменяемую переменную с именем `guess`. По другую сторону знака равенства (`=`) находится значение, к которому привязана переменная `guess`, являющееся результатом вызова функции `String::new`, возвращающей новый экземпляр типа `String`. `String` — это строковый тип, предусмотренный стандартной библиотекой, а именно наращиваемый фрагмент текста в кодировке UTF-8.

Синтаксис `::` в строке кода `::new` указывает на то, что `new` является функцией, связанной с типом `String`. Связанная функция реализуется в типе, в данном случае в типе `String`, а не в конкретном экземпляре типа `String`. В некоторых языках она называется статическим методом.

Функция `new` создает новую пустую строку. Вы найдете функцию `new` во многих типах, потому что так принято называть функцию, которая делает новое значение какого-либо рода.

Подводя итог, строка кода `let mut guess = String::new();` создала изменяемую переменную, которая в данный момент привязана к новому пустому экземпляру типа `String`. Фух!

Напомним, что мы включили в состав функциональность ввода-вывода из стандартной библиотеки, указав `std::io;` в первой строке программы. Теперь мы вызовем функцию `stdin` из модуля `io`:

```
io::stdin().read_line(&mut guess)
    .expect("Не получилось прочитать строку");
```

Если бы мы не разместили в начале программы строку `use std::io`, то мы бы могли записать вызов этой функции как `std::io::stdin`. Функция `stdin` возвращает экземпляр `std::io::Stdin`, то есть тип, который представляет собой дескриптор стандартного ввода данных для терминала.

Следующая часть кода `.read_line(&mut guess)` вызывает метод `read_line`, заданный для дескриптора стандартного ввода данных для получения данных от пользователя. Мы также передаем в метод `read_line` один аргумент: `&mut guess`.

Работа метода `read_line` состоит в том, чтобы брать все, что пользователь набирает в стандартном вводе данных, и помещать это в экземпляр типа `String`, поэтому он берет этот строковый экземпляр в качестве аргумента. Указанный аргумент должен быть изменяемым, чтобы этот метод мог изменять содержимое строкового экземпляра путем добавления вводимых пользователем данных.

Символ `&` указывает на то, что этот аргумент является ссылкой, которая дает возможность многочисленным частям кода обращаться к одному фрагменту данных без многократного копирования этих данных в память. Ссылки являются сложным языковым средством, и одно из главных преимуществ Rust состоит в том, что в нем можно совершенно безопасно и легко использовать ссылки. Для того чтобы завершить программу, не нужно знать многих подробностей. На данный момент вам нужно лишь знать, что, как и переменные, ссылки по умолчанию являются неизменяемыми. Следовательно, чтобы сделать ее изменяемой, нужно написать `&mut guess` вместо `&guess`. (В главе 4 ссылки будут объяснены подробнее.)

Обработка потенциального сбоя с помощью типа Result

Мы еще не закончили с этой строкой кода. Хотя темой нашего обсуждения до сих пор была одна-единственная строка текста, она является лишь первой частью одной логической строки кода. Вторая часть заключается в следующем методе:

```
.expect("Не получилось прочитать строку");
```

Когда вы вызываете метод с помощью синтаксиса `.foo()`, часто бывает разумно добавить новую строку и другие пробелы с целью разбиения длинных строк кода. Мы могли бы написать этот код следующим образом:

```
io::stdin().read_line(&mut guess).expect("Не получилось прочитать строку");
```

Однако одну длинную строку кода трудно читать, поэтому лучше всего ее разделить: две строки кода для двух вызовов метода. Теперь давайте рассмотрим, что делает эта строка кода.

Как уже упоминалось ранее, метод `read_line` помещает то, что пользователь набирает, в переменную типа `String`, которую мы ему передаем, но также возвращает значение — в данном случае `io::Result`. Стандартная библиотека языка Rust имеет ряд типов с именем `Result`: обобщенный тип `Result`, а также специальные версии для подмодулей, такие как тип `io::Result`.

Типы `Result` — это перечисления, часто кратко именуемые *enum*. Перечисление — это тип, который имеет фиксированное множество значений, и эти значения называются вариантами перечисления. В главе 6 перечисления будут рассмотрены подробнее.

Для типа `Result` вариантами являются `Ok` и `Err`. Вариант `Ok` указывает на то, что операция была успешной, а внутри `Ok` находится успешно сгенерированное значение. Вариант `Err` означает, что операция была неуспешной, и `Err` содержит информацию о том, как или почему операция не сработала.

Целью этих типов `Result` является кодирование информации об обработке ошибок. Значения типа `Result`, как и значения любого типа, задают методы. Экземпляр `io::Result` имеет метод `expect`, который вы можете вызывать. Если этот экземпляр `io::Result` является значением `Err`, то `expect` вызовет аварийный сбой программы и покажет сообщение, которое вы передали в метод `expect` в качестве аргумента. Если метод `read_line` возвращает `Err`, то это, скорее всего, будет результатом ошибки, исходящей из базовой операционной системы. Если этот экземпляр `io::Result` является значением `Ok`, то метод `expect` возьмет возвращаемое значение, которое содержит `Ok`, и вернет вам только это значение, которое вы сможете использовать. В данном случае указанное значение является числом в байтах, введенным пользователем в стандартный ввод данных.

Если вы не вызовете метод `expect`, то программа будет компилирована, но вы получите предупреждение[1]:

```
$ cargo build
    Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
warning: unused `std::result::Result` which must be used
  --> src/main.rs:10:5
   |
10 |     io::stdin().read_line(&mut guess);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: #[warn(unused_must_use)] on by default
```

Язык Rust предупреждает, что вы не использовали значение типа `Result`, возвращенное из метода `read_line`, указывая на то, что программа не обработала возможную ошибку.

Правильный способ блокировать это предупреждение — фактически написать обработку ошибок, но поскольку вы хотите, чтобы эта программа просто завершала свою работу аварийно, когда возникает проблема, то можете использовать метод `expect`. О восстановлении после ошибок вы узнаете в главе 9.

---

[1] `предупреждение: неиспользованное значение типа `std::result::Result`, которое должно быть использовано`

Печать значений с помощью заполнителей макрокоманды println!

Помимо закрывающих фигурных скобок осталось обсудить еще одну строку кода, а именно:

```
println!("Вы загадали: {}", guess);
```

Эта строка кода выводит строковую переменную, в которой мы сохранили введенные пользователем данные. Фигурные скобки `{}` являются заполнителем: думайте о `{}` как о маленьких клешнях краба, которые удерживают значение на месте. С помощью фигурных скобок можно вывести более одного значения: первый набор фигурных скобок содержит первое значение, приводимое после форматной строки, второй набор содержит второе значение и так далее. Вывод более одного значения в одном вызове макрокоманды `println!` выглядел бы вот так:

```
let x = 5;
let y = 10;

println!("x = {} и y = {}", x, y);
```

Этот код будет печатать x = 5 и y = 10.

Тестирование первой части

Давайте проверим первую часть игры на угадывание. Запустите ее с помощью команды `cargo run`:

```
$ cargo run
    Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
      Running `target/debug/guessing_game`
Угадайте число!
Пожалуйста, введите свою догадку.
6
Вы загадали: 6
```

На этом этапе первая часть игры закончена: мы получаем ввод с клавиатуры, а затем выводим его.

Генерирование секретного числа

Далее нам нужно сгенерировать секретное число, которое пользователь попытается угадать. Секретное число всегда должно быть разным, чтобы в игру было интересно играть несколько раз. Давайте использовать случайное число от 1 до 100, чтобы не слишком усложнять игру. Rust пока еще не содержит функциональность случайных чисел в стандартной библиотеке. Тем не менее команда разработчиков языка Rust все же предлагает упаковку `rand` по адресу https://crates.io/crates/rand/.

Использование упаковки для получения большей функциональности

Напомним, что упаковка — это множество файлов исходного кода Rust[1]. Проект, который мы до сих пор строили, представляет собой двоичную, то есть исполняемую, упаковку. Упаковка `rand` является библиотечной, то есть содержит код, предназначенный для использования в других программах.

Пакетный менеджер Cargo проявляет себя во всей красе там, где он использует внешние упаковки. Прежде чем мы сможем написать код, который задействует упаковку `rand`, нам нужно модифицировать файл Cargo.toml, включив упаковку `rand` в качестве зависимости. Теперь откройте этот файл и добавьте следующую строку в конец файла под заголовком раздела `[dependencies]`, которую Cargo создал за вас:

Cargo.toml

```
[dependencies]

rand = "0.3.14"
```

В файле Cargo.toml все, что следует за заголовком, является частью раздела, который продолжается до тех пор, пока не начнется другой. Раздел `[dependencies]` — это то место, где вы сообщаете Cargo, от каких внешних упаковок зависит ваш проект и какие версии этих упаковок вам требуются. В данном случае мы укажем упаковку `rand` с помощью семантического спецификатора версии 0.3.14. Cargo понимает семантическое управление версиями (иногда именуемое SemVer), которое является стандартом для написания номеров версий. На самом деле число `0.3.14` — это сокращение для `^0.3.14`, означающее «любая версия с публичным API, совместимым с версией 0.3.14».

Теперь, не внося никаких изменений в код, давайте соберем проект, как показано в листинге 2.2.

Листинг 2.2. Результат выполнения команды `cargo build` после добавления упаковки `rand` в качестве зависимости

```
$ cargo build
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading rand v0.3.14
 Downloading libc v0.2.14
   Compiling libc v0.2.14
   Compiling rand v0.3.14
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
```

[1] В модульной системе Rust упаковки (crate также переводится как «ящик, корзина, тара») занимают промежуточное положение между модулями и пакетами: пакет состоит из одной или нескольких исполняемых или библиотечных упаковок, а упаковки — из одного или нескольких модулей. См. https://doc.rust-lang.org/book/ch07-01-packages-and-crates.html.

Вы видите разные номера версий (но все они будут совместимы с кодом, благодаря SemVer!), причем выводимые на экран строки могут находиться в другом порядке.

Теперь, когда у нас есть внешняя зависимость, Cargo извлекает последние версии всего этого из реестра, то есть копии данных из https://crates.io/. Crates.io — это то место, где участники экосистемы языка Rust размещают свои проекты с открытым исходным кодом Rust для их использования другими разработчиками.

После обновления реестра пакетный менеджер Cargo проверяет раздел `[dependencies]` и скачивает все упаковки, которых у вас еще нет. В данном случае, несмотря на то что мы указали в качестве зависимости только `rand`, Cargo также захватил копию `libc`, потому что в своей работе `rand` зависит от `libc`. После скачивания упаковок язык Rust компилирует их, а затем компилирует проект с имеющимися зависимостями.

Если вы сразу же выполните команду `cargo build` снова, не внося никаких изменений, то на выходе вы не получите никакого результата, кроме строчки `Finished`. Cargo знает, что он уже скачал и скомпилировал зависимости и вы ничего не изменили в файле Cargo.toml. Cargo также знает, что вы ничего не изменили в коде, поэтому он не перекомпилирует и его. Он просто завершает работу, ничего не делая.

Если вы откроете файл src/main.rs, внесете незначительное изменение, а затем сохраните его и соберете снова, то увидите только две результирующие строчки:

```
$ cargo build
    Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
```

Эти строчки показывают, что Cargo обновляет сборку, используя только крошечное изменение в файле src/main.rs. Ваши зависимости не изменились, поэтому Cargo знает, что он может использовать повторно те упаковки, которые он уже скачал и скомпилировал для них. Он просто перестраивает вашу часть кода.

Обеспечение воспроизводимых сборок с помощью файла Cargo.lock

Cargo имеет механизм, который дает возможность перестройки одного и того же артефакта всякий раз, когда вы или кто-либо другой выполняете сборку вашего кода: Cargo будет использовать только те версии зависимостей, которые вы указали, до тех пор, пока вы не укажете иное. Например, что произойдет, если на следующей неделе выйдет версия 0.3.15 упаковки `rand`, содержащая важное исправление ошибок, а также регрессию, которая нарушит работу вашего кода?

Ответом на этот вопрос является файл Cargo.lock, который был создан при первом выполнении команды `cargo build` и теперь находится в каталоге guessing_game. Когда вы создаете проект в первый раз, Cargo выясняет все версии зависимостей, которые соответствуют критериям, а затем записывает их в файл Cargo.lock. Ког-

да вы будете выполнять сборку проекта в будущем, Cargo будет видеть, что файл Cargo.lock существует, и использовать указанные там версии, а не делать всю работу по выяснению версий снова. Это позволяет вам автоматически иметь воспроизводимую сборку. Другими словами, ваш проект будет оставаться в версии 0.3.14 до тех пор, пока вы не обновите его благодаря файлу Cargo.lock.

Обновление упаковки до новой версии

Когда вы хотите обновить упаковку, Cargo предоставляет еще одну команду, `update`, которая будет игнорировать файл Cargo.lock и выяснит все последние версии, которые соответствуют вашим спецификациям в Cargo.toml. Если она сработает, то Cargo запишет эти версии в файл Cargo.lock.

Но по умолчанию Cargo будет искать только версии больше 0.3.0 и меньше 0.4.0. Если упаковка `rand` выпустила две новые версии, 0.3.15 и 0.4.0, то вы увидите следующее ниже, выполнив команду `cargo update`:

```
$ cargo update
    Updating registry `https://github.com/rust-lang/crates.io-index`
    Updating rand v0.3.14 -> v0.3.15
```

На этом этапе вы также заметите изменение в файле Cargo.lock, отметив, что версия упаковки `rand`, которую вы сейчас используете, равна 0.3.15.

Если бы вы захотели использовать `rand` версии 0.4.0 или любую версию в серии 0.4.x, то вам следовало бы обновить файл Cargo.toml так, чтобы он выглядел следующим образом:

Cargo.toml

```
[dependencies]

rand = "0.4.0"
```

В следующий раз, когда вы выполните команду `cargo build`, Cargo обновит реестр имеющихся упаковок и пересмотрит потребности `rand` в соответствии с новой версией, которую вы указали.

О пакетном менеджере Cargo и его экосистеме мы еще много чего расскажем в главе 14, но пока это все, что вам нужно знать. Cargo очень упрощает повторное использование библиотек, и поэтому растиане могут писать небольшие проекты, которые собираются из многочисленных пакетов.

Генерирование случайного числа

Теперь, когда вы добавили упаковку `rand` в файл Cargo.toml, приступим к ее использованию. Следующим шагом является обновление src/main.rs, как показано в листинге 2.3.

Листинг 2.3. Добавление кода генерирования случайного числа

```
   use std::io;
❶  use rand::Rng;

   fn main() {
       println!("Угадайте число!");

    ❷  let secret_number = rand::thread_rng().gen_range(1, 101);

       println!("Секретное число равно {}", secret_number);

       println!("Пожалуйста, введите свою догадку.");

       let mut guess = String::new();

       io::stdin().read_line(&mut guess)
           .expect("Не получилось прочитать строку");

       println!("Вы загадали: {}", guess);
   }
```

Сначала мы добавляем строку кода use: `use rand::Rng` ❶. Типаж `Rng` определяет методы, которые реализуются генераторами случайных чисел, и указанный типаж должен быть в области видимости, благодаря чему мы можем использовать эти методы. В главе 10 типажи будут рассмотрены подробно.

Далее мы добавляем еще две строки кода в середине ❷. Функция `rand::thread_rng` даст нам конкретный генератор случайных чисел, который мы намерены использовать — он является локальным для текущего потока и инициализируется операционной системой. Затем мы вызываем метод `gen_range` на указанном генераторе случайных чисел. Этот метод определен типажом `Rng`, который мы ввели в область видимости с помощью инструкции `use rand::Rng`. Метод `gen_range` берет два числа в качестве аргументов и генерирует случайное число между ними. Результат включает в себя нижнюю границу, но исключает верхнюю, поэтому для того, чтобы запросить число между 1 и 100, нам нужно указать 1 и 101.

ПРИМЕЧАНИЕ

Вы не просто узнаете, какие типажи использовать и какие функции и методы вызывать из упаковки. Инструкции по использованию упаковки содержатся в сопроводительной документации к ней. Еще одно приятное свойство Cargo заключается в том, что вы можете выполнить команду `cargo doc --open`, которая выведет документацию, порождаемую всеми вашими зависимостями, локально и откроет ее в браузере. К примеру, если вас интересует прочая функциональность упаковки `rand`, то выполните `cargo doc --open` и нажмите `rand` на боковой панели слева.

Вторая строка кода, добавленная нами в середине программы, выводит секретное число. Это полезно при разработке программы и позволяет ее тестировать, но мы удалим эту строку из окончательной версии. Игра никуда не годится, если программа выводит ответ, как только запускается!

Попробуйте выполнить программу несколько раз:

```
$ cargo run
    Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
      Running `target/debug/guessing_game`
Угадайте число!
Секретное число равно 7
Пожалуйста, введите свою догадку.
4
Вы загадали: 4
$ cargo run
      Running `target/debug/guessing_game`
Угадайте число!
Секретное число равно 83
Пожалуйста, введите свою догадку.
5
Вы загадали: 5
```

Вы должны получить разные случайные числа, и все они должны быть от 1 до 100. Отлично!

Сравнение загаданного числа с секретным числом

Теперь, когда у нас есть данные, введенные пользователем, и случайное число, мы можем их сравнить. Этот шаг показан в листинге 2.4. Обратите внимание, что этот код пока что не компилируется, и позже мы объясним почему.

Листинг 2.4. Обработка возможных результатов сравнения двух чисел

src/main.rs

```
   use std::io;
❶  use std::cmp::Ordering;
   use rand::Rng;

   fn main() {

       // --пропуск--

       println!("Вы загадали: {}", guess);

       match❷ guess.cmp(&secret_number)❸ {
           Ordering::Less => println!("Слишком малое число!"),
           Ordering::Greater => println!("Слишком большое число!"),
           Ordering::Equal => println!("Вы выиграли!"),
       }
   }
```

Здесь первый новый фрагмент — это еще одна инструкция `use` ❶, вводящая тип с именем `std::cmp::Ordering` в область видимости из стандартной библиотеки.

Как и `Result`, `Ordering` («упорядочение») — это еще одно перечисление, но вариантами для перечисления `Ordering` являются `Less` («меньше»), `Greater` («больше») и `Equal` («равно»), являющиеся тремя возможными результатами, когда вы сравниваете два значения.

Затем мы добавляем внизу пять новых строк кода, которые используют тип `Ordering`. Метод `cmp` ❸ сравнивает два значения и может быть вызван для всего, что можно сравнить. Он ссылается на то, с чем вы хотите сравнить. Здесь он сравнивает загаданное число `guess` с секретным числом `secret_number`. Затем он возвращает вариант перечисления `Ordering`, который мы ввели в область видимости с помощью инструкции `use`. Мы используем выражение `match` ❷, чтобы решить, что делать дальше, основываясь на том, какой вариант упорядочения `Ordering` был возвращен из вызова метода `cmp` со значениями в `guess` и `secret_number`.

Выражение `match` состоит из ветвей. Рукав (*arm*) состоит из шаблона и кода, который должен быть исполнен, если значение, заданное в начале выражения `match`, совпадает с шаблоном ветви. Язык Rust берет значение, переданное в `match`, и по очереди просматривает шаблон каждого рукава. Конструкция `match` и шаблоны являются мощными средствами. Они позволяют выражать разнообразные ситуации, с которыми может столкнуться код, и гарантируют, что каждая ситуация будет обработана. Мы подробно разберем эти средства в главах 6 и 18 соответственно.

Давайте посмотрим, что произойдет с используемым здесь выражением `match`. Будем считать, что пользователь загадал 50, а случайно сгенерированное секретное число на этот раз равно 38. Когда код сравнивает 50 и 38, то метод `cmp` возвращает `Ordering::Greater`, потому что 50 больше 38. Выражение `match` получает значение `Ordering::Greater` и начинает проверять шаблон каждой ветви. Выражение смотрит на шаблон первой ветви `Ordering::Less` и видит, что значение `Ordering::Greater` не совпадает с `Ordering::Less`, поэтому оно игнорирует код в этом рукаве и переходит к следующему. Паттерн следующего рукава `Ordering::Greater` действительно совпадает с `Ordering::Greater`! Связанный код в ветви исполнится и выведет `Слишком большое число!`. Выражение `match` завершает свою работу, потому что нет необходимости проверять последнюю ветвь в этом сценарии.

Однако код в листинге 2.4 пока что не компилируется. Давайте попробуем это сделать[1]:

```
$ cargo build
    Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
error[E0308]: mismatched types
  --> src/main.rs:23:21
   |
23 |     match guess.cmp(&secret_number) {
   |                     ^^^^^^^^^^^^^^^ expected struct `std::string::String`,
found integral variable
```

[1] `ошибка[E0308]: несовпадающие типы`

```
    |
    = note: expected type `&std::string::String`
    = note:    found type `&{integer}`

error: aborting due to previous error
Could not compile `guessing_game`.
```

Ошибка заключается в том, что типы не совпадают. В Rust имеется сильная статическая система типов. Однако в нем также есть логический вывод типов. Когда мы написали `let mut guess = String::new()`, Rust смог логически вывести, что переменная `guess` должна иметь тип `String`, и не заставил нас указать тип. С другой стороны, переменная `secret_number` имеет числовой тип. Значение от 1 до 100 могут иметь несколько числовых типов: `i32`, 32-битное число; `u32`, беззнаковое 32-битное число; `i64`, 64-битное число и другие. По умолчанию Rust использует тип `i32`, который является типом переменной `secret_number`, при условии, что вы не добавляете информацию о типе в другом месте, что заставит компилятор логически вывести другой числовой тип. Причина ошибки здесь заключается в том, что Rust не может сравнить строковый и числовой типы.

В итоге мы хотим конвертировать значение типа `String`, которое программа читает на входе, в числовой тип, чтобы численно сравнить его с секретным числом. Мы можем это сделать, добавив следующие две строки кода в тело функции `main`:

src/main.rs

```
// --пропуск--
    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Не получилось прочитать строку");

    let guess: u32 = guess.trim().parse()
        .expect("Пожалуйста, наберите число!");

    println!("Вы загадали: {}", guess);

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Слишком малое число!"),
        Ordering::Greater => println!("Слишком большое число!"),
        Ordering::Equal => println!("Вы выиграли!"),
    }
}
```

Мы создаем переменную с именем `guess`. Но подождите, у программы ведь уже есть переменная с именем `guess`? Да, есть, но Rust позволяет нам затенить предыдущее значение переменной `guess` новым. Это языковое средство часто используется в ситуациях, когда требуется конвертировать значение из одного типа в другой. Затенение позволяет нам использовать имя переменной `guess` повторно, а не создавать две уникальные переменные, как, например, `guess_str` и `guess`. (В главе 3 затенение рассматривается подробнее.)

Мы связываем переменную `guess` с выражением `guess.trim().parse()`. Переменная `guess` в указанном выражении относится к исходной переменной `guess`, которая была экземпляром типа `String`, содержащим в себе входные данные. Метод `trim` для типа `String` устраняет любые пробелы в начале и конце. Несмотря на то что тип `u32` может содержать только числовые символы, пользователь должен нажать клавишу ENTER и тем самым выполнить метод `read_line`. Когда пользователь нажимает ENTER, символ новой строки добавляется в конец строкового значения. Например, если пользователь набирает `5` и нажимает ENTER, то переменная `guess` выглядит следующим образом: `5\n`. `\n` обозначает «новую строку», то есть результат нажатия клавиши ENTER. Метод `trim` исключает `\n`, давая в результате только `5`.

Метод `parse`, определенный для типа `String`, делает разбор строкового значения и преобразует его в какое-то число. Поскольку этот метод извлекает различные типы чисел, нужно указать точный тип числа, который мы хотим, используя `let guess: u32`. Двоеточие (`:`) после `guess` говорит о том, что мы аннотируем тип переменной. В Rust имеется несколько встроенных числовых типов: `u32`, который мы рассматриваем, является беззнаковым 32-битным целочисленным типом. Это хороший выбор по умолчанию для малого положительного числа. О других типах чисел вы узнаете в главе 3. Кроме того, аннотация `u32` в этом примере программы и сравнение с переменной `secret_number` означают, что Rust логически выведет, что переменная `secret_number` тоже должна иметь тип `u32`. И, таким образом, теперь будут сравниваться два значения одного и того же типа!

Вызов метода `parse` легко может стать причиной ошибки. Если бы, например, строка содержала `A👍%`, то было бы невозможно конвертировать ее в число. Поскольку метод `parse` может не сработать, он возвращает тип `Result` во многом так же, как и метод `read_line` (тип `Result` обсуждался в разделе «Обработка потенциального сбоя с помощью типа `Result`»). Мы будем обращаться с этим типом точно так же, снова используя метод `expect`. Если `parse` возвращает вариант типа `Result`, равный `Err`, так как он не смог создать число из строки, то вызов метода `expect` завершит игру аварийно и выведет сообщение, заданное нами. Если метод `parse` выполнит успешную конвертацию строки в число, то он вернет вариант типа `Result`, равный `Ok`, а метод `expect` вернет число, которое мы хотим получить из значения `Ok`.

Давайте прямо сейчас выполним эту программу!

```
$ cargo run
    Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     Finished dev[unoptimized + debuginfo] target(s) in 1.50 secs
      Running `target/debug/guessing_game`
Угадайте число!
Секретное число равно 58
Пожалуйста, введите свою догадку.
  76
Вы загадали: 76
Слишком большое число!
```

Здорово! Несмотря на то что перед загаданным числом были пробелы, программа все равно выяснила, что пользователь загадал 76. Выполните программу несколько раз, чтобы проверить разное поведение с разными входными данными: загадайте правильное число, слишком большое и слишком маленькое.

Сейчас у нас работает бóльшая часть игры, но пользователь может вводить только одно загаданное число. Давайте это изменим, добавив цикл!

Вывод нескольких загаданных чисел с помощью цикличности

Ключевое слово `loop` создает бесконечный цикл. Сейчас мы его добавим, предоставив пользователям больше шансов угадать число:

src/main.rs

```
// --пропуск--

    println!("Секретное число равно {}", secret_number);

    loop {
        println!("Пожалуйста, введите свою догадку.");

        // --пропуск--

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Слишком малое число!"),
            Ordering::Greater => println!("Слишком большое число!"),
            Ordering::Equal => println!("Вы выиграли!"),
        }
    }
}
```

Как видите, мы перенесли в цикл все операции — и просьбу ввести загаданное число, и все остальное. Обязательно сделайте отступы между строками кода внутри цикла еще на четыре пробела и выполните программу снова. Обратите внимание, что возникла новая проблема, потому что программа делает именно то, что мы ей сказали: бесконечно запрашивает следующую догадку! Похоже, что пользователь просто не сможет выйти из игры!

Пользователь всегда может прервать программу, нажав Ctrl-C. Но есть и другой способ избежать этого ненасытного монстра, как упоминалось при обсуждении метода `parse` в разделе «Сравнение загаданного числа с секретным числом»: если пользователь введет нечисловой ответ, то программа завершится аварийно. Пользователь может этим воспользоваться для того, чтобы выйти из программы, как показано здесь:

```
$ cargo run
    Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
```

```
     Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
      Running `target/debug/guessing_game`
Угадайте число!
Секретное число равно 59
Пожалуйста, введите свою догадку.
45
Вы загадали: 45
Слишком малое число!
Пожалуйста, введите свою догадку.
60
Вы загадали: 60
Слишком большое число!
Пожалуйста, введите свою догадку.
59
Вы загадали: 59
Вы выиграли!
Пожалуйста, введите свою догадку.
выйти
thread 'main' panicked at 'Please type a number!: ParseIntError { kind:
InvalidDigit }', src/libcore/result.rs:785
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

Набрав `quit`, вы фактически выходите из игры, но так же произойдет и при вводе любых других нечисловых данных. Мягко говоря, неразумно. Мы хотим, чтобы игра автоматически останавливалась, когда игрок угадывает правильное число.

Выход из игры после правильно угаданного числа

Давайте запрограммируем игру так, чтобы она завершалась, когда пользователь выигрывает, добавив для этого инструкцию `break`:

src/main.rs

```
// --пропуск--

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Слишком малое число!"),
            Ordering::Greater => println!("Слишком большое число!"),
            Ordering::Equal => {
                println!("Вы выиграли!");
                break;
            }
        }
    }
}
```

Добавление строки кода с инструкцией `break` после сообщения `Вы выиграли!` побуждает программу выйти из цикла, когда пользователь правильно угадывает секретное число. Выход из цикла также означает выход из программы, потому что цикл является последней частью функции `main`.

Обработка ввода недопустимых данных

Чтобы уточнить, как поведет себя игра, вместо аварийного завершения программы, когда пользователь вводит нечисловое значение, давайте сделаем так, чтобы игра игнорировала нечисловое значение, предоставив пользователю возможность продолжить угадывать. Мы можем сделать это, изменив строку кода, в которой переменная `guess` конвертируется из типа `String` в тип `u32`, как показано в листинге 2.5.

Листинг 2.5. Игнорирование нечислового загаданного числа и запрос следующего загаданного числа вместо аварийного завершения программы

src/main.rs

```
// --пропуск--

io::stdin().read_line(&mut guess)
    .expect("Не получилось прочитать строку");

let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};

println!("Вы загадали: {}", guess);

// --пропуск--
```

Переключаясь с вызова метода `expect` на выражение `match`, вы применяете способ, который обычно используется для перехода от аварийного завершения программы к обработке ошибки. Помните, что метод `parse` возвращает тип `Result`, а `Result` — это перечисление, имеющее варианты `Ok` или `Err`. Здесь мы используем выражение `match`, как и в случае с результатом типа `Ordering` метода `cmp`.

Если метод `parse` в состоянии успешно превратить строку в число, то он вернет значение `Ok`, содержащее результирующее число. Это значение `Ok` совпадет с паттерном первого рукава, а выражение `match` просто вернет значение `num`, которое метод `parse` произвел и поместил внутрь значения `Ok`. Это число окажется именно там, где мы хотим, в новой создаваемой нами переменной `guess`.

Если метод `parse` не в состоянии превратить строку в число, то он возвращает значение `Err`, содержащее дополнительную информацию об ошибке. Значение `Err` не совпадает с паттерном `Ok(num)` в первом рукаве выражения `match`, но оно совпадает с паттерном `Err(_)` во втором рукаве. Подчеркивание `_` является всеохватывающим значением. В данном примере мы хотим, чтобы совпали все значения `Err`, независимо от того, какая информация у них внутри. Поэтому программа выполнит код второго рукава `continue`, который говорит программе перейти к следующей итерации цикла и запросить еще одно загаданное число. Таким образом, по сути, программа игнорирует все ошибки, с которыми может столкнуться метод `parse`!

Теперь в программе все должно работать как следует. Давайте испытаем ее:

```
$ cargo run
    Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
      Running `target/debug/guessing_game`
Угадайте число!
Секретное число равно 61
Пожалуйста, введите свою догадку.
10
Вы загадали: 10
Слишком малое число!
Пожалуйста, введите свою догадку.
99
Вы загадали: 99
Слишком большое число!
Пожалуйста, введите свою догадку.
foo
Пожалуйста, введите свою догадку.
61
Вы загадали: 61
Вы выиграли!
```

Потрясающе! Благодаря одной крошечной финальной доработке мы завершим игру-угадайку! Напомним, что программа по-прежнему выводит секретное число. Это пригодилось для тестирования, но портит игру. Давайте удалим макрокоманду `println!`, которая выводит секретное число. В листинге 2.6 показан окончательный код.

Листинг 2.6. Полный код игры-угадайки

src/main.rs

```
use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Угадайте число!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    loop {
        println!("Пожалуйста, введите свою догадку.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("Не получилось прочитать строку");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
```

```
            };

            println!("Вы загадали: {}", guess);

            match guess.cmp(&secret_number) {
                Ordering::Less => println!("Слишком малое число!"),
                Ordering::Greater => println!("Слишком большое число!"),
                Ordering::Equal => {
                    println!("Вы выиграли!");
                    break;
                }
            }
        }
    }
```

Итоги

Итак, вы успешно создали игру-угадайку. Примите поздравления!

Этот проект на практике познакомил вас со многими новыми понятиями языка Rust: `let`, `match`, методами, связанными функциями, использованием внешних упаковок и многим другим. В следующих главах вы узнаете об этих понятиях подробнее. В главе 3 рассказывается о понятиях, которые есть в большинстве языков программирования, таких как переменные, типы данных и функции, и показывается, как их использовать в Rust. В главе 4 рассматривается идея владения — средство, отличающее Rust от других языков. В главе 5 обсуждаются структуры и синтаксис методов, а в главе 6 объясняется принцип работы перечислений.

3 Концепции программирования

Эта глава посвящена концепциям, которые есть почти в каждом языке программирования, и рассказывает о том, как они работают в Rust. Многие языки в своей основе имеют много общего. Ни одно из понятий, представленных в этой главе, не является уникальным для Rust. Однако мы обсудим их в контексте Rust и проясним моменты, связанные с использованием этих концепций.

В частности, вы узнаете о переменных, базовых типах, функциях, комментариях и управлении потоком. Эти базовые понятия лежат в основе каждой программы Rust, и если вы усвоите их, то у вас появится прочная опора для старта.

КЛЮЧЕВЫЕ СЛОВА

Как и в других языках, в Rust имеется набор ключевых слов, которые используются только в этом языке. Имейте в виду, что эти слова нельзя использовать в качестве имен переменных или функций. Большинство ключевых слов имеют специальные значения, и вы будете применять их для выполнения различных задач в программах на языке Rust. С некоторыми из них не связана текущая функциональность, но их сохранили, чтобы добавить функциональность в Rust в будущем. Список ключевых слов можно найти в приложении A в конце книги.

Переменные и изменяемость

Как уже упоминалось в главе 2, переменные по умолчанию являются неизменяемыми. Это одна из многих сторон языка Rust, которая позволяет писать код, пользуясь преимуществами безопасности и удобной конкурентности этого языка. Однако вы по-прежнему можете делать переменные изменяемыми. Давайте узнаем, как и почему язык Rust побуждает вас отдавать предпочтение неизменяемости и почему в какой-то момент вы, возможно, захотите от нее отказаться.

Если переменная является неизменяемой, то после привязки значения к имени это значение нельзя изменить. В качестве иллюстрации давайте создадим новый проект под названием variables («переменные») в каталоге проектов projects с помощью команды `cargo new variables`.

Затем в новом каталоге variables откройте src/main.rs и замените его код следующим кодом, который пока что не компилируется:

src/main.rs

```
fn main() {
    let x = 5;
    println!("Значение x равно {}", x);
    x = 6;
    println!("Значение x равно {}", x);
}
```

Сохраните и выполните программу с помощью команды `cargo run`. На выходе вы должны получить сообщение об ошибке, как показано ниже[1]:

```
error[E0384]: cannot assign twice to immutable variable `x`
--> src/main.rs:4:5
  |
2 |    let x = 5;
  |        - first assignment to `x`
3 |     println!("Значение x равно {}", x);
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable
```

В этом примере показано, как компилятор помогает находить ошибки в программах. Даже если ошибки компилятора вас расстраивают, они лишь означают, что ваша программа пока не делает безопасно то, что вы хотите. Они вовсе не означают, что вы плохой программист! Ошибки случаются даже у опытных растиан.

В сообщении об ошибке указано: причина ошибки в том, что неизменяемой переменной `x` нельзя присвоить значение дважды (`cannot assign twice to immutable variable`), потому что вы попытались присвоить второе значение неизменяемой переменной `x`.

Важно, что мы получаем ошибки времени компиляции, когда пытаемся изменить значение, которое мы ранее определили как неизменяемое. Именно эта ситуация приводит к ошибкам. Если одна часть кода работает, допуская, что значение никогда не изменится, а другая часть кода это значение изменяет, то вполне возможно, что первая часть кода не будет работать так, как было задумано при разработке. Причину таких ошибок бывает трудно отследить постфактум, в особенности когда вторая часть кода лишь иногда изменяет значение.

---

[1] `ошибка[E0384]: нельзя дважды присвоить значение неизменяемой переменной `x`

Компилятор гарантирует, что, когда вы заявляете, что значение не будет изменяться, оно действительно не будет изменяться. Это означает, что, когда вы читаете и пишете код, вам не нужно следить за тем, как и где значение может измениться. Таким образом, становится легче обдумывать код.

Но изменяемость бывает очень полезной. Переменные являются неизменяемыми только по умолчанию. Как и в главе 2, вы можете сделать их изменяемыми, добавив ключевое слово `mut` перед именем переменной. Кроме возможности изменять это значение, `mut` сообщает будущим читателям цель кода, указывая на то, что другие части кода будут изменять значение этой переменной.

Например, давайте внесем в src/main.rs следующее изменение:

src/main.rs

```
fn main() {
    let mut x = 5;
    println!("Значение x равно {}", x);
    x = 6;
    println!("Значение x равно {}", x);
}
```

Когда мы выполняем эту программу, то получаем:

```
$ cargo run
    Compiling variables v0.1.0 (file:///projects/variables)
     Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
      Running `target/debug/variables`
Значение x равно 5
Значение x равно 6
```

Нам разрешено изменить значение, к которому привязана переменная `x`, с `5` на `6`, когда используется ключевое слово `mut`. В некоторых случаях вы захотите сделать переменную изменяемой, потому что так удобнее писать код по сравнению с тем, как если бы в нем были только неизменяемые переменные.

Кроме предотвращения ошибок следует учесть целый ряд компромиссов. Например, в тех случаях, когда вы используете крупные структуры данных, изменить экземпляр, возможно, будет быстрее, чем копировать и возвращать новые размещенные в памяти экземпляры. В случае с меньшими структурами данных бывает, что создание новых экземпляров и написание их в более функциональном стиле программирования упрощает их осмысление, поэтому более низкая производительность, возможно, будет стоящей расплатой за достижение такой ясности.

Различия между переменными и константами

Неспособность изменить значение переменной, возможно, напомнила вам еще об одном понятии программирования, которое есть в большинстве других языков — константах. Как и неизменяемые переменные, константы — это значения, привя-

занные к имени, которые нельзя изменить. Но между константами и переменными есть несколько различий.

Прежде всего, с константами нельзя использовать ключевое слово `mut`. Константы являются не просто неизменяемыми по умолчанию — они неизменяемы всегда.

Вы объявляете константы с помощью ключевого слова `const` вместо ключевого слова `let`, и тип значения должен быть аннотирован. Мы рассмотрим типы и аннотации типов в разделе «Типы данных», поэтому сейчас не стоит беспокоиться о деталях. Просто знайте, что вы всегда должны аннотировать тип.

Константы можно объявлять в любой области видимости, включая глобальную область, что делает их полезными для значений, о которых необходимо знать многим частям кода.

Последнее отличие состоит в том, что константы устанавливаются равными только константным выражениям, а не результату вызова функции или любому другому значению, которое вычисляется только во время выполнения.

Ниже приведен пример объявления константы, где имя константы — `MAX_POINTS`, а ее значение равно `100 000`. (Согласованное правило Rust об именовании констант заключается в использовании прописных букв с подчеркиванием между словами. Подчеркивания можно вставлять в числовые литералы, чтобы улучшить их читаемость):

```
const MAX_POINTS: u32 = 100_000;
```

Константы являются действительными в течение всего времени выполнения программы внутри области видимости, в которой они были объявлены. Это делает их полезными для выбора значений в домене приложения, о котором, возможно, необходимо знать многочисленным частям программы, например, максимальное количество очков, которое можно заработать в игре, или скорость света.

Именовать жестко кодированные значения, используемые в программе в качестве констант, полезно для передачи смысла этого значения будущим разработчикам кода. Благодаря этому у вас есть всего одно место в коде, которое потребуется изменить, если жестко кодированное значение будет необходимо обновить в будущем.

Затенение

Как вы видели в практическом занятии с игрой на угадывание в разделе «Сравнение загаданного числа с секретным числом», вы можете объявить новую переменную с тем же именем, что и предыдущая переменная, и новая переменная затенит предыдущую. Растиане говорят, что первая переменная затеняется второй, имея в виду, что, когда переменная используется, то появляется значение именно второй переменной. Мы можем затенить переменную, используя одинаковое имя переменной и повторяя ключевое слово `let` следующим образом:

src/main.rs

```
fn main() {
    let x = 5;

    let x = x + 1;

    let x = x * 2;

    println!("Значение x равно {}", x);
}
```

Эта программа сначала привязывает `x` к значению `5`. Затем она затеняет `x` путем повтора инструкции `let x =`, беря исходное значение и прибавляя `1`, после чего значение `x` становится `6`. Третья инструкция `let` также затеняет `x`, умножая предыдущее значение на `2`, давая `x` конечное значение `12`. Когда мы выполним эту программу, она выведет следующее:

```
$ cargo run
    Compiling variables v0.1.0 (file:///projects/variables)
     Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
      Running `target/debug/variables`
Значение x равно 12
```

Затенение отличается от маркировки переменной как `mut`, потому что мы получим ошибку времени компиляции, если нечаянно попытаемся передать этой переменной новое значение без использования ключевого слова `let`. Используя `let`, мы можем выполнить несколько трансформаций значения, но при этом после завершения трансформаций переменная будет неизменяемой.

Другое различие между `mut` и затенением заключается в том, что, поскольку во время использования ключевого слова `let` еще раз мы практически создаем новую переменную, мы можем изменить тип значения, используя то же самое имя повторно. Например, программа просит пользователя показать, сколько пробелов должно быть между неким текстом, путем ввода знаков пробела, но на самом деле мы хотим сохранить этот ввод данных в виде числа:

```
let spaces = " ";
let spaces = spaces.len();
```

Эта конструкция разрешена, поскольку первая переменная `spaces` имеет строковый тип, а вторая переменная `spaces`, то есть совершенно новая переменная с тем же именем, имеет числовой тип. Таким образом, затенение избавляет нас от необходимости придумывать разные имена, такие как `spaces_str` и `spaces_num`. Вместо этого мы можем использовать более простое имя `spaces` повторно. Однако если для этого мы попытаемся использовать `mut`, как показано здесь, то получим ошибку времени компиляции:

```
let mut spaces = " ";
spaces = spaces.len();
```

Ошибка сообщает, что нам не разрешается изменять тип переменной:

```
error[E0308]: mismatched types
 --> src/main.rs:3:14
  |
3 |     spaces = spaces.len();
  |              ^^^^^^^^^^^^ expected &str, found usize
  |
  = note: expected type `&str`
             found type `usize`
```

Теперь, когда мы изучили то, как работают переменные, давайте рассмотрим другие типы данных, которые они могут иметь.

Типы данных

Каждое значение в Rust принадлежит к определенному типу данных, который говорит о том, что это за данные, и таким образом язык понимает, как с ними работать. Мы рассмотрим два подмножества типов данных: скалярные и составные.

Имейте в виду, что Rust — это статически типизированный язык, а это означает, что во время компиляции он должен знать типы всех переменных. Компилятор обычно логически выводит тип, нужный нам, основываясь на значении и на том, как мы его используем. В тех случаях, когда может быть более одного типа, например, когда мы конвертировали строковый тип в числовой с помощью метода `parse` в разделе «Сравнение загаданного числа с секретным числом», мы должны добавить аннотацию типа:

```
let guess: u32 = "42".parse().expect("Не является числом!");
```

Если не добавить аннотацию типа, то компилятор выдаст ошибку, которая приведена ниже. Она означает, что компилятору требуется больше информации, чтобы узнать, какой тип мы хотим использовать[1]:

```
error[E0282]: type annotations needed
 --> src/main.rs:2:9
  |
2 |     let guess = "42".parse().expect("Не является числом!");
  |         ^^^^^
  |         |
  |         cannot infer type for `_`
  |         consider giving `guess` a type
```

Вы увидите разные аннотации типов для разных типов данных.

[1] `Ошибка[E0282]: требуются аннотации типов`

Скалярные типы

Скалярный тип представляет одно значение. В языке Rust есть четыре первичных скалярных типа: целочисленный тип, тип чисел с плавающей точкой[1], булев тип и символьный тип. Возможно, вы встречались с ними в других языках программирования. Давайте сразу займемся тем, как они работают в языке Rust.

Целочисленные типы

Целое число — это число без дробной составляющей. В главе 2 мы использовали один целочисленный тип — `u32`. Это объявление типа указывает на то, что значение, связанное с ним, должно быть целым числом без знака (целочисленные типы со знаком начинаются с `i`, а не с `u`), которое занимает 32 бита памяти. В табл. 3.1 показаны встроенные в язык Rust целочисленные типы. Каждый вариант в столбцах «Знаковый» и «Беззнаковый» (например, `i16`) используется для объявления типа целочисленного значения.

Таблица 3.1. Целочисленные типы в языке Rust

Длина	Знаковый	Беззнаковый
8 бит	`i8`	`u8`
16 бит	`i16`	`u16`
32 бит	`i32`	`u32`
64 бит	`i64`	`u64`
128 бит	`i128`	`u128`
arch	`isize`	`usize`

Каждый вариант может быть либо знаковым, либо беззнаковым и имеет явно выраженный размер. Термины «знаковый» и «беззнаковый» обозначают, может ли число быть отрицательным или положительным — другими словами, должно ли число иметь при себе знак (знаковое) или оно всегда будет только положительным и, следовательно, представляется без знака (беззнаковое). Точно так же мы записываем числа на бумаге: когда знак важен, у числа есть знак плюс или минус. Однако, когда можно с уверенностью сказать, что число является положительным, его записывают без знака. Знаковые числа хранятся с помощью представления в виде двоичного дополнения (если вы не знаете, что это такое, можете отыскать этот термин в интернете; его объяснение выходит за рамки темы этой книги)[2].

[1] В переводе указана форма записи чисел, принятая в оригинале книги, где для отделения дробной составляющей принято использовать точку.

[2] Двоичное дополнение (two's complement) — это отрицание двоичного числа, которое реализуется путем установки всех единиц в нули и всех нулей в единицы, а затем добавления единицы к результату.

ЦЕЛОЧИСЛЕННОЕ ПЕРЕПОЛНЕНИЕ

Допустим, у вас есть переменная типа `u8`, которая может содержать значения от 0 до 255. Если вы попытаетесь изменить значение переменной на значение, выходящее за пределы этого интервала, такое как 256, то произойдет целочисленное переполнение. В Rust есть несколько интересных правил, связанных с этим поведением. Во время компиляции в отладочном режиме Rust делает проверки на предмет целочисленного переполнения, которые побуждают программу поднимать панику во время работы, если так происходит. В Rust используется термин «паника», когда работа программы завершается с ошибкой. Мы обсудим панику подробнее в разделе «Неустранимые ошибки с помощью макрокоманды panic!».

Когда вы компилируете в режиме с флагом `--release`, Rust не делает проверки на целочисленное переполнение, которые вызывают панику. Вместо этого, если происходит переполнение, то Rust выполняет циклический перенос на основе двоичного дополнения. Короче говоря, значения, превышающие возможный максимум, «сбрасываются» до минимума значений, которые тип может содержать. В случае `u8`, 256 становится 0, 257 становится 1 и так далее. Программа не будет поднимать панику, но у переменной будет значение, которое, вероятно, не будет соответствовать вашим ожиданиям. Опора на поведение с циклическим переносом при целочисленном переполнении считается ошибкой. Если вы хотите выполнить циклический перенос, то можете воспользоваться типом `Wrapping` стандартной библиотеки.

Каждый знаковый вариант может хранить числа от $-(2^{n-1})$ до $2^{n-1} - 1$ включительно, где n — это число бит, используемых вариантом. Таким образом, `i8` может хранить числа от $-(2^7)$ до $2^7 - 1$, что равно интервалу от -128 до 127. Беззнаковые варианты могут хранить числа от 0 до $2^n - 1$, поэтому `u8` может хранить числа от 0 до $2^8 - 1$, что равно интервалу от 0 до 255.

Кроме того, типы `isize` и `usize` зависят от типа компьютера, на котором выполняется программа: 64 бита, если вы используете 64-битную архитектуру, и 32 бита, если у вас 32-битная архитектура.

Вы можете писать целочисленные литералы в любой из форм, приведенных в табл. 3.2. Обратите внимание, что все числовые литералы, кроме байтового литерала, допускают суффикс типа, например `57u8`, и `_` в качестве визуального разделителя, например `1_000`.

Таблица 3.2. Целочисленные литералы в Rust

Числовые литералы	Пример
Десятичные	`98_222`
Шестнадцатеричные	`0xff`
Восьмеричные	`0o77`
Двоичные	`0b1111_0000`
Байтовый (только u8)	`b 'A'`

Как узнать, какой тип целого числа использовать? Если вы не уверены, то типы, принятые в Rust по умолчанию, достаточно хороши. Целочисленные типы имеют тип `i32` — обычно он самый быстрый, даже в 64-битных системах. Типы `isize` или `usize` по преимуществу используются в индексировании коллекции.

Типы с плавающей точкой

В языке Rust также есть два примитивных типа для чисел с плавающей точкой, то есть чисел с десятичными точками. Типами языка Rust с плавающей точкой являются `f32` и `f64`, которые имеют размер 32 бита и 64 бита соответственно. По умолчанию используется тип `f64`, потому что в современных процессорах он имеет примерно такую же скорость, как и `f32`, но при этом обладает большей точностью.

Вот пример, который показывает числа с плавающей точкой в действии:

src/main.rs

```
fn main() {
    let x = 2.0; // f64

    let y: f32 = 3.0; // f32
}
```

Числа с плавающей точкой представлены в соответствии со стандартом IEEE-754. Тип `f32` представляет собой вещественное число с одинарной точностью, а `f64` имеет двойную точность.

Числовые операции

Язык Rust поддерживает основные математические операции, которые можно производить со всеми типами чисел: сложение, вычитание, умножение, деление и получение остатка. Следующий код показывает, как использовать каждую из них в инструкции `let`:

src/main.rs

```
fn main() {
    // сложение
    let sum = 5 + 10;

    // вычитание
    let difference = 95.5 - 4.3;

    // умножение
    let product = 4 * 30;

    // деление
    let quotient = 56.7 / 32.2;

    // остаток
    let remainder = 43 % 5;
}
```

Каждое выражение в этих инструкциях использует математический оператор и оценивается в одно значение, которое затем присваивается переменной. Приложение Б в конце книги содержит список всех операторов Rust.

Булев тип

Как и в большинстве других языков программирования, булев тип в Rust имеет два возможных значения: `true` и `false`. Значения булева типа равны одному байту. Булев тип в Rust задает ключевое слово `bool`. Например:

src/main.rs

```
fn main() {
    let t = true;

    let f: bool = false; // с явной аннотацией типа
}
```

Значения булева типа применяются главным образом в условных выражениях, таких как выражение `if`. Мы рассмотрим работу выражений `if` в разделе «Управление потоком».

Символьный тип

До сих пор мы работали только с цифрами, но Rust поддерживает и буквы. Тип `char` — это самый примитивный алфавитный тип языка, и следующий код показывает один из способов его использования. (Обратите внимание, что символьные литералы задаются одинарными кавычками, в отличие от строковых литералов, которые используют двойные кавычки.)

src/main.rs

```
fn main() {
    let c = 'z';
    let z = 'ℤ';
    let heart_eyed_cat = '😻';
}
```

Тип `char` равен четырем байтам и представляет собой скалярное значение Юникод, то есть это значит, что он представляет собой гораздо больше, чем символы простой кодировки ASCII. Акцентированные буквы, китайские, японские и корейские символы, эмодзи и пробелы нулевой ширины — все это допустимые значения типа `char` в языке Rust. Скалярные значения Юникод варьируются от U+0000 до U+D7FF и U+E000 до U+10FFFF включительно. Однако термин «символ» (или «знак») на самом деле не является понятием Юникода, поэтому ваша интуиция относительно того, что такое символ, может не совпадать с тем, что есть символ в типе `char`. Мы подробно обсудим эту тему в разделе «Хранение текста в кодировке UTF-8 с помощью строк».

Составные типы

Составные типы группируют многочисленные значения в один тип. В Rust есть два примитивных составных типа: кортежи и массивы.

Кортежный тип

Кортеж — это часто встречающийся способ группирования ряда других значений с разнообразными типами в один составной тип. Кортежи имеют фиксированную длину: после объявления они не могут увеличиваться или уменьшаться в размере.

Мы создаем кортеж, записывая список значений через запятую внутри круглых скобок. Каждая позиция в кортеже имеет свой тип, а типы разных значений в кортеже не обязательно должны быть одинаковыми. В этом примере мы добавили необязательные аннотации типов:

src/main.rs

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

Переменная `tup` привязывается ко всему кортежу, поскольку кортеж считается единым составным элементом. Для получения отдельных значений кортежа мы используем сопоставление с паттерном, чтобы деструктурировать значения кортежа, например:

src/main.rs

```
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("Значение y равно {}", y);
}
```

Эта программа сначала создает кортеж и привязывает его к переменной `tup`. Затем она использует паттерн с `let`, беря `tup` и превращая его в три отдельные переменные, `x`, `y` и `z`. Это называется деструктурированием, потому что оно разделяет один кортеж на три части. Наконец, программа выводит значение `y`, то есть `6.4`.

В дополнение к деструктурированию посредством сопоставления с паттерном мы можем обращаться к кортежному элементу непосредственно с помощью точки (`.`) с последующим индексом значения, к которому мы хотим обратиться. Например:

src/main.rs

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);
```

```
    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

Эта программа создает кортеж `x`, а затем задает новые переменные для каждого элемента, используя его индекс. Как и в большинстве языков программирования, первый индекс в кортеже равен 0.

Массив

Еще один способ получить коллекцию из нескольких значений — использовать массив. В отличие от кортежа, каждый элемент массива должен иметь один и тот же тип. Массивы в Rust отличаются от массивов в некоторых других языках, потому что массивы в Rust имеют фиксированную длину, как кортежи.

В Rust входящие в массив значения записываются как список значений через запятую внутри квадратных скобок:

src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];
}
```

Массивы полезны, если вы хотите, чтобы данные размещались в стеке, а не в куче (мы рассмотрим стек и кучу подробнее в главе 4), или если вы хотите, чтобы у вас всегда было фиксированное число элементов. Однако массив не так гибок, как векторный тип. Вектор — это аналогичный коллекционный тип, предусмотренный стандартной библиотекой, который может увеличиваться или уменьшаться в размере. Если вы не уверены в том, что именно использовать — массив или вектор, то вам, вероятно, следует использовать вектор. В главе 8 векторы рассматриваются подробнее.

Возможно, вы захотите использовать массив вместо вектора, — это можно показать на примере программы, задача которой — узнать названия месяцев года. Весьма маловероятно, что в такой программе нужно будет добавить или удалить месяцы, поэтому вы используете массив — вы знаете, что в нем всегда будет 12 элементов:

```
let months = ["Январь", "Февраль", "Март", "Апрель", "Май", "Июнь", "Июль",
              "Август", "Сентябрь", "Октябрь", "Ноябрь", " Декабрь"];
```

Массивный тип записывается с помощью квадратных скобок. Внутрь скобок включают тип каждого элемента, точку с запятой, а затем число элементов в массиве:

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

Здесь `i32` — это тип каждого элемента. После точки с запятой число 5 указывает на количество элементов в массиве.

Такая запись типа массива похожа на альтернативный синтаксис инициализации массива: если вы хотите создать массив, содержащий одинаковое значение для каждого элемента, то вы можете указать в квадратных скобках начальное значение с последующей точкой с запятой, а затем написать длину массива, как показано здесь:

```
let a = [3; 5];
```

Массив с именем `a` будет содержать 5 элементов, для всех элементов которого изначально будет задано значение 3. Это то же самое, что написать `let a = [3, 3, 3, 3, 3];`, но в более сжатом виде.

Доступ к элементам массива

Массив представляет собой отдельную часть памяти, выделенную в стеке. К элементам массива можно обращаться по индексу, например:

src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];

    let first = a[0];
    let second = a[1];
}
```

В этом примере переменная с именем `first` получит значение `1`, потому что это значение находится в массиве в индексе `[0]`. Переменная с именем `second` получит значение `2` из индекса `[1]` в массиве.

Недействительный доступ к элементу массива

Что произойдет, если вы попытаетесь обратиться к элементу массива, который находится за его пределами? Допустим, вы меняете пример на следующий код, который будет компилироваться, но завершится с ошибкой при выполнении:

src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];
    let index = 10;

    let element = a[index];

    println!("Значение элемента равно {}", element);
}
```

Выполнение этого кода с командой `cargo run` произведет следующий результат:

```
$ cargo run
    Compiling arrays v0.1.0 (file:///projects/arrays)
     Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
      Running `target/debug/arrays`
thread '<main>' panicked at 'index out of bounds: the len is 5 but the index
```

```
is 10', src/main.rs:6
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

При компиляции не произошло ошибок, но программа вызвала ошибку времени выполнения и не завершилась успешно. Когда вы попытаетесь обратиться к элементу по индексу, язык Rust проверит, чтобы указанный индекс был меньше длины массива. Если индекс больше или равен длине массива, то Rust поднимет панику.

Это первый пример, иллюстрирующий принципы безопасности Rust в действии. Во многих низкоуровневых языках подобная проверка не выполняется, и когда вы указываете неправильный индекс, то можно получить доступ к недействительной памяти. Rust защищает от таких ошибок, немедленно выходя из программы, вместо того чтобы разрешить доступ к памяти и продолжить работу. В главе 9 обработка ошибок в языке Rust обсуждается подробнее.

Функции

У функций в коде Rust всеобъемлющий характер. Вы уже видели одну из самых важных функций языка – функцию `main`, которая является точкой входа многих программ. Вы также встречались с ключевым словом `fn`, которое позволяет объявлять новые функции.

Часто используется змеиный регистр написания имен функций и переменных. В змеином регистре все буквы находятся в нижнем регистре, а отдельные слова соединяются символом подчеркивания. Вот программа, которая содержит пример определения функции:

src/main.rs

```
fn main() {
    println!("Hello, World!");

    another_function();
}

fn another_function() {
    println!("Еще одна функция.");
}
```

Определения функций начинаются с ключевого слова `fn` и имеют набор скобок после имени функции. Фигурные скобки сообщают компилятору место, где начинается и заканчивается тело функции.

Мы можем вызвать любую функцию, введя ее имя с последующим набором скобок. Поскольку функция `another_function` определена в программе, ее можно вызвать изнутри функции `main`. Обратите внимание, что в исходном коде мы определили функцию `another_function` после функции `main`. Мы могли бы определить ее и раньше. Языку Rust все равно, где вы определяете свои функции; главное, чтобы они были где-то определены.

Давайте начнем новый двоичный проект functions («функции») и продолжим изучать функции. Поместите пример `another_function` в файл src/main.rs и выполните его. На выходе вы должны увидеть следующий результат:

```
$ cargo run
    Compiling functions v0.1.0 (file:///projects/functions)
     Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
      Running `target/debug/functions`
Hello, World!
Еще одна функция.
```

Строки кода исполняются в том порядке, в котором они появляются в функции `main`. Сначала выводится сообщение «Hello, World!», а затем вызывается функция `another_function` и выводится ее сообщение.

Параметры функций

Функции также могут определяться вместе с параметрами, то есть специальными переменными, являющимися частью сигнатуры функции. Когда у функции есть параметры, вы можете предоставить ей конкретные значения для этих параметров. Технически конкретные значения называются аргументами, но в обычной беседе люди склонны использовать термины «параметр» и «аргумент» взаимозаменяемо, как для переменных в определении функции, так и для конкретных значений, передаваемых при вызове функции.

Следующая переписанная версия функции `another_function` показывает, как выглядят параметры:

src/main.rs

```
fn main() {
    another_function(5);
}

fn another_function(x: i32) {
    println!("Значение x равно {}", x);
}
```

Попробуйте выполнить эту программу. На выходе вы должны получить следующий результат:

```
$ cargo run
    Compiling functions v0.1.0 (file:///projects/functions)
     Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
      Running `target/debug/functions`
Значение x равно 5
```

Объявление функции `another_function` имеет один параметр с именем `x`. Тип `x` задан как `i32`. Когда `5` передается в функцию `another_function`, макрокоманда `println!` помещает `5` туда, где в форматной строке находилась пара фигурных скобок.

В сигнатурах функций необходимо объявлять тип каждого параметра. Это решение в дизайне языка является намеренным: обязательные аннотации типов в определениях функций означают, что компилятор почти никогда не нуждается в том, чтобы вы использовали их в другом месте кода, чтобы выяснить, что вы имеете в виду.

Если вы хотите, чтобы функция имела более одного параметра, то отделите объявления параметров запятыми:

src/main.rs

```
fn main() {
    another_function(5, 6);
}

fn another_function(x: i32, y: i32) {
    println!("Значение x равно {}", x);
    println!("Значение y равно {}", y);
}
```

Этот пример создает функцию с двумя параметрами, которые имеют тип `i32`. Затем функция выводит значения обоих параметров. Обратите внимание, что параметры функции не обязательно должны быть одного типа; они просто случайно оказались такими в данном примере.

Давайте попробуем выполнить этот код. Замените программу, находящуюся в данный момент в файле src/main.rs проекта functions, на приведенный выше пример и выполните его с помощью команды `cargo run`:

```
$ cargo run
    Compiling functions v0.1.0 (file:///projects/functions)
     Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
      Running `target/debug/functions`
Значение x равно 5
Значение y равно 6
```

Поскольку мы вызвали функцию с `5` в качестве значения для `x`, а `6` передается как значение для `y`, две строки печатаются с этими значениями.

Инструкции и выражения в телах функций

Тела функций состоят из серии инструкций, которые необязательно заканчиваются выражением. До сих пор мы рассматривали только функции без конечного выражения, но вы встречали выражение как часть инструкции. Поскольку язык Rust основывается на выражениях, это различие очень важно понимать. Другие языки не имеют таких различий, поэтому давайте посмотрим, что такое инструкции и выражения и как их различия влияют на тела функций.

На самом деле инструкции и выражения мы уже использовали. Инструкции — это указания (команды) для компьютера выполнить некое действие, они не воз-

вращают значение. Выражение же оценивает результирующее значение. Давайте взглянем на несколько примеров.

Создание переменной и присвоение ей значения с помощью ключевого слова `let` — это инструкция. В листинге 3.1 `let y = 6;` является инструкцией.

Листинг 3.1. Объявление функции `main`, содержащее одну инструкцию

src/main.rs

```
fn main() {
    let y = 6;
}
```

Определения функций тоже являются инструкциями; весь приведенный выше пример является отдельной инструкцией.

Инструкции не возвращают значений. Следовательно, вы не можете назначить инструкцию `let` еще одной переменной, как это пытается сделать следующий код. Произойдет ошибка:

src/main.rs

```
fn main() {
    let x = (let y = 6);
}
```

Когда вы выполните эту программу, то полученная ошибка будет выглядеть следующим образом[1]:

```
$ cargo run
    Compiling functions v0.1.0 (file:///projects/functions)
error: expected expression, found statement (`let`)
 --> src/main.rs:2:14
  |
2 |     let x = (let y = 6);
  |     ^^^
  |
  = note: variable declaration using `let` is a statement
```

Инструкция `let y = 6` не возвращает значения, поэтому переменную `x` не к чему привязать. Это отличается от того, что происходит в других языках, таких как C и Ruby, где инструкция присваивания возвращает значение присваивания. В этих языках вы можете написать `x = y = 6`, при этом как `x`, так и `y` будут содержать значение `6`. В Rust дела обстоят иначе.

Выражения оцениваются и составляют бо́льшую часть остального кода, который вы будете писать на языке Rust. Возьмем простую математическую операцию `5 + 6`, то есть выражение, которое в результате вычисления принимает значение `11`. Выражения могут быть частью инструкций: в листинге 3.1 число `6` в инструкции

[1] `ошибка: ожидалось выражение, обнаружена инструкция (`let`)`

`let y = 6;` является выражением, которое в результате вычисления принимает значение `6`. Вызов функции является выражением. Вызов макрокоманды является выражением. Блок, который мы используем для создания новых областей, `{}`, является выражением, например:

src/main.rs

```
fn main() {
    let x = 5;

 ❶ let y = {❷
        let x = 3;
    ❸ x + 1
    };

    println!("Значение y равно {}", y);
}
```

Выражение ❷ — это блок, который в данном случае в результате вычисления принимает значение 4. Его значение привязывается к переменной `y` как часть инструкции `let` ❶. Обратите внимание на строку кода без точки с запятой в конце ❸ — она не похожа на большинство строк кода, которые вы видели до сих пор. Выражения не включают конечные точки с запятой. Если вы добавляете точку с запятой в конец выражения, то превращаете его в инструкцию, которая в таком случае не возвращает значение. Имейте это в виду при дальнейшем анализе значений и выражений, возвращаемых из функций.

Функции с возвращаемыми значениями

Функции могут возвращать значения в вызывающий их код. Мы не даем возвращаемым значениям имена, но объявляем их тип после стрелки (`->`). В Rust возвращаемое функцией значение является синонимом значения конечного выражения в блоке тела функции. Вы можете вернуться из функции досрочно, используя ключевое слово `return` и указав значение, но большинство функций неявным образом возвращают последнее выражение. Вот пример функции, которая возвращает значение:

src/main.rs

```
fn five() -> i32 {
    5
}

fn main() {
    let x = five();

    println!("Значение x равно {}", x);
}
```

В функции `five` нет вызовов функций, макрокоманд или даже инструкций `let` — только одно число `5`. В Rust такая функция совершенно допустима. Обратите вни-

мание, что тип возвращаемого значения функции также указан как `-> i32`. Попробуйте выполнить этот код. Результат должен выглядеть следующим образом:

```
$ cargo run
    Compiling functions v0.1.0 (file:///projects/functions)
     Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
      Running `target/debug/functions`
Значение x равно 5
```

Число `5` в функции `five` является возвращаемым значением функции, поэтому тип возвращаемого значения равен `i32`. Давайте изучим это подробнее. Здесь имеется два важных момента: во-первых, строка кода `let x = five();` показывает, что мы используем возвращаемое функцией значение для инициализации переменной. Поскольку функция `five` возвращает `5`, эта строка кода совпадает со следующей:

```
let x = 5;
```

Во-вторых, функция `five` не имеет параметров и определяет тип возвращаемого значения, но тело функции — просто число `5` без точки с запятой, потому что это выражение, значение которого мы хотим вернуть.

Давайте взглянем еще на один пример:

src/main.rs

```
fn main() {
    let x = plus_one(5);

    println!("Значение x равно {}", x);
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

При выполнении этого кода будет выведено

```
Значение x равно 6
```

Но если мы поставим точку с запятой в конце строки кода, содержащей `x + 1`, поменяв ее с выражения на инструкцию, то произойдет ошибка.

src/main.rs

```
fn main() {
    let x = plus_one(5);

    println!("Значение x равно {}", x);
}

fn plus_one(x: i32) -> i32 {
    x + 1;
}
```

Компиляция этого кода произведет ошибку, как показано ниже:

```
error[E0308]: mismatched types
 --> src/main.rs:7:28
  |
7 |   fn plus_one(x: i32) -> i32 {
  |  ____________________________^
8 | |     x + 1;
  | |          = help: consider removing this semicolon
9 | | }
  | |_^ expected i32, found ()
  |
  = note: expected type `i32`
             found type `()`
```

Главное сообщение об ошибке «несовпадающие типы» (*mismatched types*) раскрывает суть проблемы с этим кодом. Определение функции `plus_one` говорит о том, что она будет возвращать тип `i32`, но инструкции не вычисляют значения, что выражается через `()`, пустой кортеж. Поэтому ничего не возвращается, а это противоречит определению функции и приводит к ошибке. В приведенных выше данных язык Rust выдает сообщение, которое, возможно, поможет решить эту проблему: язык предлагает удалить точку с запятой, что и устранит ошибку.

Комментарии

Все программисты стремятся делать свой код простым в понимании, но иногда без дополнительных объяснений не обойтись. В этих случаях программисты оставляют в исходном коде заметки или комментарии, которые компилятор будет игнорировать, но люди, читающие код, возможно, найдут их полезными.

Вот простой комментарий:

```
// Здравствуй, Мир
```

В Rust комментарии должны начинаться с двух слешей и продолжаться до конца строки кода. Для комментариев, которые выходят за пределы одной строки кода, вам нужно будет включить `//` в каждую строку, например:

```
// Итак, мы делаем нечто настолько сложное и длинное, что нам понадобилось
// несколько строк, чтобы написать комментарий!
// Ух! Будем надеяться, что этот комментарий прояснит, что происходит.
```

Комментарии также могут размещаться в конце строк, содержащих код:

src/main.rs

```
fn main() {
    let lucky_number = 7; // Мне сегодня точно повезет
}
```

Но чаще вы будете видеть, что они используются в следующем формате, где комментарий находится на отдельной строке программы над кодом, который он аннотирует:

src/main.rs

```
fn main() {
    // Мне сегодня точно повезет
    let lucky_number = 7;
}
```

В Rust также есть еще один вид комментариев — документационные комментарии, которые мы обсудим в разделе «Публикация упаковки для Crates.io».

Управление потоком

Решение о том, следует ли выполнять код в зависимости от истинности условия и нужно ли повторно выполнять код до тех пор, пока условие остается истинным, — это основные блоки в большинстве языков программирования. Наиболее частыми конструкциями, позволяющими управлять потоком исполнения кода Rust, являются выражения `if` и циклы.

Выражения if

Выражение `if` позволяет ветвить код в зависимости от условий. Вы задаете условие, а затем заявляете: «если условие удовлетворено, выполнить блок кода. Если условие не удовлетворено, не выполнять блок кода».

Создайте новый проект под названием branches («ветви») в каталоге проектов projects для изучения выражения `if`. В файле src/main.rs введите следующий код:

src/main.rs

```
fn main() {
    let number = 3;

    if number < 5 {
        println!("условие было истинным");
    } else {
        println!("условие было ложным");
    }
}
```

Все выражения `if` начинаются с ключевого слова `if` и последующего условия. В данном случае условие проверяет, имеет ли переменная `number` значение меньше `5`. Блок кода, который мы хотим исполнить, если условие является истинным, помещается сразу после условия внутри фигурных скобок. Блоки кода, связанные с условиями в выражениях `if`, иногда называются рукавами, как и рукава в выражениях `match`, которые мы рассматривали в разделе «Сравнение загаданного числа с секретным числом».

При необходимости мы также можем включить выражение `else`. Это мы как раз и сделали, чтобы дать программе альтернативный блок кода для исполнения, если

в результате вычисления условие примет значение «ложь». Если вы не укажете выражение `else` и условие окажется ложным, то программа просто пропустит блок `if` и перейдет к следующему коду.

Попробуйте выполнить код ниже. На выходе вы увидите следующий результат:

```
$ cargo run
    Compiling branches v0.1.0 (file:///projects/branches)
     Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
      Running `target/debug/branches`
условие было истинным
```

Давайте изменим значение переменной `number` на значение, которое делает условие ложным, и посмотрим, что произойдет:

```
let number = 7;
```

Выполните программу еще раз и посмотрите на результат:

```
$ cargo run
    Compiling branches v0.1.0 (file:///projects/branches)
     Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
      Running `target/debug/branches`
условие было ложным
```

Также стоит отметить, что условие в этом коде должно иметь тип `bool`. Если условие не имеет типа `bool`, произойдет ошибка. Например, выполните следующий код:

src/main.rs

```
fn main() {
    let number = 3;

    if number {
        println!("число было равно трем");
    }
}
```

На этот раз условие `if` принимает значение 3, и Rust выдает ошибку:

```
error[E0308]: mismatched types
 --> src/main.rs:4:8
  |
4 |     if number {
  |        ^^^^^^ expected bool, found integral variable
  |
  = note: expected type `bool`
             found type `{integer}`
```

Ошибка указывает на то, что Rust ожидал значение типа `bool`, но получил значение целочисленного типа. В отличие от таких языков, как Ruby и JavaScript, язык Rust не будет автоматически пытаться конвертировать не-булевы типы в булев тип. Следует выражаться однозначно и всегда предоставлять выражению `if`

значение булева типа в качестве условия. К примеру, если мы хотим, чтобы блок кода `if` выполнялся только тогда, когда число не равно 0, то мы можем изменить выражение `if` на следующее:

src/main.rs

```
fn main() {
    let number = 3;

    if number != 0 {
        println!("число было отличным от нуля");
    }
}
```

При выполнении этого кода будет выведено:

```
число было отличным от нуля
```

Обработка нескольких условий с помощью else if

Вы можете иметь несколько условий, совместив `if` и `else` в выражение `else if`. Например:

src/main.rs

```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("число делится на 4");
    } else if number % 3 == 0 {
        println!("число делится на 3");
    } else if number % 2 == 0 {
        println!("число делится на 2");
    } else {
        println!("число не делится на 4, 3 и 2");
    }
}
```

У программы есть четыре варианта развития событий. После ее выполнения вы должны увидеть следующий результат:

```
$ cargo run
    Compiling branches v0.1.0 (file:///projects/branches)
     Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
      Running `target/debug/branches`
число делится на 3
```

Когда программа исполняется, она проверяет каждое выражение `if` по очереди и исполняет первое тело, для которого условие соблюдается. Обратите внимание, что даже если 6 делится на 2, мы не увидим результат `число делится на 2`, равно как и не будет текста `число не делится на 4, 3 и 2` из блока `else`. Это обусловлено тем, что Rust исполняет блок только для первого истинного условия. Как только он его находит, язык не проверяет остальные условия.

Использование слишком большого числа выражений `else if` загромождает код, поэтому, если у вас более одного выражения, вам, возможно, потребуется рефакторизовать код. В главе 6 описывается мощная конструкция ветвления в Rust под названием `match`, предназначенная для таких случаев.

Использование выражения if в инструкции let

Поскольку `if` является выражением, мы можем использовать его в правой части инструкции `let`, как в листинге 3.2.

Листинг 3.2. Назначение переменной результата выражения `if`

src/main.rs

```
fn main() {
    let condition = true;
    let number = if condition {
        5
    } else {
        6
    };

    println!("Значение числа равно {}", number);
}
```

Переменная `number` будет привязана к значению на основе результата выражения `if`. Выполните этот код и посмотрите, что произойдет:

```
$ cargo run
    Compiling branches v0.1.0 (file:///projects/branches)
     Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
      Running `target/debug/branches`
Значение числа равно 5
```

Помните, что блоки кода в результате вычисления принимают значение последнего выражения в них, и сами числа тоже являются выражениями. В данном случае значение всего выражения `if` зависит от того, какой блок кода исполняется. Это означает, что значения, которые потенциально будут результатами из каждого ветвления `if`, должны иметь одинаковый тип. В листинге 3.2 результаты как ветвления `if`, так и ветвления `else` были целыми числами `i32`. Если типы не совпадают, как в следующем примере, то мы получим ошибку:

src/main.rs

```
fn main() {
    let condition = true;

    let number = if condition {
        5
    } else {
        "шесть"
    };

    println!("Значение числа равно {}", number);
}
```

Если мы попытаемся скомпилировать этот код, то получим ошибку. Ветви `if` и `else` имеют несовместимые типы значений, и Rust точно указывает, где найти проблему в программе[1]:

```
error[E0308]: if and else have incompatible types
 --> src/main.rs:4:18
  |
4 |       let number = if condition {
  |  __________________^
5 | |         5
6 | |     } else {
7 | |         "six"
8 | |     };
  | |_____^ expected integral variable, found &str
  |
  = note: expected type `{integer}`
             found type `&str`
```

Выражение в блоке `if` в результате вычисления получает целое число, а выражение в блоке `else` — строковое значение. Это не будет работать, потому что переменные должны иметь одинаковый тип. Rust должен знать во время компиляции, какой тип у переменной `number`. Это позволяет ему верифицировать, что тип переменной `number` допустим везде, где мы ее используем. Rust не смог бы этого сделать, если бы тип переменной `number` определялся только во время выполнения. Компилятор был бы сложнее и давал бы меньше гарантий кода, если бы ему приходилось отслеживать несколько гипотетических типов для любой переменной.

Повторение с помощью циклов

Часто бывает полезным исполнить блок кода более одного раза. Для этой работы Rust предоставляет несколько видов циклов. Цикл выполняется в коде внутри тела цикла до конца, а затем сразу же начинает сначала. Для того чтобы поэкспериментировать с циклами, давайте создадим новый проект под названием loops («циклы»).

Язык Rust имеет три вида циклов: `loop`, `while` и `for`. Давайте поработаем с каждым из них.

Повторение кода с помощью цикла loop

Ключевое слово `loop` говорит о том, что нужно исполнять блок кода снова и снова бесконечное количество раз или до тех пор, пока вы не скажете ему остановиться.

В качестве примера измените файл src/main.rs в каталоге loops, чтобы он выглядел следующим образом:

---

[1] `ошибка[E0308]: if и else имеют несовместимые типы`

src/main.rs

```
fn main() {
    loop {
        println!("еще раз!");
    }
}
```

Когда мы выполним эту программу, мы увидим, как текст `еще раз!` будет печататься снова и снова до тех пор, пока мы не остановим программу вручную. Большинство терминалов поддерживают клавиатурное сокращение Ctrl-C для прерывания программ, застрявших в бесконечном цикле. Попробуйте сами:

```
$ cargo run
    Compiling loops v0.1.0 (file:///projects/loops)
     Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
      Running `target/debug/loops`
еще раз!
еще раз!
еще раз!
еще раз!
^Сеще раз!
```

Символ `^C` обозначает место, где вы нажали Ctrl-C. Вы можете и не увидеть текст `еще раз!` после `^C`, в зависимости от того, где код находился в цикле, когда он получил сигнал остановки.

К счастью, язык Rust обеспечивает еще один, более надежный способ вырваться из цикла. Вы можете разместить ключевое слово `break` внутри цикла, тем самым сообщив программе, когда следует прекратить исполнение цикла. Напомним, что мы сделали это в игре на угадывание в разделе «Выход из игры после правильного загаданного числа» (с. 57) для выхода из программы, когда пользователь победил в игре, угадав правильное число.

Возвращение значений из циклов

Одно из применений цикла `loop` состоит в повторной попытке выполнения операции, которая, как вы знаете, может не сработать, как, например, проверка того, завершил ли поток задание. Однако вам, возможно, потребуется передать результат этой операции остальной части кода. Для этого вы добавляете значение, которое хотите возвратить после выражения `break`, используемое для остановки цикла. Это значение будет возвращено из цикла, и тогда вы сможете его использовать, как показано здесь:

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
```

```
        }
    };

    println!("Результат равен {}", result);
}
```

Перед началом цикла мы объявляем переменную `counter` и инициализируем ее со значением 0. Затем мы объявляем переменную с именем `result` для хранения значения, возвращаемого из цикла. Во время каждой итерации цикла мы прибавляем 1 к переменной `counter`, а затем проверяем ее значение. Когда переменная `counter` равна 10, мы используем ключевое слово `break` со значением `counter * 2`. После цикла мы используем точку с запятой для завершения инструкции, которая присваивает значение цикла переменной `result`. Наконец мы печатаем значение в переменной `result`, которое в данном случае равно 20.

Условные циклы с использованием while

Зачастую бывает полезно, когда программа оценивает условие внутри цикла. И до тех пор, пока условие является истинным, цикл выполняется. Когда условие перестает быть истинным, программа вызывает `break`, останавливая цикл. Этот тип цикла реализуется с помощью комбинации ключевых слов `loop`, `if`, `else` и `break`. Вы можете сделать это сейчас в программе, если хотите.

Однако этот паттерн является настолько частым, что для него в Rust встроена конструкция, которая называется циклом `while`. В листинге 3.3 используется `while`: программа делает три итерации, каждый раз ведя обратный отсчет, а затем, после окончания цикла, выводит еще одно сообщение и завершает работу.

Листинг 3.3. Использование цикла `while` для выполнения кода до тех пор, пока условие остается истинным

src/main.rs

```
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{}!", number);

        number = number - 1;
        }

    println!("Поехали!!!");
}
```

Эта конструкция понятнее, и она устраняет вложенность, которая была бы необходима, если бы вы использовали `loop`, `if`, `else` и `break`. До тех пор пока условие истинно, код выполняется; в противном случае он выходит из цикла.

Осуществление цикла в коллекции с помощью for

Вы можете использовать конструкцию `while` для осуществления цикла в элементах коллекции, такой как массив. Давайте взглянем на листинг 3.4.

Листинг 3.4. Осуществление цикла в элементах коллекции с помощью while

src/main.rs

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        println!("Значение равно {}", a[index]);

        index = index + 1;
    }
}
```

Здесь код подсчитывает число элементов в массиве. Он начинается с индекса 0, а затем повторяется до тех пор, пока не достигнет конечного индекса в массиве (то есть когда выражение `index < 5` больше не является истинным). Выполнение этого кода выведет каждый элемент массива:

```
$ cargo run
    Compiling loops v0.1.0 (file:///projects/loops)
     Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
      Running `target/debug/loops`
Значение равно 10
Значение равно 20
Значение равно 30
Значение равно 40
Значение равно 50
```

Все пять значений массива, как и предполагалось, появляются в терминале. Даже если индекс в какой-то момент достигнет значения 5, цикл остановится перед попыткой извлечь шестое значение из массива.

Но при таком подходе возможны ошибки. В программе может начаться паника, если длина индекса будет неправильной. Работа при этом подходе осуществляется медленно, потому что компилятор добавляет код времени выполнения для проверки условия каждого элемента во время каждой итерации цикла.

В качестве более короткой альтернативы можно использовать цикл `for` и исполнять код для каждого элемента в коллекции. Цикл `for` выглядит, как показано в коде из листинга 3.5.

Листинг 3.5. Осуществление цикла в элементах коллекции с помощью for

src/main.rs

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a.iter() {
        println!("Значение равно {}", element);
    }
}
```

Когда мы выполним этот код, увидим тот же результат, что и в листинге 3.4. Важно, что теперь мы повысили безопасность кода и устранили вероятность ошибок, которые могут возникнуть в результате выхода за пределы массива или оттого, что некоторые элементы в массиве пропущены.

Например, если в коде из листинга 3.4 вы удалили элемент из массива, но забыли обновить условие до `while index < 4`, то код поднимет панику. Используя цикл `for`, вам не нужно помнить о том, что требуется изменить какой-либо фрагмент кода, если вы поменяли число значений в массиве.

Безопасность и лаконичность циклов `for` делают их наиболее часто используемой цикличной конструкцией в языке Rust. Даже в ситуациях, когда нужно выполнить код некоторое число раз, как в примере обратного отсчета из листинга 3.3, в котором использовался цикл `while`, большинство растиан будут применять цикл `for`. Для этого можно использовать тип `Range`, предусмотренный стандартной библиотекой, который генерирует все числа в последовательности, начинающейся с одного числа и заканчивающейся перед еще одним числом.

Вот как выглядит обратный отсчет с использованием цикла `for` и еще одного метода, о котором мы пока не говорили, — `rev` — для инвертирования интервала:

src/main.rs

```
fn main() {
    for number in (1..4).rev() {
        println!("{}!", number);
    }
    println!("Поехали!!!");
}
```

Этот код выглядит получше, не правда ли?

Итоги

У вас получилось! Глава вышла довольно объемная: вы узнали о переменных, скалярных и составных типах данных, функциях, комментариях, выражениях `if` и циклах! Если вы хотите поработать с понятиями, описанными в этой главе, попробуйте написать программу, которая:

- Конвертирует температуру из градусов по Фаренгейту в градусы Цельсия и наоборот.
- Генерирует *n*-е число Фибоначчи.
- Выводит текст рождественской песни «Двенадцать дней Рождества» (*The Twelve Days of Christmas*), пользуясь повторами в песне.

Когда вы будете готовы двигаться дальше, мы поговорим о понятии языка Rust, которое, как правило, отсутствует в других языках программирования, — о владении.

4 Концепция владения

Владение — это уникальное средство Rust, оно позволяет этому языку давать гарантии безопасности памяти, не обращаясь при этом к сборщику мусора. Поэтому очень важно понять принцип работы владения в Rust. В этой главе мы поговорим о владении, а также о нескольких связанных средствах: заимствовании, срезах (отрезках) и о том, как Rust располагает данные в памяти.

Что такое владение?

Центральным средством языка Rust является владение. Концепция владения проста, но несмотря на легкость очень важна.

Все программы должны управлять тем, как они используют память компьютера во время работы. В некоторых языках есть сбор мусора, в ходе которого происходит постоянный поиск неиспользуемой памяти во время выполнения программы. В других языках программист должен распределять и освобождать память. Rust использует третий подход: управление памятью осуществляется через систему владения с набором правил, которые компилятор проверяет во время компиляции. Во время выполнения владения ни одно из его свойств не замедляет работу программы.

Поскольку для многих программистов владение является новым понятием, требуется некоторое время на то, чтобы к нему привыкнуть. Хорошая новость состоит в том, что чем опытнее вы становитесь в работе с Rust и с правилами системы владения, тем естественнее вы сможете разрабатывать безопасный и эффективный код. Продолжайте в том же духе!

Когда вы поймете, что такое владение, у вас будет прочная основа для понимания языковых средств, которые делают Rust уникальным. В этой главе вы усвоите понятие владения, проработав несколько примеров, посвященных очень распространенной структуре данных — строкам.

СТЕК И КУЧА

Во многих языках программирования не нужно часто думать о стеке и куче. Но в языке системного программирования, таком как Rust, то, где находится значение — в стеке или куче, — существенно влияет на то, как себя ведет язык и почему приходится принимать те или иные решения. Соответствующие компоненты владения будут описаны применительно к стеку и куче далее в этой главе, поэтому ниже приведено краткое подготовительное пояснение.

И стек, и куча являются частями памяти, которые доступны коду для использования во время выполнения, но они структурированы по-разному. Стек хранит значения в том порядке, в котором он их получает, а удаляет значения в обратном порядке. Этот принцип называется «последним вошел, первым вышел». Подумайте о стопке тарелок: когда вы добавляете тарелки в стопку, вы кладете их наверх, а когда вам нужно взять тарелку, вы снимаете ее сверху. Добавить тарелки в середину и вниз или убрать их оттуда не получится! Добавление данных называется вталкиванием в стек, а удаление данных называется выталкиванием из стека.

Все данные, хранящиеся в стеке, должны иметь известный фиксированный размер. А вот данные, размер которых во время компиляции неизвестен или может измениться, должны храниться в куче. Куча менее организованна: когда вы помещаете данные в кучу, вы запрашиваете определенный объем пространства. Операционная система находит достаточно большое пустое место в куче, помечает его как используемое и возвращает указатель, являющийся адресом этого места. Такой процесс называется выделением пространства в куче и иногда сокращается до двух слов — выделение пространства. Вталкивание значений в стек не считается выделением. Поскольку указатель известен и имеет фиксированный размер, вы можете хранить указатель в стеке, но, когда вам нужны фактические данные, вы должны проследовать по указателю.

Представьте себе, как вас обслуживают в ресторане. Когда вы входите, то говорите, сколько вас, и сотрудник находит свободный столик, подходящий всем, и провожает вас. Если кто-то из вашей компании приходит позже, он может спросить, где вы сидите, чтобы вас найти.

Вталкивание в стек происходит быстрее, чем выделение пространства в куче, поскольку операционной системе не нужно искать место для хранения новых данных. Это место всегда находится в верхней части стека. В сравнении с этим выделение пространства в куче требует больше усилий, поскольку операционная система сначала должна найти достаточно большое пространство для хранения данных, а затем выполнить служебные действия по подготовке к следующему выделению.

Доступ к данным в куче происходит медленнее, чем к данным в стеке, поскольку для этого необходимо проследовать по указателю. Современные процессоры работают быстрее, если они меньше перемещаются по памяти. Продолжая аналогию, рассмотрим пример с официантом в ресторане, принимающим заказы от многих столиков. Эффективнее всего получить все заказы от одного столика и только потом переходить к следующему. Процесс получения заказа от столика A, затем от столика B, далее еще одного заказа от A, а после снова от B будет гораздо медленнее. Точно

так же процессор выполняет работу лучше, если он работает с данными, которые находятся близко к другим данным (как это происходит в стеке), а не далеко (как это бывает в куче). Выделение большого объема пространства в куче также занимает немало времени.

Когда код вызывает функцию, значения, переданные в функцию (в том числе потенциально, указатели на данные в куче) и локальные переменные функции вталкиваются в стек. Когда функция заканчивается, эти значения выталкиваются из стека.

Отслеживание того, какие части кода используют тот или иной тип данных в куче, сведение к минимуму объема повторяющихся данных и очистка неиспользуемых данных в куче, чтобы сохранить свободное пространство, — все эти проблемы решает владелец. Как только вы поймете, что такое владение, вам не нужно будет задумываться о стеке и куче. Но если вы знаете, что управление данными кучи является причиной существования владения, то вы поймете, почему оно работает именно так.

Правила владения

Прежде всего давайте взглянем на правила владения. Помните об этих правилах по мере того, как мы будем отрабатывать примеры, которые их иллюстрируют:

- Каждое значение в языке Rust имеет переменную, которая называется его владельцем.
- В каждый момент времени может существовать только один владелец.
- Если владелец выйдет из области видимости, значение будет отброшено.

Область видимости переменной

Мы уже отработали пример программы Rust в главе 2. Теперь, изучив базовый синтаксис, мы не будем включать весь код `fn main() {` в примеры, поэтому, если вы будете заниматься дальше, то вам придется помещать последующие примеры в функцию `main` вручную. В результате наши примеры будут немного короче, позволяя сосредотачиваться на деталях, а не на стереотипном коде.

В качестве первого примера владения мы рассмотрим область видимости нескольких переменных. Область видимости — это диапазон внутри программы, для которого элемент является действительным. Будем считать, что у нас есть переменная, которая выглядит следующим образом:

```
let s = "hello";
```

Переменная `s` относится к строковому литералу, где значение строки жестко закодировано в текст нашей программы. Переменная является действительной с момента ее объявления до конца текущей области видимости. В листинге 4.1 приведены комментарии, аннотирующие места, где действует переменная `s`.

Листинг 4.1. Переменная и область, в которой эта переменная действует

```
{                      // s здесь не действует; она еще не объявлена
    let s = "hello ";  // s действует с этого момента и далее

                       // что-то сделать с s
}                      // эта область закончилась, и s больше не действует
```

Другими словами, здесь имеется два важных момента во времени:

- когда переменная `s` входит в область видимости, она становится действительной;
- она остается действительной до тех пор, пока не выйдет из области видимости.

В этой точке связь между областями видимости и действительными переменными аналогична той, что существует в других языках программирования. Теперь мы будем опираться на это понимание, представив тип `String`.

Строковый тип

В целях иллюстрации правил владения нам нужен более сложный тип данных, чем те, о которых мы говорили в разделе «Типы данных». Все описанные ранее типы хранятся в стеке и выталкиваются из него, когда их область видимости заканчивается. Но мы хотим посмотреть на данные, хранящиеся в куче, и понять, каким образом Rust знает, когда нужно очищать эти данные.

В качестве примера мы будем использовать тип `String` и сосредоточимся на тех частях указанного типа, которые относятся к владению. Эти аспекты также применимы к другим сложным типам данных, предусмотренным стандартной библиотекой, а также к тем, которые вы создаете. О типе `String` мы поговорим подробнее в главе 8.

Мы уже встречали строковые литералы, где значение строки жестко кодируется в программу. Указанные литералы удобны, но они не подходят для каждой ситуации, в которой мы используем текст. Одна из причин заключается в том, что они неизменяемы. Вторая причина — не каждое строковое значение может быть известно, когда мы пишем код. Например, что делать, если мы хотим взять у пользователя данные и сохранить их? Для этих ситуаций в Rust есть второй строковый тип — `String`. Пространство для этого типа выделяется в куче, и по этой причине он способен хранить объем текста, который нам не известен во время компиляции. Вы можете создать экземпляр типа `String` из строкового литерала с помощью функции `from`, например:

```
let s = String::from("hello");
```

Двойное двоеточие (`::`) — это оператор, который позволяет использовать пространство имен конкретной функции `from` под типом `String` вместо имени наподобие `string_from`. Мы обсудим этот синтаксис подробнее в разделе «Синтаксическое оформление метода», а также при рассмотрении организации пространств

имен с помощью модулей в разделе «Пути для ссылки на элемент в дереве модулей» (с. 151).

Этот вид строки может изменяться:

```
let mut s = String::from("hello");

s.push_str(", world!"); // push_str() добавляет литерал к экземпляру типа String

println!("{}", s);      // эта инструкция выводит `hello, world!`
```

Тогда в чем же разница? Почему тип `String` может изменяться, а литералы — нет? Разница заключается в том, как эти два типа работают с памятью.

Память и выделение пространства

В случае строкового литерала мы знаем его содержимое во время компиляции, поэтому текст жестко кодируется непосредственно в конечный исполняемый файл. Вот почему строковые литералы быстры и эффективны. Но эти свойства существуют только вследствие неизменяемости строкового литерала. К сожалению, мы не можем поместить большой объем памяти в двоичный файл для каждого фрагмента текста, размер которого неизвестен при компиляции и который может измениться во время выполнения программы.

В случае с типом `String` для поддержки изменяемого, наращиваемого фрагмента текста, нам нужно выделить объем памяти в куче, неизвестный во время компиляции, для хранения содержимого. Это означает, что:

- Память должна быть запрошена из операционной системы во время выполнения.
- Нам нужен способ вернуть эту память в операционную систему, когда мы закончим работу с экземпляром типа `String`.

Эта первая часть выполняется нами: когда мы вызываем функцию `String::from`, ее реализация запрашивает необходимую память. Так же происходит и в других языках программирования.

Однако вторая часть отличается от предыдущей. В языках со сборщиком мусора (GC от англ. *garbage collector*) он отслеживает и очищает память, которая больше не используется, и нам не нужно об этом думать. Без сборщика мусора мы сами должны узнать, когда память больше не используется, и вызвать код, который возвращает ее. Мы делали так, когда ее запрашивали. Правильное выполнение этой работы исторически являлось трудновыполнимой задачей программирования. Если мы забудем, то потеряем память. Если мы сделаем это слишком рано, то получим недействительную переменную. Если мы сделаем это дважды, то это тоже будет дефектом. Нам нужно связать в пару одно `allocate` («выделение») с одним `free` («высвобождением»).

Язык Rust выбирает другой путь: память возвращается автоматически, как только переменная, которой она принадлежит, выходит из области видимости. Вот версия примера области видимости из листинга 4.1 с использованием экземпляра типа `String` вместо строкового литерала:

```
{
    let s = String::from("hello"); // s действительна с этого момента и далее

    // что-то сделать с s
}                                  // эта область закончилась, и
                                   // s больше не действует
```

Существует естественная точка, в которой мы можем вернуть память, необходимую экземпляру типа `String`, в операционную систему — когда переменная `s` выходит из области видимости. Когда переменная выходит из области видимости, Rust вызывает специальную функцию. Эта функция называется `drop` («отбросить»), и именно туда автор экземпляра типа `String` может поместить код для возврата памяти. Rust вызывает функцию `drop` автоматически в месте, где расположена закрывающая фигурная скобка.

ПРИМЕЧАНИЕ

В языке C++ паттерн высвобождения ресурсов в конце жизненного цикла элемента иногда называется «Получение ресурса — это инициализация» (то есть получение ресурса совмещается с инициализацией, а высвобождение — с уничтожением объекта, от англ. *Resource Acquisition Is Initialization*, RAII). Функция `drop` в языке Rust, вероятно, вам знакома, если вы использовали паттерны RAII.

Этот паттерн оказывает глубокое влияние на то, как код пишется на Rust. Сейчас все кажется простым, но поведение кода окажется неожиданным в более сложных ситуациях, когда нужно, чтобы данные, для которых мы выделили пространство в куче, использовались более одной переменной. Давайте рассмотрим некоторые ситуации.

Способы взаимодействия переменных и данных: перемещение

В Rust многочисленные переменные взаимодействуют с одними и теми же данными по-разному. Рассмотрим пример использования целого числа в листинге 4.2.

Листинг 4.2. Передача целочисленного значения от переменной `x` переменной `y`

```
let x = 5;
let y = x;
```

Мы догадываемся, что здесь происходит: «связать значение `5` с `x`; затем сделать копию значения в `x` и связать его с `y`». Теперь у нас есть две переменные, `x` и `y`, и обе равны `5`. Это действительно то, что происходит, потому что целые числа — это простые значения с известным фиксированным размером, и эти два значения `5` помещаются в стек.

Теперь давайте посмотрим на версию с экземпляром типа `String`:

```
let s1 = String::from("hello");
let s2 = s1;
```

Этот код очень похож на предыдущий. Поэтому мы будем исходить из того, что характер его работы будет таким же, то есть вторая строка кода сделает копию значения в `s1` и свяжет ее с `s2`. Но это не совсем верно.

Взгляните на рис. 4.1, и вы увидите, что на самом деле происходит с экземпляром типа `String`. Экземпляр типа `String` состоит из трех частей, показанных слева: указателя на память, в которой находится содержимое строки, ее длины и емкости. Эта группа данных хранится в стеке. Справа находится память в куче, в которой расположено ее содержимое.

Длина — это объем памяти в байтах, который использует содержимое экземпляра типа `String` в настоящее время. Емкость — это общий объем памяти в байтах, который экземпляр типа `String` получил от операционной системы. Разница между длиной и емкостью имеет значение, но не в этом случае, поэтому на данный момент емкость можно проигнорировать.

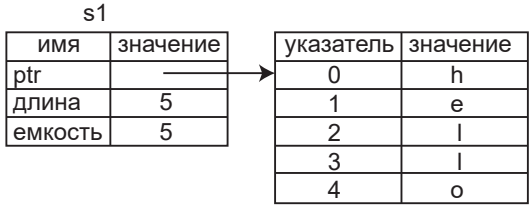

Рис. 4.1. Представление в памяти экземпляра типа String, содержащего значение «hello», привязанное к переменной s1

Когда мы присваиваем переменную `s1` переменной `s2`, данные экземпляра типа `String` копируются, то есть указатель, длина и емкость, которые находятся в стеке. Мы не копируем данные в куче, на которую ссылается указатель. Другими словами, представление данных в памяти выглядит так, как показано на рис. 4.2.

Это представление не похоже на рис. 4.3, а именно так выглядела бы память, если бы вместо этого Rust копировал данные кучи. Если бы язык сделал это, то операция `s2 = s1` могла бы быть очень затратной с точки зрения производительности времени выполнения, если бы данные в куче были объемными.

Ранее мы говорили, что, когда переменная выходит из области видимости, Rust автоматически вызывает функцию `drop` и очищает память кучи для этой переменной. Но на рис. 4.2 показано, как оба указателя данных показывают одно и то же место. В этом как раз и проблема: когда `s2` и `s1` выходят из области видимости, они обе будут пытаться высвободить одну и ту же память. Такая ситуация называется ошибкой двойного освобождения (*double free error*) и является одной из ошибок

обеспечения безопасности доступа к памяти, о которых мы упоминали ранее. Высвобождение памяти дважды приводит к ее повреждению, что потенциально может вызвать уязвимость в системе безопасности.

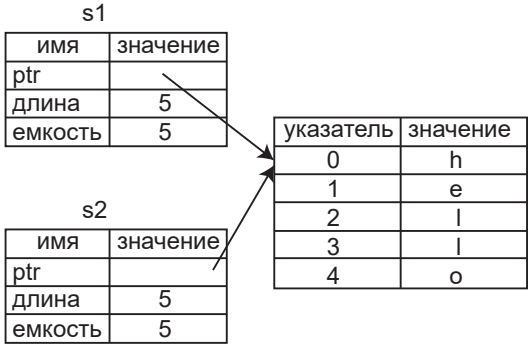

Рис. 4.2. Представление в памяти переменной s2, имеющей копию указателя, длины и емкости s1

Рис. 4.3. Еще один вариант, что выражение s2 = s1 могло бы сделать, если бы язык Rust копировал данные кучи

Чтобы обеспечить безопасность памяти в Rust в этой ситуации, происходит еще одна вещь. Язык не пытается скопировать выделенную память, а считает, что переменная `s1` больше не является действительной. Поэтому Rust не нужно ничего высвобождать, когда `s1` выходит из области видимости. Убедитесь сами, что произойдет, когда вы пытаетесь использовать `s1` после создания `s2`. Ничего не выйдет:

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

Вы получите ошибку ниже, потому что Rust мешает использовать недействительную ссылку[1]:

```
error[E0382]: use of moved value: `s1`
 --> src/main.rs:5:28
  |
3 |     let s2 = s1;
  |         -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value used here after move
  |
  = note: move occurs because `s1` has type `std::string::String`, which does
not implement the `Copy` trait
```

Если во время работы с другими языками вы слышали термины «мелкая копия» и «глубокая копия», то копирование указателя, длины и емкости без копирования данных, вероятно, звучит как создание мелкой копии. Но так как язык Rust также делает недействительной первую переменную, эта операция называется не «мелкая копия», а «перемещение» (move). В этом примере мы бы сказали, что `s1` была перемещена в `s2`. Поэтому то, что происходит на самом деле, показано на рис. 4.4.

И это решает проблему! Только переменная `s2` действительная; когда она выйдет из области видимости, эта переменная высвободит память, и дело сделано.

В дополнение, из этого вытекает одно конструктивное решение: Rust никогда не будет автоматически создавать «глубокие» копии данных. Следовательно, любое автоматическое копирование считается незатратным с точки зрения производительности времени выполнения.

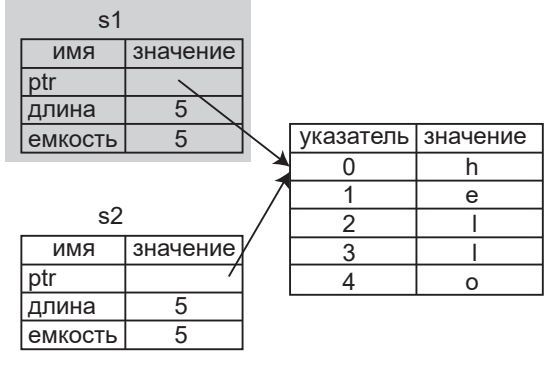

Рис. 4.4. Представление в памяти после того, как переменная s1 была сделана недействительной

[1] `ошибка[E0382]: использование перемещенного значения: `s1``

Пути взаимодействия переменных и данных: Clone

Если мы все-таки хотим сделать глубокую копию данных `String` из кучи, а не только данных из стека, то можно использовать часто встречающийся метод `clone`. Мы обсудим синтаксис методов в главе 5, но так как методы являются языковым средством, часто встречающимся во многих языках программирования, вы, вероятно, видели их раньше.

Вот пример метода `clone` в действии:

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

Он хорошо работает и ведет себя так, как показано на рис. 4.3, где данные кучи действительно копируются.

Когда вы видите вызов метода `clone`, вы знаете, что исполняется произвольный код, который может быть затратным. Указанный метод является визуальным индикатором того, что происходит что-то другое.

Данные только из стека: Copy

Есть еще одна загвоздка, о которой мы пока не говорили. Приведенный ниже код с целыми числами, часть которого была показана в листинге 4.2, работает и является допустимым:

```
let x = 5;
let y = x;

println!("x = {}, y = {}", x, y);
```

Но этот код, похоже, противоречит тому, о чем мы только что узнали: у нас нет вызова метода `clone`, но переменная `x` по-прежнему действует и не была перемещена в `y`.

Причина заключается в том, что такие типы, как целые числа, размер которых во время компиляции известен, хранятся полностью в стеке, и поэтому копии фактических значений создаются быстро. Это означает, что нет причин, по которым следовало бы запретить переменной `x` быть действительной после того, как мы создадим переменную `y`. Другими словами, здесь нет разницы между глубоким и мелким копированием, поэтому вызов метода `clone` будет делать то, что бывает при обычном мелком копировании, и мы можем его исключить.

В Rust имеется специальная аннотация, именуемая типажом `Copy`, которую можно размещать в типах, таких как целые числа, хранящихся в стеке (подробнее о типажах мы поговорим в главе 10). Если у типа есть типаж `Copy`, то более старая переменная по-прежнему пригодна для использования после ее присвоения другой переменной. Язык Rust не позволит нам аннотировать тип с помощью типажа

`Copy`, если тип или любая из его частей реализовали типаж `Drop`. Если типу нужно, чтобы происходило нечто особенное, когда значение выходит из области видимости, а мы добавляем аннотацию `Copy` в этот тип, то происходит ошибка времени компиляции. Чтобы узнать, как добавлять аннотацию `Copy` в тип, см. приложение B в конце книги.

Итак, какие типы аннотирует типаж `Copy`? Вы можете обратиться к документации по тому или иному типу и узнать об этом, но по общему правилу любая группа простых скалярных значений аннотируется типажом `Copy`. Типаж `Copy` не аннотирует ничего, что требует выделения пространства в куче или является некоторой формой ресурса. Вот несколько типов, которые аннотируются:

- Все целочисленные типы, такие как `u32`.
- Булев тип `bool` со значениями `true` и `false`.
- Символьный тип `char`.
- Все типы с плавающей точкой, такие как `f64`.
- Кортежи, если только они содержат типы, которые также аннотируются. Например, `(i32, i32)` аннотируются, а `(i32, String)` — нет.

Владение и функции

Семантика передачи значения функции аналогична семантике присвоения значения переменной. Передача переменной функции приведет к перемещению или копированию, так же как и присвоение значения переменной. В листинге 4.3 приведен пример с примечаниями, показывающими, где переменные входят в область видимости, а где выходят из нее.

Листинг 4.3. Функции с аннотированными владением и областью видимости

src/main.rs

```
fn main() {
    let s = String::from("hello"); // s входит в область видимости

    takes_ownership(s);            // значение s перемещается в функцию...
                                   // ... и поэтому больше здесь не действует

    let x = 5;                     // x входит в область видимости

    makes_copy(x);                 // x переместится в функцию, но
                                   // i32 копируема, поэтому нормально,
                                   // если x будет использоваться после этого

} // Здесь x выходит из области видимости, а затем s. Но поскольку
  // значение s было перемещено, ничего особенного не происходит.

fn takes_ownership(some_string: String) { // some_string входит
                                          // в область видимости
    println!("{}", some_string);
```

```
} // Здесь some_string выходит из области видимости и вызывается `drop`.
  // Поддерживающая память высвобождается.

fn makes_copy(some_integer: i32) { // some_integer входит в область видимости
    println!("{}", some_integer);
} // Здесь some_integer выходит из области видимости.
  // Ничего особенного не происходит.
```

Если бы мы попытались использовать переменную `s` после вызова функции `takes_ownership`, то Rust выдал бы ошибку времени компиляции. Эти статические проверки защищают от ошибок. Попробуйте добавить в функцию `main` код, который использует переменные `s` и `x`, чтобы увидеть, где их можно использовать, а где правила владения не позволяют этого делать.

Возвращаемые значения и область видимости

Возвращаемые значения также передают владение. Листинг 4.4 — это пример с аннотациями, похожими на приведенные в листинге 4.3.

Листинг 4.4. Передача владения возвращаемых значений

src/main.rs

```
fn main() {
    let s1 = gives_ownership();        // gives_ownership перемещает свое
                                       // возвращаемое значение в s1

    let s2 = String::from("hello");    // s2 входит в область видимости

    let s3 = takes_and_gives_back(s2); // s2 перемещается в
                                       // takes_and_gives_back, которая также
                                       // перемещает свое возвращаемое
                                       // значение в s3
} // Здесь s3 выходит из области видимости и отбрасывается. s2 выходит
  // из области видимости, но была перемещена, поэтому ничего не происходит.
  // s1 выходит из области и отбрасывается.

fn gives_ownership() -> String {             // gives_ownership переместит свое
                                             // возвращаемое значение в функцию,
                                             // которая ее вызывает

    let some_string = String::from("hello"); // some_string входит
                                             // в область видимости

    some_string                              // some_string возвращается и
                                             // выносится в вызывающую функцию
}

// takes_and_gives_back возьмет String и вернет String
fn takes_and_gives_back(a_string: String) -> String { // a_string приходит
                                                      // в область видимости

    a_string // a_string возвращается и выносится в вызывающую функцию
}
```

Владение переменной всякий раз следует одному и тому же паттерну: присвоение (передача) значения другой переменной перемещает его. Если переменная, содержащая данные в куче, выходит из области видимости, то значение будет очищено методом `drop`, если только данные не были перемещены в другую переменную.

Брать во владение, а затем возвращать владение назад при использовании каждой функции немного утомляет. Что делать, если мы хотим позволить функциям использовать значение, но не брать его во владение? Несколько раздражает, что все, что передается внутрь, также должно быть передано обратно, если нужно использовать это снова в дополнение к любым другим данным из тела функции, которые мы также, возможно, захотим вернуть.

С помощью кортежа можно возвращать несколько значений, как показано в листинге 4.5.

Листинг 4.5. Возвращение владения параметрами

src/main.rs

```
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("Длина '{}' равна {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() возвращает длину экземпляра типа String

    (s, length)
}
```

Но для идеи, которая должна быть общей, получается слишком много работы. К счастью, для этой идеи в Rust есть специальное языковое средство. Оно называется ссылками.

Ссылки и заимствование

Проблема с кортежным кодом в листинге 4.5 заключается в том, что мы должны вернуть экземпляр типа `String` вызывающей функции, чтобы по-прежнему можно было использовать экземпляр типа `String` после вызова функции `calculate_length`, потому что экземпляр типа `String` был перенесен в `calculate_length`.

Вот как, не беря значение во владение, можно определить и использовать функцию `calculate_length`, у которой есть ссылка на объект в качестве параметра:

src/main.rs

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);
```

```
    println!("Длина '{}' равна {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

Во-первых, обратите внимание, что исчезли кортежный код в объявлении переменной и возвращаемое значение функции. Во-вторых, заметьте, что в функцию `calculate_length` мы передаем `&s1` и для ее определения берем не `String`, а `&String`.

Амперсанды являются ссылками, они позволяют ссылаться на некое значение, не беря его во владение. На рис. 4.5 показана схема.

Рис. 4.5. Схема, демонстрирующая, как &String s указывает на String s1

ПРИМЕЧАНИЕ

Противоположностью процедуре референции, то есть созданию указателя на существующее значение с использованием оператора референции **&**, является процедура разыменования, то есть следование по указателю к существующему значению с использованием оператора разыменования *. Мы рассмотрим некоторые виды использования оператора разыменования в главе 8 и обсудим детали разыменования в главе 15.

Давайте подробнее рассмотрим вызов функции:

```
let s1 = String::from("hello");

let len = calculate_length(&s1);
```

Синтаксис `&s1` позволяет создать ссылку, которая ссылается на значение переменной `s1`, но не владеет им. Поскольку она не является ее владельцем, значение, на которое ссылка указывает, не будет отброшено, когда ссылка выйдет из области видимости.

Схожим образом в сигнатуре функции используется **&**, говоря о том, что тип параметра `s` является ссылкой. Давайте добавим несколько пояснительных аннотаций:

```
fn calculate_length(s: &String) -> usize { // s — это ссылка на экземпляр
                                           // типа String
    s.len()
} // Здесь s выходит из области видимости. Но поскольку она не владеет тем,
  // на что она ссылается, ничего не происходит.
```

Область, в которой переменная `s` является действительной, совпадает с областью любого функционального параметра. Но мы не отбрасываем то, на что указывает ссылка, когда она выходит из области видимости, потому что у нас нет владения. Когда у функций есть ссылки в виде параметров, а не в виде фактических значений, нам не нужно возвращать значения, чтобы передать владение назад, потому что у нас никогда не было владения.

Мы называем использование ссылок в качестве функциональных параметров заимствованием. Как и в реальной жизни, если человек владеет неким предметом, вы можете этот предмет у него позаимствовать. Когда дело сделано, вам придется вернуть его обратно.

Тогда что произойдет, если мы попытаемся изменить то, что заимствуем? Попробуйте использовать код из листинга 4.6. Спойлер: он не работает!

Листинг 4.6. Попытка модифицировать заимствованное значение

src/main.rs

```
fn main() {
    let s = String::from("hello");

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

Вот ошибка[1]:

```
error[E0596]: cannot borrow immutable borrowed content `*some_string` as mutable
 --> error.rs:8:5
  |
7 | fn change(some_string: &String) {
  |                        ------- use `&mut String` here to make mutable
8 |     some_string.push_str(", world");
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ cannot borrow as mutable
```

Ссылки, точно так же, как и переменные, по умолчанию неизменяемы. Мы не можем модифицировать то, на что у нас есть ссылка.

Изменяемые ссылки

Мы можем исправить ошибку в коде из листинга 4.6 с помощью небольшой поправки:

[1] `ошибка[E0596]: не получается заимствовать неизменяемое заимствованное содержимое `*some_string` как изменяемое`

src/main.rs

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

Сначала нам пришлось внести изменение в `s`, добавив ключевое слово `mut`. Затем мы создали изменяемую ссылку с помощью `&mut s` и приняли ее с помощью `some_string: &mut String`.

Но изменяемые ссылки имеют одно существенное ограничение: у вас может быть только одна изменяемая ссылка на отдельный фрагмент данных в отдельной области видимости. Приведенный ниже код не сработает:

src/main.rs

```
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;

println!("{}, {}", r1, r2);
```

Вот ошибка[1]:

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
 --> src/main.rs:5:14
  |
4 |     let r1 = &mut s;
  |              ------ first mutable borrow occurs here
5 |     let r2 = &mut s;
  |              ^^^^^^ second mutable borrow occurs here
6 |
7 |     println!("{}, {}", r1, r2);
  |                         -- first borrow later used here
```

Указанное ограничение допускает изменение, но это можно строго контролировать. С этим с трудом справляются новые растиане, потому что большинство языков позволяют осуществлять изменение, когда вам захочется.

Преимущество этого ограничения в том, что Rust предотвращает гонку данных во время компиляции. Гонка данных похожа на гоночную ситуацию и случается при следующих обстоятельствах:

[1] `ошибка[E0499]: не получается заимствовать переменную `s` как изменяемую более чем за раз`

- Два или более указателей одновременно обращаются к одним и тем же данным.
- По крайней мере один из указателей используется для записи в данные.
- Не используется никакого механизма синхронизации доступа к данным.

Гонки данных вызывают неопределенное поведение, их трудно диагностировать и устранять, когда вы пытаетесь отследить их во время выполнения. Язык Rust предотвращает эту проблему, потому что код с гонками данных он даже не будет компилировать!

Как всегда, мы используем фигурные скобки для создания новой области видимости, допускающей использование более одной изменяемой ссылки (эти ссылки не являются одновременными):

```
let mut s = String::from("hello");

{
    let r1 = &mut s;
} // r1 здесь выходит из области видимости, поэтому мы можем без проблем сделать
  // новую ссылку.

let r2 = &mut s;
```

Схожее правило существует для комбинирования изменяемых и неизменяемых ссылок. Этот код приводит к ошибке:

```
let mut s = String::from("hello");

let r1 = &s; // нет проблем
let r2 = &s; // нет проблем
let r3 = &mut s; // БОЛЬШАЯ ПРОБЛЕМА

println!("{}, {}, and {}", r1, r2, r3);
```

Вот ошибка[1]:

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
immutable
 --> src/main.rs:6:14
  |
4 |     let r1 = &s; // нет проблем
  |              -- immutable borrow occurs here
5 |     let r2 = &s; // нет проблем
6 |     let r3 = &mut s; // БОЛЬШАЯ ПРОБЛЕМА
  |              ^^^^^^ mutable borrow occurs here
7 |
8 |     println!("{}, {}, and {}", r1, r2, r3);
  |                                 -- immutable borrow later used here
```

[1] ошибка[E0502]: не получается позаимствовать переменную `s` как изменяемую, потому что она также заимствуется как неизменяемая

Вот те на! Мы также не можем иметь изменяемую ссылку, пока у нас есть неизменяемая. Пользователи неизменяемой ссылки не ожидают, что значения внезапно изменятся! Тем не менее, если имеется более одной неизменяемой ссылки, то все в порядке, потому что если кто-то просто читает данные, у него нет возможности повлиять на чтение данных кем-либо другим.

Даже если эти ошибки иногда расстраивают, помните, что компилятор Rust указывает на потенциальный дефект заблаговременно (во время компиляции, а не во время выполнения) и сообщает, где именно находится проблема. Благодаря этому вам не приходится отслеживать причину, почему данные не такие, как вы думали.

Висячие ссылки

В языках с указателями легко можно по ошибке создать висячий указатель, то есть указатель, ссылающийся на место в памяти, которое, возможно, было отдано кому-то другому, за счет освобождения некоторой памяти с сохранением указателя на нее. В Rust, напротив, компилятор гарантирует, что ссылки никогда не будут висячими. Если у вас есть ссылка на данные, то компилятор обеспечит, чтобы данные не вышли из области видимости до того, как это сделает ссылка на данные.

Давайте попробуем создать висячую ссылку, которую Rust предотвратит с ошибкой времени компиляции:

src/main.rs

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");

    &s
}
```

Вот ошибка[1]:

```
error[E0106]: missing lifetime specifier
 --> main.rs:5:16
  |
5 | fn dangle() -> &String {
  |                ^expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but there is
    no value for it to be borrowed from
  = help: consider giving it a 'static lifetime
```

[1] ошибка[E0106]: пропущен спецификатор жизненного цикла

Это сообщение об ошибке относится к языковому средству, которое мы еще не рассмотрели, — жизненным циклам. Мы подробно обсудим жизненный цикл в главе 10. Но если вы пропустите часть, посвященную жизненным циклам, то указанное сообщение действительно содержит ключ к тому, почему этот код является проблемным[1]:

```
this function's return type contains a borrowed value, but there is no value for
it to be borrowed from.
```

Давайте посмотрим, что именно происходит на каждом этапе кода функции `dangle`:

src/main.rs

```
fn dangle() -> &String { // dangle возвращает ссылку на String

    let s = String::from("hello"); // s — это новый экземпляр типа String

    &s // мы возвращаем ссылку на String, s
} // Здесь s выходит из области видимости и отбрасывается. Ее память уходит.
  // Поберегись!
```

Так как переменная `s` создана внутри функции `dangle`, когда код функции `dangle` завершен, место для указанной переменной освобождается. Но мы попытались вернуть ссылку на нее. Это означает, что ссылка будет указывать на недействительный экземпляр типа `String`. Это никуда не годится! Rust не позволит нам это сделать.

Решение заключается в том, чтобы вернуть экземпляр типа `String` напрямую:

src/main.rs

```
fn no_dangle() -> String {
    let s = String::from("hello");

    s
}
```

Этот код работает без проблем. Владение перемещено наружу, и ничто не высвобождается.

Правила ссылок

Давайте повторим пройденный материал про ссылки:

- В любой момент времени у вас может быть один из двух вариантов, но не оба: одна изменяемая ссылка либо любое число неизменяемых ссылок.
- Ссылки всегда должны быть действительными.

Далее мы рассмотрим совсем другой вид ссылок — срезы.

---

[1] Тип возвращения из этой функции содержит заимствованное значение, но значение, которое она могла бы заимствовать, отсутствует.

Срезовый тип

Еще один тип данных, в котором не предусмотрено владение, — это срез (или отрезок). Срезы позволяют ссылаться не на всю коллекцию, а на сплошную последовательность элементов в коллекции.

Вот небольшая задача по программированию: нужно написать функцию, которая берет строку и возвращает первое слово, которое она находит в этой строке. Если функция не находит пробел в строке, то вся строка является одним словом, поэтому ее нужно возвратить целиком.

Давайте подумаем о сигнатуре этой функции:

```
fn first_word(s: &String) -> ?
```

Указанная функция `first_word` имеет в качестве параметра `&String`. Владение не требуется, так что все в порядке. Но что мы должны вернуть? На самом деле мы не можем говорить о части строки. Однако мы могли бы вернуть индекс конца слова. Давайте попробуем это сделать, как показано в листинге 4.7.

Листинг 4.7. Функция first_word, которая возвращает значение байтового индекса в параметр типа String

src/main.rs

```
fn first_word(s: &String) -> usize {
 ❶ let bytes = s.as_bytes();

    for (i, &item) ❷ in bytes.iter()❸.enumerate() {
     ❹ if item == b' ' {
            return i;
        }
    }

 ❺ s.len()
}
```

Поскольку нужно проверить экземпляр типа `String` поэлементно и посмотреть, что текущее значение является пробелом, мы выполним конвертирование нашего экземпляра в массив байт с помощью метода `as_bytes` ❶. Далее мы создадим итератор по массиву байт, используя метод `iter` ❸.

Подробнее об итераторах мы поговорим в главе 13. Пока же знайте, что `iter` — это метод, который возвращает каждый элемент в коллекции, а метод `enumerate` обертывает результат метода `iter` и возвращает каждый элемент как часть кортежа. Первый элемент кортежа, возвращаемого из `enumerate`, является индексом, а второй — ссылкой на элемент. Это немного удобнее, чем вычислять индекс самим.

Поскольку метод `enumerate` возвращает кортеж, мы можем использовать паттерны для деструктурирования этого кортежа, как и везде в Rust. Поэтому в цикле `for` мы указываем паттерн, который имеет `i` для индекса в кортеже и `&item` для

отдельного байта в кортеже ❷. Поскольку мы получаем ссылку на элемент из `.iter().enumerate()`, мы используем в паттерне оператор `&`.

Внутри цикла `for` мы ищем байт, который представляет пробел, используя синтаксис байтового литерала ❹. Если мы находим пробел, то возвращаем его позицию. В противном случае мы возвращаем длину строки с помощью `s.len()` ❺.

Теперь у нас есть способ выяснить индекс конца первого слова в строке, но есть одна проблема. Мы возвращаем тип `usize` отдельно, но это число имеет значение только в контексте `&String`. Другими словами, поскольку это значение является отдельным от типа `String`, нет никакой гарантии, что оно будет по-прежнему действительным в будущем. Рассмотрим программу из листинга 4.8, которая использует функцию `first_word` из листинга 4.7.

Листинг 4.8. Сохранение результата, полученного после вызова функции first_word, и изменение содержимого экземпляра типа String

src/main.rs

```
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s); // переменная word получит значение 5

    s.clear(); // это опустошает экземпляр типа String, делая его равным ""

    // здесь word по-прежнему имеет значение 5, но больше нет строки,
    // с которой мы могли бы осмысленно использовать значение 5.
    // переменная word теперь полностью недействительна!
}
```

Эта программа компилируется без каких-либо ошибок и также будет компилироваться, если мы используем переменную `word` после вызова `s.clear()`. Поскольку переменная `word` вообще не связана с состоянием переменной `s`, переменная `word` по-прежнему содержит значение `5`. Мы могли бы использовать значение `5` с переменной `s`, чтобы попытаться извлечь первое слово, но это было бы ошибкой, потому что с тех пор, как мы сохранили `5` в `word`, содержимое `s` изменилось.

Беспокойство о том, что индекс в `word` выходит из синхронизации с данными в `s`, утомительно, и могут быть ошибки! Управление этими индексами становится еще более деликатным, если мы пишем функцию `second_word`. Ее сигнатура должна выглядеть вот так:

```
fn second_word(s: &String) -> (usize, usize) {
```

Теперь мы отслеживаем начальный и конечный индексы, и у нас еще больше значений, которые были рассчитаны из данных в отдельно взятом состоянии, но совсем не привязаны к этому состоянию Теперь у нас три свободные несвязанные переменные, которые нужно держать в синхронизированном состоянии.

К счастью, в Rust есть решение этой проблемы — строковые срезы.

Строковые срезы

Строковый срез — это ссылка на часть значения типа `String`. Он выглядит следующим образом:

```
let s = String::from("hello world");

let hello = &s[0..5];
❶ let world = &s[6..11];
```

Он похож на взятие ссылки на все значение типа `String`, но с дополнительным фрагментом `[0..5]`. Вместо ссылки на все значение типа `String` срез является ссылкой на часть значения типа `String`.

Мы можем создавать срезы, используя интервал внутри скобок, указав `[starting_index..ending_index]`, где `starting_index` — это первая позиция в срезе, а `ending_index` — на одну позицию больше, чем последняя позиция в срезе. Внутренне срезовая структура данных хранит начальную позицию и длину среза, которая соответствует `ending_index` минус `starting_index`. Таким образом, в ❶ переменная `world` будет срезом, который содержит указатель на 7-й байт переменной `s` со значением длины, равным `5`.

Рисунок 4.6 показывает это на схеме.

Рис. 4.6. Срез значения типа String, ссылающийся на часть значения

С помощью интервального синтаксиса `..` языка Rust, если вы хотите начать с первого индекса (нуля), то вы можете отбросить значение перед двумя точками. Другими словами, варианты ниже равны:

```
let s = String::from("hello");

let slice = &s[0..2];
let slice = &s[..2];
```

Точно так же, если ваш срез содержит последний байт значения типа `String`, то вы можете удалить конечное число. Это означает, что следующие варианты равны:

```
let s = String::from("hello");

let len = s.len();

let slice = &s[3..len];
let slice = &s[3..];
```

Вы также можете отбросить оба значения, взяв срез всего значения типа `String` целиком. Поэтому следующие варианты равны:

```
let s = String::from("hello");

let len = s.len();

let slice = &s[0..len];
let slice = &s[..];
```

ПРИМЕЧАНИЕ

Индексы интервала строкового среза должны располагаться в допустимых границах символов UTF-8. Если вы попытаетесь создать строковый срез в середине многобайтового символа, то программа завершится с ошибкой. Для введения строковых срезов в этом разделе мы исходим из использования только символов ASCII. Более подробное описание работы с UTF-8 приведено в разделе «Хранение текста в кодировке UTF-8 с помощью строк» (с. 175).

Имея в виду всю эту информацию, давайте перепишем функцию `first_word` так, чтобы вернуть срез. Тип, обозначающий «строковый срез», записывается как `&str`:

src/main.rs

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

Мы получаем индекс конца слова таким же образом, как и в листинге 4.7, путем отыскания первого вхождения пробела. Когда мы отыскиваем пробел, мы возвращаем строковый срез, используя начало строки и индекс пробела в качестве начального и конечного индексов.

Теперь, вызывая функцию `first_word`, мы получаем обратно единственное значение, которое привязано к опорным данным. Значение состоит из ссылки на начальную точку среза и числа элементов в срезе.

Возвращение среза также будет работать для функции `second_word`:

```
fn second_word(s: &String) -> &str {
```

Теперь у нас есть простой API, который гораздо сложнее испортить, потому что компилятор будет обеспечивать, чтобы ссылки на значение типа `String` оставались действительными. Помните дефект в программе из листинга 4.8, когда мы получили индекс на конец первого слова, но затем очистили строку, в результате чего индекс стал недействительным? Этот код был логически неправильным, но не показывал никаких ошибок. Проблемы проявились бы позже, если бы мы продолжили использовать первый индекс слова с очищенной строкой. Срезы делают этот дефект невозможным и гораздо раньше сообщают о том, что у нас есть проблема с кодом. Использование срезовой версии функции `first_word` вызовет ошибку компиляции:

src/main.rs

```
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s);

    s.clear(); // error!

    println!("первое слово равно {}", word);
}
```

Вот ошибка компилятора[1]:

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
immutable
  --> src/main.rs:18:5
   |
16 |     let word = first_word(&s);
   |                           -- immutable borrow occurs here
17 |
18 |     s.clear(); // error!
   |     ^^^^^^^^^ mutable borrow occurs here
19 |
20 |     println!("the first word is: {}", word);
   |                                       ---- immutable borrow later used here
```

Вспомните правила заимствования, в которых говорится, что если у нас есть неизменяемая ссылка на что-то, то мы не можем брать еще и изменяемую ссылку. Поскольку методу `clear` нужно обрезать значение типа `String`, он пытается взять

[1] ошибка[E0502]: не получается позаимствовать переменную `s` как изменяемую, потому что она также заимствуется как неизменяемая

изменяемую ссылку, а это не получается. Язык Rust не только упростил использование API, но и устранил целый класс ошибок во время компиляции!

Строковые литералы — это срезы

Напомним, что мы говорили о строковых литералах, хранящихся внутри двоичного файла. Теперь, когда мы знаем о срезах, мы можем правильно понять строковые литералы:

```
let s = "Hello, world!";
```

Тип переменной `s` здесь равен `&str`. Это срез, указывающий на конкретную точку двоичного файла. Именно поэтому строковые литералы являются неизменяемыми, `&str` — это неизменяемая ссылка.

Строковые срезы в качестве параметров

Знание о том, что можно брать срезы литералов и значений типа `String`, приводит к еще одному улучшению в функции `first_word`, к изменениям в его сигнатуре:

```
fn first_word(s: &String) -> &str {
```

Более опытный растианин написал бы сигнатуру, показанную в листинге 4.9, поскольку она позволяет использовать одну и ту же функцию для значений `String` и `&str`.

Листинг 4.9. Улучшение функции `first_word` с помощью строкового среза для типа параметра `s`

```
fn first_word(s: &str) -> &str {
```

Если у нас есть строковый срез, то можно передать его напрямую. Если у нас есть значение типа `String`, то мы можем передать срез всего значения типа `String`. Определение функции для взятия строкового среза вместо ссылки на значение типа `String` делает API более общим и полезным без потери функциональности:

src/main.rs

```
fn main() {
    let my_string = String::from("hello world");

    // first_word работает на срезах экземпляров типа `String`
    let word = first_word(&my_string[..]);

    let my_string_literal = "hello world";

    // first_word работает на срезах строковых литералов
    let word = first_word(&my_string_literal[..]);

    // Так как строковые литералы уже *являются* строковыми срезами,
    // это также работает без срезового синтаксиса!
    let word = first_word(my_string_literal);
}
```

Другие срезы

Строковые срезы, как вы можете догадаться, являются специфичными для строк. Но есть и более общий срезовый тип. Возьмем вот этот массив:

```
let a = [1, 2, 3, 4, 5];
```

Мы вполне можем сослаться на часть массива точно так же, как мы ссылаемся на часть строки. Можно сделать это так:

```
let a = [1, 2, 3, 4, 5];

let slice = &a[1..3];
```

Этот срез имеет тип `&[i32]`. Он работает так же, как и строковые срезы, сохраняя ссылку на первый элемент и длину. Вы будете использовать этот срез для всех видов других коллекций. Мы подробно обсудим коллекции, когда будем говорить о векторах в главе 8.

Итоги

Идеи владения, заимствования и срезов обеспечивают безопасность доступа к памяти в программах на Rust во время компиляции. Язык Rust позволяет контролировать использование памяти таким же образом, как и другие языки системного программирования. Однако наличие владельца данных автоматически очищает эти данные, когда владелец выходит из области видимости, подразумевая, что не нужно писать и отлаживать лишний код с целью получения этого контроля.

Владение влияет на то, как работают другие части Rust, поэтому мы будем говорить об этих идеях в дальнейшем на протяжении всей книги. Давайте перейдем к главе 5 и посмотрим на группирование фрагментов данных вместе в одной структуре `struct`.

5 Использование структур для связанных данных

Структура, или `struct`, — это настраиваемый тип данных, который позволяет вам именовать и упаковывать вместе несколько связанных значений, составляющих смысловую группу. Если вы знакомы с объектно-ориентированным языком, то структура подобна атрибутам данных в объекте. В этой главе мы сравним кортежи со структурами и противопоставим их, продемонстрируем приемы использования структур, а также обсудим вопросы определения методов и связанных функций для детализации поведения, связанного с данными в структуре. Структуры и перечисления (обсуждаемые в главе 6) являются блоками для создания новых типов в домене программы, в полной мере использующих преимущество проверки типов языка Rust во время компиляции.

Определение и инстанцирование структур

Структуры подобны кортежам, которые были рассмотрены в главе 3. Как и кортежи, части структуры могут быть разных типов. В отличие от кортежей, вы будете называть каждый фрагмент данных, чтобы было понятно, о каких значениях идет речь. Благодаря этим именам структуры гибче, чем кортежи: вы не зависите от порядка следования данных в указании или получении значений экземпляра.

Для того чтобы определить структуру, мы вводим ключевое слово `struct` и называем всю структуру. Имя структуры должно выразительно описывать сгруппированные фрагменты данных. Затем, внутри фигурных скобок, мы определяем имена и типы элементов данных, которые называются «поля». Например, листинг 5.1 показывает структуру, хранящую информацию об учетной записи пользователя.

Листинг 5.1. Определение структуры User

```
struct User {
    username: String,
    email: String,
```

```
    sign_in_count: u64,
    active: bool,
}
```

Для того чтобы использовать структуру после ее определения, мы ее инстанцируем, то есть создаем экземпляр этой структуры, указывая конкретные значения для каждого из полей. Мы создаем экземпляр путем задания имени структуры с последующим добавлением фигурных скобок, содержащих пары ключ-значение, где ключи — это имена полей, а значения — данные, которые будут храниться в этих полях. Нам не нужно указывать поля в том же порядке, в котором мы их объявили в структуре. Другими словами, определение структуры выглядит как общая заготовка для типа, а экземпляры заполняют эту заготовку конкретными данными, создавая значения типа. Например, можно объявить конкретного пользователя, как показано в листинге 5.2.

Листинг 5.2. Создание экземпляра структуры User

```
let user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};
```

Для того чтобы получить конкретное значение из структуры, мы используем точечную нотацию. Если бы нам нужен был только адрес электронной почты этого пользователя, мы бы использовали `user1.email` везде, где следовало бы использовать это значение. Если экземпляр изменяемый, то можно изменить значение с помощью точечной нотации, передав значение в определенное поле. Листинг 5.3 показывает, как изменять значение в поле `email` изменяемого экземпляра структуры `User`.

Листинг 5.3. Изменение значения в поле email экземпляра структуры User

```
let mut user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};

user1.email = String::from("anotheremail@example.com");
```

Обратите внимание, что весь экземпляр должен быть изменяемым. Rust не позволяет помечать только определенные поля как изменяемые. Как и в любом выражении, мы конструируем новый экземпляр структуры как последнее выражение в теле функции, неявно возвращая этот новый экземпляр.

Листинг 5.4 показывает функцию `build_user`, которая возвращает экземпляр структуры `User` с заданной электронной почтой и именем пользователя. Поле `active` получает значение `true`, а `sign_in_count` — значение `1`.

Листинг 5.4. Функция build_user, которая принимает электронную почту и имя пользователя и возвращает экземпляр структуры User

```
fn build_user(email: String, username: String) -> User {
    User {
        email: email,
        username: username,
        active: true,
        sign_in_count: 1,
    }
}
```

Имеет смысл называть параметры функции так же, как и поля структуры, но повторять имена полей `email` и `username` и переменные немного утомительно. Если бы в структуре было больше полей, то необходимость повторять каждое имя раздражала бы еще больше. К счастью, есть удобное краткое написание!

Использование краткой инициализации полей: когда у переменных и полей одинаковые имена

Поскольку имена параметров и имена полей структуры в листинге 5.4 совпадают, можно использовать синтаксис краткой инициализации полей для написания функции `build_user` по-новому, чтобы она вела себя точно так же, но при этом не имела повторений `email` и `username`, как показано в листинге 5.5.

Листинг 5.5. Функция build_user, которая использует краткую инициализацию полей, потому что параметры email и username совпадают с именами полей структуры

```
fn build_user(email: String, username: String) -> User {
    User {
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
}
```

Мы создаем новый экземпляр структуры `User`, у которого есть поле с именем `email`. Мы хотим установить значение в поле `email` равным значению параметра `email` функции `build_user`. Поскольку у поля `email` и параметра `email` одинаковые имена, то вместо `email: email` нам нужно написать только `email`.

Создание экземпляров из других экземпляров с помощью синтаксиса обновления структуры

Часто бывает полезным создать новый экземпляр структуры, который использует большую часть значений старого экземпляра, но изменяет некоторые из них. Это делается с помощью синтаксиса обновления структуры.

Сначала листинг 5.6 показывает, как создается новый экземпляр структуры `User` в переменной `user2` без синтаксиса обновления. Мы устанавливаем новые значения полей `email` и `username`, но в остальном используем те же значения из `user1`, которые создали в листинге 5.2.

Листинг 5.6. Создание нового экземпляра структуры User с использованием некоторых значений из user1

```
let user2 = User {
    email: String::from("another@example.com"),
    username: String::from("anotherusername567"),
    active: user1.active,
    sign_in_count: user1.sign_in_count,
};
```

Используя синтаксис обновления структуры, мы можем достичь того же эффекта с меньшим объемом кода, как показано в листинге 5.7. Синтаксис `..` указывает на то, что остальные поля, не заданные явно, должны иметь то же значение, что и поля в конкретном экземпляре.

Листинг 5.7. Применение синтаксиса обновления структуры для установки новых значений полей email и username в экземпляре структуры User, но использование остальных значений из полей экземпляра из переменной user1

```
let user2 = User {
    email: String::from("another@example.com"),
    username: String::from("anotherusername567"),
    ..user1
};
```

Код в листинге 5.7 также создает экземпляр в переменной `user2`, у которого есть другое значение полей `email` и `username`, но те же значения полей `active` и `sign_in_count`, что и в переменной `user1`.

Использование кортежных структур без именованных полей для создания разных типов

Вы также можете определять структуры, похожие на кортежи, которые называются «кортежные структуры». Кортежные структуры имеют дополнительный смысл, который обеспечивает имя структуры, но у них нет имен, связанных с их полями. Вместо этого они имеют типы полей. Кортежные структуры полезны, когда вы хотите дать всему кортежу имя и сделать кортеж типом, отличающимся от других кортежей, а необходимость называть каждое поле, как в обычной структуре, многословна или избыточна.

Для того чтобы определить кортежную структуру, начните с ключевого слова `struct` и имени структуры с последующими типами в кортеже. Вот определения и использование двух кортежных структур с именами `Color` и `Point`:

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

let black = Color(0, 0, 0);
let origin = Point(0, 0, 0);
```

Обратите внимание, что значения `black` и `origin` — это разные типы, поскольку они являются экземплярами разных кортежных структур. Каждая определяемая вами структура имеет собственный тип, даже если у полей внутри структуры одни и те же типы. Например, функция, берущая параметр типа `Color`, не может брать `Point` в качестве аргумента, даже если оба типа состоят из трех значений `i32`. В иных случаях экземпляры кортежной структуры ведут себя как кортежи: вы можете деструктировать их на отдельные части и использовать `.` с последующим индексом для доступа к отдельному значению и так далее.

Unit-подобные структуры без полей

Вы также можете определять структуры, у которых нет никаких полей! Они называются unit-подобными, или пустыми структурами, потому что ведут себя аналогично `()`, типу `unit`, пустому типу. Пустые структуры бывают полезны в ситуациях, когда нужно реализовать типаж в каком-либо типе, но у вас нет данных, которые вы хотите хранить в самом типе. Мы обсудим типажи в главе 10.

ВЛАДЕНИЕ ДАННЫМИ СТРУКТУР

В определении структуры `User` в листинге 5.1 мы используем обладаемый тип `String` вместо типа `&str` строкового среза. Это сознательный выбор, потому что нужно, чтобы экземпляры этой структуры владели всеми ее данными и чтобы эти данные были действительными до тех пор, пока вся структура является действительной.

Структуры могут хранить ссылки на данные, принадлежащие кому-то другому, но для этого требуется использование жизненных циклов — средства языка Rust, которое мы обсудим в главе 10. Жизненный цикл обеспечивает, чтобы данные, на которые структура ссылается, были действительными до тех пор, пока она существует. Допустим, вы пытаетесь сохранить ссылку в структуре без указания жизненных циклов, как в примере ниже. Это не сработает:

src/main.rs

```
struct User {
    username: &str,
    email: &str,
    sign_in_count: u64,
    active: bool,
}

fn main() {
    let user1 = User {
```

```
        email: "someone@example.com",
        username: "someusername123",
        active: true,
        sign_in_count: 1,
    };
}
```

Компилятор пожалуется, что ему нужны спецификаторы жизненных циклов:

```
error[E0106]: missing lifetime specifier
 -->
  |
2 |     username: &str,
  |               ^ expected lifetime parameter

error[E0106]: missing lifetime specifier
 -->
  |
3 |     email: &str,
  |            ^ expected lifetime parameter
```

В главе 10 мы обсудим вопрос устранения этих ошибок, в результате чего вы сможете хранить в структурах ссылки. Сейчас мы устраним подобные ошибки, используя обладаемые типы, такие как `String` вместо ссылок типа `&str`.

Пример программы с использованием структур

Для того чтобы разобраться в том, когда следует использовать структуры, давайте напишем программу, которая вычисляет площадь прямоугольника. Мы начнем с отдельных переменных, а затем проведем рефакторинг, пока вместо них не станем использовать структуры.

Давайте создадим новый двоичный проект с помощью Cargo под названием rectangles («прямоугольники»), который возьмет ширину и высоту прямоугольника, заданного в пикселах, и рассчитает площадь прямоугольника. Листинг 5.8 показывает короткую программу с одним из способов, как это сделать в проекте src/main.rs.

Листинг 5.8. Расчет площади прямоугольника, заданной отдельными переменными ширины width и высоты height

src/main.rs

```
fn main() {
    let width1 = 30;
    let height1 = 50;

    println!(
        "Площадь прямоугольника равна {} квадратным пикселам.",
        area(width1, height1)
```

```
    );
}

fn area(width: u32, height: u32) -> u32 {
    width * height
}
```

Теперь выполните эту программу с помощью команды `cargo run`:

```
Площадь прямоугольника равна 1500 квадратным пикселам.
```

Хотя листинг 5.8 работает и рассчитывает площадь прямоугольника, вызывая функцию `area` для каждой размерности, мы можем сделать ее лучше. Ширина и высота связаны друг с другом, потому что вместе описывают один прямоугольник.

Трудность с этим кодом проявляется в сигнатуре функции `area`:

```
fn area(width: u32, height: u32) -> u32 {
```

Функция `area` должна рассчитывать площадь одного прямоугольника, но у функции, которую мы написали, два параметра. Параметры являются связанными, но в программе это нигде не выражено. Она будет более читаемой и управляемой, если сгруппировать ширину и высоту. Мы уже обсуждали один из способов сделать это в разделе «Кортежный тип» путем использования кортежей.

Рефакторинг с использованием кортежей

Листинг 5.9 показывает еще одну версию программы, теперь с использованием кортежей.

Листинг 5.9. Указание ширины и высоты прямоугольника с помощью кортежа

src/main.rs

```
fn main() {
    let rect1 = (30, 50);

    println!(
        "Площадь прямоугольника равна {} квадратным пикселам.",
      ❶ area(rect1)
    );
}

fn area(dimensions: (u32, u32)) -> u32 {
 ❷  dimensions.0 * dimensions.1
}
```

С одной стороны, эта программа — лучше. Кортежи позволяют добавить структуру, и теперь мы передаем всего один аргумент ❶. Но, с другой стороны, эта версия менее ясная: кортежи не именуют свои элементы, и в результате вычисление становится запутаннее, потому что нужно индексировать части кортежа ❷.

Вовсе не имеет значения, если при расчете площади мы перепутаем ширину и высоту, но если надо нарисовать прямоугольник на экране, то это будет иметь зна-

чение! Мы должны учитывать, что ширина — это кортежный индекс 0, а высота — кортежный индекс 1. Если бы кто-то другой работал над этим кодом, то ему следовало бы это понимать и держать в уме. Легко забыть или перепутать эти значения и вызвать ошибки, потому что мы не передали смысл данных в коде.

Рефакторинг с использованием структур: добавление большего смысла

Мы используем структуры, для того чтобы улучшить смысловое содержание путем маркировки данных. Можно трансформировать используемый кортеж в тип данных с именем для целого и с именами для частей, как показано в листинге 5.10.

Листинг 5.10. Определение структуры `Rectangle`

src/main.rs

```
❶ struct Rectangle {
    ❷ width: u32,
      height: u32,
  }

  fn main() {
    ❸ let rect1 = Rectangle { width: 30, height: 50 };

      println!(
          "Площадь прямоугольника равна {} квадратным пикселам.",
          area(&rect1)
      );
  }

❹ fn area(rectangle: &Rectangle) -> u32 {
    ❺ rectangle.width * rectangle.height
  }
```

Мы определили структуру и назвали ее `Rectangle` ❶. Внутри фигурных скобок поля определены как `width` и `height`, оба имеют тип `u32` ❷. Затем в функции `main` мы создали конкретный экземпляр структуры `Rectangle`, у которого ширина 30 и высота 50 ❸.

Функция `area` теперь определяется с одним параметром, который мы назвали `rectangle`, его тип является неизменяемым заимствованием экземпляра структуры `Rectangle` ❹. Как уже упоминалось в главе 4, мы хотим заимствовать структуру, а не брать ее во владение. Таким образом, функция `main` сохраняет свое владение и продолжает использовать `rect1`, что является причиной того, почему мы используем `&` в сигнатуре функции и где мы вызываем эту функцию.

Функция `area` обращается к полям `width` и `height` экземпляра структуры `Rectangle` ❺. Сигнатура функции `area` теперь соответствует тому, что мы имеем в виду: нужно вычислить площадь прямоугольника, используя поля ширины `width` и высоты `height`. Благодаря этому передается идея о том, что ширина и высота связа-

ны друг с другом, и значениям даются описательные имена вместо использования индексных значений 0 и 1 кортежа. Это точно выигрышный вариант.

Добавление полезной функциональности с использованием типажей с атрибутом derived

Было бы неплохо иметь возможность выводить экземпляр структуры `Rectangle` во время отладки программы и видеть значения всех ее полей. Листинг 5.11 пытается использовать макрокоманду `println!`, как мы уже делали в предыдущих главах. Однако это не сработает.

Листинг 5.11. Попытка вывести экземпляр структуры Rectangle

src/main.rs

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };

    println!("rect1 равен {}", rect1);
}
```

Когда мы выполняем этот код, происходит ошибка со следующим ключевым сообщением[1]:

```
error[E0277]: `Rectangle` doesn't implement `std::fmt::Display`
```

Макрокоманда `println!` может выполнять много видов форматирования. По умолчанию фигурные скобки говорят макрокоманде `println!` использовать форматирование, именуемое `Display:`, предназначенное для непосредственного потребления конечным пользователем. Примитивные типы, которые мы видели до этого, реализуют формат `Display` по умолчанию, потому что есть только один способ показать пользователю единицу или любой другой примитивный тип. Но в работе со структурами принцип форматирования результата макрокомандой `println!` менее ясен ввиду больших возможностей вывода на экран: «Вам нужны запятые или нет? Надо ли печатать фигурные скобки? Показать все поля?» Из-за этой неоднозначности язык Rust не пытается угадывать наши пожелания, а у структур нет предусмотренной реализации формата `Display`.

Если мы продолжим читать сообщения об ошибках, то найдем вот это полезное примечание[2]:

---

[1] `ошибка[E0277]: ‘Rectangle` не реализует `std:: fmt:: Display`

[2] `= справка: типаж `std::fmt::Display` не реализован для `Rectangle`
`= примечание: в форматных строках вместо этого вы можете использовать `{:?}` (или {:#?} для структурированной печати)`

```
= help: the trait `std::fmt::Display` is not implemented for `Rectangle`
= note: in format strings you may be able to use `{:?}` (or {:#?} for
prettyprint) instead
```

Давайте попробуем! Вызов макрокоманды `println!` теперь будет выглядеть как `println!("rect1 равен {:?}", rect1);`. Размещение спецификатора `:?` внутри фигурных скобок говорит макрокоманде `println!` о том, что мы хотим использовать выходной формат под названием `Debug`. Типаж `Debug` дает возможность печатать структуру в полезном для разработчиков виде, благодаря которому видно ее значение во время отладки кода.

Выполните код с этим изменением. Проклятие! Все равно есть ошибка:

```
error[E0277]: `Rectangle` doesn't implement `std::fmt::Debug`
```

И снова компилятор дает нам полезное примечание[1]:

```
= help: the trait `std::fmt::Debug` is not implemented for `Rectangle`
= note: add `#[derive(Debug)]` or manually implement `std::fmt::Debug`
```

Язык Rust включает в себя функциональность печати отладочной информации, но мы должны согласиться с тем, чтобы сделать эту функциональность доступной для структуры. Для этого нужно добавить аннотацию `#[derive(Debug)]` непосредственно перед определением структуры, как показано в листинге 5.12.

Листинг 5.12. Добавление аннотации для генерирования типажа Debug и печати экземпляра структуры Rectangle с использованием отладочного форматирования

src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };

    println!("rect1 равен {:?}", rect1);
}
```

Теперь, выполнив эту программу, мы не получим никаких ошибок и увидим следующий результат:

```
rect1 равен Rectangle { width: 30, height: 50 }
```

Здорово! Этот результат не самый красивый, но он показывает значения всех полей для этого экземпляра, что определенно поможет во время отладки. Когда есть

[1] `= справка: типаж `std::fmt::Debug` не реализован для `Rectangle`
= примечание: добавьте `#[derive(Debug)]` либо реализуйте `std::fmt::Debug` вручную`

более крупные структуры, полезно иметь результат, который легче читать. В этих случаях вместо `{:?}` в строке макрокоманды `println!` мы можем использовать `{:#?}`. Если в данном примере применить стиль `{:#?}`, то результат будет выглядеть следующим образом:

```
rect1 is Rectangle {
    width: 30,
    height: 50
}
```

Язык Rust предоставляет ряд типажей для использования с аннотацией `derive`, которые добавляют полезные свойства в настраиваемые типы. Эти типажи и их свойства перечислены в приложении B в конце книги. Мы рассмотрим технические приемы реализации этих типажей с настраиваемыми свойствами, а также способы создания собственных типажей в главе 10.

Функция площади `area` очень специфична: она вычисляет площадь только прямоугольников. Было полезно теснее связать это свойство со структурой `Rectangle`, потому что оно не будет работать с другим типом. Давайте посмотрим, как продолжить рефакторинг этого кода, превратив функцию `area` в метод `area`, определенный для типа `Rectangle`.

Синтаксис методов

Методы похожи на функции: они объявляются с помощью ключевого слова `fn` и имени, у них могут быть параметры и возвращаемое значение, они содержат код, который выполняется, когда их вызывают из другого места. Однако методы отличаются от функций тем, что они определяются внутри контекста структуры (или перечисления, или типажа, которые мы рассмотрим в главах 6 и 17 соответственно). Их первым параметром всегда является параметр `self`, представляющий экземпляр структуры, для которого метод вызывается.

Определение методов

Давайте изменим функцию `area`, которая в качестве параметра берет экземпляр структуры `Rectangle`, и вместо этого создадим метод `area`, определенный для структуры `Rectangle`, как показано в листинге 5.13.

Листинг 5.13. Определение метода area для структуры Rectangle

src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}
```

```
❶ impl Rectangle {
    ❷ fn area(&self) -> u32 {
          self.width * self.height
      }
  }

  fn main() {
      let rect1 = Rectangle { width: 30, height: 50 };

      println!(
          "Площадь прямоугольника равна {} квадратным пикселам.",
      ❸   rect1.area()
      );
  }
```

Для того чтобы определить функцию внутри контекста структуры `Rectangle`, мы начинаем блок `impl` (реализация) ❶. Затем переносим функцию `area` в фигурные скобки `impl` ❷ и изменяем первый (и в этом случае единственный) параметр на параметр `self` в сигнатуре и в теле. В функции `main`, где мы вызывали функцию `area` и передавали `rect1` в качестве аргумента, мы вместо этого используем синтаксис вызова метода `area` для экземпляра структуры `Rectangle` ❸. Синтаксис вызова метода идет после экземпляра: мы добавляем точку с последующим именем метода, скобками и любыми аргументами.

В сигнатуре для функции `area` мы используем `&self` вместо `rectangle: &Rectangle`, потому что язык Rust знает, что типом параметра `self` является `Rectangle`, поскольку этот метод находится внутри контекста `impl Rectangle`. Обратите внимание, что нам по-прежнему нужно использовать `&` перед параметром `self`, так же, как мы делали в `&Rectangle`. Методы могут брать `self` во владение, заимствовать параметр `self` неизменяемо, как мы это сделали, либо заимствовать параметр `self` изменяемо, как и любой другой параметр.

Мы выбрали `&self` по той же причине, по которой использовали `&Rectangle` в функциональной версии. Мы не хотим владеть, нужно просто читать данные в структуре, а не писать их. Если бы мы хотели изменить экземпляр, в котором вызвали метод, в рамках того, что этот метод делает, то в качестве первого параметра использовали бы `&mut self`. Метод, который берет экземпляр во владение, используя в качестве первого параметра только параметр `self`, встречается редко. Этот технический прием обычно используется, когда метод трансформирует параметр `self` во что-то другое, и вы хотите запретить вызывающей его стороне использовать исходный экземпляр после этой трансформации.

Главное преимущество использования методов вместо функций, помимо применения синтаксиса методов и отсутствия необходимости повторять тип параметра `self` в сигнатуре каждого метода, заключается в организации. Мы поместили все, что можем сделать с экземпляром типа в один блок `impl` вместо того, чтобы заставлять будущих пользователей кода искать возможности структуры `Rectangle` в различных местах библиотеки, которая есть в языке.

ГДЕ ОПЕРАТОР ->?

В языках C и C++ для вызова методов используются два разных оператора: вы пользуетесь оператором `.`, если вызываете метод непосредственно для объекта, и оператором `->`, если вызываете метод для указателя на объект и сначала должны проследовать по указателю к значению с помощью оператора разыменования (говоря технически, дереференцировать указатель). Другими словами, если объект является указателем, то выражение `object->something()` аналогично `(*object).something()`.

В языке Rust нет эквивалента оператору `->`. Вместо него есть языковое средство, которое называется автоматической референцией (создание указателя на существующее значение) и дереференция (следование по указателю к существующему значению)[1]. Вызов методов — это одна из немногих характеристик языка Rust, которая так себя ведет.

Вот как это работает: когда вы вызываете метод с помощью `object.something()`, язык Rust автоматически добавляет `&`, `&mut` или `*`, благодаря чему объект совпадает с сигнатурой метода. Другими словами, варианты ниже одинаковы:

```
p1.distance(&p2);
(&p1).distance(&p2);
```

Первый выглядит гораздо яснее. Автоматическое создание ссылки работает, потому что методы имеют четкого получателя — тип параметра `self`. С учетом получателя и имени метода язык Rust точно знает, что делает метод: читает (`&self`), изменяет (`&mut self`) или потребляет (`self`). Тот факт, что язык Rust делает заимствование неявным для получателей метода, очень важен и делает владение эргономичным на практике.

Методы с большим числом параметров

Давайте позанимаемся применением методов путем реализации второго метода для структуры `Rectangle`. На этот раз мы хотим, чтобы экземпляр структуры `Rectangle` брал еще один экземпляр структуры `Rectangle` и возвращал `true`, если второй прямоугольник `Rectangle` полностью помещается внутрь параметра `self`. В противном случае он должен вернуть `false`. То есть мы хотим написать программу, показанную в листинге 5.14, как только определим метод `can_hold`.

Листинг 5.14. Использование пока еще не написанного метода can_hold

src/main.rs

```
fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };
    let rect2 = Rectangle { width: 10, height: 40 };
    let rect3 = Rectangle { width: 60, height: 45 };
```

[1] Референция (referencing) — это операция косвенного обращения, которая заключается в создании указателя на существующее значение путем получения его адреса в памяти (с помощью оператора референции &). Разыменование (dereferencing) — это операция косвенного обращения, которая заключается в следовании по указателю к существующему значению (с помощью оператора разыменования *).

```
    println!("Может ли rect1 содержать в себе rect2? {}", rect1.can_hold(&rect2));
    println!("Может ли rect1 содержать в себе rect3? {}", rect1.can_hold(&rect3));
}
```

И ожидаемый результат будет выглядеть следующим образом, потому что обе размерности переменной `rect2` меньше, чем размерности переменной `rect1`, но переменная `rect3` шире, чем переменная `rect1`:

```
Может ли rect1 содержать в себе rect2? true
Может ли rect1 содержать в себе rect3? false
```

Мы знаем, что хотим определить метод, поэтому он будет находиться внутри блока `impl Rectangle`. Имя метода `can_hold`, он будет брать неизменяемое заимствование еще одной структуры `Rectangle` в качестве параметра. Мы можем определить тип параметра, посмотрев на код, который вызывает метод: `rect1.can_hold(&rect2)` передает аргумент `&rect2`, который является неизменяемым заимствованием переменной `rect2` экземпляра структуры `Rectangle`. Это имеет смысл, потому что нам нужно только читать `rect2` (а не писать, что означало бы, что нам нужно было изменяемое заимствование). Также нужно, чтобы функция `main` сохранила владение переменной `rect2`, в результате чего мы сможем использовать его снова после вызова метода `can_hold`. Значение, возвращаемое из `can_hold`, будет булевым, и реализация проверит, являются ли ширина и высота параметра `self` соответственно больше ширины и высоты другой структуры `Rectangle`. Давайте добавим новый метод `can_hold` в блок `impl` из листинга 5.13, показанный в листинге 5.15.

Листинг 5.15. Реализация метода can_hold в структуре Rectangle, который берет еще один экземпляр структуры Rectangle в качестве параметра

src/main.rs

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

Когда мы выполним этот код с помощью функции `main` в листинге 5.14, то получим желаемый результат. Методы могут принимать многочисленные параметры, которые мы добавляем в сигнатуру после параметра `self`, и они работают так же, как параметры в функциях.

Связанные функции

Еще одним полезным свойством блоков `impl` является то, что нам разрешено определять функции в блоках `impl`, которые не берут `self` в качестве параметра.

Эти функции называются связанными, поскольку они присоединены к структуре. Они по-прежнему остаются функциями, а не методами, потому что у них нет экземпляра структуры, с которым можно работать. Вы уже использовали связанную функцию `String::from`.

Связанные функции часто используются для конструкторов, которые возвращают новый экземпляр структуры. Например, мы могли бы предоставить связанную функцию, у которой будет один размерный параметр, и она будет использовать его как ширину и как высоту, что облегчит создание квадратной структуры `Rectangle` вместо того, чтобы указывать одно и то же значение дважды:

src/main.rs

```
impl Rectangle {
    fn square(size: u32) -> Rectangle {
        Rectangle { width: size, height: size }
    }
}
```

Для того чтобы вызвать эту связанную функцию, мы используем синтаксис `::` с именем структуры. Инструкция `let sq = Rectangle::square(3);` является примером. Эта функция находится в пространстве имен структуры: синтаксис `::` используется как для связанных функций, так и для пространств имен, создаваемых модулями. Мы обсудим модули в главе 7.

Несколько блоков impl

Каждая структура может иметь несколько блоков `impl`. Например, листинг 5.15 эквивалентен коду, показанному в листинге 5.16, в котором у каждого метода собственный блок `impl`.

Листинг 5.16. Написание листинга 5.15 по-новому с использованием нескольких блоков impl

src/main.rs

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

Нет необходимости разделять эти методы на несколько блоков `impl`, но такой синтаксис является допустимым. В главе 10, рассказывающей про обобщения и типажи, мы покажем пример, где будет полезно использовать несколько блоков `impl`.

Итоги

Структуры позволяют создавать настраиваемые типы, значимые для вашего домена. Используя структуры, вы можете связывать фрагменты данных друг с другом и называть каждый фрагмент, делая код понятным. Методы позволяют указывать свойства, присущие экземплярам структур, а связанные функции дают возможность использовать функциональность пространства имен, характерную только для вашей структуры, не имея при этом экземпляра в наличии.

Но структуры — это не единственный способ создания типов, настраиваемых под нужды пользователя. Давайте добавим еще один инструмент в свой арсенал и обратимся к средству Rust, которое называется «перечисления».

6 Перечисления и сопоставление с паттернами

В этой главе мы обратимся к перечислениям, так называемым enum. Перечисления позволяют определять тип путем перечисления его возможных значений. Сначала мы определим и применим перечисление, чтобы показать, как оно кодирует значение вместе с данными. Далее мы познакомимся с особенно полезным перечислением под названием `Option`, которое выражает идею о том, что значение бывает либо чем-то, либо ничем. Затем мы посмотрим, как сопоставление с паттернами в выражении `match` позволяет легко выполнять разный код для разных значений перечисления. Наконец, мы покажем, что конструкция `if let` — это еще одна удобная и лаконичная идиома, доступная для обработки перечислений в коде.

Перечисления входят в состав средств многих языков программирования, но в каждом языке их свойства различаются. Перечисления языка Rust наиболее похожи на алгебраические типы данных в функциональных языках, таких как F#, OCaml и Haskell.

Определение перечисления

Давайте взглянем на ситуацию, выраженную в коде, и посмотрим, почему перечисления в этом случае являются полезными и более уместными, чем структуры. Допустим, нам нужно работать с IP-адресами. В настоящее время для IP-адресов используются два главных стандарта: версия четыре и версия шесть. Эти возможности для IP-адреса являются единственными, с которыми столкнется программа: мы можем перечислить все возможные значения, отсюда и происходит название перечисления.

Любой IP-адрес может быть адресом либо версии четыре, либо версии шесть, но не обеими версиями одновременно. Это свойство IP-адресов делает структуру данных enum оправданной, поскольку значения перечисления принимают только один из вариантов. Обе версии адресов — четыре и шесть — в своей основе по-прежнему являются IP-адресами, поэтому их следует рассматривать как один и тот же тип, когда код обрабатывает ситуации, применимые к любому виду IP-адреса.

Мы можем выразить эту идею в коде, определив перечисление `IpAddrKind` и перечислив возможные варианты видов IP-адреса, `V4` и `V6`. Они называются вариантами перечисления:

```
enum IpAddrKind {
    V4,
    V6,
}
```

Теперь `IpAddrKind` является настраиваемым типом данных, который можно использовать в любом месте кода.

Значения перечисления

Мы можем создавать экземпляры каждого из двух вариантов перечисления `IpAddrKind` следующим образом:

```
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;
```

Обратите внимание, что варианты перечисления находятся в пространстве имен под его идентификатором, и чтобы их отделить, мы используем двойное двоеточие. Это полезно потому, что теперь оба значения, `IpAddrKind::V4` и `IpAddrKind::V6`, имеют один и тот же тип: `IpAddrKind`. Затем мы можем, к примеру, определить функцию, которая принимает любой вариант `IpAddrKind`:

```
fn route(ip_kind: IpAddrKind) { }
```

Можно вызвать эту функцию с любым вариантом:

```
route(IpAddrKind::V4);
route(IpAddrKind::V6);
```

Использование перечислений имеет еще больше преимуществ. Если вновь обратиться к типу IP-адреса, на данный момент у нас нет способа хранить фактические данные IP-адреса; нам лишь известен его вид. Учитывая, что вы узнали о структурах в главе 5, можно решить эту задачу, как показано в листинге 6.1.

Листинг 6.1. Хранение данных и варианта IpAddrKind IP-адреса с использованием структуры

```
❶ enum IpAddrKind {
      V4,
      V6,
  }

❷ struct IpAddr {
    ❸ kind: IpAddrKind,
    ❹ address: String,
  }

❺ let home = IpAddr {
```

```
        kind: IpAddrKind::V4,
        address: String::from("127.0.0.1"),
    };

❻  let loopback = IpAddr {
        kind: IpAddrKind::V6,
        address: String::from("::1"),
    };
```

Здесь мы определили структуру `IpAddr` ❷, у которой есть два поля: поле вида `kind` ❸, которое имеет тип `IpAddrKind` (перечисление, определенное нами ранее ❶), и поле адреса `address` ❹ типа `String`. У нас есть два экземпляра этой структуры. Первый, `home` ❺, имеет значение `IpAddrKind::V4` как его вид `kind` с соответствующими адресными данными `127.0.0.1`. Второй экземпляр, `loopback` ❻, имеет другой вариант `IpAddrKind` в качестве своего значения `kind`, `V6`, и связанный с ним адрес `::1`. Мы использовали структуру для компоновки значений вида и адреса вместе, поэтому теперь вариант связан с этим значением.

Мы можем представить ту же самую идею в более сжатом виде, используя вместо перечисления внутри структуры только одно перечисление, поместив данные непосредственно в каждый вариант перечисления. Это новое определение перечисления `IpAddr` говорит о том, что оба варианта, `V4` и `V6`, будут иметь связанные значения `String`:

```
enum IpAddr {
    V4(String),
    V6(String),
}

let home = IpAddr::V4(String::from("127.0.0.1"));

let loopback = IpAddr::V6(String::from("::1"));
```

Мы прикрепляем данные к каждому варианту перечисления напрямую, поэтому дополнительная структура не нужна.

Есть еще одно преимущество использования перечисления вместо структуры: каждый вариант может иметь разные типы и объемы связанных данных. У IP-адресов версии четыре всегда будет четыре числовых компонента, каждый из которых будет иметь значения от 0 до 255. Если бы мы хотели хранить адреса `V4` как четыре значения `u8`, но выражать адреса `V6` как одно значение `String`, то не смогли бы сделать это со структурой. Перечисления легко с этим справляются.

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

let home = IpAddr::V4(127, 0, 0, 1);

let loopback = IpAddr::V6(String::from("::1"));
```

Мы показали несколько разных способов определения структур данных для хранения IP-адресов версий четыре и шесть. Однако, как оказалось, пользователи часто желают хранить IP-адреса и определять в коде их вид, поэтому в стандартной библиотеке есть определение, которое можно использовать. Давайте посмотрим, как стандартная библиотека определяет `IpAddr`. В ней есть точно такое же перечисление и варианты, которые мы определили и использовали, но библиотека вставляет адресные данные внутрь вариантов в форме двух разных структур, которые определяются по-разному для каждого варианта:

```
struct Ipv4Addr {
    // --пропуск--
}

struct Ipv6Addr {
    // --пропуск--
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

Этот код иллюстрирует, что внутрь варианта перечисления можно помещать любые данные, например строковые, числовые типы или структуры. Вы даже можете включить еще одно перечисление! Кроме того, типы стандартной библиотеки часто не намного сложнее того, что вы могли бы придумать сами.

Обратите внимание, что, даже если стандартная библиотека содержит определение для `IpAddr`, все равно можно создавать и использовать собственное определение бесконфликтно, потому что мы не ввели определение стандартной библиотеки в область видимости. Мы еще поговорим о введении типов в область видимости в главе 7.

Давайте посмотрим на очередной пример перечисления в листинге 6.2. В варианты этого перечисления встроено большое разнообразие типов.

Листинг 6.2. Перечисление Message, в каждом из вариантов которого хранятся разные объемы и типы значений

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

Это перечисление имеет четыре варианта с разными типами:

- `Quit` вообще не имеет связанных с ним данных.
- `Move` включает в себя анонимную структуру.

- `Write` включает в себя одно значение `String`.
- `ChangeColor` включает в себя три значения `i32`.

Определение перечисления с вариантами, подобными приведенным в листинге 6.2, похоже на определение структур разных видов. Однако различие состоит в том, что перечисление не использует ключевое слово `struct`, и все варианты сгруппированы вместе, в соответствии с типом `Message`. Следующие структуры могли бы содержать те же данные, что и варианты предыдущего перечисления:

```
struct QuitMessage;  // пустая структура
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String);              // кортежная структура
struct ChangeColorMessage(i32, i32, i32); // кортежная структура
```

Но если бы мы использовали разные структуры, у каждой из которых был бы собственный тип, то мы бы не смогли так же легко определить функцию для приема любого из этих типов сообщений, как можно было бы сделать с перечислением `Message`, определенным в листинге 6.2, которое является единым типом.

Между перечислениями и структурами существует еще одно сходство: для перечислений можно определять методы с помощью ключевого слова `impl` точно так же, как для структур. Вот метод с именем `call`, который мы можем определить в перечислении `Message`:

```
impl Message {
    fn call(&self) {
     ❶ // здесь будет определено тело метода
    }
}

❷ let m = Message::Write(String::from("hello"));
m.call();
```

Тело метода будет использовать параметр `self`, чтобы получить значение, для которого мы вызвали этот метод. В этом примере мы создали переменную `m` ❷ со значением `Message::Write(String::from("hello"))`, и это то, как будет выглядеть параметр `self` в теле метода `call` ❶, когда выполняется `m.call()`.

Давайте рассмотрим еще одно полезное перечисление из стандартной библиотеки, которое встречается очень часто, — `Option`.

Перечисление Option и его преимущества по сравнению со значениями null

В предыдущем разделе мы видели, как перечисление `IpAddr` позволяет использовать систему типов языка Rust, чтобы кодировать в программу больше информации, чем просто данные. В этом разделе проводится тематический анализ перечис-

ления `Option`, то есть еще одного перечисления стандартной библиотеки. Тип `Option` используется во многих местах, потому что он кодирует крайне распространенный сценарий, в котором значение может быть чем-то либо не может быть ничем. Выражение этого понятия в терминах системы типов означает, что компилятор может проверить, что вы учли все необходимые случаи. Эта функция предотвращает дефекты, которые чрезвычайно распространены в других языках программирования.

Дизайн языка программирования часто рассматривается с точки зрения того, какие средства вы включаете, но средства, которые вы исключаете, также важны. Язык Rust не имеет языкового оформления для null, которое есть во многих других языках. Null — это значение, которое подразумевает, что нет никакого значения. В языках с null переменные всегда могут находиться в одном из двух состояний: null или не-null.

В своей презентации 2009 года «Ссылки на null: ошибка на миллиард долларов» (*Null References: The Billion Dollar Mistake*) Тони Хоар (Tony Hoare), изобретатель null, сказал следующее:

> *Я называю это своей ошибкой на миллиард долларов. В то время я занимался дизайном первой всесторонней системы типов для ссылок в объектно-ориентированном языке. Цель была в том, чтобы использование ссылок было абсолютно безопасным, с автоматической проверкой, выполняемой компилятором. Но я не смог устоять перед искушением вставить пустую ссылку (null-ссылку) просто потому, что ее было так легко реализовать. Это привело к бесчисленным ошибкам, уязвимостям и системным сбоям, которые за последние сорок лет причинили боли и ущерба, пожалуй, на миллиард долларов.*

Проблема со значениями null заключается в том, что если вы попытаетесь использовать значение null в качестве значения не-null, то возникнет та или иная ошибка. Поскольку это свойство null или не-null является всеобъемлющим, очень легко сделать такую ошибку.

Однако идея, которую пытается выразить null, по-прежнему остается полезной: null — это значение, которое в настоящее время недействительно или по какой-либо причине отсутствует.

Проблема на самом деле не в идее, а в конкретной реализации. В таком качестве язык Rust не имеет null, но у него есть перечисление, которое кодирует идею присутствующего или отсутствующего значения. Этим перечислением является `Option<T>`, и стандартная библиотека определяет его следующим образом:

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

Перечисление `Option<T>` является настолько полезным, что оно даже включено в прелюдию. Вам не нужно вводить его в область видимости. То же самое касается и его вариантов: вы можете использовать `Some` и `None` непосредственно без пре-

фикса `Option::`. Перечисление `Option<T>` по-прежнему является обычным перечислением, а `Some(T)` и `None` — по-прежнему варианты типа `Option<T>`.

Синтаксис `<T>` является средством языка Rust, о котором мы еще не говорили. Это параметр обобщенного типа, мы рассмотрим обобщения подробнее в главе 10. На данный момент вам нужно знать только одно: `<T>` означает, что вариант `Some` перечисления `Option` может содержать один фрагмент данных любого типа. Вот несколько примеров использования значений перечисления `Option` для числовых и строковых типов:

```
let some_number = Some(5);
let some_string = Some("строковый литерал");

let absent_number: Option<i32> = None;
```

Если вместо `Some` мы используем `None`, то нам нужно указать, какой тип `Option<T>` у нас есть, потому что компилятор не может логически вывести тип, который вариант `Some` будет содержать, только из значения `None`.

Когда есть значение `Some` (то есть какое-то значение), мы знаем, что оно присутствует и содержится внутри `Some`. Когда у нас есть значение `None` (то есть нет никакого значения), в некотором смысле это означает то же самое, что и null: у нас нет действительного значения. Тогда почему `Option<T>` лучше, чем null?

Если коротко, так как `Option<T>` и `T` (где `T` может быть любым типом) являются разными типами, компилятор не позволит использовать значение `Option<T>`, как если бы это значение было действительным. Например, этот код не компилируется, потому что он пытается прибавить значение типа `i8` к значению типа `Option<i8>`:

```
let x: i8 = 5;
let y: Option<i8> = Some(5);

let sum = x + y;
```

Если мы выполним этот код, то получим сообщение об ошибке[1]:

```
error[E0277]: the trait bound `i8: std::ops::Add<std::option::Option<i8>>` is
not satisfied
 -->
  |
5 |     let sum = x + y;
  |                 ^ no implementation for `i8 + std::option::Option<i8>`
  |
```

Круто! Это сообщение об ошибке означает, что компилятор не понимает, как сложить типы `i8` и `Option<i8>`, потому что это разные типы. Когда у нас есть значение

[1] `ошибка[E0277]: граница типажа`i8: std::ops::Add<std::option::Option<i8>>` не удовлетворена`

типа `i8`, компилятор обеспечит, чтобы значение всегда было действительным. Мы можем действовать уверенно, не проводя проверок на null перед использованием этого значения. Нам приходится беспокоиться о возможном отсутствии значения, только когда у нас есть `Option<i8>` (или любой другой тип значения, с которым мы работаем), а компилятор сделает так, чтобы мы справились с задачей, прежде чем использовать это значение.

Другими словами, вы должны конвертировать тип `Option<T>` в `T`, прежде чем сможете выполнять операции с `T`. Как правило, это помогает выявлять одну из самых распространенных проблем с null: допущение, что нечто не является null, когда в действительности оно является таковым.

Если вам не нужно беспокоиться о том, как избежать ошибок при работе со значениями, которые не являются null, то вы можете быть уверены в своем коде. Для того чтобы иметь значение, которое может быть null, вы должны сделать тип этого значения типом `Option<T>`. Когда вы используете это значение, вам следует проработать ситуацию, при которой значение равно null. Если значение имеет тип, который не является `Option<T>`, можно с уверенностью допустить, что значение не является null. Решение разработчиков языка Rust ограничить всеобъемлющее свойство null и повысить безопасность кода Rust было сознательным.

Как получить значение типа `T` из некоторого варианта и использовать его, когда есть значение типа `Option<T>`? Перечисление `Option<T>` имеет большое число методов, которые полезны в различных ситуациях. Вы можете обратиться к ним в документации. Знакомство с методами, относящимися к `Option<T>`, будет чрезвычайно полезным в процессе изучения Rust.

В общем, для того чтобы использовать значение перечисления `Option<T>`, нужен код, который будет обрабатывать каждый вариант. Требуется некоторый код, который будет выполняться только в случае, когда у вас есть значение `Some(T)`, и этому коду разрешается использовать внутренний `T`. Нужно, чтобы выполнялся какой-то другой код, если у вас есть значение `None`, а у этого кода нет значения `T`. Выражение `match` является той конструкцией управления потоком, которая делает именно это при использовании с перечислениями: она будет выполнять разный код в зависимости от того, какой вариант перечисления у нее есть, а этот код будет использовать данные внутри совпадающего значения.

Выражение match как оператор управления потоком

В языке Rust есть чрезвычайно мощное выражение, выступающее как оператор управления потоком, под названием `match`. Оно позволяет сравнивать значение с серией паттернов, а затем исполнять код, основываясь на том, какой паттерн совпадает. Паттерны могут состоять из буквенных значений, имен переменных, подстановочных знаков и других элементов. В главе 18 рассказывается о разных

видах паттернов и описывается их работа. В основе выражения `match` лежит мощь паттернов, а также то, что компилятор проверяет, что все возможные случаи учтены.

Подумайте о выражении `match` как о машине для сортировки монет: монеты скользят по дорожке, вдоль которой расположены отверстия разного размера. Каждая монета падает в первое попавшееся отверстие, в которое помещается. Значения в выражении `match` проходят через каждый паттерн точно таким же образом, и в первом паттерне, в который значение «помещается», оно попадает в связанный блок кода, который будет использоваться во время исполнения.

Поскольку мы только что упомянули монеты, давайте используем их в качестве примера применения выражения `match`! Мы напишем функцию, которая берет неизвестную монету Соединенных Штатов и, подобно счетной машине, выясняет номинал этой монеты и возвращает его в центах, как показано в листинге 6.3.

Листинг 6.3. Перечисление и выражение `match`, которое в качестве паттернов содержит варианты перечисления

```
❶ enum Coin {
      Penny,
      Nickel,
      Dime,
      Quarter,
  }

  fn value_in_cents(coin: Coin) -> u8 {
   ❷ match coin {
       ❸ Coin::Penny => 1,
         Coin::Nickel => 5,
         Coin::Dime => 10,
         Coin::Quarter => 25,
      }
  }
```

Давайте разберем выражение `match` в функции `value_in_cents`. Сначала мы пишем ключевое слово `match`, а затем — выражение, которым в данном случае является значение переменной `coin` ❷. Это выражение очень похоже на то, которое используется с `if`, но есть большая разница: с `if` выражение должно возвращать логическое значение, а здесь оно может быть любого типа. Типом переменной `coin` в данном примере является перечисление `Coin`, которое мы определили в ❶.

Далее идут рукава выражения `match`. Рукав состоит из двух частей: паттерна и кода. Здесь первый рукав имеет паттерн, являющийся значением `Coin::Penny` с последующим оператором `=>`, который отделяет паттерн и исполняемый код ❸. Кодом в данном случае является просто значение `1`. Каждый рукав отделен от следующего запятой.

Когда выражение `match` исполняется, оно по порядку сравнивает полученное значение с паттерном каждого рукава. Если паттерн совпадает со значением, то

код, связанный с этим паттерном, исполняется. Если этот паттерн не совпадает со значением, то исполнение продолжается и переходит к следующему рукаву, как в машине для сортировки монет. У нас может быть столько рукавов, сколько нам нужно: в листинге 6.3 у выражения `match` четыре рукава.

Код, связанный с каждым рукавом, является выражением, а результирующее значение выражения в совпадающем рукаве является значением, возвращаемым для всего выражения `match`.

В типичной ситуации фигурные скобки не используются, если код рукава выражения `match` короткий, как показано в листинге 6.3, где каждый рукав просто возвращает значение. Если вы хотите выполнить несколько строк кода в рукаве выражения `match`, то используете фигурные скобки. Например, следующий код будет выводить «Монетка на счастье!» всякий раз, когда метод вызывается с `Coin::Penny`, но все равно возвращать последнее значение блока, `1`:

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => {
            println!("Монетка на счастье!");
            1
        },
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Паттерны, которые привязываются к значениям

Еще одним полезным свойством рукавов выражения `match` является то, что они могут привязываться к частям значений, которые совпадают с паттерном. Именно так мы можем извлекать значения из вариантов перечисления.

В качестве примера давайте изменим один из наших вариантов перечисления так, чтобы он содержал данные внутри себя. С 1999 по 2008 год США чеканили четвертаки с разными рисунками с одной стороны для каждого из 50 штатов. Ни одна другая монета не имеет рисунка, соответствующего своему штату, поэтому только у четвертаков есть эта дополнительная ценность. Мы можем добавить эту информацию в перечисление, изменив вариант `Quarter` и включив в него значение `UsState`, хранящееся внутри него. Это мы сделали в листинге 6.4.

Листинг 6.4. Перечисление Coin, в котором вариант Quarter также имеет значение UsState

```
#[derive(Debug)] // чтобы проверить штат сразу
enum UsState {
    Alabama,
    Alaska,
    // --пропуск--
```

```
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

Давайте представим, что у нас есть друг, который пытается собрать четвертаки всех 50 штатов. Пока мы сортируем разменную монету по типам, мы также будем называть штат, связанный с каждым четвертаком. Таким образом, в случае, если у друга нет такой монеты, он сможет добавить ее в свою коллекцию.

В выражении `match` для этого кода мы добавляем переменную `state` в паттерн, который сопоставляется со значениями варианта `Coin::Quarter`. Если `Coin::Quarter` совпадет, переменная `state` будет привязана к значению штата этого четвертака. Затем можно использовать `state` в коде для этого рукава:

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("Четвертак из штата {:?}!", state);
            25
        },
    }
}
```

Если бы нужно было вызвать `value_in_cents(Coin::Quarter(UsState::Alaska))`, то переменная `coin` была бы `Coin::Quarter(UsState::Alaska)`. Когда мы сравниваем это значение с каждым рукавом выражения `match`, ни один из них не совпадет до тех пор, пока мы не достигнем `Coin::Quarter(state)`. В этом месте привязкой для `state` будет значение `UsState::Alaska`. Затем можно использовать эту привязку в выражении `println!`, получив таким образом внутреннее значение штата из варианта перечисления `Coin` для `Quarter`.

Сопоставление с Option<T>

В предыдущем разделе мы хотели получить внутреннее значение `T` из случая `Some` при использовании перечисления `Option<T>`. Можно обрабатывать `Option<T>`, используя выражение `match`, так же как это было с перечислением `Coin`! Вместо сравнения монет мы будем сравнивать варианты перечисления `Option<T>`, но характер работы выражения `match` останется тем же.

Допустим, мы хотим написать функцию, которая берет перечисление `Option<i32>` и, если внутри есть значение, то эта функция прибавляет к нему 1. Если внутри

нет значения, то она должна возвращать значение `None` и не выполнять какие-либо операции.

Эта функция очень проста в написании благодаря `match` и будет выглядеть как в листинге 6.5.

Листинг 6.5. Функция, использующая выражение match для перечисления Option<i32>

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
      ❶ None => None,
      ❷ Some(i) => Some(i + 1),
    }
}

let five = Some(5);
let six = plus_one(five); ❸
let none = plus_one(None); ❹
```

Давайте рассмотрим первое исполнение функции `plus_one` подробнее. Когда мы вызываем `plus_one(five)` ❸, переменная `x` в теле `plus_one` имеет значение `Some(5)`. Затем мы сравниваем его с каждым рукавом выражения `match`.

Значение `Some(5)` не совпадает с паттерном `None` ❶, поэтому мы переходим к следующему рукаву. Совпадает ли `Some(5)` с `Some(i)` ❷? Разумеется, да! У нас такой же вариант. Переменная `i` привязывается к значению, содержащемуся в `Some`, поэтому `i` принимает значение `5`. Затем выполняется код в рукаве выражения `match`, мы прибавляем 1 к значению `i` и создаем новое значение `Some` с суммарным числом 6 внутри.

Теперь рассмотрим второй вызов `plus_one` в листинге 6.5, где `x` равно `None` ❹. Мы входим в `match` и сравниваем с первым рукавом ❶.

И оно совпадает! Там нет никакого прибавляемого значения, поэтому программа останавливается и возвращает значение `None` справа от `=>`. Поскольку первый рукав совпадает, никакие другие рукава не сравниваются.

Комбинирование выражения `match` и перечислений оказывается полезным во многих ситуациях. Вы будете встречать указанный паттерн в коде Rust очень часто: сопоставление с вариантами перечисления, привязка переменной к данным внутри, а затем исполнение кода на их основе. Поначалу все это немного сложновато, но как только вы привыкнете к этому паттерну, вы пожалеете, что его нет во всех языках. Он неизменно является излюбленным среди пользователей языка Rust.

Совпадения являются исчерпывающими

Нам нужно обсудить еще один аспект выражения `match`. Рассмотрим такую версию функции `plus_one`, в которой есть ошибка, и она не компилируется:

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        Some(i) => Some(i + 1),
    }
}
```

Мы не учитываем случай `None`, поэтому указанный код вызовет ошибку. К счастью, Rust умеет выявлять эту ошибку. Если мы попытаемся скомпилировать код, то получим такую ошибку[1]:

```
error[E0004]: non-exhaustive patterns: `None` not covered
 -->
  |
6 |        match x {
  |              ^ pattern `None` not covered
```

Rust знает, что мы не учли все возможные случаи, и даже знает, какой мы забыли паттерн! Совпадения в Rust являются исчерпывающими: для того чтобы код был допустимым, мы должны исчерпать все до последней возможности. Особенно в случае с `Option<T>`, когда Rust не дает нам забыть про обработку `None`. Язык защищает от ошибочного предположения о наличии значения, когда вместо него может быть null, то есть от упомянутой ранее ошибки на миллиард долларов.

Заполнитель _

В Rust также имеется паттерн, который можно использовать, когда мы не хотим перечислять все возможные значения. Например, `u8` имеет допустимые значения в интервале от 0 до 255. Если нас интересуют только значения 1, 3, 5 и 7, то нам не придется перечислять 0, 2, 4, 6, 8, 9 вплоть до 255. К счастью, мы не обязаны это делать: вместо этого мы можем использовать специальный паттерн `_`:

```
let some_u8_value = 0u8;
match some_u8_value {
    1 => println!("один"),
    3 => println!("три"),
    5 => println!("пять"),
    7 => println!("семь"),
    _ => (),
}
```

Паттерн `_` будет соответствовать любому значению. Если поставить его после других рукавов, `_` будет совпадать со всеми возможными случаями, которые не указаны до него. `()` является всего лишь пустым значением, поэтому в случае `_` ничего не произойдет. В результате можно сказать, что ничего не нужно делать со всеми возможными значениями, которые мы не перечисляем до заполнителя `_`.

Однако выражение `match` бывает несколько избыточным в ситуации, когда нас интересует только один из случаев. Для этого в Rust есть конструкция `if let`.

---

[1] `ошибка[E0004]: неисчерпывающие паттерны: не учтен случай `None``

Сжатое управление потоком с помощью if let

Синтаксис `if let` позволяет кратко совместить `if` и `let` для обработки значений, которые совпадают с одним паттерном, игнорируя остальные. Рассмотрим программу в листинге 6.6, которая имеет совпадение в значении `Option<u8>`, но исполнит код только в том случае, если значение равно `3`.

Листинг 6.6. Выражение match, которое исполнит код, только когда значение равно Some(3)

```
let some_u8_value = Some(0u8);
match some_u8_value {
    Some(3) => println!("три"),
    _ => (),
}
```

Нам нужно произвести определенные действия с совпадением `Some(3)`, но ничего не нужно делать с любым другим значением `Some<u8>` или значением `None`. Для того чтобы выражение `match` было удовлетворено, нам приходится добавить `_ => ()` после обработки только одного варианта, что в итоге приводит к добавлению большого объема стереотипного кода.

Вместо этого мы могли бы сделать более краткую запись, используя `if let`. Следующий код ведет себя так же, как и выражение `match` в листинге 6.6:

```
if let Some(3) = some_u8_value {
    println!("три");
}
```

Синтаксис `if let` берет паттерн и выражение, отделенные знаком равенства. Он работает так же, как и синтаксис `match`, где выражение передается в `match`, а этот паттерн является его первым рукавом.

Использование `if let` означает, что нужно печатать меньше текста, делать меньше отступов и писать меньше стереотипного кода. Однако вы теряете исчерпывающую проверку, которую обеспечивает выражение `match`. Выбор между `match` и `if let` зависит от того, что вы делаете в конкретной ситуации и является ли краткость подходящим компромиссом в случае потери исчерпывающей проверки.

Другими словами, вы можете думать об `if let` как о синтаксическом сахаре для `match`, которое выполняет код, когда значение совпадает с одним паттерном, а потом игнорирует все остальные значения.

Мы можем добавить вариант `else` в `if let`. Блок кода, который сопровождает `else`, является таким же, как блок кода, который будет идти вместе с `_` в случае выражения `match`, что равносильно `if let` и `else`. Вспомните определение перечисления `Coin` в листинге 6.4, где вариант `Quarter` также содержал значение `UsState`. Если бы мы хотели пересчитать все монеты, не являющиеся четвертаками, которые ви-

дим, и при этом называть штаты четвертаков, то мы могли бы сделать это с помощью такого выражения `match`:

```
let mut count = 0;
match coin {
    Coin::Quarter(state) => println!("Четвертак из штата {:?}!", state),
    _ => count += 1,
}
```

Или мы могли бы использовать выражения `if let` и `else`:

```
let mut count = 0;
if let Coin::Quarter(state) = coin {
    println!("Четвертак из штата {:?}!", state);
} else {
    count += 1;
}
```

В ситуации, когда логика программы слишком многословна, чтобы выразить ее с помощью `match`, помните, `if let` также находится в арсенале инструментов Rust.

Итоги

Выше мы показали, как использовать перечисления для создания настраиваемых типов, которые могут принимать одно из множества перечисляемых значений. Мы продемонстрировали, как тип `Option<T>` стандартной библиотеки помогает использовать систему типов для предотвращения ошибок. Когда значения перечисления содержат данные, вы можете использовать `match` или `if let` для извлечения и использования этих значений, в зависимости от того, сколько случаев вам нужно учитывать.

Ваши программы на Rust теперь могут выражать понятия в домене с помощью структур и перечислений. Создание настраиваемых типов для использования в API обеспечивает безопасность типов: компилятор сделает так, чтобы функции получали только те значения типа, которые ожидает каждая функция.

Для того чтобы дать вашим пользователям хорошо организованный API, простой в использовании и предоставляющий только то, что нужно, давайте теперь обратимся к модулям.

7 Управление растущими проектами с помощью пакетов, упаковок и модулей

При написании крупных программ очень важна работа по организации кода, потому что отслеживать всю программу станет невозможным. Группируя связанную функциональность и объединяя код в отдельные функциональные элементы, вы будете четко понимать, где найти код, реализующий тот или иной элемент, и где внести изменение в характер работы этого элемента.

Программы, которые мы писали до сих пор, были в одном модуле одного файла. По мере развития проекта можно организовывать код, деля его на многочисленные модули, а затем на многочисленные файлы. Пакет может содержать более одной двоичной упаковки и, возможно, одну библиотечную упаковку. По мере роста пакета вы можете извлекать части в отдельные упаковки, которые становятся внешними зависимостями. В данной главе описаны все эти технические решения. В случае очень крупных проектов, состоящих из совокупности взаимосвязанных пакетов, которые разрабатываются вместе, пакетный менеджер Cargo предоставляет рабочие пространства. Мы рассмотрим их в разделе «Рабочие пространства Cargo» (с. 354).

Кроме группирования функциональности инкапсуляция деталей реализации позволяет использовать код повторно на более высоком уровне. После того как вы реализовали операцию, другой код может вызывать этот код через публичный интерфейс кода, не зная, как эта реализация работает. То, как вы пишете код, определяет, какие части являются публичными для использования другим кодом, а какие — частными деталями реализации, право изменять которые вы оставляете за собой. Это еще один способ ограничить объем деталей, которые нужно держать в голове.

Связанным понятием является область видимости: вложенный контекст, в котором пишется код, имеет ряд имен, определяемых как находящиеся «внутри области». При чтении, написании и компиляции кода программистам и компиляторам нужно знать, относится ли то или иное имя в определенном месте к переменной, функции, структуре, перечислению, модулю, константе или другому элементу и что этот элемент означает. Вы можете создавать области и изменять то, какие имена находятся внутри области или вне ее. У вас не может быть двух элементов с одинаковым именем в одной области. Существуют инструменты, которые занимаются урегулированием конфликтов имен.

Язык Rust имеет ряд средств, позволяющих управлять организацией кода, включая то, какие детали демонстрируются, какие детали являются конфиденциальными и какие имена находятся в каждой области программ. Эти средства, иногда коллективно именуемые модульной системой, включают в себя:

Пакеты

Пакетный менеджер Cargo, который позволяет создавать, тестировать и совместно использовать упаковки.

Упаковки

Дерево модулей, которое производит библиотеку или исполняемый файл.

Модули и use

Позволяют управлять организацией, областью видимости и приватностью путей.

Пути

Способ именования элемента, такого как структура, функция или модуль.

В этой главе мы изучим все эти средства, обсудим вопрос их взаимодействия и объясним способы использования с целью управления областью видимости. К концу главы у вас будет четкое представление о модульной системе, и вы сможете работать с областями видимости как профессионал!

Пакеты и упаковки

Первые части модульной системы, которые мы рассмотрим, — это пакеты и упаковки. Упаковка (*crate*) — это двоичный или библиотечный файл. Корень упаковки (*crate root*) — это исходный файл, с которого компилятор Rust начинает работу, он составляет корневой модуль упаковки (мы подробно объясним модули в разделе «Определение модулей для управления областью видимости и конфиденциальностью»). Пакет — это одна или несколько упаковок, которые обеспечивают функциональность. Пакет содержит файл Cargo.toml, который описывает то, как создавать эти упаковки.

Несколько правил обусловливают возможное содержимое пакета. Пакет должен включать ноль или одну библиотечную упаковку, но не более. Он может содержать столько двоичных упаковок, сколько вам нужно, но в нем должна быть по крайней мере одна упаковка (библиотечная либо двоичная).

Давайте посмотрим на этапы создания пакета. Сначала мы вводим команду `cargo new`:

```
$ cargo new my-project
    Created binary (application) `my-project` package
$ ls my-project
Cargo.toml
```

```
src
$ ls my-project/src
main.rs
```

Когда мы ввели команду, пакетный менеджер Cargo создал файл Cargo.toml, обеспечив нас пакетом. Глядя на содержимое файла Cargo.toml, мы видим, что там нет упоминания о src/main.rs, потому что Cargo следует соглашению о том, что src/main.rs является упаковочным корнем двоичной упаковки с тем же именем, что и пакет. Кроме того, Cargo знает, что если каталог пакета содержит src/lib.rs, то указанный пакет содержит библиотечную упаковку с тем же именем, что и пакет, а src/lib.rs является его упаковочным корнем. Cargo передает файлы корня упаковки в `rustc` для построения библиотеки или двоичного файла.

Здесь мы имеем дело с пакетом, который содержит только src/main.rs, то есть только двоичную упаковку с именем `my-project`. Если в пакете есть src/main.rs и src/lib.rs, то он имеет две упаковки: библиотечную и двоичную — обе с тем же именем, что и пакет. Пакет может иметь несколько двоичных упаковок, помещая файлы в каталог src/bin: каждый файл будет отдельной двоичной упаковкой.

Упаковка будет группировать связанную функциональность в область видимости, вследствие чего эту функциональность можно легко использовать совместно в нескольких проектах. Например, упаковка `rand`, которую мы использовали в разделе «Генерирование секретного числа», предоставляет функциональность, генерирующую случайные числа. Мы можем использовать эту функциональность в собственных проектах, введя упаковку `rand` внутрь области видимости своего проекта. Вся функциональность, предусмотренная упаковкой `rand`, доступна благодаря ее имени, `rand`.

Функциональность упаковки в ее собственной области видимости проясняет, где определяется та или иная функциональность — в нашей упаковке либо в упаковке `rand`, и предотвращает потенциальные конфликты. Например, упаковка `rand` предоставляет типаж с именем `Rng`. Мы также можем определить структуру с именем `Rng` в нашей упаковке. Поскольку функциональность упаковки организуется в пространстве имен в ее собственной области видимости, когда мы добавляем `rand` в качестве зависимости, компилятор не путает смысл имени `Rng`. В нашей упаковке это относится к определенной нами структуре `Rng`. Можно получить доступ к типажу `rand` из упаковки `rand` как `rand::Rnd`.

Теперь давайте поговорим о модульной системе!

Определение модулей для управления областью видимости и конфиденциальностью

В этом разделе мы поговорим о модулях и других частях модульной системы, а именно о путях, позволяющих называть элементы, ключевом слове `use`, которое вводит путь в область видимости, и ключевом слове `pub`, которое делает элементы

публичными. Мы также обсудим ключевое слово `as`, внешние пакеты и оператор `glob`. А пока давайте сфокусируемся на модулях!

Модули позволяют организовывать код внутри упаковки в группы для удобства чтения и многоразового использования. Модули также контролируют конфиденциальность элементов, которая заключается в том, может ли элемент использоваться внешним кодом (публичный) или же он является деталью внутренней реализации и не доступен для внешнего использования (конфиденциальный).

В качестве примера давайте напишем библиотечную упаковку, которая обеспечивает функциональность ресторана. Мы определим сигнатуры функций, но оставим их тела пустыми, чтобы сосредоточиться на организации кода, а не фактически реализовать ресторан в коде.

В ресторанной индустрии некоторые части ресторана называются передней частью заведения (или гостевой), а другие — задней частью (или служебной). В передней части заведения находятся клиенты; здесь хозяева рассаживают гостей, официанты принимают заказы и оплату, а бармены готовят напитки. Задняя часть — это место, где шеф-повара и кухонные работники готовят на кухне, посудомоечные машины моют посуду, а руководители выполняют административную работу.

Для того чтобы структурировать упаковку так же, как работает настоящий ресторан, мы можем организовать функции во вложенные модули. Создайте новую библиотеку с именем restaurant («ресторан»), выполнив команду `cargo new --lib restaurant`. Затем поместите код из листинга 7.1 в src/lib.rs, чтобы определить несколько модулей и сигнатур функций.

Листинг 7.1. Модуль front_of_house, содержащий другие модули, которые затем содержат функции

src/lib.rs

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
    }
}
```

Мы определяем модуль, начиная с ключевого слова `mod`, а затем указываем имя модуля (в данном случае `front_of_house`) и помещаем фигурные скобки вокруг тела модуля. Внутри модулей у нас могут быть другие модули, как в данном случае с модулями `hosting` («размещение») и `serving` («обслуживание»). Модули

также могут содержать определения других элементов, таких как структуры, перечисления, константы, типажи или — как в листинге 7.1 — функции.

Используя модули, мы можем группировать взаимосвязанные определения вместе и называть причину, почему они связаны. Программистам, использующим этот код, будет легче отыскивать определения, которые они хотят использовать, потому что они смогут перемещаться по коду на основе групп, а не прочитывать все определения. Программисты, добавляющие новую функциональность в этот код, будут знать, где размещать код, поддерживая программу организованной.

Ранее мы упоминали, что src/main.rs и src/lib.rs называются корнями упаковки. Причина такого названия заключается в том, что содержимое любого из этих двух файлов формирует модуль с именем `crate` («упаковка») в корне модульной упаковочной структуры, именуемой деревом модулей.

В листинге 7.2 показано дерево модулей для организационной структуры в листинге 7.1.

Листинг 7.2. Дерево модулей для кода в листинге 7.1

```
crate
 └── front_of_house
     ├── hosting
     │   ├── add_to_waitlist
     │   └── seat_at_table
     └── serving
         ├── take_order
         ├── serve_order
         └── take_payment
```

Это дерево показывает, что некоторые модули вставлены друг в друга (например, `hosting` вложен внутрь `front_of_house`). Дерево также показывает, что некоторые модули являются одноуровневыми друг другу, имея в виду, что они определены в одном модуле (модули `hosting` и `serving` определяются в `front_of_house`). Продолжая метафору семьи, если модуль A содержится внутри модуля B, то мы говорим, что модуль A является дочерним к модулю B, а модуль B является родительским по отношению к модулю A. Обратите внимание, что все дерево модулей коренится в скрытом модуле с именем `crate`.

Дерево модулей, возможно, напомнит вам каталожное дерево файловой системы на вашем компьютере: такое сравнение является очень удачным! Как и каталоги в файловой системе, вы используете модули для организации кода. И точно так же, как файлы в каталоге, нам нужен способ отыскивать модули.

Пути для ссылки на элемент в дереве модулей

Для того чтобы показать языку Rust, где в дереве модулей отыскать элемент, мы используем путь точно так же, как во время навигации по файловой системе. Если мы хотим вызвать функцию, то нам нужно знать ее путь.

Путь принимает две формы:

- Абсолютный путь начинается с корня упаковки, используя имя упаковки либо литерал `crate`.
- Относительный путь начинается с текущего модуля и использует `self`, `super` или идентификатор в текущем модуле.

Как абсолютные, так и относительные пути сопровождаются одним или несколькими идентификаторами, разделенными двойными двоеточиями (`::`).

Давайте вернемся к примеру из листинга 7.1. Как вызвать функцию `add_to_waitlist` («добавить в список ожидания»)? Это то же самое, что спросить, каков путь функции `add_to_waitlist`? В листинге 7.3 мы немного упростили код, удалив некоторые модули и функции. Мы покажем два способа вызова функции `add_to_waitlist` из новой функции `eat_at_restaurant` («поесть в ресторане»), определенной в корне упаковки. Функция `eat_at_restaurant` является частью публичного API библиотечной упаковки, поэтому мы помечаем ее ключевым словом `pub`. В разделе «Демонстрация путей с помощью ключевого слова `pub`» мы рассмотрим `pub` подробнее. Обратите внимание, что этот пример пока не компилируется. Вскоре мы объясним почему.

Листинг 7.3. Вызов функции add_to_waitlist с использованием абсолютного и относительного путей

src/lib.rs

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // Абсолютный путь
    crate::front_of_house::hosting::add_to_waitlist();

    // Относительный путь
    front_of_house::hosting::add_to_waitlist();
}
```

Когда мы вызываем функцию `add_to_waitlist` в `eat_at_restaurant` в первый раз, мы используем абсолютный путь. Функция `add_to_waitlist` определена в той же упаковке, что и функция `eat_at_restaurant`. Это означает, что мы можем использовать ключевое слово `crate` для начала абсолютного пути.

После `crate` мы включаем каждый из последующих модулей, пока не доберемся до `add_to_waitlist`. Можно представить файловую систему с такой же организационной структурой; для выполнения программы `add_to_waitlist` мы бы указали путь `/front_of_house/hosting/add_to_waitlist`. Использовать имя `crate` для начала движения от корня упаковки — это все равно что использовать `/` для начала движения от корня файловой системы в оболочке.

Когда мы вызываем `add_to_waitlist` в `eat_at_restaurant` во второй раз, мы используем относительный путь. Путь начинается с `front_of_house`, имени модуля, определенного на том же уровне дерева модулей, что и `eat_at_restaurant`. Здесь эквивалент файловой системы будет использовать путь `front_of_house/hosting/add_to_waitlist`. Начало с имени означает, что путь является относительным.

Принимать решение о выборе относительного или абсолютного пути вы будете, основываясь на своем проекте. Указанное решение будет зависеть от того, станете ли вы перемещать код определения элемента отдельно или вместе с кодом, использующим этот элемент. Например, если мы переместим модуль `front_of_house` и функцию `eat_at_restaurant` в модуль `customer_experience` («удовлетворенность качеством обслуживания»), то нам нужно будет обновить абсолютный путь до `add_to_waitlist`, но относительный путь все равно будет допустим. Однако если мы переместим функцию `eat_at_restaurant` отдельно в модуль с именем `dining` («ужин в ресторане»), то абсолютный путь к вызову `add_to_waitlist` останется прежним, а относительный путь необходимо будет обновить. Мы предпочитаем указывать абсолютные пути, поскольку существует бо́льшая вероятность того, что определения кода и вызовы элементов будут перемещаться независимо друг от друга.

Давайте попробуем скомпилировать листинг 7.3 и выясним, почему он еще не компилируется!

Полученная нами ошибка показана в листинге 7.4[1].

Листинг 7.4. Ошибки компилятора при построении кода в листинге 7.3

```
$ cargo build
    Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: module `hosting` is private
 --> src/lib.rs:9:28
  |
9 |     crate::front_of_house::hosting::add_to_waitlist();
  |                            ^^^^^^^

error[E0603]: module `hosting` is private
  --> src/lib.rs:12:21
   |
12 |     front_of_house::hosting::add_to_waitlist();
   |                     ^^^^^^^
```

Сообщения об ошибках говорят о том, что модуль `hosting` является конфиденциальным. Другими словами, у нас есть правильные пути для модуля `hosting` и функции `add_to_waitlist`, но Rust не позволит нам их использовать, потому что у него нет доступа к конфиденциальным разделам.

Модули полезны не только для организации своего кода. Они также определяют границу конфиденциальности в Rust: контур, инкапсулирующий детали реализа-

[1] `ошибка[E0603]: модуль `hosting` является конфиденциальным`

ции, о которых внешний код не должен знать, которые он не должен вызывать или на которые он не должен полагаться. Поэтому, если вы хотите сделать элемент, такой как функция или структура, конфиденциальным, вам нужно поместить его в модуль.

Принцип работы конфиденциальности в Rust заключается в том, что все элементы (функции, методы, структуры, перечисления, модули и константы) конфиденциальны по умолчанию. Элементы в родительском модуле не могут использовать конфиденциальные элементы внутри дочерних модулей, но элементы дочерних модулей могут использовать элементы в своих предковых модулях. Причина этого заключается в том, что дочерние модули обертывают и скрывают детали своей реализации, но они видят контекст, в котором определены. Продолжая ресторанную метафору, подумайте о правилах конфиденциальности как о служебном помещении ресторана: то, что происходит там, скрыто от клиентов, но сотрудники могут видеть и делать в ресторане, в котором они работают, абсолютно все.

В Rust функционирование модульной системы организовано так, чтобы сокрытие деталей внутренней реализации осуществлялось по умолчанию. Благодаря этому вы знаете, какие части внутреннего кода вы можете изменять, не нарушая внешний код. Но вы можете показывать внутренние части кода дочерних модулей внешним предковым модулям, используя ключевое слово `pub`, делая элемент публичным.

Демонстрация путей с помощью ключевого слова pub

Давайте вернемся к ошибке в листинге 7.4, которая сообщила о том, что модуль `hosting` является конфиденциальным. Мы хотим, чтобы функция `eat_at_restaurant` в родительском модуле имела доступ к функции `add_to_waitlist` в дочернем модуле, поэтому помечаем модуль `hosting` ключевым словом `pub`, как показано в листинге 7.5.

Листинг 7.5. Объявление модуля hosting как pub, чтобы использовать его из eat_at_restaurant

src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // Абсолютный путь
    crate::front_of_house::hosting::add_to_waitlist();

    // Относительный путь
    front_of_house::hosting::add_to_waitlist();
}
```

К сожалению, код в листинге 7.5 по-прежнему приводит к ошибке, как показано в листинге 7.6[1].

Листинг 7.6. Ошибки компилятора при построении кода в листинге 7.5

```
$ cargo build
    Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: function `add_to_waitlist` is private
 --> src/lib.rs:9:37
  |
9 |     crate::front_of_house::hosting::add_to_waitlist();
  |                                     ^^^^^^^^^^^^^^^

error[E0603]: function `add_to_waitlist` is private
  --> src/lib.rs:12:30
   |
12 |     front_of_house::hosting::add_to_waitlist();
   |                              ^^^^^^^^^^^^^^^
```

Что случилось? Добавление ключевого слова `pub` перед `mod hosting` делает модуль публичным. С этим изменением, в случае если мы можем обратиться к `front_of_house`, можно обратиться к модулю `hosting`. Но содержимое модуля `hosting` по-прежнему является конфиденциальным; делая модуль публичным, вы не делаете публичным его содержимое. Ключевое слово `pub` в модуле позволяет коду ссылаться на него только в его предковых модулях.

Ошибки в листинге 7.6 говорят о том, что функция `add_to_waitlist` является конфиденциальной. Правила конфиденциальности применяются к структурам, перечислениям, функциям и методам, а также модулям.

Давайте также сделаем функцию `add_to_waitlist` публичной, добавив ключевое слово `pub` перед ее определением, как в листинге 7.7.

Листинг 7.7. Добавление ключевого слова pub в mod hosting и fn add_to_waitlist позволяет вызывать функцию из eat_at_restaurant

src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // Абсолютный путь
    crate::front_of_house::hosting::add_to_waitlist();

    // Относительный путь
    front_of_house::hosting::add_to_waitlist();
}
```

[1] `ошибка[E0603]: функция `add_to_waitlist` является конфиденциальной`

Теперь код будет компилироваться! Давайте посмотрим на абсолютный и относительный путь и перепроверим, почему добавление ключевого слова `pub` позволяет использовать эти пути в `add_to_waitlist` в соответствии с правилами конфиденциальности.

В абсолютном пути мы начинаем с `crate`, корня дерева модулей упаковки. Затем в корне упаковки определяется модуль `front_of_house`. Модуль `front_of_house` не является публичным, но так как функция `eat_at_restaurant` определена в том же модуле, что и модуль `front_of_house` (то есть `eat_at_restaurant` и `front_of_house` являются одноуровневыми), мы можем ссылаться на модуль `front_of_house` из функции `eat_at_restaurant`. Далее идет модуль `hosting`, помеченный ключевым словом `pub`. Мы можем обратиться к родительскому модулю `hosting`, поэтому можно обратиться и к модулю `hosting`. Наконец, функция `add_to_waitlist` помечена ключевым словом `pub`, и можно получить доступ к ее родительскому модулю, поэтому вызов функции работает!

В относительном пути логика такая же, как и в абсолютном, за исключением первого шага: вместо того, чтобы начинать с корня упаковки, путь начинается с модуля `front_of_house`. Модуль `front_of_house` определяется в том же модуле, что и функция `eat_at_restaurant`, поэтому относительный путь, начинающийся с модуля, в котором определена функция `eat_at_restaurant`, работает. Тогда, поскольку модуль `hosting` и функция `add_to_waitlist` помечены ключевым словом `pub`, остальная часть пути работает и вызов функции является допустимым!

Начало относительных путей с помощью super

Мы также можем конструировать относительные пути, которые начинаются в родительском модуле, используя в начале пути ключевое слов `super`. Это похоже на начало пути файловой системы с помощью синтаксиса `..`. Почему нам хотелось бы это делать?

Рассмотрим код в листинге 7.8, моделирующий ситуацию, в которой шеф-повар исправляет неправильный заказ и лично выносит его клиенту. Функция `fix_incorrect_order` («исправить неправильный заказ») вызывает функцию `serve_order` («обслужить заказ»), указав путь к `serve_order`, начинающийся с ключевого слова `super`.

Листинг 7.8. Вызов функции с использованием относительного пути, начинающегося с super

src/lib.rs

```
fn serve_order() {}

mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        super::serve_order();
```

```
    }

    fn cook_order() {}
}
```

Функция `fix_incorrect_order` находится в модуле `back_of_house`, поэтому мы можем использовать `super` для перехода к родительскому модулю `back_of_house`, который в данном случае является `crate`, корнем. Таким образом, мы ищем `serve_order` и находим его. Получилось! Мы думаем, что модуль `back_of_house` и функция `serve_order`, скорее всего, останутся в том же отношении друг к другу и будут перемещены вместе, в случае если мы решим реорганизовать дерево модулей упаковки. Следовательно, мы использовали `super`, чтобы в будущем у нас было меньше мест для обновления кода, если этот код будет перенесен в другой модуль.

Обозначение структур и перечислений как публичных

Мы также можем использовать ключевое слово `pub` для обозначения структур и перечислений как публичных, но тут есть несколько дополнительных деталей. Если мы используем `pub` перед определением структуры, то делаем структуру публичной, но поля структуры все равно будут конфиденциальными. Можно сделать каждое поле публичным на индивидуальной основе. В листинге 7.9 мы определили публичную структуру `back_of_house::Breakfast` с публичным полем `toast`, но конфиденциальным полем `seasonal_fruit`. Так моделируется случай в ресторане, где клиент сам выбирает хлеб, который ему принесут вместе с блюдом, но шеф-повар решает, какие фрукты подать, исходя из времени года и наличия. Список доступных фруктов быстро меняется, поэтому клиенты не могут ни выбрать фрукты, ни даже узнать, какие именно фрукты им принесут.

Листинг 7.9. Структура, в которой несколько публичных и конфиденциальных полей

src/lib.rs

```
mod back_of_house {
    pub struct Breakfast {
        pub toast: String,
        seasonal_fruit: String,
    }

    impl Breakfast {
        pub fn summer(toast: &str) -> Breakfast {
            Breakfast {
                toast: String::from(toast),
                seasonal_fruit: String::from("персики"),
            }
        }
    }
}

pub fn eat_at_restaurant() {
```

```
    // Заказать летом завтрак с ржаным тостом
    let mut meal = back_of_house::Breakfast::summer("ржаной");
    // Изменить мнение о том, какой хлеб мы бы хотели
    meal.toast = String::from("пшеничный");
    println!("Я бы хотел {} тост, пожалуйста", meal.toast);

    // Следующая строка не будет компилироваться, если мы раскомментируем ее;
    // нам запрещено просматривать или изменять сезонные фрукты,
    // которые принесут с едой
    // meal.seasonal_fruit = String::from("черника");
}
```

Поскольку поле `toast` в структуре `back_of_house::Breakfast` является публичным, в функции `eat_at_restaurant` мы можем читать поле `toast` и писать в него с помощью точечной нотации. Обратите внимание, что мы не можем использовать поле `seasonal_fruit` в функции `eat_at_restaurant`, потому что `seasonal_fruit` является конфиденциальным. Попробуйте раскомментировать строку, изменив значение поля `seasonal_fruit`, чтобы увидеть, какая ошибка произойдет!

Кроме того, обратите внимание, что, поскольку структура `back_of_house::Breakfast` имеет конфиденциальное поле, она должна предоставить публичную связанную функцию, которая конструирует экземпляр структуры `Breakfast` (мы назвали ее здесь `summer`). Если бы в структуре `Breakfast` не было такой функции, то мы бы не смогли создать экземпляр указанной структуры в функции `eat_at_restaurant`, потому что нам бы не удалось задать значение конфиденциального поля `seasonal_fruit` в функции `eat_at_restaurant`.

Напротив, если мы сделаем перечисление публичным, то тогда все его варианты будут публичными. Нам нужно ключевое слово `pub` только перед ключевым словом `enum`, как показано в листинге 7.10.

Листинг 7.10. Обозначение перечисления как публичного делает все его варианты публичными

src/lib.rs

```
mod back_of_house {
    pub enum Appetizer {
        Soup,
        Salad,
    }
}

pub fn eat_at_restaurant() {
    let order1 = back_of_house::Appetizer::Soup;
    let order2 = back_of_house::Appetizer::Salad;
}
```

Поскольку мы сделали перечисление `Appetizer` («закуска») публичным, мы можем использовать варианты `Soup` и `Salad` в функции `eat_at_restaurant`. Перечисления не очень полезны, если их варианты не являются публичными. Необ-

ходимость комментировать все варианты перечисления с помощью pub в каждом случае могла бы раздражать, поэтому по умолчанию варианты перечисления являются публичными. Структуры часто бывают полезными, когда их поля не публичны, поэтому поля структуры следуют общему правилу, что все должно быть конфиденциальным по умолчанию, если только не аннотировано с помощью pub.

Есть еще одна ситуация, связанная с pub, о которой мы не говорили. Это последнее средство модульной системы — ключевое слово use. Сначала мы рассмотрим указанное ключевое слово отдельно, а затем покажем, как сочетать pub и use.

Введение путей в область видимости с помощью ключевого слова use

Может показаться, что пути, которые мы до этого писали для вызова функций, слишком длинные и часто повторяются. Например, в листинге 7.7, независимо от того, выбираем ли мы абсолютный или относительный путь к функции add_to_waitlist, всякий раз, когда мы хотим вызвать add_to_waitlist, нужно указывать front_of_house и hosting тоже. К счастью, есть способ упростить этот процесс. Мы можем ввести путь в область видимости один раз, а затем вызывать элементы в этом пути, как если бы они были локальными элементами, с помощью ключевого слова use.

В листинге 7.11 мы вводим модуль crate::front_of_house::hosting в область функции eat_at_restaurant, и поэтому для вызова функции add_to_waitlist в eat_at_restaurant нам нужно только указать hosting::add_to_waitlist.

Листинг 7.11. Введение модуля в область видимости с помощью use

src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

Добавление use и пути в область видимости аналогично созданию символической ссылки в файловой системе. Благодаря добавлению use crate::front_of_house::hosting в корень упаковки модуль hosting сейчас является допустимым именем в этой области видимости, как будто модуль hosting был определен в кор-

не упаковки. Пути, введенные в область видимости с помощью `use`, проверяют конфиденциальность так же, как и любые другие пути.

Указание относительного пути с помощью `use` немного отличается. Вместо того, чтобы начинать с имени в текущей области видимости, мы должны начинать путь, переданный инструкции `use`, с ключевого слова `self`. В листинге 7.12 показано, как указать относительный путь для получения того же результата, что и в листинге 7.11.

Листинг 7.12. Введение модуля в область видимости с помощью use и относительный путь, начинающийся с self

src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use self::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

Обратите внимание, что такое использование `self` в будущем может не понадобиться. Оно представляет собой нестыковку в языке, над устранением которой работают разработчики Rust.

Создание идиоматических путей use

Глядя на листинг 7.11, можно задаться вопросом, почему мы указали `use crate::front_of_house::hosting`, а затем вызвали `hosting::add_to_waitlist` в функции `eat_at_restaurant` вместо указания пути `use` вплоть до функции `add_to_waitlist` для достижения того же результата, что и в листинге 7.13.

Листинг 7.13. Введение функции add_to_waitlist в область видимости с помощью use, которое не является идиоматическим

src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting::add_to_waitlist;

pub fn eat_at_restaurant() {
    add_to_waitlist();
```

```
    add_to_waitlist();
    add_to_waitlist();
}
```

Хотя оба листинга — 7.11 и 7.13 — выполняют одну и ту же задачу, листинг 7.11 является идиоматическим способом введения функции в область видимости с помощью `use`. Введение родительского модуля функции в область видимости с помощью `use` так, что нам приходится указывать родительский модуль при вызове функции, дает понять, что указанная функция не определена локально, но при этом минимизирует повторы полного пути. Код в листинге 7.13 неясен относительно того, где определена функция `add_to_waitlist`.

С другой стороны, при введении структур, перечислений и других элементов с помощью `use`, идиоматический способ состоит в указании полного пути. В листинге 7.14 показан идиоматический способ введения структуры `HashMap` стандартной библиотеки в область двоичной упаковки.

Листинг 7.14. Введение структуры HashMap в область видимости идиоматическим способом

src/main.rs

```
use std::collections::HashMap;

fn main() {
    let mut map = HashMap::new();
    map.insert(1, 2);
}
```

Эта идиома не имеет никакой серьезной причины, это просто возникшая условность, и люди привыкли читать и писать код на Rust таким образом.

Исключение из этой идиомы возникает, если мы вводим в область видимости два элемента с одинаковым именем с помощью инструкций `use`, потому что язык Rust не разрешает это делать. В листинге 7.15 показано, как ввести в область видимости два типа `Result`, у которых одинаковые имена, но разные родительские модули, и как на них ссылаться.

Листинг 7.15. Введение двух типов с одинаковыми именами в одну область требует использования их родительских модулей

src/lib.rs

```
use std::fmt;
use std::io;

fn function1() -> fmt::Result {
    // --пропуск--
}

fn function2() -> io::Result<()> {
    // --пропуск--
}
```

Как вы видите, использование родительских модулей различает два типа `Result`. Если бы вместо этого мы указали `use std::fmt::Result` и `use std::io::Result`, то у нас было бы два типа `Result` в одной области, и язык Rust не знал бы, какой из них мы имели в виду, когда использовали `Result`.

Предоставление новых имен с помощью ключевого слова as

Существует еще одно решение проблемы введения двух типов одного и того же имени в одну область видимости с помощью `use`: после пути мы можем указать `as` и новое локальное имя, или псевдоним, для типа. В листинге 7.16 показан еще один способ написания кода из листинга 7.15 путем переименования одного из двух типов `Result` с помощью `as`.

Листинг 7.16. Переименование типа, когда он вводится в область видимости с помощью ключевого слова as

src/lib.rs

```
use std::fmt::Result;
use std::io::Result as IoResult;

fn function1() -> Result {
    // --пропуск--
}

fn function2() -> IoResult<()> {
    // --пропуск--
}
```

Во второй инструкции `use` мы выбрали новое имя `IoResult` для типа `std::io::Result`, которое не будет конфликтовать с `Result` из `std::fmt`, который мы также ввели в область видимости. Листинг 7.15 и листинг 7.16 считаются идиоматическими, поэтому выбор за вами!

Реэкспорт имен с использованием pub

Когда мы вводим имя в область видимости с помощью ключевого слова `use`, то имя, доступное в новой области, является конфиденциальным. Чтобы дать возможность коду, который вызывает наш код, ссылаться на это имя так, как если бы оно было определено в области видимости этого кода, мы можем совместить `pub` и `use`. Этот прием называется реэкспортом, потому что мы вводим элемент в область видимости, но также делаем его доступным, благодаря чему другие пользователи могут ввести его в свою область видимости.

Листинг 7.17 показывает код из листинга 7.11 с заменой `use` в корневом модуле на `pub use`.

Листинг 7.17. Предоставление имени для использования любым кодом из новой области с помощью pub use

src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

Используя `pub use`, внешний код теперь может вызывать функцию `add_to_waitlist` с помощью `hosting::add_to_waitlist`. Если бы мы не указали `pub use`, то функция `eat_at_restaurant` могла бы вызывать `hosting::add_to_waitlist` в своей области видимости, но внешний код не смог бы воспользоваться этим новым путем.

Реэкспорт полезен, когда внутренняя структура кода отличается от того, как программисты, вызывающие ваш код, будут думать о домене. Например, в ресторанной метафоре люди, управляющие рестораном, думают о «передней» и «задней» части заведения. Но клиенты, посещающие ресторан, вероятно, не будут думать о частях ресторана в таких определениях. При использовании `pub use` мы можем писать код с одной структурой, но демонстрировать другую структуру. Это хорошо организует нашу библиотеку для программистов, работающих над библиотекой, и программистов, вызывающих ее.

Использование внешних пакетов

В главе 2 мы создали программу проекта игры-угадайки, который использовал внешний пакет под названием `rand` с целью получения случайных чисел. Для того чтобы использовать пакет `rand` в проекте, мы добавили такую строку в Cargo.toml:

Cargo.toml

```
[dependencies]
rand = "0.5.5"
```

Добавление `rand` в качестве зависимости в Cargo.toml поручает Cargo загрузить пакет `rand` и любые зависимости из https://crates.io/ и сделать `rand` доступным для проекта.

Затем, для того чтобы ввести определения `rand` в область видимости пакета, мы добавили строку `use`, начинающуюся с имени пакета `rand`, и перечислили элементы, которые мы хотели бы ввести в область видимости. Напомним, что в разде-

ле «Генерирование случайного числа» мы ввели в область видимости типаж `Rng` и вызывали функцию `rand::thread_rng`:

```
use rand::Rng;
fn main() {
    let secret_number = rand::thread_rng().gen_range(1, 101);
}
```

Члены сообщества Rust предоставили в общее пользование большое число пакетов по адресу https://crates.io/, и ввод любого из них в ваш пакет предусматривает те же шаги: перечисление их в файле Cargo.toml вашего пакета и использование `use` для введения элементов в область видимости.

Обратите внимание, что стандартная библиотека (`std`) также является упаковкой, внешней по отношению к пакету. Поскольку стандартная библиотека поставляется с языком Rust, нам не нужно изменять Cargo.toml для включения `std`. Но мы все равно должны обращаться к нему с помощью `use`, чтобы ввести оттуда элементы в область видимости пакета. Например, в случае с `HashMap` мы бы использовали такую строку:

```
use std::collections::HashMap;
```

Она представляет собой абсолютный путь, начинающийся с `std`, имени упаковки стандартной библиотеки.

Использование вложенных путей для очистки больших списков use

Если мы используем несколько элементов, определенных в одном пакете или одном модуле, указание каждого элемента в отдельной строке будет занимать много вертикального пространства в файлах. Например, следующие две инструкции `use`, которые мы использовали в игре на угадывание в листинге 2.4, вводят в область видимости элементы из `std`:

src/main.rs

```
use std::io;
use std::cmp::Ordering;
// --пропуск--
```

Вместо этого мы можем использовать вложенные пути, вводя те же самые элементы в область видимости в одной строке. Мы делаем это, указывая общую часть пути с последующими двумя двоеточиями, а затем фигурными скобками вокруг списка частей путей, которые отличаются друг от друга, как показано в листинге 7.18.

Листинг 7.18. Указание вложенного пути для введения в область видимости нескольких элементов с одинаковым префиксом

src/main.rs

```
use std::{io, cmp::Ordering};
// --пропуск--
```

В более крупных программах введение в область видимости большого числа элементов из одного пакета или модуля с помощью вложенных путей значительно сокращает число необходимых для этого отдельных инструкций `use`.

Мы можем использовать вложенный путь на любом уровне пути, что полезно при объединении двух инструкций `use`, совместно использующих вложенный путь. Например, листинг 7.19 показывает две инструкции: одна вводит в область видимости `std::io`, а другая вводит в область видимости `std::io::Write`.

Листинг 7.19. Две инструкции, где одна является подпутем другой

src/lib.rs

```
use std::io;
use std::io::Write;
```

Общей частью обоих путей является путь `std::io`, он представляет собой полный первый путь. Для объединения этих двух путей в одну инструкцию `use` мы можем использовать `self` во вложенном пути, как показано в листинге 7.20.

Листинг 7.20. Объединение путей в листинге 7.19 в одну инструкцию `use`

src/lib.rs

```
use std::io::{self, Write};
```

Эта строка вводит в область видимости `std::io` и `std::io::Write`.

Оператор glob

Если мы хотим ввести в область видимости все публичные элементы, определенные в пути, то можно указать этот путь с последующим оператором glob*:

```
use std::collections::*;
```

Эта инструкция `use` вводит в текущую область видимости все публичные элементы, определенные в `std::collections`. При использовании оператора glob будьте осторожны! Он может затруднить определение того, какие имена находятся в области видимости и где было определено имя, используемое в программе.

Оператор glob часто используется при тестировании с целью введения всего, что тестируется, в модуль `tests`. Мы поговорим об этом в разделе «Как писать тесты». Оператор glob также иногда используется как часть паттерна прелюдии. Для получения дополнительной информации об этом паттерне обратитесь к документации стандартной библиотеки по адресу https://doc.rust-lang.org/stable/std/prelude/index.html#other-preludes.

Разделение модулей на разные файлы

До сих пор все примеры в этой главе определяли несколько модулей в одном файле. Когда модули станут крупными, вы, возможно, захотите переместить их определения в отдельный файл, чтобы облегчить навигацию по коду.

Давайте начнем с кода в листинге 7.17 и переместим модуль `front_of_house` в его собственный файл src/front_of_house.rs, изменив файл корня упаковки таким образом, чтобы он содержал код, показанный в листинге 7.21. В этом случае файлом корня упаковки является src/lib.rs, но эта процедура также работает с двоичными упаковками, чьим файлом корня упаковки является src/main.rs.

Листинг 7.21. Объявление модуля front_of_house, тело которого будет находиться в src/front_of_house.rs

src/lib.rs

```
mod front_of_house;

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

И src/front_of_house.rs получает определения из тела модуля `front_of_house`, как показано в листинге 7.22.

Листинг 7.22. Определения внутри модуля front_of_house в src/front_of_house.rs

src/front_of_house.rs

```
pub mod hosting {
    pub fn add_to_waitlist() {}
}
```

Использование точки с запятой после `mod front_of_house` вместо блока сообщает языку Rust о том, что нужно загрузить содержимое модуля из другого файла с тем же именем, что и модуль. В продолжение нашего примера, чтобы извлечь модуль `hosting` в его собственный файл, мы изменим src/front_of_house.rs так, чтобы он содержал только объявление модуля `hosting`:

src/front_of_house.rs

```
pub mod hosting;
```

Затем мы создаем каталог src/front_of_house и файл src/front_of_house/hosting.rs, содержащий определения, сделанные в модуле `hosting`:

src/front_of_house/hosting.rs

```
pub fn add_to_waitlist() {}
```

Дерево модулей остается тем же самым, а вызовы функций в `eat_at_restaurant` будут работать без каких-либо изменений, даже если определения располагаются в разных файлах. Этот технический прием позволяет перемещать модули в новые файлы по мере увеличения их размера.

Обратите внимание, что инструкция `pub use crate::front_of_house::hosting` в src/lib.rs тоже не изменилась, а инструкция `use` также не оказывает никакого влияния на то, какие файлы компилируются как часть упаковки. Ключевое слово `mod` объявляет модули, и Rust ищет в файле с тем же именем, что и модуль, код, который входит в этот модуль.

Итоги

Rust позволяет организовывать пакеты в упаковки, а упаковки в модули, благодаря чему вы можете ссылаться на элементы, определенные в одном модуле, из другого модуля. Это можно делать, указывая абсолютные или относительные пути. Эти пути могут быть введены в область видимости с помощью инструкции `use`, благодаря чему вы можете задействовать более короткий путь для повторного использования элемента в этой области. Код модуля по умолчанию является конфиденциальным, но вы можете сделать определения публичными, добавив ключевое слово `pub`.

В следующей главе мы рассмотрим некоторые коллекционные структуры данных из стандартной библиотеки, которые вы можете использовать в своем отлично организованном коде.

8 Общие коллекции

Стандартная библиотека Rust включает в себя ряд очень полезных структур данных, которые называются коллекциями. Большинство других типов данных представляют одно конкретное значение, но коллекции могут содержать многочисленные значения. В отличие от встроенных массивного и кортежного типов, данные, на которые указывают эти коллекции, хранятся в куче, то есть необходимости знать объем данных во время компиляции нет, и они увеличиваются или уменьшаются в ходе выполнения программы. Каждый вид коллекции имеет разные возможности и затраты, а умение выбирать подходящий вид для текущей ситуации является навыком, который вы приобретете с течением времени. В этой главе мы обсудим три коллекции, которые очень часто используются в программах Rust:

- Вектор позволяет хранить переменное число значений рядом.
- Строка представляет собой коллекцию символов. Мы уже упоминали тип `String` ранее, но в этой главе мы поговорим о нем подробнее.
- Хешируемое отображение позволяет связывать значение с определенным ключом. Оно является конкретной реализацией более общей структуры данных, которая называется отображением (map).

Дополнительные сведения о других типах коллекций, предусмотренных стандартной библиотекой, см. в документации по адресу https://doc.rust-lang.org/stable/std/collections/.

Мы обсудим способы создания и обновления векторов, строк и хеш-отображений, а также то, что делает каждую из коллекций особенной.

Хранение списков значений с помощью векторов

Первым типом коллекции, который мы рассмотрим, является `Vec<T>`, так называемый «вектор». Векторы позволяют хранить более одного значения в одной структуре данных, которая помещает все значения рядом в памяти. Векторы хра-

нят значения только одного типа. Они полезны, когда у вас есть список элементов, таких как строки текста в файле или цены товаров в корзине покупок.

Создание нового вектора

Для того чтобы создать новый пустой вектор, мы вызываем функцию `Vec::new`, как показано в листинге 8.1.

Листинг 8.1. Создание нового пустого вектора для хранения значений типа i32

```
let v: Vec<i32> = Vec::new();
```

Обратите внимание, что мы добавили аннотацию типа. Поскольку мы не вставляем никаких значений в этот вектор, Rust не знает, какие элементы мы намерены хранить. Это очень важный момент. Векторы реализуются с помощью обобщений. Мы рассмотрим способы использования обобщений с собственными типами в главе 10. Пока же знайте, что тип `Vec<T>`, предусмотренный стандартной библиотекой, может содержать любой тип, и когда конкретный вектор содержит конкретный тип, этот тип указывается в угловых скобках. В листинге 8.1 мы сказали языку Rust о том, что `Vec<T>` в `v` будет содержать элементы типа `i32`.

В более реалистичном коде Rust часто логически выводит тип значения, которое вы хотите хранить после того, как вставляете значения, поэтому от вас редко требуется аннотировать этот тип. Чаще создается `Vec<T>`, который имеет начальные значения, и для удобства Rust предоставляет макрокоманду `vec!`. Указанная макрокоманда будет создавать новый вектор со значениями, которые вы зададите. Листинг 8.2 создает новый `Vec<i32>`, который содержит значения 1, 2 и 3.

Листинг 8.2. Создание нового вектора, содержащего значения

```
let v = vec![1, 2, 3];
```

Поскольку мы задали начальные значения типа `i32`, Rust логически выводит, что типом `v` является `Vec<i32>`, и аннотация типа не нужна. Далее мы посмотрим, как модифицировать вектор.

Обновление вектора

Для того чтобы создать вектор и затем добавить в него элементы, мы используем метод `push`, как показано в листинге 8.3.

Листинг 8.3. Использование метода push для добавления значений в вектор

```
let mut v = Vec::new();

v.push(5);
v.push(6);
v.push(7);
v.push(8);
```

Как и в случае с любой переменной, если мы хотим иметь возможность изменять его значение, нам нужно сделать его изменяемым с помощью ключевого слова `mut`, как описано в главе 3. Все числа, которые мы помещаем внутрь, относятся к типу `i32`, а язык Rust выводит это логически из данных, поэтому аннотация `Vec<i32>` не нужна.

Отбрасывание вектора отбрасывает его элементы

Как и любая другая структура, вектор высвобождается, когда он выходит из области видимости, как указано в листинге 8.4.

Листинг 8.4. Показывает точку, где вектор и его элементы отбрасываются

```
{
    let v = vec![1, 2, 3, 4];

    // что-то сделать с v

} // <- здесь v выходит из области видимости и высвобождается
```

Когда вектор отбрасывается, все его содержимое тоже отбрасывается, то есть те целые числа, которые он содержит, будут очищены. Этот момент, возможно, покажется простым, но он окажется немного сложнее, когда вы начнете вводить ссылки на элементы вектора. Давайте разберемся с этим вопросом далее.

Чтение элементов вектора

Теперь, когда вы знаете, как векторы создаются, обновляются и уничтожаются, пора познакомиться с тем, как читать их содержимое. Существует два способа ссылаться на значение, хранящееся в векторе. В примерах мы ради большей ясности аннотировали типы значений, возвращаемых этими функциями.

Листинг 8.5 показывает оба метода доступа к значению в векторе, с помощью синтаксиса индексирования либо с помощью метода `get`.

Листинг 8.5. Использование синтаксиса индексирования либо метода `get` для доступа к элементу в векторе

```
let v = vec![1, 2, 3, 4, 5];

let third: &i32 = &v[2];
println!("Третий элемент равен {}", third);

match v.get(2) {
    Some(third) => println!("Третий элемент равен {}", third),
    None => println!("Третий элемент отсутствует."),
}
```

Обратите внимание на две детали. Во-первых, мы используем индексное значение 2, чтобы получить третий элемент: векторы индексируются по числу, начиная

с нуля. Во-вторых, два упомянутых способа получить третий элемент предусматривают использование `&` и `[]`, в результате чего мы получаем ссылку, либо использование метода `get` с индексом, переданным в качестве аргумента, в результате чего мы получаем `Option<&T>`.

Rust может ссылаться на элемент двумя способами, благодаря чему можно выбирать, как программа себя ведет при попытке использовать значение индекса, для которого вектор не имеет элемента. В качестве примера давайте посмотрим, что программа будет делать, если она имеет вектор, содержащий пять элементов, а затем пытается обратиться к элементу с индексом 100, как показано в листинге 8.6.

Листинг 8.6. Попытка обратиться к элементу с индексом 100 в векторе, содержащем пять элементов

```
let v = vec![1, 2, 3, 4, 5];

let does_not_exist = &v[100];
let does_not_exist = v.get(100);
```

Когда мы выполним этот код, первый метод `[]` станет причиной того, что программа поднимет панику, потому что он ссылается на несуществующий элемент. Этот метод лучше всего использовать, когда вы хотите, чтобы программа завершила работу аварийно, если есть попытка обратиться к элементу после окончания вектора.

Когда метод `get` получает индекс, который находится вне вектора, он возвращает `None` без паники. Вы можете использовать этот метод, если обращение к элементу вне диапазона вектора происходит нерегулярно при нормальных обстоятельствах. Тогда код будет иметь логику для обработки, имея либо `Some(&element)`, либо `None`, как описано в главе 6. Например, индекс может исходить от человека, вводящего число. Если он ненароком вводит слишком большое число и программа получает значение `None`, то вы можете сообщить пользователю о количестве элементов, находящихся в текущем векторе, и дать ему еще один шанс ввести допустимое значение. Это было бы для пользователя удобнее, чем аварийный сбой программы из-за опечатки!

Когда у программы есть действительная ссылка, контролер заимствования применяет правила владения и заимствования (описанные в главе 4) и обеспечивает, чтобы эта ссылка и любые другие ссылки на содержимое вектора оставались действительными. Вспомните правило, которое гласит, что вы не можете иметь изменяемые и неизменяемые ссылки в одной и той же области. Это правило применяется в листинге 8.7, где содержится неизменяемая ссылка на первый элемент вектора, а мы пытаемся добавить элемент в конец. Это не будет работать.

Листинг 8.7. Попытка добавить элемент в вектор при сохранении ссылки на элемент

```
let mut v = vec![1, 2, 3, 4, 5];

let first = &v[0];

v.push(6);

println!("Первый элемент равен {}", first);
```

Компиляция этого кода приведет к ошибке[1]:

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as
immutable
 --> src/main.rs:6:5
  |
4 |     let first = &v[0];
  |                  - immutable borrow occurs here
5 |
6 |     v.push(6);
  |     ^^^^^^^^^ mutable borrow occurs here
7 |
8 |     println!("The first element is: {}", first);
  |                                          ----- immutable borrow later used
                                                 here
```

Код в листинге 8.7 выглядит так, будто должен работать. Почему для использования ссылки на первый элемент должно быть важно, что изменяется в конце вектора? Эта ошибка связана с тем, как работают векторы: добавление нового элемента в конец вектора может потребовать выделения новой памяти и копирования старых элементов в новое пространство, если не хватает места для того, чтобы поместить все элементы рядом, туда, где в данный момент находится вектор. В этом случае ссылка на первый элемент будет указывать на освобожденную память. Правила заимствования не позволяют программам оказываться в такой ситуации.

ПРИМЕЧАНИЕ

Дополнительные сведения о реализации типа `Vec<T>` см. в «Растономиконе», книге тайных знаний о Rust, по адресу https://doc.rust-lang.org/stable/nomicon/vec.html, которая затрагивает самые «ужасные» подробности языка.

Перебор значений в векторе

Если мы хотим обращаться к каждому элементу вектора по очереди, то можем перебирать все элементы в цикле вместо того, чтобы использовать индексы для доступа по одному за раз. В листинге 8.8 показано, как использовать цикл `for`, получая неизменяемые ссылки на каждый элемент вектора значений типа `i32` и печатая их.

Листинг 8.8. Печать каждого элемента в векторе путем перебора элементов с помощью цикла for

```
let v = vec![100, 32, 57];
for i in &v {
    println!("{}", i);
}
```

[1] ошибка[E0502]: не получается позаимствовать переменную `v` как изменяемую, потому что она также заимствуется как неизменяемая

Мы также можем перебирать изменяемые ссылки на каждый элемент в изменяемом векторе для того, чтобы внести изменения во все элементы. Цикл `for` в листинге 8.9 прибавит по 50 в каждый элемент.

Листинг 8.9. Перебор изменяемых ссылок, указывающих на элементы в векторе

```
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}
```

Для того чтобы изменить значение, на которое ссылается изменяемая ссылка, приходится использовать оператор разыменования (*), чтобы получить значение в `i` перед тем, как использовать оператор `+=`. Подробнее об операторе разыменования мы поговорим в разделе «Следование по указателю к значению с помощью оператора разыменования» (с. 370).

Использование перечисления для хранения нескольких типов

В начале этой главы мы говорили, что векторы хранят значения только одного типа. Это бывает неудобно. Безусловно, существуют варианты использования, в которых требуется хранить список элементов разных типов. К счастью, варианты перечисления определяются под одним и тем же типом enum, поэтому, когда нам нужно сохранить элементы другого типа в векторе, мы можем определить и использовать перечисление.

Предположим, что мы хотим получить значения из строки электронной таблицы, в которой одни столбцы в строке таблицы содержат целые числа, другие — числа с плавающей точкой, а третьи — строковые значения. Мы можем определить перечисление, варианты которого будут содержать разные типы значений, и тогда все варианты перечисления будут считаться одним и тем же типом — типом enum. Затем мы можем создать вектор, который содержит это перечисление и поэтому, в конечном счете, включает разные типы. Мы продемонстрировали это в листинге 8.10.

Листинг 8.10. Определение перечисления для хранения значений разных типов в одном векторе

```
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}

let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("синий")),
    SpreadsheetCell::Float(10.12),
];
```

Компилятору необходимо знать, какие типы будут находиться в векторе во время компиляции, чтобы определить, сколько потребуется памяти в куче для хранения каждого элемента. Дополнительное преимущество заключается в том, что мы можем четко определить, какие типы разрешены в этом векторе. Если бы Rust разрешал вектору содержать любой тип, то был бы шанс, что один или несколько типов вызовут ошибки при операциях, выполняемых над элементами вектора. Использование перечисления плюс выражения `match` означает, что во время компиляции язык Rust будет обеспечивать обработку всех возможных вариантов, как описано в главе 6.

При написании программы, если вы не знаете исчерпывающий набор типов, которые программа получит во время работы для хранения в векторе, то техническое решение с использованием перечисления работать не будет. Вместо него вы можете применить типажный объект, который мы рассмотрим в главе 17.

Теперь, когда мы обсудили несколько наиболее распространенных способов использования векторов, обязательно ознакомьтесь с документацией API о всевозможных полезных методах, определенных для `Vec<T>` стандартной библиотекой. Например, в дополнение к методу `push` метод `pop` выталкивает из вектора и возвращает последний элемент. Давайте перейдем к следующему коллекционному типу — типу `String`!

Хранение текста в кодировке UTF-8 с помощью строк

Мы говорили о строках в главе 4, а теперь рассмотрим их подробнее. Новички, как правило, застревают на строках по трем причинам: из-за склонности Rust к выявлению возможных ошибок, из-за большей сложности структуры данных строк, чем считают многие программисты, и вследствие UTF-8. Комбинацию этих факторов бывает трудно воспринять, если до этого вы имели дело с другими языками программирования.

Целесообразно обсуждать строки в контексте коллекций, поскольку строки реализованы как коллекция байтов, плюс некоторые методы обеспечивают полезную функциональность, когда эти байты интерпретируются как текст. В этом разделе мы поговорим об операциях в типе `String`, которые есть у каждого коллекционного типа, такого как создание, обновление и чтение. Мы также обсудим отличия типа `String` от других коллекций, а именно как индексирование в `String` осложняется различиями в том, как люди и компьютеры интерпретируют данные типа `String`.

Что такое тип String?

Сначала мы определим, что подразумевается под термином строковый тип, `String`. В Rust имеется только один строковый тип в ядре языка, а именно строковый срез

str, обычно видимый в его заимствованной форме &str. В главе 4 мы говорили о строковых срезах, то есть ссылках на строковые данные в кодировке UTF-8, хранящиеся в другом месте. Строковые литералы, например, хранятся в двоичном коде программы и поэтому являются строковыми срезами.

Тип String, который предусмотрен стандартной библиотекой Rust, а не закодирован в языке, является наращиваемым, изменяемым, обладаемым строковым типом в кодировке UTF-8. Когда растиане говорят о «строках», они обычно имеют в виду тип String и тип строкового среза &str, а не только один из этих типов. Хотя этот раздел в основном посвящен типу String, оба типа широко используются в стандартной библиотеке Rust, и как String, так и строковые срезы кодируются в UTF-8.

Стандартная библиотека также включает в себя ряд других строковых типов, таких как OsString, OsStr, CString и CStr. Библиотечные упаковки могут предоставить еще больше возможностей для хранения строковых данных. Вы заметили, что все эти имена заканчиваются на String или Str? Они относятся к обладаемым и заимствованным вариантам, как и типы String и str, которые вы видели ранее. К примеру, эти строковые типы могут хранить текст в разных кодировках или быть представлены в памяти по-разному. В этой главе мы не будем обсуждать другие строковые типы. Для получения дополнительной информации о том, как их использовать и когда подходит каждый из них, обратитесь к соответствующей документации API.

Создание нового экземпляра типа String

Многие операции, доступные для типа Vec<T>, также доступны для String, начиная с функции new, которая помогает создать экземпляр типа String. Эта функция показана в листинге 8.11.

Листинг 8.11. Создание нового пустого экземпляра типа String

```
let mut s = String::new();
```

Эта строка кода создает новый пустой экземпляр типа String под названием s, в который затем можно загрузить данные. Часто у нас есть некоторые исходные данные, с которых мы хотим начать строковый экземпляр. Для этого мы используем метод to_string, который доступен для любого типа, реализующего типаж Display, как это делают строковые литералы. В листинге 8.12 приведены два примера.

Листинг 8.12. Использование метода to_string для создания экземпляра типа String из строкового литерала

```
let data = "начальное содержимое";

let s = data.to_string();

// этот метод также работает непосредственно на литерале:
let s = "начальное содержимое".to_string();
```

Этот код создает строку, содержащую начальное содержимое.

Для создания экземпляра типа `String` из строкового литерала мы также можем использовать функцию `String::from`. Код в листинге 8.13 эквивалентен коду из листинга 8.12, который использует `to_string`.

Листинг 8.13. Использование функции String::from для создания экземпляра типа String из строкового литерала

```
let s = String::from("начальное содержимое");
```

Поскольку строки используются для большого количества вещей, мы можем использовать много разных обобщенных API для строк, обеспечивая себя целым рядом вариантов. Некоторые из них могут показаться излишними, но все они имеют свое место! В данном случае `String::from` и `to_string` делают одно и то же, поэтому ваш выбор является вопросом стиля.

Помните, что строки имеют кодировку UTF-8, поэтому мы можем включить в них любые надлежаще кодированные данные, как показано в листинге 8.14.

Листинг 8.14. Хранение приветствий на разных языках в строках

```
let hello = String::from("السلام عليكم");
let hello = String::from("Dobrý den");
let hello = String::from("Hello");
let hello = String::from("שָׁלוֹם");
let hello = String::from("नमस्ते");
let hello = String::from("こんにちは");
let hello = String::from("안녕하세요");
let hello = String::from("你好");
let hello = String::from("Olá");
let hello = String::from("Здравствуйте");
let hello = String::from("Hola");
```

Все они являются допустимыми значениями типа `String`.

Обновление строки

Экземпляр типа `String` может увеличиваться в размере, а его содержимое может изменяться, как и содержимое экземпляра типа `Vec<T>`, если вы добавляете в него больше данных. Кроме того, можно удобно использовать оператор + или макрокоманду `format!` для конкатенации значений типа `String`.

Добавление в конец значения типа String с помощью push_str и push

Мы можем наращивать значение типа `String`, используя метод `push_str` для добавления строкового среза в конец, как показано в листинге 8.15.

Листинг 8.15. Добавление строкового среза в конец значения типа String с помощью метода push_str

```
let mut s = String::from("foo");
s.push_str("bar");
```

После этих двух строк `s` будет содержать `foobar`. Метод `push_str` берет строковый срез, потому что мы не обязательно хотим брать этот параметр во владение. Например, код в листинге 8.16 показывает, что было бы очень хорошо, если бы мы не смогли использовать `s2` после добавления его содержимого в конец `s1`.

Листинг 8.16. Использование строкового среза после добавления его содержимого в конец значения типа String

```
let mut s1 = String::from("foo");
let s2 = "bar";
s1.push_str(s2);
println!("s2 равна {}", s2);
```

Если бы метод `push_str` брал `s2` во владение, то мы бы не смогли вывести его значение в последней строке кода. Однако этот код работает так, как мы и ожидали!

Метод `push` берет один символ в качестве параметра и добавляет его в конец значения типа `String`. В листинге 8.17 показан код, который добавляет букву `l` в значение типа `String` с помощью метода `push`.

Листинг 8.17. Добавление одного символа в значение типа String с помощью push

```
let mut s = String::from("lo");
s.push('l');
```

В результате выполнения этого кода переменная `s` будет содержать `lol`.

Конкатенация с помощью оператора + или макрокоманды format!

У вас часто будет возникать потребность в объединении двух существующих строк. Один из способов предполагает использование оператора `+`, как показано в листинге 8.18.

Листинг 8.18. Использование оператора + для объединения двух значений типа String в новое значение типа String

```
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // примечание: s1 перенесено сюда и больше не может
                   // использоваться
```

В результате выполнения этого кода переменная `s3` будет содержать `Hello, world!`. Причина, почему переменная `s1` больше не является действительной после сложения и отчего мы использовали ссылку на `s2`, связана с сигнатурой метода, который вызывается при использовании оператора `+`. Оператор `+` использует метод `add`, сигнатура которого выглядит примерно так:

```
fn add(self, s: &str) -> String {
```

Это не совсем та сигнатура, которая имеется в стандартной библиотеке: в стандартной библиотеке метод `add` определяется с помощью обобщений. Здесь мы смо-

трим на сигнатуру `add`, в которой конкретные типы заменены на обобщения, что как раз и происходит, когда мы вызываем этот метод со значениями типа `String`. Мы обсудим обобщения в главе 10. Эта сигнатура дает нам подсказки, которые нужны, чтобы понять хитрости оператора `+`.

Во-первых, переменная `s2` содержит `&`, имея в виду, что в первую строку мы добавляем ссылку на вторую строку из-за параметра `s` в методе `add`: в значение типа `String` мы можем добавить только `&str`. Мы не можем сложить два значения типа `String`. Но постойте, типом `&s2` является `&String`, а не `&str`, как указано во втором параметре метода `add`. Тогда почему листинг 8.18 компилируется?

Причина, по которой мы можем использовать `&s2` в вызове метода `add`, заключается в том, что компилятор неявно приводит аргументный тип `&String` к типу `&str`. Когда мы вызываем метод `add`, Rust использует принудительное приведение типа посредством `deref`, которое здесь неявно превращает `&s2` в `&s2[..]`. Мы обсудим это приведение типов подробнее в главе 15. Поскольку метод `add` не берет параметр `s` во владение, переменная `s2` по-прежнему будет действительным экземпляром типа `String` после этой операции.

Во-вторых, по сигнатуре мы видим, что метод `add` берет параметр `self` во владение, потому что `self` не имеет `&`. Это означает, что `s1` в листинге 8.18 будет перемещен в вызов метода `add` и после этого больше не будет действителен. Таким образом, хотя инструкция `let s3 = s1 + &s2;` выглядит так, как будто она копирует обе строки и создает новую, она фактически берет `s1` во владение, добавляет в конец копию содержимого `s2`, а затем возвращает владение результата. Другими словами, внешне выглядит, будто она делает много копий, но на самом деле это не так. Ее реализация эффективнее, чем простое копирование.

Если нам нужно конкатенировать несколько строк, то результат работы с оператором `+` становится громоздким:

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = s1 + "-" + &s2 + "-" + &s3;
```

В этой точке переменная `s` будет равняться `tic-tac-toe`. Со всеми этими символами `+` и `"` становится трудно понять, что происходит. Для более сложного комбинирования строк мы можем применить макрокоманду `format!`:

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = format!("{}-{}-{}", s1, s2, s3);
```

Этот код также устанавливает переменную `s` равной `tic-tac-toe`. Макрокоманда `format!` работает так же, как и макрокоманда `println!`, но вместо печати резуль-

тата на экран она возвращает экземпляр типа `String` с его содержимым. Версия кода с использованием макрокоманды `format!` гораздо легче читается и не берет во владение ни один из своих параметров.

Индексирование в строках

Во многих других языках программирования доступ к отдельным символам в строке со ссылкой на них по индексу является допустимой и часто встречающейся операцией. Однако если вы попытаетесь обратиться к частям значения типа `String` с помощью синтаксиса индексирования в языке Rust, то получите ошибку. Рассмотрим недопустимый код в листинге 8.19.

Листинг 8.19. Попытка использовать синтаксис индексирования со значением типа String

```
let s1 = String::from("hello");
let h = s1[0];
```

Этот код приведет к следующей ошибке[1]:

```
error[E0277]: the trait bound `std::string::String: std::ops::Index<{integer}>`
is not satisfied
 -->
  |
3 |     let h = s1[0];
  |             ^^^^^ the type `std::string::String` cannot be indexed by `{integer}`
  |
  = help: the trait `std::ops::Index<{integer}>` is not implemented
                    for `std::string::String`
```

Указанная ошибка и примечание говорят сами за себя: строки Rust не поддерживают индексирование. Но почему? Для того чтобы ответить на этот вопрос, нам нужно узнать, как Rust хранит строки в памяти.

Внутреннее представление

Тип `String` представляет собой обертку для `Vec<u8>`. Давайте посмотрим на некоторые примеры строк из листинга 8.14, надлежаще кодированные в UTF-8. Прежде всего, на эту:

```
let len = String::from("Hola").len();
```

В данном случае переменная `len` будет равна 4, то есть вектор, хранящий приветствие `"Hola"`, имеет длину 4 байта. Каждая из этих букв занимает 1 байт при кодировке в UTF-8. Но как насчет следующей строки? (Обратите внимание, что

[1] ошибка[E0277]: граница типажа `std::string::String: std::ops::Index<{integer}>` не удовлетворена

эта строка начинается с заглавной кириллической буквы З, а не с арабской цифры 3.)

```
let len = String::from("Здравствуйте").len();
```

На вопрос о том, какова длина строки, вы можете ответить: 12. Однако ответ языка Rust будет 24: это число байтов, необходимое для кодирования приветствия `"Привет"` в UTF-8, потому что каждое скалярное значение Юникода в данной строке занимает 2 байта памяти. Следовательно, индекс в байты строки не всегда будет коррелировать с допустимым скалярным значением Юникода. Для демонстрации рассмотрим недопустимый код Rust:

```
let hello = "Здравствуйте";
let answer = &hello[0];
```

Каким должно быть значение переменной `answer`? Должно ли оно равняться `З`, первой букве? При кодировке в UTF-8 первый байт буквы `З` равен `208`, а второй — `151`, поэтому переменная `answer` должна равняться `208`, но `208` само по себе не является допустимым символом. Число `208`, скорее всего, не является тем, что пользователь хотел бы вернуть, если бы он запросил первую букву этой строки. Однако это единственные данные, которые есть у Rust в байтовом индексе 0. Пользователи вообще-то не хотят возвращать значение байта, даже если строка содержит только латинские буквы: если бы выражение `&"hello"[0]` было допустимым кодом, который возвращал бы значение байта, то он вернул бы `104`, а не `h`. Во избежание возвращения неожиданного значения и возникновения ошибок, которые, возможно, будут обнаружены не сразу, язык Rust вообще не компилирует этот код и предотвращает недоразумения на ранних стадиях процесса разработки.

Байты, скалярные значения и графемные кластеры! О боже!

Еще один момент, связанный с UTF-8, заключается в том, что с точки зрения Rust есть три релевантных способа рассматривать строки: как байты, как скалярные значения и как графемные кластеры (наиболее близкие к тому, что мы называем буквами).

Если мы взглянем на слово «नमस्ते» на языке хинди, написанное слоговым письмом деванагари, то оно хранится в виде вектора значений типа `u8`, который выглядит следующим образом:

```
[224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164, 224,
165, 135]
```

Это 18 байт, и именно так компьютеры хранят эти данные. Если мы посмотрим на них как на скалярные значения Юникод, то есть на то, что в языке Rust является типом `char`, то эти байты выглядят следующим образом:

```
['न', 'म', 'स', '्', 'त', 'े']
```

Здесь шесть значений `char`, но четвертое и шестое не являются буквами — это диакритические знаки, которые сами по себе не имеют смысла. Наконец, если мы по-

смотрим на них как на графемные кластеры, мы получим то, что человек назвал бы четырьмя буквами, составляющими слово на хинди:

```
["न", "म", "स्", "ते"]
```

Rust предоставляет разные способы интерпретации сырых строковых данных, хранящихся в компьютерах. Благодаря этому каждая программа может выбирать необходимую ей интерпретацию, независимо от того, на каком человеческом языке хранятся данные.

Последняя причина, по которой Rust не позволяет нам индексировать значение типа `String` для получения символа, заключается в том, что от операций индексирования ожидается, что они всегда будут занимать постоянное время (O(1)). Но гарантировать такую производительность со значением типа `String` невозможно, поскольку для того, чтобы выяснить число допустимых символов, компилятору пришлось бы проверить его содержимое с самого начала до индекса.

Нарезка строк

Зачастую индексирование в строке — плохая идея, поскольку неясно, каким должен быть тип возвращаемого значения операции индексирования строки: байтовым значением, символом, графемным кластером или строковым срезом. Поэтому язык Rust просит вас быть точнее, если вам действительно нужно использовать индексы для создания строковых срезов. Для того чтобы быть точнее при индексировании и указать, что вам нужен строковый срез, а не индексирование с помощью `[]` с одним единственным числом, вы можете использовать `[]` с интервалом, создав строковый срез, содержащий отдельные байты:

```
let hello = "Здравствуйте";

let s = &hello[0..4];
```

Здесь переменная `s` будет строковым срезом `&str`, который содержит первые 4 байта строки. Ранее мы упоминали, что каждый из этих символов имеет размер в 2 байта, а это значит, что переменная `s` будет содержать `Зд`.

Что бы произошло, если бы мы применили `&hello[0..1]`? Ответ: язык Rust поднял бы панику во время выполнения точно так же, как если бы в векторе произошло обращение к недопустимому индексу[1]:

```
thread 'main' panicked at 'byte index 1 is not a char boundary; it is inside
'З' (bytes 0..2) of `Здравствуйте`', src/libcore/str/mod.rs:2188:4
```

Использовать интервалы для создания строковых срезов следует с осторожностью, потому что это может привести к аварийному сбою программы.

---

[1] `поток 'main' вызвал панику в точке 'байтовый индекс 1 не является границей char; он находится внутри 'З' (байты 0..2) цепочки `Здравствуйте`'`

Методы перебора строк

К счастью, вы можете обращаться к элементам в строке другими способами.

Если вам требуется выполнять операции с отдельными скалярными значениями Юникода, то самый лучший способ сделать это — использовать метод `chars`. Вызов метода `chars` на `"नमस्ते"` выделяет и возвращает шесть значений типа `char`, и вы можете сделать перебор результата, чтобы получить доступ к каждому элементу:

```
for c in "नमस्ते".chars() {
    println!("{}", c);
}
```

Этот код выводит следующее:

```
न
म
स
◌्
त
े
```

Метод `bytes` возвращает каждый сырой байт, который, возможно, подойдет для вашего домена:

```
for b in "नमस्ते".bytes() {
    println!("{}", b);
}
```

Код выводит 18 байт, которые составляют это значение типа `String`:

```
224
164
// --пропуск--
165
135
```

Но обязательно помните, что допустимые скалярные значения Юникод могут состоять из более чем 1 байта.

Получение графемных кластеров из строк — сложная процедура, поэтому данная функциональность стандартной библиотекой не предусмотрена. Упаковки можно найти по адресу https://crates.io/, если это именно та функциональность, которая вам нужна.

Строки не так просты

Подводя итог, можно сказать, что строки имеют сложный характер. В разных языках программирования по-разному решается, как представить эту сложность программисту. Компилятор решил сделать правильную обработку данных типа `String` свойством по умолчанию для всех программ Rust, имея в виду, что про-

граммисты должны заблаговременно уделять больше внимания обработке данных UTF-8. Этот компромисс раскрывает большую сложность строк, чем, по-видимому, в других языках программирования. Но этот же компромисс помогает избежать необходимости обрабатывать ошибки, связанные с символами, не являющимися ASCII, на более поздних этапах жизненного цикла разработки.

Давайте переключимся на что-то более простое — на хеш-отображения!

Хранение ключей со связанными значениями в хеш-отображениях

Последней из распространенных коллекций будет хеш-отображение. Тип `HashMap<K, V>` хранит отображение ключей типа `K` в значения типа `V`. Он делает это с помощью хеширующей функции, которая обусловливает то, как он помещает эти ключи и значения в память. Многие языки программирования поддерживают этот тип структуры данных, но в них часто используются другие названия, такие как хеш, отображение, объект, хеш-таблица, словарь или ассоциативный массив, и это лишь несколько вариантов наименований.

Хеш-отображения полезны, когда вы хотите искать данные не с помощью индекса, как это можно делать с векторами, а с помощью ключа любого типа. Например, в игре вы можете отслеживать баллы каждой команды в хеш-отображении, в котором каждый ключ — это название команды, а значения — баллы каждой команды. При наличии названия команды вы можете узнать ее балл.

В этом разделе мы познакомимся с базовым API хеш-отображений, но много других полезных вещей скрывается в функциях, определенных стандартной библиотекой для `HashMap<K, V>`. Как всегда, для получения дополнительной информации обратитесь к документации стандартной библиотеки.

Создание нового хеш-отображения

Вы можете создать пустое хеш-отображение с помощью функции `new` и добавить элементы, используя функцию `insert`. В листинге 8.20 мы следим за баллами двух команд, названия которых «Синяя» и «Желтая». Синяя команда начинает с 10 баллов, а Желтая — с 50.

Листинг 8.20. Создание нового хеш-отображения и вставка нескольких ключей и значений

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Синяя"), 10);
scores.insert(String::from("Желтая"), 50);
```

Обратите внимание, что сначала нужно использовать тип `HashMap` из раздела коллекций стандартной библиотеки. Из трех часто встречающихся коллекций эта используется реже всего, поэтому она не входит в состав средств, включенных в прелюдию автоматически. Хеш-отображения также имеют меньшую поддержку со стороны стандартной библиотеки: например, для их проектирования нет встроенной макрокоманды.

Как и векторы, хеш-отображения хранят данные в куче. Этот экземпляр типа `HashMap` имеет ключи типа `String` и значения типа `i32`. Как и векторы, хеш-отображения однородны: у всех ключей должен быть один и тот же тип, и у всех значений тоже должен быть один и тот же тип.

Еще один способ построения хеш-отображения предусматривает использование метода `collect` в векторе кортежей, где каждый кортеж состоит из ключа и его значения. Метод `collect` собирает данные в несколько коллекционных типов, включая `HashMap`. Например, если бы у нас были названия команд и начальные баллы в двух отдельных векторах, то мы бы могли применить метод `zip` для создания вектора кортежей, где название «Синяя» связано с 10, и так далее. Затем можно было бы применить метод `collect`, чтобы превратить этот вектор кортежей в хеш-отображение, как показано в листинге 8.21.

Листинг 8.21. Создание хеш-отображения из списка команд и списка баллов

```
use std::collections::HashMap;

let teams = vec![String::from("Синяя"), String::from("Желтая")];
let initial_scores = vec![10, 50];

let scores: HashMap<_, _> = teams.iter().zip(initial_scores.iter()).collect();
```

Аннотация типа `HashMap<_, _>` здесь необходима, потому что метод `collect` может производить сборку в целый ряд разных структур данных, и язык Rust не знает, чего вы хотите, если ее не укажете. Однако для параметров типов ключей и значений мы используем подчеркивания, и язык Rust может логически вывести типы, которые содержит хеш-отображение, основываясь на типах данных в векторах.

Хеш-отображения и владение

Для типов, реализующих типаж `Copy`, таких как `i32`, значения копируются внутрь хеш-отображения. Для обладаемых значений, таких как `String`, значения будут перемещены, а хеш-отображение будет владельцем этих значений, как показано в листинге 8.22.

Листинг 8.22. Показывает, что ключи и значения находятся во владении хеш-отображения после их вставки

```
use std::collections::HashMap;

let field_name = String::from("Любимый цвет");
```

```
let field_value = String::from("Синий");

let mut map = HashMap::new();
map.insert(field_name, field_value);
// field_name и field_value в этом месте недопустимы, попробуйте
// их использовать и увидите, какая ошибка компилятора произойдет!
```

Мы не можем использовать переменные `field_name` и `field_value` после того, как они были перемещены в хеш-отображение вызовом метода `insert`.

Если мы вставим ссылки на значения в хеш-отображение, то значения не будут перемещены в хеш-отображение. Значения, на которые указывают ссылки, должны быть действительными по крайней мере до тех пор, пока действует хеш-отображение. Более подробно об этих трудностях мы поговорим в разделе «Проверка ссылок с помощью жизненных циклов».

Доступ к значениям в хеш-отображении

Мы можем получить значение из хеш-отображения, предоставив его ключ методу `get`, как показано в листинге 8.23.

Листинг 8.23. Доступ к баллу Синей команды, хранящемуся в хеш-отображении

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Синяя"), 10);
scores.insert(String::from("Желтая"), 50);

let team_name = String::from("Синяя");
let score = scores.get(&team_name);
```

Здесь у `score` будет значение, связанное с Синей командой, и результат будет `Some(&10)`. Результат обернут в `Some`, потому что метод `get` возвращает `Option<&V>`. Если для этого ключа значения в хеш-отображении нет, то метод `get` вернет `None`. Программа должна будет обрабатывать тип `Option` одним из способов, которые мы рассмотрели в главе 6.

Мы можем перебрать каждую пару ключ/значение хеш-отображения аналогично тому, как мы делаем с векторами, используя цикл `for`:

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Синяя"), 10);
scores.insert(String::from("Желтая"), 50);

for (key, value) in &scores {
    println!("{}: {}", key, value);
}
```

Этот код выводит каждую пару в произвольном порядке:

```
Желтая: 50
Синяя: 10
```

Обновление хеш-отображения

Хотя число ключей и значений может расти, с каждым ключом одновременно может быть связано только одно значение. Если вы хотите изменить данные в хеш-отображении, то вы должны решить, как обрабатывать случай, когда за ключом уже закреплено значение. Вы могли бы заменить старое значение новым, проигнорировав старое значение. Вы могли бы оставить старое значение и проигнорировать новое. Добавить новое значение можно было бы, только если у ключа еще нет значения. Или же вы могли бы объединить старое и новое значения. Давайте посмотрим, как выполнить каждый из этих вариантов.

Перезапись значения

Если мы вставим ключ и значение в хеш-отображение, а затем вставим тот же ключ с другим значением, то значение, связанное с этим ключом, будет заменено. Даже если код в листинге 8.24 вызывает метод `insert` дважды, хеш-отображение будет содержать только одну пару ключ/значение, потому что мы оба раза вставляем значение для ключа Синей команды.

Листинг 8.24. Замена сохраненного значения с конкретным ключом

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Синяя"), 10);
scores.insert(String::from("Синяя"), 25);

println!("{:?}", scores);
```

Этот код выводит `{"Синяя": 25}`. Исходное значение `10` было перезаписано.

Вставка значения, только если ключ не имеет значения

Часто принято выполнять проверку на наличие у ключа конкретного значения, и если его нет, то вставлять значение для ключа. Хеш-отображения имеют специальный API для этого под названием `entry`. Метод `entry` берет в качестве параметра ключ, который вы хотите проверить, и возвращает перечисление `Entry`, представляющее значение, которого может и не существовать. Допустим, мы хотим проверить, что значение связано с ключом Желтой команды. Если это не так, то мы хотим вставить значение `50`, и то же самое для Синей команды. При использовании API `entry` код выглядит так, как показано в листинге 8.25.

Листинг 8.25. Использование метода entry для вставки, только если ключ еще не имеет значения

```
use std::collections::HashMap;

let mut scores = HashMap::new();
scores.insert(String::from("Синяя"), 10);

scores.entry(String::from("Желтая")).or_insert(50);
scores.entry(String::from("Синяя")).or_insert(50);

println!("{:?}", scores);
```

Метод `or_insert` для перечисления `Entry` определен для возвращения изменяемой ссылки на значение соответствующего ключа `Entry`, если этот ключ существует. Если же ключа нет, то он вставляет параметр в качестве нового значения для этого ключа и возвращает изменяемую ссылку на новое значение. Этот прием гораздо яснее, чем самостоятельное написание алгоритма и, вдобавок, более слаженно взаимодействует с контролером заимствования.

При выполнении кода в листинге 8.25 будет выведено

```
{"Желтая": 50, "Синяя": 10}
```

При первом вызове метода `entry` будет вставлен ключ для Желтой команды со значением 50, потому что у Желтой команды еще нет значения. Второй вызов метода `entry` не изменит хеш-отображение, потому что у Синей команды уже есть значение 10.

Обновление значения на основе старого значения

Еще один распространенный вариант использования хеш-отображений состоит в поиске значения ключа и последующем его обновлении на основе старого значения. Например, в листинге 8.26 показан код, который подсчитывает число вхождений каждого слова в текст. Мы используем хеш-отображение со словами в качестве ключей и увеличиваем значение, отслеживая число появлений этого слова. Если мы встречаем слово впервые, то сначала вставляем значение `0`.

Листинг 8.26. Подсчет числа вхождений слов с помощью хеш-отображения, которое хранит слова и количества

```
use std::collections::HashMap;

let text = "здравствуй мир, чудесный мир";

let mut map = HashMap::new();

for word in text.split_whitespace() {
    let count = map.entry(word).or_insert(0);
    *count += 1;
}

println!("{:?}", map);
```

Этот код выводит

```
{"мир": 2, "здравствуй": 1, "чудесный": 1}
```

Метод `or_insert` фактически возвращает изменяемую ссылку (`&mut V`) на значение этого ключа. Здесь мы храним эту изменяемую ссылку в переменной `count`, поэтому для того, чтобы передать это значение, мы сначала должны пройти по ссылке `count` к этому значению, используя оператор разыменования (`*`). Изменяемая ссылка выходит из области видимости в конце цикла `for`, поэтому все эти изменения безопасны и разрешены правилами заимствования.

Хеширующие функции

По умолчанию тип `HashMap` использует криптографически сильную хеширующую функцию, которая обеспечивает устойчивость к DoS-атакам. Это не самый быстрый алгоритм хеширования из имеющихся, но компромисс ради большей безопасности, который приходит с падением производительности, стоит того. Если вы профилируете код и обнаруживаете, что принятая по умолчанию хеш-функция является слишком медленной для ваших целей, то вы можете переключиться на другую функцию, указав другой хешер. Хешер (hasher) — это тип, которым реализуется типаж `BuildHasher`. О типажах и способах их реализации мы поговорим в главе 10. Вам не обязательно реализовывать собственный хешер с нуля; https://crates.io/ содержит библиотеки, распространяемые другими пользователями языка Rust, которые предоставляют хешеры, реализующие многие часто встречающиеся алгоритмы хеширования.

Итоги

Векторы, строки и хеш-отображения обеспечат большой объем функциональности, необходимый в программах, когда вам нужно хранить и изменять данные, а также обращаться к ним. Ниже приведено несколько упражнений — теперь вы достаточно подготовлены, чтобы их решить:

- При наличии списка целых чисел использовать вектор и вернуть среднее (среднее арифметическое значение), медиану (после сортировки взять значение в серединной позиции) и моду (значение, которое встречается чаще всего, — здесь будет полезно хеш-отображение) списка.
- Конвертировать строки в «поросячью латынь». Первая согласная каждого слова перемещается в конец слова, куда добавляется «ay», в результате чего «first» становится «irst-fay». В конец слов, которые начинаются с гласной, добавляется «hay» («apple» становится «apple-hay»). Учитывайте сведения о кодировке UTF-8!
- Используя хеш-отображения и векторы, создать текстовый интерфейс, позволяющий пользователю добавлять имена сотрудников в отдел компании. На-

пример, «Добавить Салли в инженерный отдел» или «Добавить Амира в коммерческий отдел». Затем пользователь извлекает список всех сотрудников отдела или всех сотрудников компании по отделам, отсортированными в алфавитном порядке.

Документация API стандартной библиотеки описывает методы, имеющиеся для векторов, строк и хеш-отображений, которые будут полезны для этих упражнений.

Мы переходим к более сложным программам, в которых операции могут завершаться безуспешно, поэтому наступает идеальное время, чтобы обсудить вопрос обработки ошибок. Мы сделаем это далее!

9 Обработка ошибок

Надежность Rust распространяется и на обработку ошибок. Ошибки — это правда жизни в ПО, поэтому в Rust имеется ряд средств для обработки ситуаций, в которых что-то идет не так. Во многих случаях Rust требует от вас признать возможность ошибки и предпринять некие действия перед компиляцией кода. Это требование делает программу более надежной и обязывает вас устранять ошибки надлежащим образом, прежде чем развернуть код в производство.

Rust группирует ошибки в две основные категории: устранимые и неустранимые. В случае устранимых ошибок, например «Файл не найден», целесообразно сообщить о проблеме пользователю и повторить операцию. Неустранимые ошибки всегда являются симптомами дефектов, таких как попытка доступа к позиции за пределами массива.

Большинство языков не различают эти виды ошибок и обрабатывают их одинаково, используя такие механизмы, как исключения. В Rust нет исключений. Вместо этого в нем имеется тип `Result<T, E>` для устранимых ошибок и макрокоманда `panic!`, которая останавливает исполнение, когда программа обнаруживает неустранимую ошибку. В этой главе сначала речь пойдет о вызове макрокоманды `panic!`, а после мы расскажем о возвращении значений `Result<T, E>`. Кроме того, мы рассмотрим, что следует делать при возникновении ошибки: пытаться восстановить работу после ошибки или остановить исполнение.

Неустранимые ошибки и макрокоманда panic!

Иногда в коде происходит что-то плохое, и вы ничего не можете с этим поделать. Для таких случаев в Rust есть макрокоманда `panic!`. При выполнении макрокоманды `panic!` программа будет печатать сообщение об ошибке, размотает и очистит стек, а затем завершит работу. Чаще всего это происходит, когда обнаружен какой-то дефект и программисту не ясно, как с ним справиться.

РАЗМАТЫВАНИЕ СТЕКА, ИЛИ ПРЕРЫВАНИЕ РАБОТЫ В ОТВЕТ НА ПАНИКУ

По умолчанию, когда возникает паника, программа начинает разматываться, то есть язык Rust отступает назад в стеке и очищает данные в каждой функции, с которой он сталкивается. Но этот процесс сопряжен с большой работой. Альтернативой является немедленное прерывание, которое завершает программу без очистки. Память, которую программа использовала, затем должна быть очищена операционной системой. Если в проекте нужно сделать результирующий двоичный файл как можно меньше, то при возникновении паники вы можете переключиться с разматывания на прерывание, добавив `panic = 'abort'` в соответствующие разделы `[profile]` в файле Cargo.toml. Например, если вы хотите прервать работу при возникновении паники в релизном режиме, добавьте:

```
[profile.release]
panic = 'abort'
```

Давайте попробуем вызвать `panic!` в простой программе:

src/main.rs

```
fn main() {
    panic!("полное фиаско");
}
```

Когда вы выполните указанную программу, то увидите что-то вроде этого[1]:

```
$ cargo run
    Compiling panic v0.1.0 (file:///projects/panic)
     Finished dev [unoptimized + debuginfo] target(s) in 0.25 secs
      Running 'target/debug/panic'
thread 'main' panicked at 'полное фиаско', src/main.rs:2:5
note: Run with 'RUST_BACKTRACE=1' for a backtrace.
```

Вызов макрокоманды `panic!` приводит к появлению сообщения об ошибке, содержащегося в последних двух строках кода. Первая строка кода показывает сообщение о панике и место в исходном коде, где произошла паника: src/main.rs:2:5 указывает на то, что это вторая строка, пятый символ файла src/main.rs.

В данном случае указанная строка является частью кода, и если мы перейдем к этой строке, то увидим вызов макрокоманды `panic!`. В других случаях вызов макрокоманды `panic!` может быть в коде, который вызывается нашим кодом, а имя файла и номер строки в сообщении об ошибке будут чужим кодом, где макрокоманда `panic!` была вызвана, а не той строкой нашего кода, которая в конечном итоге привела к вызову макрокоманды. Мы можем использовать обратную трассировку функций, откуда пришел вызов макрокоманды `panic!`, чтобы выяснить, какая часть кода служит причиной этой проблемы. Далее мы подробнее обсудим понятие обратной трассировки.

---

[1] `поток 'main' вызвал панику при 'полном фиаско', src/main.rs:2:5`
`примечание: выполните с опцией 'RUST_BACKTRACE=1' для получения обратной трассировки`

Использование обратной трассировки при вызове panic!

Давайте взглянем еще на один пример и посмотрим, что происходит, когда вызов макрокоманды `panic!` поступает не из нашего кода, который вызывает макрокоманду напрямую, а из библиотеки в связи с дефектом в нашем коде. В листинге 9.1 есть код, который пытается обратиться к элементу по индексу в векторе.

Листинг 9.1. Попытка обратиться к элементу за пределами вектора, что приведет к вызову макрокоманды panic!

src/main.rs

```
fn main() {
    let v = vec![1, 2, 3];

    v[99];
}
```

Здесь мы пытаемся обратиться к сотому элементу вектора (который находится в индексе 99, потому что индексация начинается с нуля), но в нем только три элемента. В этой ситуации возникнет состояние паники. Использование `[]` предположительно должно возвращать элемент, но, если передается недопустимый индекс, то здесь нет ни одного правильного элемента, который Rust мог бы вернуть.

Другие языки, такие как C, в этой ситуации попытаются дать вам именно то, что вы просили, даже если это не то, что вы хотите: вы получите все, что находится в том месте в памяти, которое соответствовало бы этому элементу в векторе, даже если память не принадлежит вектору. Это называется переполнением буфера и приводит к уязвимостям в системе безопасности, если злоумышленнику удается манипулировать индексом так, чтобы считывать данные, хранящиеся после массива, которые запрещено читать.

В целях защиты программы от такого рода уязвимостей, если вы попытаетесь прочитать элемент в несуществующем индексе, то язык Rust остановит исполнение и откажется продолжать работу. Давайте попробуем и посмотрим:

```
$ cargo run
    Compiling panic v0.1.0 (file:///projects/panic)
     Finished dev [unoptimized + debuginfo] target(s) in 0.27 secs
      Running 'target/debug/panic'
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is
99', libcore/slice/mod.rs:2448:10
note: Run with 'RUST_BACKTRACE=1' for a backtrace.
```

Приведенная выше ошибка указывает на файл, который мы не писали, — libcore/slice/mod.rs. Он представляет собой реализацию среза `slice` в исходном коде Rust. Код, который запускается на исполнение, когда мы используем `[]` в векторе `v`, располагается в файле libcore/slice/mod.rs, и именно там на самом деле происходит `panic!`.

Следующая далее строка примечания говорит о том, что мы можем установить переменную среды `RUST_BACKTRACE`, чтобы получить обратную трассировку того,

что вызвало ошибку. Обратная трассировка — это список всех функций, которые были вызваны, чтобы добраться до этой точки. Обратные трассировки в Rust работают так же, как и в других языках. Ключ к чтению обратной трассировки — начать сверху и читать до тех пор, пока вы не увидите файлы, которые вы писали. Именно в этом месте возникла проблема. Строки над строками с упоминанием ваших файлов показывают код, который вызвал ваш код; строки ниже показывают код, который был вызван вашим кодом. Эти строки могут включать основной код Rust, код стандартной библиотеки или используемые упаковки. Давайте попробуем получить обратную трассировку, установив переменную среды `RUST_BACKTRACE`, равной любому значению, кроме 0. Листинг 9.2 показывает результат, аналогичный тому, что вы увидите.

Листинг 9.2. Обратная трассировка, сгенерированная вызовом макрокоманды panic!, появляется при установке переменной среды RUST_BACKTRACE

```
$ RUST_BACKTRACE=1 cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is
99', libcore/
slice/mod.rs:2448:10
stack backtrace:
   0: std::sys::unix::backtrace::tracing::imp::unwind_backtrace
             at libstd/sys/unix/backtrace/tracing/gcc_s.rs:49
   1: std::sys_common::backtrace::print
             at libstd/sys_common/backtrace.rs:71
             at libstd/sys_common/backtrace.rs:59
   2: std::panicking::default_hook::{{closure}}
             at libstd/panicking.rs:211
   3: std::panicking::default_hook
             at libstd/panicking.rs:227
   4: <std::panicking::begin_panic::PanicPayload<A> as core::panic::BoxMeUp>::get
             at libstd/panicking.rs:476
   5: std::panicking::continue_panic_fmt
             at libstd/panicking.rs:390
   6: std::panicking::try::do_call
             at libstd/panicking.rs:325
   7: core::ptr::drop_in_place
             at libcore/panicking.rs:77
   8: core::ptr::drop_in_place
             at libcore/panicking.rs:59
   9: <usize as core::slice::SliceIndex<[T]>>::index
             at libcore/slice/mod.rs:2448
  10: core::slice::<impl core::ops::index::Index<I> for [T]>::index
             at libcore/slice/mod.rs:2316
  11: <alloc::vec::Vec<T> as core::ops::index::Index<I>>::index
             at liballoc/vec.rs:1653
  12: panic::main
             at src/main.rs:4
  13: std::rt::lang_start::{{closure}}
             at libstd/rt.rs:74
  14: std::panicking::try::do_call
```

```
           at libstd/rt.rs:59
           at libstd/panicking.rs:310
  15: macho_symbol_search
           at libpanic_unwind/lib.rs:102
  16: std::alloc::default_alloc_error_hook
           at libstd/panicking.rs:289
           at libstd/panic.rs:392
           at libstd/rt.rs:58
  17: std::rt::lang_start
           at libstd/rt.rs:74
  18: panic::main
```

Уж очень длинный результат! Точный результат, который вы увидите, возможно, будет отличаться в зависимости от вашей операционной системы и версии Rust. Для получения обратных трассировок с этой информацией необходимо использовать отладочные символы. Отладочные символы задействуются по умолчанию при использовании команды `cargo build` или `cargo run` без флага `--release`, как у нас здесь.

Строка 12 обратной трассировки в листинге 9.2 указывает на строку кода в проекте, которая служит причиной проблемы, — строку 4 файла src/main.rs. Если мы не хотим, чтобы программа поднимала панику, то место, на которое указывает первая строка, упоминающая написанный нами файл, является местом, откуда мы должны начать расследование. В листинге 9.1, где мы намеренно написали код, который будет вызывать панику, чтобы продемонстрировать использование обратных трассировок, способ устранить панику состоит в том, чтобы не запрашивать элемент с индексом 99 из вектора, который содержит только три элемента. Когда код будет поднимать панику в будущем, вам нужно будет выяснить, какие действия и с какими значениями выполняет код, которые в итоге становятся причиной паники, и что код должен вместо этого делать.

Мы еще вернемся к макрокоманде `panic!` и к вопросу о том, когда следует, а когда не следует использовать макрокоманду `panic!` для обработки условий ошибки, в разделе «Паниковать! Или не паниковать». Далее мы посмотрим, как восстанавливать работу после ошибки с помощью `Result`.

Устранимые ошибки с помощью Result

Большинство ошибок не настолько серьезны, чтобы требовать полной остановки программы. Порой, когда функция не срабатывает, это происходит по причине, которую можно легко объяснить и на которую можно отреагировать. Например, если вы попытаетесь открыть файл, а операция завершится безуспешно, так как файл не существует, то вместо окончания процесса вы можете создать файл.

Напомним из раздела «Обработка потенциального сбоя с помощью типа Result» (с. 45), что `Result` определяется как перечисление, имеющее два варианта — `Ok` и `Err`, — следующим образом:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

`T` и `E` — это параметры обобщенных типов. Мы обсудим обобщения подробнее в главе 10. Сейчас вам нужно знать только то, что `T` представляет тип значения, которое будет возвращено внутри варианта `Ok` в случае успеха, а `E` — это тип ошибки, которая будет возвращена внутри варианта `Err` в случае провала. Поскольку `Result` имеет эти параметры обобщенных типов, мы можем использовать тип `Result` и функции, определенные для него стандартной библиотекой, в различных ситуациях, когда значение успеха и значение ошибки, которые мы хотим вернуть, могут отличаться.

Давайте вызовем функцию, которая возвращает значение `Result`, потому что функция может не сработать. В листинге 9.3 мы попытаемся открыть файл.

Листинг 9.3. Открытие файла

src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");
}
```

Как узнать, что функция `File::open` возвращает `Result`? Мы могли бы посмотреть документацию API стандартной библиотеки либо спросить компилятор! Если мы дадим переменной `f` аннотацию типа, о котором мы знаем, что он не является типом, возвращаемым из указанной функции, а затем попытаемся скомпилировать код, то компилятор сообщит нам, что типы не совпадают. Затем сообщение об ошибке скажет о том, каким является тип `f`. Давайте попробуем! Мы знаем, что тип, возвращаемый из `File::open`, не относится к типу `u32`, поэтому изменим инструкцию `let f` на другую:

```
let f: u32 = File::open("hello.txt");
```

Попытка скомпилировать код теперь дает следующий результат:

```
error[E0308]: mismatched types
 --> src/main.rs:4:18
  |
4 |     let f: u32 = File::open("hello.txt");
  |                  ^^^^^^^^^^^^^^^^^^^^^^^ expected u32, found enum
`std::result::Result`
  |
  = note: expected type `u32`
             found type `std::result::Result<std::fs::File, std::io::Error>`
```

Речь о том, что типом, возвращаемым функцией `File::open`, является `Result<T, E>`. Обобщенный параметр `T` был заполнен здесь типом успешного значения,

`std::fs::File`, то есть файловым дескриптором. Типом параметра `E`, используемого в значении ошибки, является `std::io::Error`.

Этот возвращаемый тип означает, что вызов `File::open` может завершиться успешно и вернуть файловый дескриптор, который мы прочитаем или в который запишем. Кроме того, вызов функции может не сработать. Например, возможно, файл не существует или у нас не будет разрешения на доступ к файлу. Функция `File::open` должна иметь возможность сообщить об успешности или неуспешности выполнения и в то же время дать нам либо файловый дескриптор, либо информацию об ошибке. Именно эту информацию передает перечисление типа `Result`.

В случае успешного выполнения `File::open` значение переменной `f` будет экземпляром `Ok`, содержащим файловый дескриптор. В случае, когда выполнение оказывается безуспешным, значение в `f` будет являться экземпляром `Err`, содержащим дополнительную информацию о типе произошедшей ошибки.

Нужно добавить код в листинг 9.3, он будет предпринимать разные действия в зависимости от значения, которое возвращает `File::open`. В листинге 9.4 показан один из способов обработки перечисления `Result` с помощью базового инструмента — выражения `match`, которое мы обсуждали в главе 6.

Листинг 9.4. Использование выражения match для обработки вариантов перечисления Result, которые могут быть возвращены

src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => {
            panic!("Проблема с открытием файла: {:?}", error)
        },
    };
}
```

Обратите внимание, что, как и перечисление `Option`, перечисление `Result` и его варианты были введены в область видимости прелюдией, поэтому не нужно указывать `Result::` перед вариантами `Ok` и `Err` в рукавах выражения `match`.

Здесь мы говорим языку Rust о том, что, когда результат будет равным `Ok`, надо вернуть внутреннее значение `file` из варианта `Ok`, а затем мы присвоим это значение файлового дескриптора переменной `f`. После выражения `match` можно использовать указанный файловый дескриптор для чтения или записи.

Другой рукав выражения `match` улаживает случай, когда мы получаем значение `Err` из `File::open`. В этом примере мы решили вызвать макрокоманду `panic!`.

Если в текущем каталоге нет файла с именем hello.txt, а мы выполним этот код, то из макрокоманды `panic!` будут получены такие денные[1]:

```
thread 'main' panicked at 'Problem opening the file: Error { repr: Os { code:
2, message: "No such file or directory" } }', src/main.rs:9:12
```

Как обычно, этот результат сообщает, что конкретно пошло не так.

Применение выражения match с разными ошибками

Код в листинге 9.4 поднимет `panic!` независимо от того, почему функция `File::open` не сработала. Вместо этого мы хотим предпринимать разные действия по разным причинам сбоя. Если функция `File::open` не сработала, потому что файл не существует, то требуется создать файл и вернуть дескриптор на новый файл. Если же функция `File::open` не сработала по какой-либо другой причине — например, потому, что у нас не было разрешения на открытие файла, — нужно, чтобы код по-прежнему поднимал `panic!` точно так же, как это было сделано в листинге 9.4. Посмотрите на листинг 9.5, в который добавлено внутреннее выражение `match`.

Листинг 9.5. Обработка разных видов ошибок разными способами

src/main.rs

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Проблема с созданием файла: {:?}", e),
            },
            other_error => panic!("Проблема с открытием файла: {:?}", other_error),
        },
    };
}
```

Тип значения, которое функция `File::open` возвращает внутри варианта `Err`, равен `io::Error`, то есть структуре, предусмотренной стандартной библиотекой. Эта структура имеет метод `kind`, который мы можем вызвать, чтобы получить значение `io::ErrorKind`. Перечисление `io::ErrorKind` предусмотрено стандартной библиотекой, и в нем есть варианты, представляющие разные виды ошибок, которые

[1] `поток 'main' поднял панику в точке 'Проблема с открытием файла: Error { repr: Os { code: 2, message: "Нет такого файла или каталога" } }'`

возникают в результате операции `io`. Вариантом, который мы хотим использовать, является `ErrorKind::NotFound`. Он указывает на то, что файл, который мы пытаемся открыть, пока еще не существует. Поэтому мы применяем `match` для `f`, но у нас также есть внутреннее выражение `match` для `error.kind()`.

Во внутреннем выражении `match` мы хотим проверить условие о том, является ли значение, возвращаемое из `error.kind()`, вариантом `NotFound` перечисления `ErrorKind`. Если это так, то мы попытаемся создать файл с помощью функции `File::create`.

Однако, поскольку `File::create` тоже может не сработать, то внутреннему выражению `match` нужен второй рукав. Когда создать файл не получается, выводится другое сообщение об ошибке. Второй рукав внешнего выражения `match` остается неизменным, поэтому программа поднимает панику при любой ошибке, помимо ошибки отсутствующего файла.

Уж очень много выражений `match`! Выражение `match` очень полезно, но и во многом примитивно. В главе 13 вы узнаете о замыканиях. В типе `Result<T, E>` имеется много методов, которые принимают замыкание и реализуются с помощью выражений `match`. Использование этих методов сделает код лаконичнее. Более опытный растианин, возможно, написал бы вместо листинга 9.5 такой код:

src/main.rs

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt").unwrap_or_else(|error| {
        if error.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|error| {
                panic!("Проблема с созданием файла: {:?}", error);
            })
        } else {
            panic!("Проблема с открытием файла: {:?}", error);
        }
    });
}
```

Несмотря на то что этот код ведет себя так же, как и в листинге 9.5, он не содержит никаких выражений `match` и более понятен для чтения. Вернитесь к этому примеру после изучения главы 13 и найдите метод `unwrap_or_else` в документации стандартной библиотеки. Есть еще много методов, которые могут очищать огромные вложенные выражения `match`, когда вы работаете над ошибками.

Краткие формы для паники в случае ошибки: unwrap и expect

Выражение `match` работает неплохо, но оно бывает несколько многословным и не всегда хорошо передает намерение. Для типа `Result<T, E>` определено много вспо-

могательных методов для выполнения разных видов работ. Один из этих методов, именуемый `unwrap`, представляет собой укороченный метод, который реализован точно так же, как выражение `match`, описанное в листинге 9.4. Если значение `Result` является вариантом `Ok`, то метод `unwrap` вернет значение внутрь `Ok`. Если же `Result` равно варианту `Err`, то метод `unwrap` вызовет макрокоманду `panic!`. Вот пример метода `unwrap` в действии:

src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").unwrap();
}
```

Если мы выполним этот код без файла hello.txt, то увидим сообщение об ошибке из макрокоманды `panic!`, которую метод `unwrap` вызывает[1]:

```
thread 'main' panicked at 'called 'Result::unwrap()' on an 'Err' value: Error
{ repr: Os { code: 2, message: "No such file or directory" } }', /src/libcore/
result.rs:906:4
```

Еще один метод, `expect`, похожий на `unwrap`, позволяет помимо этого выбирать сообщение об ошибке макрокоманды `panic!`. Использование метода `expect` вместо `unwrap` и корректные сообщения об ошибках лучше передают ваши намерения и облегчают отслеживание источника паники. Синтаксис метода `expect` выглядит следующим образом:

src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").expect("Не получилось открыть hello.txt");
}
```

Мы используем метод `expect` таким же образом, как и метод `unwrap`, — чтобы вернуть файловый дескриптор или вызвать макрокоманду `panic!`. Сообщение об ошибке, используемое методом `expect` в вызове макрокоманды `panic!`, будет параметром, который мы передаем методу `expect`, а не стандартным сообщением макрокоманды `panic!`, которое используется методом `unwrap`. Вот как оно выглядит[2]:

```
thread 'main' panicked at 'Не получилось открыть hello.txt: Error { repr: Os
{ code: 2, message: "No such file or directory" } }',
/src/libcore/result.rs:906:4
```

---

[1] `поток 'main' вызвал панику в точке 'Вызвана функция 'Result::unwrap()' для значения 'Err': Error { repr: Os { code: 2, message: "Нет такого файла или каталога" } }'`

[2] `поток 'main' вызвал панику в точке 'Не получилось открыть hello.txt: Error { repr: Os { code: 2, message: "Нет такого файла или каталога" } }'`

Поскольку это сообщение об ошибке начинается с указанного нами текста — `Не получилось открыть hello.txt`, — то становится легче найти место, откуда приходит это сообщение об ошибке. Если мы используем метод `unwrap` в нескольких местах, то потребуется больше времени, чтобы выяснить, какой именно метод `unwrap` служит причиной паники, потому что все вызовы метода `unwrap`, которые поднимают панику, печатают одно и то же сообщение.

Распространение ошибок

Когда вы пишете функцию, реализация которой вызывает что-то, что, возможно, не сработает, вместо того, чтобы обрабатывать ошибку внутри этой функции, вы можете вернуть ошибку в вызывающий код, и уже он будет решать, что делать. Это называется «распространение» ошибки, оно дает вызывающему коду больше контроля над тем, где, возможно, больше информации или алгоритмов, которые диктуют то, как следует обрабатывать ошибку, а не то, что у вас есть в контексте кода.

Например, листинг 9.6 показывает функцию, которая читает имя пользователя из файла. Если файл не существует или его не получается прочитать, функция вернет эти ошибки в код, вызвавший данную функцию.

Листинг 9.6. Функция, которая возвращает ошибки в вызывающий код с помощью выражения match

src/main.rs

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error>❶ {
 ❷ let f = File::open("hello.txt");

 ❸ let mut f = match f {
       Ok(file) => file,
       Err(e) => return Err(e),
   };

 ❹ let mut s = String::new();

 ❺ match f.read_to_string(&mut s)❻ {
       Ok(_) => Ok(s)❼,
       Err(e) => Err(e)❽,
   }
}
```

Указанную функцию можно записать гораздо короче, но мы начнем с того, что проделаем большую часть работы вручную, чтобы познакомиться с обработкой ошибок. В конце мы покажем более короткий способ. Давайте сначала посмотрим на возвращаемый из функции тип: `Result<String, io::Error>` ❶. Он означает, что

функция возвращает значение типа `Result<T, E>`, где обобщенный параметр `T` был заполнен конкретным типом `String`, а обобщенный тип `E` был заполнен конкретным типом `io::Error`. Если эта функция работает успешно без каких-либо проблем, то код, вызывающий эту функцию, получит значение `Ok`, содержащее значение экземпляра `String` — имя пользователя, которое эта функция прочитала из файла ❼. Если же эта функция столкнется с какими-либо проблемами, то код, вызывающий ее, получит значение `Err`, содержащее экземпляр `io::Error` с дополнительной информацией о характере этих проблем. В качестве типа, возвращаемого из этой функции, мы выбрали `io::Error`, поскольку он оказался типом значения ошибки, возвращаемой из обеих операций, которые могут завершиться сбоем: функции `File::open` и метода `read_to_string`, вызываемых нами в теле этой функции.

Тело функции начинается с вызова функции `File::open` ❷. Затем мы обрабатываем значение `Result`, возвращаемое выражением `match` ❸, аналогичного выражению `match` в листинге 9.4. Только вместо вызова `panic!` в случае `Err` мы возвращаемся из этой функции досрочно и передаем значение ошибки из функции `File::open` обратно в вызывающий код как значение ошибки этой функции ❹. Если функция `File::open` срабатывает успешно, то мы сохраняем файловый дескриптор в переменной `f` и продолжаем работу.

Затем мы создаем новый экземпляр типа `String` в переменной `s` ❺ и вызываем метод `read_to_string` для файлового дескриптора в `f`, чтобы прочитать содержимое файла в переменную `s` ❻. Метод `read_to_string` также возвращает `Result`, так как он может завершиться сбоем, даже если функция `File::open` сработала успешно. Поэтому нам нужно еще одно выражение `match`, чтобы обработать этот экземпляр типа `Result`: если метод `read_to_string` срабатывает, то функция сработала, и мы возвращаем имя пользователя из файла, которое теперь находится в переменной `s`, обернутое в `Ok` ❼. Если метод `read_to_string` не срабатывает, то мы возвращаем значение ошибки таким же образом, как вернули значение ошибки в выражении `match`, которое обрабатывало значение, возвращаемое из функции `File::open` ❽. Однако нам не нужно указывать инструкцию `return`, потому что это выражение является последним в функции.

Код, который вызывает этот код, затем займется обработкой, получив либо значение `Ok`, содержащее имя пользователя, либо значение `Err`, содержащее `io::Error`. Мы не знаем, что именно вызывающий код сделает с этими значениями. Если вызывающий код получит значение `Err`, то он может либо вызвать `panic!` и аварийно завершить программу, либо воспользоваться именем пользователя по умолчанию, либо найти имя пользователя где-то еще помимо файла. У нас нет достаточной информации о том, что именно вызывающий код попытается сделать по факту, поэтому мы распространяем всю информацию об успехе или ошибке, чтобы обработать ее надлежащим образом.

Этот паттерн распространения ошибки является в Rust настолько частым, что для его упрощения в языке предоставлен оператор `?`.

Краткая форма распространения ошибок – оператор ?

Листинг 9.7 показывает реализацию функции `read_username_from_file` («прочитать имя пользователя из файла»), которая имеет ту же функциональность, что и в листинге 9.6, но данная реализация использует оператор `?`.

Листинг 9.7. Функция, которая возвращает ошибки в вызывающий код с помощью оператора ?

src/main.rs

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

Если поместить оператор `?` после значения типа `Result`, то он будет работать почти так же, как выражения `match`, которые мы определяли для обработки значений `Result` в листинге 9.6. Если значение типа `Result` равно `Ok`, то из этого выражения будет возвращено значение внутри `Ok` и программа продолжит работу. Если же значение равно `Err`, то из всей функции будет возвращено `Err`, как если бы мы использовали ключевое слово `return`, а значение ошибки распространится в вызывающий код.

Существует разница в том, что делает выражение `match` из листинга 9.6 и оператор `?`: значения ошибок, при которых вызывается оператор `?`, проходящий через функцию `from`. Эта функция определена в типаже `From` в стандартной библиотеке и используется для конвертирования ошибок из одного типа в другой. Когда оператор `?` вызывает функцию `from`, полученный тип ошибки конвертируется в ошибку, определенную в типе, возвращаемом текущей функцией. Это полезно, когда функция возвращает один тип ошибки, представляя таким образом все варианты сбоя функции, даже если ее части не срабатывают по разным причинам. Пока каждый тип ошибки реализует функцию `from`, определяя способ своего конвертирования в возвращаемый тип ошибки, оператор `?` позаботится о конвертации автоматически.

В контексте листинга 9.7 оператор `?` в конце вызова функции `File::open` будет возвращать значение внутри `Ok` в переменную `f`. Если произойдет ошибка, то оператор `?` будет возвращаться из всей функции досрочно и передавать любое значение `Err` в вызывающий код. То же самое относится и к оператору `?` в конце вызова метода `read_to_string`.

Оператор `?` устраняет большой объем стереотипности и упрощает реализацию этой функции. Мы могли бы сократить этот код еще больше, выстроив вызовы методов в цепочку сразу после оператора `?`, как показано в листинге 9.8.

Листинг 9.8. Выстраивание вызовов методов в цепочку после оператора ?

src/main.rs

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut s = String::new();

    File::open("hello.txt")?.read_to_string(&mut s)?;

    Ok(s)
}
```

Мы перенесли создание нового экземпляра типа `String` в переменной `s` в начало функции; эта часть не изменилась. Вместо того чтобы создавать переменную `f`, мы сцепили вызов `read_to_string` непосредственно с результатом `File::open("hello.txt")?`. У нас по-прежнему есть оператор `?` в конце вызова `read_to_string`, и мы все так же возвращаем значение `Ok`, содержащее в `s` имя пользователя, когда и `File::open`, и `read_to_string` срабатывают успешно, а не возвращают ошибки. Функциональность снова та же, что и в листингах 9.6 и 9.7, просто этот способ написания более эргономичен.

Говоря о разных способах написания этой функции, листинг 9.9 показывает, что имеется способ сделать это еще короче.

Листинг 9.9. Использование `fs::read_to_string` вместо открытия и последующего чтения файла

src/main.rs

```
use std::io;
use std::fs;

fn read_username_from_file() -> Result<String, io::Error> {
    fs::read_to_string("hello.txt")
}
```

Операция чтения файла в строковую переменную встречается довольно часто, поэтому язык Rust предоставляет удобную функцию `fs::read_to_string`, которая открывает файл, создает новый экземпляр типа `String`, считывает содержимое файла, помещает его содержимое в этот экземпляр и возвращает его. Разумеется, использование `fs::read_to_string` не дает нам возможности объяснить весь процесс обработки ошибок, поэтому сначала мы сделали это более длинным способом.

Оператор ? может использоваться только в функциях, возвращающих Result

Оператор `?` может использоваться только в функциях, которые имеют возвращаемый тип `Result` , поскольку он определен для работы таким же образом, как и выражение `match`, которое мы определили в листинге 9.6. Часть выражения `match`,

которая требует возвращаемый тип `Result`, представлена фрагментом `return Err(e)`, поэтому тип, возвращаемый из функции, должен иметь тип `Result` в целях совместимости с этим `return`.

Давайте посмотрим, что произойдет, если мы применим оператор `?` в функции `main`, которая, как вы помните, возвращает тип `()`:

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt")?;
}
```

При компиляции этого кода мы получаем следующее сообщение об ошибке[1]:

```
error[E0277]: the `?` operator can only be used in a function that returns
`Result` or `Option` (or another type that implements `std::ops::Try`)
 --> src/main.rs:4:13
  |
4 |     let f = File::open("hello.txt")?;
  |             ^^^^^^^^^^^^^^^^^^^^^^^^ cannot use the `?` operator in a
  function that returns `()`
  |
  = help: the trait `std::ops::Try` is not implemented for `()`
  = note: required by `std::ops::Try::from_error`
```

Эта ошибка указывает на то, что использовать оператор `?` разрешено только в той функции, которая возвращает `Result<T, E>`. Когда вы пишете код в функции, которая не возвращает `Result<T, E>`, и хотите использовать оператор `?` при вызове других функций, возвращающих `Result<T, E>`, то у вас есть два варианта решения этой проблемы. Одним из них является изменение типа значения, возвращаемого из функции, на `Result<T, E>`, если у вас нет никаких ограничений, этому препятствующих. Другой заключается в использовании выражения `match` либо одного из методов `Result<T, E>` для обработки `Result<T, E>` любым подходящим способом.

Функция `main` является особой, и существуют ограничения на то, каким должен быть возвращаемый из нее тип. Для функции `main` одним из допустимых типов возвращения является `()`, и, что удобно, еще одним допустимым типом возвращения является `Result<T, E>`, как показано ниже:

```
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), Box<dyn Error>> {
    let f = File::open("hello.txt")?;

    Ok(())
}
```

[1] `ошибка[E0277]: оператор `?` используется только в той функции, которая возвращает `Result` либо `Option` (либо какой-то другой тип, который реализует `std::ops::Try`)`

Тип `Box<dyn Error>` называется типажным объектом, о котором мы поговорим в разделе «Использование типажных объектов, допускающих значения разных типов». Пока же вы можете истолковать смысл выражения `Box<dyn Error>` как «любой вид ошибки». Применение оператора `?` в функции `main` с этим типом возвращаемого значения допускается.

Теперь, когда мы обсудили детали вызова макрокоманды `panic!` и возвращения экземпляра `Result`, давайте вернемся к вопросу о том, каким образом определить, что уместно использовать и в каких случаях.

Паниковать! Или не паниковать!

Итак, как же определить, когда следует вызывать макрокоманду `panic!`, а когда нужно возвращать `Result`? Когда код поднимает панику, нет никакого способа восстановить работу. Вы можете вызвать макрокоманду `panic!` при любой ошибочной ситуации, независимо от того, есть ли возможный способ ее устранения или нет, но тогда вы принимаете решение от имени кода, вызывающего ваш код, о том, что такая ситуация является неустранимой. Когда вы решаете вернуть значение типа `Result`, вы даете вызывающему коду варианты, а не принимаете решение за него. Вызывающий код мог бы попытаться устранить ситуацию способом, который является уместным, либо он мог бы принять решение о том, что значение `Err` в данном случае устранить невозможно, и поэтому он может вызвать макрокоманду `panic!` и превратить устранимую ошибку в неустранимую. Следовательно, возвращение экземпляра типа `Result` — это хороший выбор по умолчанию, когда вы определяете функцию, которая может завершиться сбоем.

В редких случаях целесообразнее писать код, который вместо возвращения экземпляра типа `Result` поднимает панику. Давайте разберемся в причинах, почему уместно поднимать панику в примерах, прототипном коде и тестах. Затем мы обсудим ситуации, в которых компилятор не способен распознать, что сбой невозможен, но вы, как человек, способны. Эта глава завершится несколькими общими принципами в отношении целесообразности паники в библиотечном коде.

Примеры, прототипный код и тесты

Когда вы пишете пример для иллюстрации какой-то идеи, наличие в этом примере еще и кода с надежной обработкой ошибок сделает этот пример менее ясным. В примерах понятно, что вызов метода, такого как `unwrap`, который мог бы поднять панику, задуман как заполнитель для способа обработки ошибок, который вы хотите внедрить в программу, а он может отличаться в зависимости от того, что делает остальная часть кода.

Точно так же методы `unwrap` и `expect` очень удобны во время прототипирования до того, как вы решите, как обрабатывать ошибки. Они оставляют в коде четкие маркеры для момента, когда вы будете готовы сделать программу более надежной.

Если вызов метода не срабатывает в тесте, то вы хотели бы, чтобы не срабатывал весь тест, даже если тестируется функциональность вовсе не этого метода. Поскольку именно вызов макрокоманды `panic!` маркирует тест как несработавший, должен произойти именно вызов методов `unwrap` или `expect`.

Случаи, когда у вас больше информации, чем у компилятора

Также уместно вызывать метод `unwrap`, когда у вас есть какой-то алгоритм, который обеспечивает, чтобы экземпляр типа `Result` имел значение `Ok`. Но компилятор не понимает этот алгоритм. У вас по-прежнему будет экземпляр типа `Result`, который нужно обработать: любая операция, которую вы вызываете, по-прежнему может завершиться сбоем в целом, даже если в конкретной ситуации это логически невозможно. Если вы можете добиться, проверяя код вручную, чтобы у вас никогда не было варианта `Err`, то вполне допустимо вызывать метод `unwrap`. Вот пример:

```
use std::net::IpAddr;

let home: IpAddr = "127.0.0.1".parse().unwrap();
```

Мы создаем экземпляр `IpAddr`, разбирая жестко кодированное значение типа `String`. Мы видим, что значение `127.0.0.1` является допустимым IP-адресом, поэтому здесь можно использовать метод `unwrap`. Однако наличие жестко кодированного допустимого значения типа `String` не изменяет тип, возвращаемый из метода `parse`: мы все равно получаем значение типа `Result`, и компилятор заставит нас обрабатывать `Result`, как если бы вариант `Err` был возможен. Это произойдет потому, что компилятор недостаточно умен, чтобы увидеть, что это значение типа `String` всегда является допустимым IP-адресом. Если бы значение типа `String` с IP-адресом не было жестко закодировано в программе, а исходило от пользователя и поэтому была бы вероятность сбоя, то мы определенно хотели бы обрабатывать значение типа `Result` более надежным способом.

Принципы обработки ошибок

Рекомендуется, чтобы код поднимал панику, когда существует возможность, что код в итоге окажется в плохом состоянии. В данном контексте плохое состояние подразумевает нарушение какого-либо предположения, гарантии, контракта или инварианта, например, когда в код передаются недействительные, противоречивые либо пропущенные значения — плюс один или несколько из следующих пунктов:

- Плохое состояние — то, от чего не ожидают, что это будет происходить время от времени.
- Код после этого момента должен опираться на то, что он не находится в плохом состоянии.

- Хороший способ закодировать эту информацию в типах, которые вы используете, отсутствует.

Если кто-то вызывает ваш код и передает внутрь значения, которые не имеют смысла, то лучшим вариантом выбора, возможно, будет вызов `panic!` и оповещение человека, использующего вашу библиотеку, о дефекте в его коде. Благодаря этому он сможет устранить дефект во время разработки. Схожим образом макрокоманда `panic!` часто уместна, если вы вызываете внешний код, который находится вне вашего контроля, и он возвращает недопустимое состояние, которое вы не можете исправить.

Однако, когда сбой ожидаем, уместнее вернуть `Result`, чем устраивать вызов `panic!`. Примеры включают в себя парсер (синтаксический анализатор), которому передаются деформированные данные или HTTP-запрос, возвращающий статус, который указывает, что вы исчерпали число запросов. В этих случаях возвращение экземпляра `Result` указывает, что сбой является ожидаемой возможностью и вызывающий код должен решить, как его обработать.

Когда код выполняет операции над значениями, сначала он должен проверить, являются ли значения действительными, и поднять панику, если значения недействительны. Это главным образом делается из соображений безопасности: попытка выполнять операции над недействительными данными приводит к уязвимостям кода. Это главная причина, по которой стандартная библиотека будет вызывать макрокоманду `panic!`, если вы попытаетесь обратиться к памяти за пределами границ. Попытка обратиться к памяти, которая не принадлежит текущей структуре данных, является распространенной проблемой нарушения безопасности. В функциях часто есть контракты: их поведение гарантируется только в том случае, если входные данные удовлетворяют определенным требованиям. Поднимать панику, когда контракт нарушается, имеет смысл по той причине, что нарушение контракта всегда указывает на дефект со стороны вызывающего кода. И это не та ошибка, которую вызывающий код был бы обязан явно обрабатывать. На самом деле нет никакого разумного способа восстановить вызывающий код — исправлять код должны вызывающие его программисты. Контракты для функции, в особенности когда нарушение будет причиной паники, должны объясняться в документации API для этой функции.

Однако большое количество проверок на наличие ошибок во всех функциях было бы избыточным и могло бы раздражать. К счастью, можно использовать систему типов языка Rust (отсюда и проверка типов, выполняемая компилятором) для проведения многих проверок за вас. Если у функции есть конкретный тип в качестве параметра, то вы можете продолжить логику кода, зная, что компилятор уже обеспечил действительное значение. Например, если вместо `Option` у вас есть тип, то программа ожидает получить что-то определенное. И тогда коду не придется обрабатывать два случая для вариантов `Some` и `None`: у него будет только один случай — точное наличие значения. Код, пытающийся передать ничто в функцию, даже не скомпилируется, поэтому функция не должна проверять этот случай во

время выполнения. Еще один пример — использование беззнакового целочисленного типа, например `u32`, который обеспечивает нетрицательность параметра.

Создание настраиваемых типов для проверки допустимости

Давайте разовьем идею о том, что можно использовать систему типов языка Rust, чтобы у нас было действительное значение. Мы посмотрим, как создать настраиваемый тип для проверки. Вспомните игру на угадывание в главе 2, в которой код просил пользователя угадать число от 1 до 100. Мы так и не подтвердили, находится ли догадка пользователя в интервале между 1 и 100, перед тем как сверить его с секретным числом; мы только подтвердили, что догадка является положительным числом. В том случае последствия были не очень страшными: наш результат «Слишком большое» или «Слишком маленькое» все равно был бы правильным. Но все же было бы полезно усовершенствовать этот код, направляя пользователя к допустимым отгадкам и действуя по-разному, когда пользователь загадывает число, которое находится вне интервала, и когда вместо этого, к примеру, пользователь набирает на клавиатуре буквы.

Один из способов это сделать — выполнять разбор загаданного числа не только как тип `u32`, но и как тип `i32`, допустив потенциально отрицательные числа, а затем добавить проверку на предмет того, что число находится внутри интервала, например, вот так:

```
loop {
    // --пропуск--

    let guess: i32 = match guess.trim().parse() {
        Ok(num) => num,
        Err(_) => continue,
    };

    if guess < 1 || guess > 100 {
        println!("Секретное число будет между 1 и 100.");
        continue;
    }

    match guess.cmp(&secret_number) {
    // --пропуск--
}
```

Выражение `if` проверяет, находится ли значение вне интервала, сообщает пользователю о проблеме и вызывает `continue`, начиная следующую итерацию цикла и запрашивая еще одно загаданное число. После выражения `if` мы можем продолжить сравнение загаданного числа `guess` с секретным числом, зная, что загаданное число находится между 1 и 100.

Однако это неидеальное решение: если бы было чрезвычайно важно, чтобы программа работала только со значениями от 1 до 100 и у нее было бы много функций

с этим требованием, то такая проверка в каждой функции была бы утомительной (и могла бы повлиять на производительность).

Вместо повторных проверок всего кода мы можем создать новый тип и поместить проверки на допустимость в функцию, создав экземпляр типа. Благодаря этому функции будут безопасно использовать новый тип в своих сигнатурах и уверенно применять получаемые ими значения. Листинг 9.10 показывает один из способов определения типа `Guess`, который будет создавать экземпляр типа `Guess` только в том случае, если новая функция получает значение между 1 и 100.

Листинг 9.10. Тип Guess, который продолжит работу только со значениями между 1 и 100

```
❶ pub struct Guess {
      value: i32,
  }

  impl Guess {
   ❷ pub fn new(value: i32) -> Guess {
       ❸ if value < 1 || value > 100 {
          ❹ panic!("Значение догадки должно быть между 1 и 100,
                    получено {}.", value);
          }

       ❺ Guess {
              value
          }
      }

    ❻ pub fn value(&self) -> i32 {
          self.value
      }
  }
```

Сначала мы определяем структуру под названием `Guess`, имеющую поле с именем `value`, которое содержит значение тип `i32` ❶. Именно тут будет храниться число.

Затем мы реализуем связанную функцию `new` в структуре `Guess`, которая создает экземпляры значений типа `Guess` ❷. `new` определяется как функция, имеющая один параметр `value` типа `i32` и возвращающая тип `Guess`. Код в теле функции `new` проверяет параметр `value`, убеждаясь, что он находится между 1 и 100 ❸. Если параметр `value` этот тест не проходит, то мы делаем вызов макрокоманды `panic!` ❹, которая будет оповещать программиста, пишущего вызывающий код, о том, что у него есть дефект, который необходимо устранить, потому что создание экземпляра типа `Guess` со значением вне этого интервала нарушит контракт, на который полагается функция `Guess::new`. Условия, при которых функция `Guess::new`, возможно, будет паниковать, должны обсуждаться в документации к ее публичному API. В главе 14 мы рассмотрим документационные соглашения, указывающие на возможность вызова макрокоманды `panic!` в создаваемой вами документации API. Если параметр `value` тест не проходит, то мы создаем новый экземпляр

типа `Guess`, устанавливая его поле `value` равным параметру `value`, и возвращаем `Guess` ❺.

Далее мы реализуем метод с именем `value`, который заимствует `self`, не имеет каких-либо других параметров и возвращает тип `i32` ❻. Этот вид метода иногда называют геттером (методом чтения), потому что его цель — получить данные из своих полей и вернуть их. Этот публичный метод необходим потому, что поле `value` структуры `Guess` конфиденциально. Важно, чтобы поле `value` было конфиденциальным. Это нужно для того, чтобы код, использующий структуру `Guess`, не мог устанавливать его значение напрямую: код вне модуля должен использовать функцию `Guess::new` для создания экземпляра `Guess`, тем самым обеспечивая, чтобы структура `Guess` не имела никакого способа получить `value`, которое не было проверено условиями в функции `Guess::new`.

Функция, которая имеет параметр либо возвращает только числа между 1 и 100, могла бы затем объявить в своей сигнатуре, что она берет или возвращает `Guess` вместо `i32` и ей не нужно делать никаких дополнительных проверок в своем теле.

Итоги

Средства языка Rust по обработке ошибок предназначены для того, чтобы помочь вам писать более надежный код. Макрокоманда `panic!` сигнализирует, что программа находится в состоянии, которое она не может обработать, и позволяет вам остановить процесс, а не продолжать работу с недействительными или неправильными значениями. Перечисление `Result` использует систему типов языка Rust, чтобы сообщать о том, что операции могут дать сбой так, что код сможет восстановиться. Вы можете использовать тип `Result`, чтобы сообщить коду, который вызывает ваш код, что он должен обработать потенциальный успех или провал. Использование макрокоманды `panic!` и типа `Result` в соответствующих ситуациях сделает ваш код намного надежнее перед лицом неизбежных проблем.

Теперь, ознакомившись с тем, как стандартная библиотека использует обобщения с перечислениями `Option` и `Result`, мы поговорим о принципе работы обобщений и о том, как их использовать в коде.

10 Обобщенные типы, типажи и жизненный цикл

В каждом языке программирования есть инструменты для эффективной работы с дублирующим друг друга понятиями. В Rust одним из таких инструментов являются обобщения. Обобщения — это абстрактные заменители конкретных типов или других свойств. Когда мы пишем код, мы можем выражать поведение обобщений или их связи с другими обобщениями, не имея сведений о том, что будет на их месте во время компиляции и выполнения кода.

Функции могут принимать параметры не только конкретного типа, такого как `i32` или `String`, но и некоего обобщенного типа, подобно тому как они берут параметры с неизвестными значениями, чтобы выполнить один и тот же код для многочисленных конкретных значений. На самом деле обобщения уже использовались в главе 6 (`Option<T>`), главе 8 (`Vec<T>` и `HashMap<K, V>`) и главе 9 (`Result<T, E>`). В этой главе вы узнаете способы определения собственных типов, функций и методов с помощью обобщений!

Сначала мы рассмотрим технический прием по извлечению функции, чтобы уменьшить повторы в коде. Далее мы воспользуемся тем же приемом для создания обобщенной функции из двух функций, которые отличаются только типами параметров. Мы также объясним, как использовать обобщенные типы в определениях структур и перечислений.

Затем вы узнаете, как использовать типажи для определения поведения в обобщенном виде. Вы можете комбинировать типажи с обобщенными типами, ограничивая обобщенный тип только теми типами, у которых есть определенные свойства, в отличие от просто любого типа.

Наконец, мы обсудим жизненный цикл, то есть различные обобщения, которые дают компилятору информацию о том, как ссылки соотносятся друг с другом. Жизненный цикл позволяет заимствовать значения во многих ситуациях, при этом давая компилятору возможность проверять действительность ссылок.

Удаление повторов путем извлечения функции

Прежде чем углубляться в синтаксис обобщенных типов, давайте сначала посмотрим, как удалять повторы, которые не связаны с обобщенными типами, путем извлечения функции. Затем мы применим этот прием для извлечения обобщенной функции! Вы начнете распознавать повторяющийся код, который может использовать обобщения, точно так же, как вы распознаете повторяющийся код и извлекаете его в функцию.

Рассмотрим короткую программу, которая находит наибольшее число в списке, как показано в листинге 10.1.

Листинг 10.1. Код для поиска наибольшего числа в списке чисел

src/main.rs

```
fn main() {
 ❶ let number_list = vec![34, 50, 25, 100, 65];

 ❷ let mut largest = number_list[0];

 ❸ for number in number_list {
     ❹ if number > largest {
         ❺ largest = number;
       }
   }

   println!("Наибольшее число равно {}", largest);
}
```

Этот код помещает список целых чисел в переменную `number_list` ❶ и передает первое число в списке в переменную с именем `largest` ❷. Затем он перебирает все числа в списке ❸, и если текущее число больше, чем число, хранящееся в переменной `largest` ❹, он заменяет число в этой переменной ❺. Однако если текущее число меньше встречавшегося до сих пор наибольшего числа, то переменная не изменяется и код переходит к следующему числу в списке. После рассмотрения всех чисел в списке переменная `largest` должна содержать наибольшее число, которое в данном случае равно 100.

Чтобы найти наибольшее число в двух разных списках чисел, мы можем повторить код в листинге 10.1 и применить тот же алгоритм в двух разных местах программы, как показано в листинге 10.2.

Листинг 10.2. Код для поиска наибольшего числа в двух списках чисел

src/main.rs

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = number_list[0];

    for number in number_list {
```

```
            if number > largest {
                largest = number;
            }
        }

        println!("Наибольшее число равно {}", largest);

        let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

        let mut largest = number_list[0];

        for number in number_list {
            if number > largest {
                largest = number;
            }
        }

        println!("Наибольшее число равно {}", largest);
    }
```

Этот код работает. Вместе с тем необходимость повторять код утомляет, и возможны ошибки. Если мы хотим внести изменения в код, нам также приходится обновлять его в нескольких местах.

Для того чтобы устранить этот повтор, мы можем создать абстракцию, определив функцию, которая работает с любым списком целых чисел, передаваемых ей в параметре. Это решение делает код понятнее и позволяет выразить идею поиска наибольшего числа в списке абстрактно.

В листинге 10.3 мы извлекли код, который отыскивает наибольшее число, в функцию с именем `largest`. В отличие от кода в листинге 10.1, который находит наибольшее число только в одном конкретном списке, эта программа отыскивает наибольшее число в двух разных списках.

Листинг 10.3. Абстрактный код для поиска наибольшего числа в двух списках

src/main.rs

```
fn largest(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
```

```
    println!("Наибольшее число равно {}", result);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let result = largest(&number_list);
    println!("Наибольшее число равно {}", result);
}
```

Функция `largest` имеет параметр `list`, представляющий любой конкретный срез значений типа `i32`, которые можно передать в функцию. В результате, когда мы вызываем функцию, указанный код выполняется для конкретных значений, которые мы передаем внутрь.

В общем случае вот шаги, которые мы предприняли, чтобы изменить код из листинга 10.2 на код в листинге 10.3:

1. Выявить повторяющийся код.
2. Извлечь повторяющийся код в тело функции и указать входные данные и возвращаемые значения этого кода в сигнатуре функции.
3. Обновить два экземпляра повторяющегося кода так, чтобы они вызывали функцию.

Далее мы предпримем те же самые шаги с обобщениями, чтобы сократить повторяющийся код другими способами. Обобщения позволяют коду производить операции над абстрактными типами точно так же, как тело функции способно производить операции над абстрактным списком вместо конкретных значений.

Допустим, у нас есть две функции: одна отыскивает наибольший элемент в срезе значений типа `i32`, а другая ищет наибольший элемент в срезе значений типа `char`. Как нам устранить этот повтор? Давайте выясним!

Обобщенные типы данных

Мы можем использовать обобщения для создания определений таких элементов, как сигнатуры функций или структуры, которые затем могут использоваться с разными конкретными типами данных. Давайте сначала посмотрим на способы определения функций, структур, перечислений и методов с использованием обобщений. Затем мы обсудим вопрос влияния обобщений на производительность кода.

В определениях функций

Во время определения функции, которая использует обобщения, мы помещаем обобщения в сигнатуру функции, где обычно указываем типы параметров и возвращаемое значение. Это делает код более гибким и обеспечивает большей функ-

циональностью тех, кто вызывает нашу функцию, при этом предотвращая повторы кода.

Продолжая пример функции `largest`, листинг 10.4 показывает две функции — обе отыскивают наибольшее значение в срезе.

Листинг 10.4. Две функции, отличающиеся только именами и типами в сигнатурах

src/main.rs

```
fn largest_i32(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> char {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }
    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&number_list);
    println!("Наибольшее число равно {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&char_list);
    println!("Наибольший символ равен {}", result);
}
```

Функция `largest_i32` — это именно та функция, которую мы извлекли в листинге 10.3, она отыскивает в срезе наибольшее значение типа `i32`. Функция `largest_char` ищет в срезе наибольшее значение типа `char`. Тела функций имеют одинаковый код, поэтому давайте исключим повторы, введя параметр обобщенного типа в единственную функцию.

В целях параметризации типов в новой определяемой нами функции нужно назвать типовой параметр, так же как мы делаем это для входных параметров, принимаемых функцией. В качестве имени типового параметра можно использовать любой идентификатор. Но мы будем использовать `T`, потому что, по соглашению,

в языке Rust используются короткие имена параметров, часто состоящие из единственной буквы, а именование типов в языке Rust по соглашению предусматривает применение верблюжьего регистра. По умолчанию большинство программистов, пишущих на Rust, выбирают `T` — «тип».

Когда мы используем параметр в теле функции, нам приходится объявлять имя параметра в сигнатуре, благодаря чему компилятор знает, что это имя означает. Схожим образом, когда мы используем имя типового параметра в сигнатуре функции, мы должны объявить имя типового параметра, прежде чем его использовать. Для того чтобы определить обобщенную функцию `largest`, поместите объявления имен типов в угловые скобки `<>` между именем функции и списком параметров, как показано ниже:

```
fn largest<T>(list: &[T]) -> T {
```

Мы читаем это определение следующим образом: функция `largest` является обобщением некоего типа `T`. Эта функция имеет один параметр с именем `list`, который является срезом значений типа `T`. Функция `largest` будет возвращать значение того же типа `T`.

Листинг 10.5 показывает объединенное определение функции `largest` с использованием обобщенного типа данных в ее сигнатуре. В листинге также показано, как вызвать функцию с использованием среза значений типа `i32` или значений типа `char`. Обратите внимание, что этот код пока что не компилируется, но мы исправим это позже в данной главе.

Листинг 10.5. Определение функции largest, которая использует параметры обобщенного типа, но пока не компилируется

src/main.rs

```
fn largest<T>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("Наибольшее число равно {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("Наибольший символ равен {}", result);

}
```

Если мы скомпилируем этот код прямо сейчас, то получим такую ошибку[1]:

```
error[E0369]: binary operation `>` cannot be applied to type `T`
 --> src/main.rs:5:12
  |
5 |         if item > largest {
  |            ^^^^^^^^^^^^^^
  |
  = note: an implementation of `std::cmp::PartialOrd` might be missing for `T`
```

В примечании упоминается `std::cmp::PartialOrd`, то есть *типаж*. Мы поговорим о типажах в разделе «Типажи: определение совместного поведения». А пока указанная ошибка констатирует, что тело функции `largest` не будет работать для всех возможных типов, которыми `T` может быть. Поскольку мы хотим сравнивать значения типа `T` в теле, мы можем использовать только те типы, значения которых могут упорядочиваться. Для обеспечения возможности сравнений в стандартной библиотеке имеется типаж `std::cmp::PartialOrd`, который вы можете реализовать в типах (более подробную информацию об этом типаже см. в приложении B в конце книги). Вы увидите, что у обобщенного типа есть определенный типаж, в разделе «Типажи в качестве параметров». Но давайте сначала изучим другие приемы использования параметров обобщенного типа.

В определениях структуры

Используя синтаксис `<>`, мы также можем определять структуры с использованием параметров обобщенного типа в одном или нескольких полях. Листинг 10.6 показывает, как определить структуру `Point<T>` для хранения значений координат `x` и `y` любого типа.

Листинг 10.6. Структура Point<T>, содержащая значения x и y типа T

src/main.rs

```
struct Point<T>❶ {
    x: T❷,
    y: T❸,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

Синтаксис использования обобщений в определениях структур аналогичен тому, который применяется в определениях функций. Сначала мы объявляем имя типового параметра внутри угловых скобок сразу после имени структуры ❶. Затем используем обобщенный тип в определении структуры, где в ином случае мы указывали бы конкретные типы данных ❷❸.

[1] ошибка[E0369]: бинарная операция `>` не применяется к типу `T`

Обратите внимание, что, поскольку для определения `Point<T>` мы использовали только один обобщенный тип, это определение говорит о том, что структура `Point<T>` является обобщением над неким типом `T`, а поля `x` и `y` имеют одинаковый тип, каким бы он ни был. Если мы создадим экземпляр `Point<T>`, который имеет значения разных типов, как в листинге 10.7, то код компилироваться не будет.

Листинг 10.7. Поля x и y должны быть одного типа, поскольку они имеют одинаковый обобщенный тип данных, T

src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}
```

В этом примере, когда мы передаем целочисленное значение `5` полю `x`, мы сообщаем компилятору, что обобщенный тип `T` для этого экземпляра `Point<T>` будет целочисленным. Затем, когда мы передаем `4.0` полю `y`, которое мы определили как имеющее тот же тип, что и у `x`, мы получим ошибку несовпадения типов, как здесь:

```
error[E0308]: mismatched types
 --> src/main.rs:7:38
  |
7 |     let wont_work = Point { x: 5, y: 4.0 };
  |                                      ^^^ expected integral variable, found
floating-point variable
  |
  = note: expected type `{integer}`
             found type `{float}`
```

Для того чтобы определить структуру `Point`, где поля `x` и `y` являются обобщенными, но могут иметь разные типы, мы можем использовать несколько разных параметров обобщенного типа. Например, в листинге 10.8 можно изменить определение структуры `Point` так, чтобы оно стало обобщением над типами `T` и `U`, где поле `x` имеет тип `T`, а поле `y` — тип `U`.

Листинг 10.8. Point<T, U> является обобщением двух типов, в результате чего x и y могут быть значениями разных типов

src/main.rs

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
```

```
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

Теперь все приведенные выше экземпляры структуры `Point` разрешены! Вы можете использовать в определении столько параметров обобщенного типа, сколько захотите, но использование более одного-двух параметров затрудняет чтение кода. Если в коде требуется много обобщенных типов, то это означает, что код нуждается в реструктуризации на меньшие части.

В определениях перечислений

Как и в случае со структурами, мы можем определять перечисления, которые содержат в своих вариантах обобщенные типы данных. Давайте еще раз взглянем на перечисление `Option<T>`, предоставляемое стандартной библиотекой, которое мы использовали в главе 6:

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

Теперь это определение должно иметь больше смысла. Как вы видите, перечисление `Option<T>` является обобщением над типом `T` и имеет два варианта: `Some`, который содержит одно значение типа `T`, и `None`, который не содержит никакого значения. Используя перечисление `Option<T>`, мы можем выражать абстрактную идею наличия необязательного значения, а поскольку `Option<T>` является обобщением, можно использовать эту абстракцию независимо от типа варианта значения.

Перечисления также могут использовать более одного обобщенного типа. Определение перечисления `Result`, которое мы использовали в главе 9, является одним из таких примеров:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Перечисление `Result` является обобщением двух типов — `T` и `E`. У него два варианта: `Ok`, который содержит значение типа `T`, и `Err`, который содержит значение типа `E`. Это определение позволяет удобно использовать перечисление `Result` везде, где есть операция, которая может быть успешной (возвращать значение некоторого типа `T`) или неуспешной (возвращать ошибку некоторого типа `E`). По сути дела, это то, что мы использовали для открытия файла в листинге 9.3, где `T` заполнялся типом `std::fs::File`, когда файл успешно открывался, а `E` заполнялся типом `std::io::Error`, когда возникали проблемы с открытием файла.

Когда вы замечаете, что в коде несколько определений структур или перечислений отличаются только типами значений, которые они содержат, вы можете избежать повторов, используя обобщенные типы.

В определениях методов

Мы можем реализовать методы в структурах и перечислениях (как мы делали в главе 5), а также использовать обобщенные типы в их определениях. Листинг 10.9 показывает структуру `Point<T>`, которую мы ранее определили в листинге 10.6, с реализованным в ней методом `x`.

Листинг 10.9. Реализация метода x в структуре Point<T>, который будет возвращать ссылку на поле x типа T

src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

Здесь мы определили метод `x` для структуры `Point<T>`, который возвращает ссылку на данные в поле `x`.

Обратите внимание, что сразу после ключевого слова `impl` приходится объявлять `T`, в результате чего можно использовать его для уточнения, что мы реализуем методы в типе `Point<T>`. Благодаря объявлению `T` как обобщенного типа после `impl`, Rust будет отождествлять тип в угловых скобках в `Point` как обобщенный, а не конкретный. Например, мы могли бы реализовать методы только в экземплярах типа `Point<f32>`, а не в экземплярах типа `Point<T>` с любым обобщенным типом. В листинге 10.10 мы используем конкретный тип `f32`, то есть не объявляем никаких типов после `impl`.

Листинг 10.10. Блок impl, который применим только к структуре с отдельно взятым конкретным типом для параметра обобщенного типа T

```
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

Этот код означает, что тип `Point<f32>` имеет метод `distance_from_origin`, а для других экземпляров типа `Point<T>`, где `T` не относится к типу `f32`, этот метод определен не будет. Указанный метод измеряет расстояние нашей точки от точки в координатах (0.0, 0.0) и использует математические операции, доступные только для типов с плавающей точкой.

Параметры обобщенного типа в определении структуры не всегда совпадают с параметрами, используемыми в сигнатурах методов этой структуры. Например, листинг 10.11 определяет метод `mixup` для структуры `Point<T, U>` из листинга 10.8. Указанный метод берет в качестве параметра еще одну структуру `Point`, которая имеет другие типы, в отличие от структуры `Point` с именем `self`, для которой мы вызываем метод `mixup`. Данный метод создает новый экземпляр типа `Point` со значениями `x` из структуры `Point` с именем `self` (типа `T`) и значение `y` из переданной внутрь метода структуры `Point` (типа `W`).

Листинг 10.11. Метод, который использует другие обобщенные типы, отличные от определенной структуры

src/main.rs

```
struct Point<T, U> {
    x: T,
    y: U,
}

impl<T, U>❶ Point<T, U> {
    fn mixup<V, W>❷(self, other: Point<V, W>) -> Point<T, W> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
  ❸ let p1 = Point { x: 5, y: 10.4 };
  ❹ let p2 = Point { x: "Привет", y: 'c'};

  ❺ let p3 = p1.mixup(p2);

  ❻ println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

В функции `main` мы определили структуру `Point`, которая имеет тип `i32` для поля `x` (со значением `5`) и тип `f64` для поля `y` (со значением `10.4` ❸). Переменная `p2` — это структура `Point`, которая имеет тип строкового среза для поля `x` (со значением `"Привет"`) и тип `char` для поля `y` (со значением `c` ❹). Вызов метода `mixup` для `p1` с аргументом `p2` дает нам переменную `p3` ❺, которая будет иметь тип `i32` для поля `x`, потому что поле `x` пришло из `p1`. Переменная `p3` будет иметь тип `char` для поля `y`, потому что поле `y` пришло из `p2`. Вызов макрокоманды `println!` выводит `p3.x = 5, p3.y = c` ❻.

Цель этого примера — продемонстрировать ситуацию, в которой одни обобщенные параметры объявляются с помощью ключевого слова `impl`, а другие — с помощью определения метода. Здесь обобщенные параметры `T` и `U` объявляются после `impl`❶, поскольку они идут вместе с определением структуры. Обобщенные параметры `V` и `W` объявляются после `fn mixup` ❷, поскольку они относятся только к методу.

Производительность кода с использованием обобщений

Вы, наверное, задаетесь вопросом, сказывается ли использование параметров обобщенного типа на времени выполнения. Хорошая новость — язык Rust реализует обобщенные типы таким образом, что, используя их, код не работает медленнее, чем при применении конкретных типов.

Rust достигает этого за счет мономорфизации кода, который использует обобщенные методы во время компиляции. Мономорфизация — это процесс превращения обобщенного кода в конкретный путем вставки конкретных типов, которые используются во время компиляции.

В этом процессе компилятор выполняет действия, противоположные шагам, которые мы предпринимали для создания обобщенной функции в листинге 10.5: компилятор просматривает все места, где вызывается обобщенный код, и генерирует код для конкретных типов, с которыми вызывается обобщенный код.

Давайте посмотрим, как это работает. Пример ниже использует перечисление `Option<T>` из стандартной библиотеки:

```
let integer = Some(5);
let float = Some(5.0);
```

Когда Rust компилирует этот код, он выполняет мономорфизацию. Во время этого процесса компилятор читает значения, которые были использованы в экземплярах типа `Option<T>`, и определяет два типа `Option<T>`: один — `i32`, а другой — `f64`. По этой причине он расширяет обобщенное определение типа `Option<T>` в `Option_i32` и `Option_f64`, тем самым заменяя обобщенное определение конкретными определениями.

Мономорфизированная версия кода выглядит следующим образом. Обобщенный тип `Option<T>` заменяется конкретными определениями, создаваемыми компилятором:

src/main.rs

```
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
```

```
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

Поскольку Rust компилирует обобщенный код в код, который конкретизирует тип в каждом экземпляре, на время выполнения в связи с использованием обобщений это никак не влияет. Когда код выполняется, он работает точно так же, как если бы мы дублировали каждое определение вручную. Процесс мономорфизации делает обобщения чрезвычайно эффективными во время выполнения.

Типажи: определение совместного поведения

Типаж[1] сообщает компилятору языка Rust о функциональности, которой обладает тот или иной тип и которой он может делиться с другими типами. Мы можем использовать типажи для определения совместного поведения в абстрактном виде. Мы можем использовать границы типажа, чтобы показать — обобщением может быть любой тип, который ведет себя определенным образом.

ПРИМЕЧАНИЕ

Типажи похожи на средство, которое в других языках часто называется интерфейсами, хотя и с некоторыми отличиями.

Определение типажа

Поведение типа состоит из методов, которые мы вызываем для этого типа. Разные типы имеют одинаковое поведение, если мы вызываем одинаковые методы для всех этих типов. Определения типажей являются способом группирования вместе сигнатур методов с целью описания множества поведений, необходимых для достижения некой цели.

Например, у нас есть пара структур, которые содержат различные виды и объемы текста: структура `NewsArticle` содержит новостной сюжет, зарегистрированный в том или ином месте, а структура `Tweet` включает не более 280 символов вместе с метаданными, описывающими вид твита: новый твит, ретвит либо ответ на другой твит.

[1] Типаж (trait), или композиционный признак, в языке Rust представляет собой атрибут композиционного построения, который используется для абстракций, хорошо знакомых из объектно-ориентированного программирования, но вместо наследования он использует принцип композиции. Композиция позволяет определять совокупность атрибутов, которые можно безопасно смешивать и сопоставлять.

Мы хотим создать библиотеку-медиаагрегатор, которая показывает сводки данных, хранящиеся в экземплярах структур `NewsArticle` или `Tweet`. Для этого нужна сводка от каждого типа, и надо запрашивать эту сводку путем вызова метода `summarize` для экземпляра. Листинг 10.12 показывает определение типажа `Summary`, который выражает это поведение.

Листинг 10.12. Типаж Summary, поведение которого предусмотрено методом summarize

src/lib.rs

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

Здесь мы объявляем типаж, используя ключевое слово `trait` с последующим именем типажа, которым в данном случае является `Summary`. Внутри фигурных скобок мы объявляем сигнатуры методов, описывающие поведения типов, реализующих этот типаж, которое в данном случае представлено функцией `fn summarize(&self) -> String`.

После сигнатуры метода вместо предоставления реализации в фигурных скобках мы используем точку с запятой. У каждого типа, реализующего этот типаж, должно быть собственное индивидуальное поведение для тела метода. Компилятор обеспечит, чтобы в любом типе, который имеет типаж `Summary`, был определен метод `summarize` именно с такой сигнатурой.

Типаж может иметь в теле более одного метода: сигнатуры методов перечисляются по одной на строку кода, и каждая строка заканчивается точкой с запятой.

Реализация типажа в типе

Теперь, когда мы определили желаемое поведение с помощью типажа `Summary`, можно реализовать его в типах медиаагрегатора. Листинг 10.13 показывает реализацию типажа `Summary` в структуре `NewsArticle`, которая использует заголовок, автора и местоположение для создания значения, возвращаемого из метода `summarize`. Для структуры `Tweet` мы определяем метод `summarize` как имя пользователя, за которым следует весь текст твита, исходя из того, что содержимое твита уже ограничено 280 символами.

Листинг 10.13. Реализация типажа `Summary` в типах `NewsArticle` и `Tweet`

src/lib.rs

```
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
```

```
    fn summarize(&self) -> String {
        format!("{}, {} ({})", self.headline, self.author, self.location)
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

Реализация типажа в типе похожа на реализацию обычных методов. Разница заключается в том, что после ключевого слова `impl` мы помещаем имя типажа, который хотим реализовать, потом мы используем ключевое слово `for`, а затем указываем имя типа, в котором собираемся реализовать типаж. В блок `impl` мы помещаем сигнатуры методов, которые были обозначены определением типажа. Вместо точки с запятой после каждой сигнатуры мы используем фигурные скобки и заполняем тело метода конкретным поведением, которое должно быть у методов типажа для некоторого типа.

После реализации типажа можно вызывать методы для экземпляров типов `NewsArticle` и `Tweet` таким же образом, как мы вызываем обычные методы, например:

```
let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from("конечно, как вы, наверное, уже знаете, люди"),
    reply: false,
    retweet: false,
};

println!("1 новый твит: {}", tweet.summarize());
```

Этот код выводит `1 новый твит: horse_ebook: конечно, как вы, наверное, уже знаете, люди`.

Обратите внимание, поскольку мы определили типаж `Summary` и типы `NewsArticle` и `Tweet` в одном и том же файле lib.rs в листинге 10.13, все они находятся в одной области видимости. Допустим, этот lib.rs предназначен для упаковки, которую мы назвали `aggregator`, и кто-то хочет использовать функциональность нашей упаковки для реализации типажа `Summary` в структуре, определенную внутри области видимости собственной библиотеки. Этому человеку нужно сначала ввести этот типаж в свою область видимости. Он сделает это, указав `use aggregator::Summary;`, что даст ему возможность реализовать `Summary` для своего типа. Для того чтобы дру-

гая упаковка смогла реализовать типаж `Summary`, этот типаж также должен быть публичным, поэтому мы ставим ключевое слово `pub` перед `trait` в листинге 10.12.

При реализации типажей следует обратить внимание на одно ограничение — мы можем реализовать типаж в типе только в том случае, если либо типаж, либо тип являются локальными для упаковки. Например, можно реализовать типажи стандартной библиотеки, такие как `Display`, в настраиваемом типе `Tweet` в рамках функциональности упаковки `aggregator`, потому что тип `Tweet` является локальным для упаковки `aggregator`. Мы также можем реализовать `Summary` в `Vec<T>` в упаковке `aggregator`, поскольку типаж `Summary` является локальным для упаковки `aggregator`.

Но мы не можем реализовывать внешние типажи во внешних типах. Например, мы не можем реализовать типаж `Display` в `Vec<T>` в упаковке `aggregator`, потому что `Display` и `Vec<T>` определены в стандартной библиотеке и не являются локальными для упаковки `aggregator`. Это ограничение является частью свойства программ — когерентностью, а конкретнее — «сиротским правилом», названным так потому, что родительский тип отсутствует. Это правило обеспечивает, чтобы код других людей не мог нарушать ваш код, и наоборот. Без этого правила две упаковки могли бы реализовать один и тот же типаж в одном и том же типе, и язык Rust не знал бы, какую реализацию использовать.

Реализации по умолчанию

Иногда бывает полезно иметь поведение по умолчанию для некоторых или всех методов в типаже вместо того, чтобы требовать реализации всех методов в каждом типе. Тогда, по мере реализации типажа в конкретном типе, мы можем оставлять или переопределять поведение по умолчанию каждого метода.

Листинг 10.14 показывает, как конкретизировать строковое значение по умолчанию для метода `summarize` типажа `Summary`, а не только определять сигнатуру этого метода, как мы сделали в листинге 10.12.

Листинг 10.14. Определение типажа Summary с реализацией по умолчанию метода summary

src/lib.rs

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Читать дальше...)")
    }
}
```

Чтобы воспользоваться реализацией по умолчанию для суммирования экземпляров типа `NewsArticle`, вместо определения настраиваемой реализации мы указываем пустой блок `impl` с инструкцией `impl Summary for NewsArticle {}`.

Хотя мы больше не определяем метод `summary` непосредственно для `NewsArticle`, мы предоставили реализацию по умолчанию и указали, что `NewsArticle` реализует

типаж Summary. В результате по-прежнему можно вызвать метод summary для экземпляра NewsArticle, как в примере:

```
let article = NewsArticle {
    headline: String::from("Пингвины выигрывают Кубок Стэнли!"),
    location: String::from("Питтсбург, шт. Пенсильвания, США"),
    author: String::from("Айсбург"),
    content: String::from("Питтсбург Пингвинз снова является лучшей
    хоккейной командой в НХЛ."),
};

println!("Есть новая статья! {}", article.summarize());
```

Этот код выведет

```
Есть новая статья! (Читать дальше...).
```

Создание реализации по умолчанию для метода summary не требует никаких изменений в реализации Summary для типа Tweet в листинге 10.13. Причина в том, что синтаксис для переопределения реализации по умолчанию совпадает с синтаксисом для реализации типажного метода, у которого нет реализации по умолчанию.

Реализации по умолчанию могут вызывать другие методы с тем же типажом, даже если эти методы не имеют реализации по умолчанию. Благодаря этому типаж может обеспечивать много полезной функциональности и требовать, чтобы разработчики указывали только небольшую ее часть. Например, мы могли бы определить типаж Summary с методом summarize_author, реализация которого является обязательной, а затем определить метод summarize, имеющий реализацию по умолчанию, которая вызывает метод summarize_author:

```
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Читать дальше в {}...)", self.summarize_author())
    }
}
```

Чтобы воспользоваться этой версией Summary, нам нужно определить функцию summarize_author, только когда мы реализуем типаж в типе:

```
impl Summary for Tweet {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}
```

После того как мы определили функцию summarize_author, можно вызывать метод summarize для экземпляров структуры Tweet, и реализация по умолчанию метода summarize будет вызывать определение функции summarize_author, которую мы предоставили. Поскольку мы реализовали функцию summarize_author, типаж

`Summary` обусловил поведение метода `summarize`, не требуя, чтобы мы писали больше кода.

```
let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from("конечно, как вы, наверное, уже знаете, люди"),
    reply: false,
    retweet: false,
};

println!("1 новый твит: {}", tweet.summarize());
```

Этот код выводит

```
1 новый твит: (Читать дальше в @horse_ebooks...).
```

Обратите внимание, что не существует возможности вызывать реализацию по умолчанию из переопределяющей реализации того же метода.

Типажи в качестве параметров

Теперь, когда вы знаете, как определять и реализовывать типажи, можно узнать, как использовать типажи для определения функций, которые принимают много разных типов.

Например, в листинге 10.13 мы реализовали типаж `Summary` для типов `NewsArticle` и `Tweet`. Мы можем определить функцию `notify` — она вызывает метод `summarize` для параметра `item`, который имеет некий тип, реализующий типаж `Summary`. Для этого можно использовать синтаксис `impl Trait`:

```
pub fn notify(item: impl Summary) {
    println!("Срочные новости! {}", item.summarize());
}
```

Вместо конкретного типа для параметра `item` мы указываем ключевое слово `impl` и имя типажа. Этот параметр принимает любой тип, реализующий указанный типаж. В теле функции `notify` мы можем вызывать любые методы для `item`, которые исходят из типажа `Summary`, например метод `summarize`. Можно вызвать функцию `notify` и передать внутрь экземпляр любого из двух типов, `NewsArticle` или `Tweet`. Код, вызывающий указанную функцию с любым другим типом, например `String` или `i32`, не компилируется, поскольку эти типы не реализуют `Summary`.

Синтаксис границы типажа

Синтаксис `impl Trait` работает в простых случаях, но на самом деле он является синтаксическим сахаром для более длинной формы, которая называется границей типажа[1]. Вот как он выглядит:

[1] Граница типажа (*trait bound*) — это ограничение типа или типажа. Например, если граница помещена в аргумент, который функция принимает, то передаваемые этой функции типы должны подчиняться этому ограничению.

```
pub fn notify<T: Summary>(item: T) {
    println!("Срочные новости! {}", item.summarize());
}
```

Эта более длинная форма эквивалентна примеру в предыдущем разделе, но в ней больше слов. Мы помещаем границы типажа с помощью объявления параметра обобщенного типа после двоеточия внутрь угловых скобок.

Синтаксис `impl Trait` удобен и способствует более сжатому коду в простых случаях. В других ситуациях синтаксис границы типажа выражает больше сложности. Например, у нас может быть два параметра, которые реализуют `Summary`. Использование синтаксиса `impl Trait` выглядит следующим образом:

```
pub fn notify(item1: impl Summary, item2: impl Summary) {
```

Если бы мы хотели, чтобы эта функция позволяла `item1` и `item2` иметь разные типы, то использование `impl Trait` было бы уместным (если оба типа реализуют `Summary`). Если нужно, чтобы у обоих параметров был один и тот же тип, то это можно выразить только с помощью границы типажа:

```
pub fn notify<T: Summary>(item1: T, item2: T) {
```

Обобщенный тип `T`, указанный в качестве типа параметров `item1` и `item2`, ограничивает функцию таким образом, что конкретный тип значения, передаваемого в качестве аргумента для `item1` и `item2`, должен быть одинаковым.

Указание нескольких границ типажа с помощью синтаксиса +

Мы также можем указывать несколько границ типажа. Скажем, мы хотим, чтобы функция `notify` использовала дисплейное форматирование для `item`, а также метод `summarize`. В определении `notify` мы указываем, что `item` должен реализовать как `Display`, так и `Summary`. Можно сделать это, используя синтаксис `+`:

```
pub fn notify(item: impl Summary + Display) {
```

Синтаксис `+` также допустим с границами типажа для обобщенных типов:

```
pub fn notify<T: Summary + Display>(item: T) {
```

С двумя указанными границами типажа тело `notify` может вызвать метод `summarize` и использовать `{}` для форматирования параметра `item`.

Более четкие границы типажа с условием where

Использование слишком большого числа границ типажа имеет свои недостатки. У каждого обобщенного типа есть свои границы типажа, поэтому функции с многочисленными параметрами обобщенного типа могут содержать много информации о границах типажа между именем функции и списком ее параметров, что затрудняет чтение сигнатуры функции. По этой причине в языке Rust имеется

альтернативный синтаксис для указания границ типажа внутри условия `where` после сигнатуры функции. Поэтому чтобы не писать:

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: T, u: U) -> i32 {
```

мы можем применить условие `where`:

```
fn some_function<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,
        U: Clone + Debug
{
```

Сигнатура этой функции более четкая: имя функции, список параметров и тип возвращаемого значения находятся близко друг к другу, подобно функции без большого числа границ типажа.

Возвращение типов, реализующих типажи

Мы также можем использовать синтаксис `impl Trait` в позиции `return` для возвращения значения некоего типа, реализующего типаж, как показано здесь:

```
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from("конечно, как вы, наверное, уже знаете, люди"),
        reply: false,
        retweet: false,
    }
}
```

Используя `impl Summary` для возвращаемого типа, мы уточняем, что функция `returns_summarizable` возвращает некий тип, который реализует типаж `Summary`, не называя конкретный тип. В этом случае `returns_summarizable` возвращает тип `Tweet`, но код, вызывающий эту функцию, об этом не знает.

Возможность возвращать тип, описываемый только тем типажом, который им реализуется, особенно полезна в контексте замыканий и итераторов, которые мы рассмотрим в главе 13. Замыкания и итераторы создают типы, о которых знает только компилятор, либо типы, которые очень долго нужно описывать. Синтаксис `impl Trait` позволяет кратко описать, что функция возвращает некий тип, реализующий типаж `Iterator`, без необходимости расписывать очень длинный тип.

Однако вы можете использовать `impl Trait` только в том случае, если возвращаете один тип. Например, код ниже, который возвращает либо `NewsArticle`, либо `Tweet` с возвращаемым типом, описываемым как `impl Summary`, работать не будет:

```
fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        NewsArticle {
            headline: String::from("Пингвины выигрывают Кубок Стэнли!"),
```

```
                location: String::from("Питтсбург, шт. Пенсильвания, США"),
                author: String::from("Айсбург"),
                content: String::from("Питтсбург Пингвинз снова является лучшей
                хоккейной командой в НХЛ."),
            }
        } else {
            Tweet {
                username: String::from("horse_ebooks"),
                content: String::from("конечно, как вы, наверное,
                уже знаете, люди"),
                reply: false,
                retweet: false,
            }
        }
    }
```

Возвращение либо `NewsArticle`, либо `Tweet` не допускается из-за ограничений вокруг того, как реализован синтаксис `impl Trait` в компиляторе. Мы рассмотрим вопрос написания функции с таким поведением в разделе «Использование типажных объектов, допускающих значения разных типов».

Исправление функции largest с помощью границ типажа

Теперь, когда вы знаете, как описывать поведение, которое вы хотите использовать с помощью границы параметра обобщенного типа, давайте вернемся к листингу 10.5, чтобы исправить определение функции `largest`, которая использует параметр обобщенного типа. Последний раз, когда мы пытались выполнить этот код, произошла следующая ошибка:

```
error[E0369]: binary operation `>` cannot be applied to type `T`
 --> src/main.rs:5:12
  |
5 |         if item > largest {
  |            ^^^^^^^^^^^^^^
  |
  = note: an implementation of `std::cmp::PartialOrd` might be missing for `T`
```

В теле функции `largest` мы хотели сравнить два значения типа `T`, используя оператор больше, чем (`>`). Поскольку этот оператор определен как метод по умолчанию для типажа из стандартной библиотеки `std::cmp::PartialOrd`, нам нужно описать `PartialOrd` в границах типажа для `T`, чтобы функция `largest` могла работать в срезах любого типа, которые мы могли бы сравнивать. Не нужно вводить типаж `PartialOrd` в область видимости, потому что он находится в прелюдии. Измените сигнатуру функции `largest`, чтобы она выглядела так[1]:

```
fn largest<T: PartialOrd>(list: &[T]) -> T {
```

[1] `ошибка[E0508]: не получается переместить из типа ' [T]', некопируемый срез`
`ошибка[E0507]: не получается переместить из заимствованного содержимого`

На этот раз при компиляции кода мы получаем другой набор ошибок:

```
error[E0508]: cannot move out of type `[T]`, a non-copy slice
 --> src/main.rs:2:23
  |
2 |     let mut largest = list[0];
  |                       ^^^^^^^
  |                       |
  |                       cannot move out of here
  |                       help: consider using a reference instead: `&list[0]`

error[E0507]: cannot move out of borrowed content
 --> src/main.rs:4:9
  |
4 |     for &item in list.iter() {
  |         ^----
  |         ||
  |         |hint: to prevent move, use `ref item` or `ref mut item`
  |         cannot move out of borrowed content
```

Ключевой строчкой в этой ошибке является `cannot move out of type [T], a non-copy slice`. Работая с необобщенными версиями функции `largest`, мы пытались найти наибольшее значение только типов `i32` или `char`. Как обсуждалось в разделе «Данные только из стека: Copy» (с. 99) такие типы, как `i32` и `char`, которые имеют известный размер, хранятся в стеке, поэтому они реализуют типаж `Copy`. Но когда мы сделали функцию `largest` обобщенной, стало возможным, чтобы список параметров имел в ней типы, которые не реализуют типаж `Copy`. Как следствие, мы не сможем переместить значение из `list[0]` в переменную `largest`, что и приводит к этой ошибке.

Для того чтобы вызывать этот код только с теми типами, которые реализуют типаж `Copy`, мы можем добавить `Copy` в границы типажа для `T`! Листинг 10.15 показывает полный код обобщенной функции `largest`, которая будет компилироваться, пока типы значений в срезе, которые мы передаем внутрь функции, реализуют типажи `PartialOrd` и `Copy`, как это делают типы `i32` и `char`.

Листинг 10.15. Рабочее определение функции largest, которая работает в любом обобщенном типе, реализующем типажи PartialOrd и Copy

src/main.rs

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("Наибольшее число равно {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("Наибольший символ равен {}", result);
}
```

Если мы не хотим ограничивать функцию `largest` типами, которые реализуют типаж `Copy`, можно уточнить, что у `T` есть граница типажа `Clone` вместо `Copy`. Тогда мы могли бы клонировать каждое значение в срезе, чтобы функция `largest` могла иметь владение. Использование функции `clone` означает, что в случае с типами, которые владеют данными кучи, такими как `String`, потенциально выполняется больше операций выделения памяти в куче, и такое выделение будет медленным, если мы работаем с крупными объемами данных.

Еще один способ, которым мы могли бы реализовать функцию `largest`, заключается в том, чтобы указанная функция возвращала ссылку на значение `T` в срезе. Если мы заменим тип возвращаемого значения с `T` на `&T`, тем самым изменив тело функции так, чтобы она возвращала ссылку, то нам не понадобятся границы типажа `Clone` или `Copy`, и мы сможем избежать выделения памяти в куче. Попробуйте реализовать эти альтернативные решения самостоятельно!

Использование границ типажа для условной реализации методов

Используя типаж с блоком `impl`, который применяет параметры обобщенного типа, мы можем условно реализовать методы для типов, реализующие перечисленные типажи. Например, тип `Pair<T>` в листинге 10.16 всегда реализует функцию `new`. Но `Pair<T>` реализует метод `cmp_display` только в том случае, если его внутренний тип `T` реализует типаж `PartialOrd`, позволяющий сравнивать, и типаж `Display`, позволяющий печатать.

Листинг 10.16. Условная реализация методов для обобщенного типа в зависимости от границ типажа

```
use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
```

```
    fn new(x: T, y: T) -> Self {
        Self {
            x,
            y,
        }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("Наибольший член x равен {}", self.x);
        } else {
            println!("Наибольший член y равен {}", self.y);
        }
    }
}
```

Мы также можем условно реализовать типаж для любого типа, который реализует еще один типаж. Реализации типажа в любом типе, удовлетворяющем границам типажа, называются полными реализациями и широко используются в стандартной библиотеке языка Rust. Например, стандартная библиотека реализует типаж `ToString` в любом типе, реализующем типаж `Display`. Блок `impl` в стандартной библиотеке выглядит аналогично коду, приведенному ниже:

```
impl<T: Display> ToString for T {
    // --пропуск--
}
```

Поскольку в стандартной библиотеке есть полная реализация, мы можем вызвать метод `to_string`, определенный типажом `ToString`, для любого типа, реализующего типаж `Display`. Например, можно превратить целые числа в соответствующие им значения типа `String`, потому что целые числа реализуют `Display`:

```
let s = 3.to_string();
```

Полные реализации описаны в документации о типажах в разделе «Разработчики».

Типажи и границы типажа позволяют писать код, который использует параметры обобщенного типа не только чтобы сократить повторы, но и чтобы у обобщенного типа были определенные свойства. Компилятор затем использует информацию о границе типажа и проверяет, что все используемые в коде конкретные типы обеспечивают правильное поведение. В динамически типизированных языках произошла бы ошибка времени выполнения, если бы мы вызвали метод для типа, который этот тип не реализовал. Но Rust смещает эти ошибки на время компиляции, поэтому мы вынуждены устранять проблемы еще до того, как код сможет работать. В дополнение к этому, нам не нужно писать код, который проверяет поведение во время выполнения, потому что мы уже это сделали во время компиляции. Это повышает производительность, и гибкость обобщений сохраняется.

Еще один вид обобщения, который мы уже использовали, называется «жизненный цикл». Он не отвечает за то, чтобы у типа были определенные свойства, но обеспечивает, чтобы ссылки были действительными до тех пор, пока они нам нужны. Давайте посмотрим, как жизненный цикл это делает.

Проверка ссылок с помощью жизненных циклов

В разделе «Ссылки и заимствование» (с. 102) мы не обсудили одну деталь, а именно: у каждой ссылки в Rust есть жизненный цикл, то есть протяженность, в течение которой эта ссылка действительна. В большинстве случаев жизненные циклы выводятся логически неявно, так же как и типы, которые чаще всего выводятся логически. Необходимо аннотировать типы, когда существует возможность нескольких типов. Мы должны аннотировать жизненный цикл схожим образом, когда жизненные циклы у ссылок могут быть взаимосвязаны разными способами. Язык Rust требует, чтобы мы аннотировали связи, применяя параметры обобщенных жизненных циклов, чтобы фактические ссылки, используемые во время выполнения, были безусловно действительными.

Понятие жизненного цикла несколько отличается от инструментов в других языках программирования, что, возможно, делает жизненный цикл в Rust наиболее характерным средством. Хотя в этой главе мы не охватим жизненный цикл в полном объеме, но обсудим часто встречающиеся случаи, когда вы, возможно, будете сталкиваться с синтаксисом жизненного цикла, что позволит вам освоиться с этим понятием.

Предотвращение висячих ссылок с помощью жизненного цикла

Главная цель жизненного цикла — предотвращать висячие ссылки, из-за которых программа ссылается на данные, отличные от тех, на которые она должна ссылаться. Рассмотрим программу из листинга 10.17, которая имеет внешнюю и внутреннюю области видимости.

Листинг 10.17. Попытка использовать ссылку, значение которой вышло из области видимости

```
{
 ❶ let r;

    {
     ❷ let x = 5;
     ❸ r = &x;
 ❹ }

 ❺ println!("r: {}", r);
}
```

ПРИМЕЧАНИЕ

Примеры в листингах 10.17, 10.18 и 10.24 объявляют переменные без указания их начального значения, поэтому имя переменной существует во внешней области видимости. На первый взгляд это может показаться противоречащим тому, что Rust не имеет значений null. Однако если мы попытаемся использовать переменную перед тем, как дать ей значение, то получим ошибку времени компиляции, которая показывает, что Rust действительно не допускает значений null.

Внешняя область видимости объявляет переменную с именем `r` без начального значения ❶, а внутренняя область объявляет переменную с именем `x` с начальным значением `5` ❷. Внутри внутренней области мы пытаемся установить значение `r` как ссылку на `x` ❸. Затем внутренняя область заканчивается ❹, и мы пытаемся напечатать значение в `r` ❺. Этот код не компилируется, потому что значение, на которое `r` ссылается, вышло из области видимости до того, как мы попытаемся его использовать. Вот сообщение об ошибке[1]:

```
error[E0597]: `x` does not live long enough
 --> src/main.rs:7:5
  |
6 |         r = &x;
  |              - borrow occurs here
7 |     }
  |     ^ `x` dropped here while still borrowed
...
10 | }
   | - borrowed value needs to live until here
```

Переменная `x` «живет недостаточно долго». Причина в том, что `x` будет вне области видимости, когда внутренняя область закончится в ❹. Но переменная `r` все еще является действительной для внешней области; поскольку ее область крупнее, мы говорим, что указанная переменная «живет дольше». Если бы Rust позволил этому коду работать, то переменная `r` ссылалась бы на память, которая высвободилась, когда переменная `x` вышла из области видимости, и все, что мы попытались бы сделать с `r`, сработало бы некорректно. Тогда каким образом язык Rust выясняет, что этот код недопустим? Он использует контролер заимствования.

Контролер заимствования

В компиляторе Rust имеется контролер заимствования, который сравнивает области видимости, чтобы выяснить, являются ли все заимствования действительными. В листинге 10.18 приведен тот же код, что и в листинге 10.17, но с аннотациями, показывающими жизненный цикл переменных.

[1] `ошибка[E0597]: `x` живет недостаточно долго`

Листинг 10.18. Аннотации жизненных циклов `r` и `x`, именуемые соответственно `'a` и `'b`

```
{
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: {}", r); //          |
}                         // ---------+
```

Здесь мы аннотировали жизненный цикл `r` как `'a`, а жизненный цикл `x` как `'b`. Как вы видите, внутренний блок `'b` намного меньше внешнего блока `'a`. Компилятор сравнивает размер двух жизненных циклов и видит, что переменная `r` имеет жизненный цикл `'a`, но ссылается на память с жизненным циклом `'b`. Программа отклоняется, потому что `'b` короче `'a`: предмет ссылки не живет так же долго, как ссылка.

В листинге 10.19 этот код исправлен таким образом, что у него нет висячей ссылки и он компилируется без ошибок.

Листинг 10.19. Действительная ссылка, поскольку у данных более продолжительный жизненный цикл, чем у ссылки

```
{
    let x = 5;            // ----------+-- 'b
                          //           |
    let r = &x;           // --+-- 'a  |
                          //   |       |
    println!("r: {}", r); //   |       |
                          // --+       |
}                         // ----------+
```

Здесь переменная `x` имеет жизненный цикл `'b`, который в данном случае больше `'a`. Это означает, что переменная `r` может ссылаться на переменную `x`, потому что компилятор знает, что ссылка в `r` всегда будет действительной, пока переменная `x` является действительной.

Теперь, когда вы знаете, где находится жизненный цикл ссылок и как Rust анализирует жизненный цикл, чтобы ссылки всегда оставались действительными, давайте изучим обобщенные жизненные циклы параметров и возвращаемых значений в контексте функций.

Обобщенные жизненные циклы в функциях

Давайте напишем функцию, которая возвращает более длинный строковый срез из двух. Эта функция будет принимать два строковых среза и возвращать строко-

вый срез. После того как мы реализовали функцию `longest`, код в листинге 10.20 должен вывести:

```
Самая длинная строка равна abcd.
```

Листинг 10.20. Функция main, вызывающая функцию longest для поиска самого длинного строкового среза из двух

src/main.rs

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("Самая длинная строка равна {}", result);
}
```

Обратите внимание, что нужно, чтобы функция брала строковые срезы, то есть ссылки, а не чтобы функция `longest` владела своими параметрами. Мы хотим разрешить этой функции принимать срезы типа `String` (тип, хранящийся в переменной `string1`), а также строковые литералы (то, что содержит переменная `string2`).

Обратитесь к разделу «Строковые срезы в качестве параметров» (с. 114), где в подробностях обсуждается вопрос, почему параметры, используемые в листинге 10.20, — это как раз то, что нужно.

Если мы попытаемся реализовать функцию `longest`, как показано в листинге 10.21, то она компилироваться не будет.

Листинг 10.21. Реализация функции longest, которая возвращает самый длинный строковый срез из двух, но которая пока еще не компилируется

src/main.rs

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Вместо этого происходит следующая ошибка, которая говорит о жизненном цикле[1]:

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:1:33
  |
1 | fn longest(x: &str, y: &str) -> &str {
  |                                 ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the
signature does not say whether it is borrowed from `x` or `y`
```

[1] `ошибка[E0106]: отсутствующий спецификатор жизненного цикла`

Текст справки раскрывает, что возвращаемый тип нуждается в параметре обобщенного жизненного цикла, потому что Rust не может отличить, к чему относится возвращаемая ссылка — к `x` или к `y`. На самом деле мы тоже этого не знаем, потому что блок `if` в теле этой функции возвращает ссылку на `x`, а блок `else` возвращает ссылку на `y`!

Когда мы определяем эту функцию, мы не знаем конкретных значений, которые будут переданы внутрь нее, а потому не знаем, какой случай будет исполняться — `if` или `else`. Нам также неизвестны конкретные жизненные циклы ссылок, которые будут передаваться, поэтому мы не можем посмотреть на области, как это было в листингах 10.18 и 10.19, чтобы выяснить, что ссылка, которую мы возвращаем, всегда будет действительной. Контролер заимствования тоже не может это выяснить, потому что он не знает, как жизненный цикл `x` и `y` соотносится с жизненным циклом возвращаемого значения. Для устранения этой ошибки мы добавим параметры обобщенных жизненных циклов, которые определяют связь между ссылками, благодаря чему контролер заимствования сможет выполнить анализ.

Синтаксис аннотаций жизненных циклов

Аннотации жизненных циклов не изменяют протяженность жизни ссылки. Функции могут принимать ссылки с любым жизненным циклом, уточняя параметр обобщенного жизненного цикла подобно тому, как они принимают любой тип, когда сигнатура конкретизирует параметр обобщенного типа. Аннотации жизненных циклов описывают связи между жизненными циклами нескольких ссылок друг с другом, не влияя на жизненный цикл.

У аннотаций жизненных циклов несколько необычный синтаксис: имена параметров должны начинаться с одной кавычки (`'`). Обычно все они находятся в нижнем регистре и очень короткие, как и обобщенные типы. Большинство людей используют имя `'a`. Мы помещаем аннотации параметров жизненных циклов после знака `&` ссылки, используя пробел для отделения аннотации от типа ссылки.

Вот несколько примеров: ссылка на тип `i32` без параметра жизненного цикла, ссылка на тип `i32`, которая имеет параметр жизненного цикла с именем `'a`, и изменяемая ссылка на тип `i32`, которая также имеет жизненный цикл `'a`.

```
&i32        // ссылка
&'a i32     // ссылка с явно выраженным жизненным циклом
&'a mut i32 // изменяемая ссылка с явно выраженным жизненным циклом
```

Одна аннотация жизненного цикла сама по себе не имеет большого значения, потому что аннотации задуманы для того, чтобы сообщать компилятору, как параметры обобщенных жизненных циклов нескольких ссылок связаны друг с другом. Допустим, у нас есть функция с параметром `first`, который является ссылкой на тип `i32` с жизненным циклом `'a`. Указанная функция также имеет еще один параметр `second`, который является еще одной ссылкой на тип `i32`, у которого тоже есть жизненный цикл `'a`. Аннотации жизненных циклов указывают на то, что обе ссылки, `first` и `second`, должны жить в течение этого обобщенного жизненного цикла.

Аннотации жизненных циклов в сигнатурах функций

Теперь давайте рассмотрим аннотации жизненных циклов в контексте функции `longest`. Как и в случае с параметрами обобщенных типов, нужно объявить параметры обобщенных жизненных циклов внутри угловых скобок, между именем функции и списком параметров. В этой сигнатуре мы хотим выразить ограничение, которое заключается в том, что все ссылки в параметрах и возвращаемое значение должны иметь одинаковый жизненный цикл. Мы дадим жизненному циклу имя `'a` и затем добавим его в каждую ссылку, как показано в листинге 10.22.

Листинг 10.22. Определение функции longest, описывающее, что все ссылки в сигнатуре должны иметь одинаковый жизненный цикл 'a

src/main.rs

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Этот код должен компилироваться и производить желаемый результат, когда мы используем его внутри функции `main` в листинге 10.20.

Сигнатура функции теперь сообщает Rust о том, что для некоторого жизненного цикла `'a` функция берет два параметра, оба они являются строковыми срезами, живущими, по крайней мере, в течение жизненного цикла `'a`. Сигнатура функции также говорит о том, что возвращаемый из функции строковый срез будет жить, по крайней мере, в течение жизненного цикла `'a`. Мы хотим, чтобы Rust обеспечил соблюдение именно этих ограничений. Помните, что, когда мы описываем параметры жизненных циклов в этой сигнатуре функции, мы не изменяем жизненный цикл каких-либо передаваемых или возвращаемых значений. Наоборот, мы описываем, что контролер заимствования должен отклонять любые значения, которые не соответствуют этим ограничениям. Обратите внимание, что функции `longest` не нужно знать точную протяженность жизни `x` и `y`, а только то, что некая область может быть заменена на `'a`, которая будет удовлетворять этой сигнатуре.

Во время аннотирования жизненных циклов в функциях аннотации находятся в сигнатуре функции, а не в теле функции. Rust может анализировать код внутри функции без какой-либо помощи. Но когда у функции есть ссылки на код или ссылки из кода вне этой функции, компилятору становится почти невозможным выяснить жизненный цикл параметров или возвращаемых значений самостоятельно. Всякий раз, когда функция вызывается, жизненные циклы могут быть разными. Вот почему нужно аннотировать жизненный цикл вручную.

Когда мы передаем конкретные ссылки в функцию `longest`, конкретный жизненный цикл, подставляемый вместо `'a`, является частью области видимости `x`, которая частично совпадает с областью `y`. Другими словами, обобщенный жизненный

цикл `'a` получит конкретный жизненный цикл, равный меньшему из жизненных циклов `x` и `y`. Поскольку мы аннотировали возвращаемую ссылку тем же параметром жизненного цикла `'a`, возвращаемая ссылка также будет действительна в течение длины меньшего из жизненных циклов `x` и `y`.

Давайте посмотрим, как аннотации жизненных циклов ограничивают функцию `longest` путем передачи ссылок, у которых разные конкретные жизненные циклы. Листинг 10.23 — это простой пример.

Листинг 10.23. Использование функции longest со ссылками на значения типа String, имеющие разные конкретные жизненные циклы

src/main.rs

```
fn main() {
    let string1 = String::from("длинная строка является длинной");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("Самая длинная строка равна {}", result);
    }
}
```

В этом примере `string1` действительна вплоть до конца внешней области видимости, `string2` действительна вплоть до конца внутренней области, и `result` ссылается на то, что является действительным вплоть до конца внутренней области. Выполните этот код, и вы увидите, что контролер заимствования одобряет этот код; он скомпилирует и выведет:

```
Самая длинная строка равно длинная строка является длинной.
```

Далее, давайте посмотрим на пример, который показывает, что жизненный цикл ссылки в переменной `result` должен быть меньшим жизненным циклом двух аргументов. Мы переместим объявление переменной `result` за пределы внутренней области, но оставим передачу значения переменной `result` внутри области с переменной `string2`. Затем мы перенесем макрокоманду `println!`, которая использует `result` вне внутренней области, после того, как внутренняя область закончилась. Код, приведенный в листинге 10.24, не компилируется.

Листинг 10.24. Попытка использовать переменную result после того, как string2 вышла из области видимости

src/main.rs

```
fn main() {
    let string1 = String::from("длинная строка является длинной");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("Самая длинная строка равна {}", result);
}
```

Когда мы попытаемся скомпилировать этот код, произойдет ошибка:

```
error[E0597]: `string2` does not live long enough
 --> src/main.rs:15:5
   |
14 |         result = longest(string1.as_str(), string2.as_str());
   |                                             ------- borrow occurs here
15 |     }
   |     ^ `string2` dropped here while still borrowed
16 |     println!("The longest string is {}", result);
17 | }
   | - borrowed value needs to live until here
```

Ошибка показывает, что для того, чтобы переменная `result` была действительной для инструкции `println!`, переменная `string2` должна быть действительной вплоть до конца внешней области видимости. Rust знает это, потому что мы аннотировали жизненный цикл параметров функции и возвращаемых значений, используя один и тот же параметр жизненного цикла `'a`.

Глядя на этот код, мы видим, что переменная `string1` длиннее переменной `string2`, и поэтому результат будет содержать ссылку на `string1`. Поскольку `string1` еще не вышла из области видимости, ссылка на `string1` по-прежнему будет действительной для инструкции `println!`. Однако компилятор не способен видеть, что в данном случае ссылка является действительной. Мы уже сообщили Rust, что жизненный цикл ссылки, возвращаемой функцией `longest`, совпадает с меньшим жизненным циклом переданных ссылок. Следовательно, контролер заимствования запрещает код в листинге 10.24, поскольку в нем, возможно, есть недействительная ссылка.

Попробуйте разработать дополнительные эксперименты, в которых изменяются значения и жизненный цикл ссылок, передаваемых в функцию `longest`, а также характер использования возвращаемой ссылки. Выдвиньте предположения о том, смогут ли эти эксперименты пройти проверку контролера заимствования, перед компиляцией; затем проверьте, правы ли вы.

Мышление в терминах жизненных циклов

То, как вам нужно описывать параметры жизненных циклов, зависит от того, что делает функция. Например, если бы мы изменили реализацию функции `longest` так, чтобы она вместо самого длинного строкового среза всегда возвращала первый параметр, нам не нужно было бы указывать жизненный цикл для параметра `y`. Код ниже будет компилироваться:

src/main.rs

```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

В этом примере мы задали параметр жизненного цикла `'a` для параметра `x` и возвращаемого типа, но не для параметра `y`, поскольку жизненный цикл `y` не имеет связи с жизненным циклом `x` или возвращаемым значением.

Во время возвращения ссылки из функции параметр жизненного цикла для возвращаемого типа должен совпадать с параметром жизненного цикла одного из параметров. Если возвращаемая ссылка не ссылается на один из параметров, то она должна ссылаться на значение, созданное внутри этой функции. В итоге получится висячая ссылка, потому что значение выйдет из области видимости в конце функции. Рассмотрим попытку реализации функции `longest`, которая не будет компилироваться:

src/main.rs

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("реально длинная строка");
    result.as_str()
}
```

Даже если мы описали параметр жизненного цикла `'a` для возвращаемого типа, эту реализацию здесь не получится скомпилировать, потому что жизненный цикл возвращаемого значения вообще не связан с жизненным циклом параметров. Вот сообщение об ошибке:

```
error[E0597]: `result` does not live long enough
 --> src/main.rs:3:5
  |
3 |     result.as_str()
  |     ^^^^^^ does not live long enough
4 | }
  | - borrowed value only lives until here
  |
note: borrowed value must be valid for the lifetime 'a as defined on the
function body at 1:1...
 --> src/main.rs:1:1
  |
1 | / fn longest<'a>(x: &str, y: &str) -> &'a str {
2 | |     let result = String::from("really long string");
3 | |     result.as_str()
4 | | }
  | |_^
```

Проблема в том, что переменная `result` выходит из области видимости и очищается в конце функции `longest`. Мы также пытаемся вернуть ссылку на `result` из функции. Мы никак не можем описать параметры жизненных циклов, которые изменили бы висячую ссылку, а компилятор не позволит создать висячую ссылку. В этом случае лучше всего было бы вернуть тип данных, находящийся во владении, а не ссылку, чтобы затем вызывающая функция отвечала за очистку значения.

В конечном счете, синтаксис жизненных циклов предназначен для соединения жизненных циклов различных параметров и возвращаемых значений функций.

После того как они будут соединены, у языка Rust будет достаточно информации, чтобы разрешить безопасные для памяти операции и запретить операции, которые могли бы создать висячие указатели или иным образом нарушить безопасность памяти.

Аннотации жизненных циклов в определениях структур

До сих пор мы определяли структуры только для хранения обладаемых типов. Структуры могут содержать ссылки, но в этом случае потребуется добавить аннотацию жизненного цикла в каждую ссылку в определении структуры. Листинг 10.25 имеет структуру `ImportantExcerpt`, содержащую строковый срез.

Листинг 10.25. Структура, содержащая ссылку, поэтому ее определение нуждается в аннотации жизненного цикла

src/main.rs

```
❶ struct ImportantExcerpt<'a> {
    ❷ part: &'a str,
  }

  fn main() {
    ❸ let novel = String::from("Зовите меня Измаил. Несколько лет тому назад...");
    ❹ let first_sentence = novel.split('.')
          .next()
          .expect("Не смог отыскать '.'");

      ❺ let i = ImportantExcerpt { part: first_sentence };
  }
```

Эта структура имеет одно поле `part`, которое содержит строковый срез, то есть ссылку ❷. Как и в случае с обобщенными типами данных, мы объявляем имя параметра обобщенного жизненного цикла в угловых скобках после имени структуры, поэтому можно использовать параметр жизненного цикла в теле определения структуры ❶. Эта аннотация означает, что экземпляр `ImportantExcerpt` не может пережить ссылку, которую он содержит в поле `part`.

Функция `main` здесь создает экземпляр структуры `ImportantExcerpt` ❺, который содержит ссылку на первое предложение повести `string` ❹, которым владеет переменная `novel` ❸. Данные в `novel` существуют перед тем, как создается экземпляр структуры `ImportantExcerpt`. В дополнение к этому, переменная `novel` не выходит из области видимости, пока из области видимости не выйдет экземпляр структуры `ImportantExcerpt`, поэтому ссылка в экземпляре структуры `ImportantExcerpt` является действительной.

Пропуск жизненного цикла

Вы узнали, что у каждой ссылки есть жизненный цикл и что нужно описывать параметры жизненных циклов для функций или структур, которые используют

ссылки. Однако в листинге 4.9 у нас была функция, снова изображенная в листинге 10.26, которая компилировалась без аннотаций жизненных циклов.

Листинг 10.26. Функция, определенная в листинге 4.9, которая компилировалась без аннотаций жизненных циклов, даже если параметр и возвращаемый тип являются ссылками

src/lib.rs

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

Причина, по которой эта функция компилируется без аннотаций жизненных циклов, имеет свою историю: в ранних версиях (до 1.0) этот код не компилировался, потому что каждая ссылка требовала явно выраженного жизненного цикла. В то время сигнатура функции была бы записана следующим образом:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

Написав большой объем кода, команда разработчиков обнаружила, что в некоторых ситуациях программисты снова и снова вводят одни и те же аннотации жизненных циклов. Эти ситуации были предсказуемы и подчинялись нескольким детерминированным паттернам. Разработчики запрограммировали эти паттерны в код компилятора, чтобы контролер заимствования мог логически выводить жизненный цикл в этих ситуациях и не нуждался в явных аннотациях.

Эта часть истории языка Rust имеет актуальное значение, потому что существует возможность, что появятся более детерминированные паттерны, которые добавят в компилятор. Возможно, в будущем еще меньше аннотаций жизненных циклов будут обязательными.

Паттерны, запрограммированные в анализ ссылок языка Rust, называются «правила пропуска жизненного цикла». Это не те правила, которым должны следовать программисты. Это набор частных случаев, которые рассмотрит компилятор, и если код укладывается в эти случаи, то не нужно явно прописывать жизненный цикл.

Правила пропуска не обеспечивают полного логического вывода. Если Rust детерминированно применяет эти правила, но все еще существует неопределенность относительно того, какой жизненный цикл есть у ссылок, то компилятор не догадается, каким должен быть жизненный цикл у остальных ссылок. В этом случае, не строя догадок, компилятор выдаст ошибку. Ее можно устранить, доба-

вив аннотации жизненных циклов, которые описывают то, как ссылки соотносятся друг с другом.

Жизненный цикл для параметров функций или методов называется входным жизненным циклом, а жизненный цикл для возвращаемых значений — выходным жизненным циклом.

Чтобы выяснить, какие жизненные циклы есть у ссылок, когда нет явно заданных аннотаций, компилятор использует три правила. Первое правило применяется к входным жизненным циклам, а второе и третье — к выходным. Если компилятор дойдет до конца этих трех правил и все еще останутся ссылки, для которых он не может выяснить жизненный цикл, то компилятор остановится с ошибкой. Эти правила применимы к определениям `fn`, а также к блокам `impl`.

Первое правило состоит в том, что каждый параметр, являющийся ссылкой, получает собственный параметр жизненного цикла. Другими словами, функция с одним параметром получает один параметр жизненного цикла: `fn foo<'a>(x: &'a i32)`; функция с двумя параметрами получает два отдельных параметра жизненного цикла: `fn foo<'a, 'b>(x: &'a i32, y: &'b i32)` и так далее.

Второе правило состоит в том, что, если имеется ровно один параметр входного жизненного цикла, то этот параметр жизненного цикла назначается всем параметрам выходных жизненных циклов: `fn foo<'a>(x: &'a i32) -> &'a i32`.

Третье правило заключается в том, что, если имеется несколько параметров входных жизненных циклов, но один из них является `&self` или `&mut self`, так как это метод, то жизненный цикл параметра `self` назначается всем параметрам выходных жизненных циклов. Это третье правило делает методы гораздо удобнее для чтения и записи, поскольку требуется меньше символов.

Давайте представим, что мы — компилятор. Мы применим эти правила, чтобы выяснить продолжительность жизненного цикла ссылок в сигнатуре функции `first_word` в листинге 10.26. Сигнатура начинается без каких-либо жизненных циклов, связанных со ссылками:

```
fn first_word(s: &str) -> &str {
```

Затем компилятор применяет первое правило, которое описывает, что каждый параметр получает свой собственный жизненный цикл. Как обычно, мы назовем его `'a`, поэтому теперь сигнатура будет такой:

```
fn first_word<'a>(s: &'a str) -> &str {
```

Второе правило применяется потому, что существует ровно один входной жизненный цикл. Второе правило описывает, что жизненный цикл одного входного параметра назначается выходному жизненному циклу, поэтому сигнатура теперь будет выглядеть так:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

Теперь все ссылки в этой сигнатуре функции имеют жизненный цикл, и компилятор может продолжать анализ. Участие программиста, который аннотировал бы жизненный цикл в сигнатуре этой функции, не требуется.

Давайте посмотрим еще на один пример, на этот раз используя функцию `longest`, которая не имела никаких параметров жизненных циклов, когда мы начали работать с ней в листинге 10.21:

```
fn longest(x: &str, y: &str) -> &str {
```

Давайте применим первое правило: каждый параметр получает собственный жизненный цикл. На этот раз у нас два параметра вместо одного, поэтому имеется два жизненных цикла:

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

Вы видите, что второе правило не применяется, потому что существует более одного входного жизненного цикла. Третье правило тоже неприменимо, потому что `longest` является функцией, а не методом, поэтому ни один из параметров не является параметром `self`. Поработав со всеми тремя правилами, мы все еще не выяснили продолжительность жизненного цикла возвращаемого типа. Вот почему мы получили ошибку, пытаясь скомпилировать код в листинге 10.21: компилятор работал с правилами исключения жизненного цикла, но так и не смог вычислить все жизненные циклы ссылок в сигнатуре.

Поскольку третье правило применяется только в сигнатурах методов, далее мы рассмотрим жизненный цикл в данном контексте, чтобы увидеть, почему третье правило означает, что нам не нужно очень часто аннотировать жизненный цикл в сигнатурах методов.

Аннотации жизненных циклов в определениях методов

Когда мы реализуем методы в структуре с жизненным циклом, то используем тот же синтаксис, что и для параметров обобщенного типа, показанный в листинге 10.11. То, где мы объявляем и используем параметры жизненных циклов, зависит от того, связаны ли они с полями структуры либо с параметрами метода и возвращаемыми значениями.

Имена жизни для полей структуры всегда должны объявляться после ключевого слова `impl` и затем использоваться после имени структуры, потому что эти жизненные циклы являются частью типа структуры.

В сигнатурах методов внутри блока `impl` ссылки могут быть привязаны к жизненному циклу ссылок в полях структуры, либо они могут быть независимыми. В дополнение к этому, правила исключения жизненного цикла приводят к тому, что аннотации жизненных циклов не нужны в сигнатурах методов. Давайте посмотрим на несколько примеров использования структуры `ImportantExcerpt`, которую мы определили в листинге 10.25.

Сначала мы будем использовать метод с именем `level`, единственным параметром которого является ссылка на `self`, а его возвращаемое значение — значение типа `i32`, которое не является ссылкой на что-либо:

```
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}
```

Объявление параметра жизненного цикла после `impl` и использование после имени типа является обязательным, но от нас не требуется аннотировать жизненный цикл ссылки на параметр `self` из-за первого правила элизии.

Вот пример, где применяется третье правило пропуска жизненного цикла:

```
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) -> &str {
        println!("Пожалуйста, внимание: {}", announcement);
        self.part
    }
}
```

Здесь имеется два входных жизненных цикла, поэтому компилятор применяет первое правило пропуска жизненного цикла и дает как `&self`, так и `announcement` их собственные жизненные циклы. Затем, поскольку одним из параметров является `&self`, возвращаемый тип получает жизненный цикл `&self`, и все жизненные циклы будут вычислены.

Статический жизненный цикл

Мы должны обсудить один особый жизненный цикл — `'static`, который обозначает всю продолжительность программы. Все строковые литералы имеют жизненный цикл `'static`, который мы обозначаем следующим образом:

```
let s: &'static str = "У меня статический жизненный цикл.";
```

Текст этого строкового значения хранится непосредственно в двоичном файле программы, который всегда доступен. Следовательно, жизненным циклом всех строковых литералов является `'static`.

Вы, возможно, увидите рекомендации использовать жизненный цикл `'static` в сообщениях об ошибках. Но прежде чем указать `'static` в качестве жизненного цикла для ссылки, подумайте о том, живет ли ваша ссылка в действительности все жизненные циклы программы или нет. Вы, возможно, задумаетесь над тем, хотите ли вы, чтобы она жила так долго, даже если бы смогла. В большинстве случаев проблема возникает из-за попытки создать висячую ссылку или из-за несовпадения имеющихся жизненных циклов. В таких случаях решение заключается в устранении этих проблем, а не в указании статического жизненного цикла `'static`.

Параметры обобщенного типа, границы типажа и жизненный цикл вместе

Давайте вкратце рассмотрим синтаксис описания параметров обобщенного типа, границ типажа и жизненных циклов в одной функции.

```
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(x: &'a str, y: &'a str, ann: T) -> &'a
str
    where T: Display
{
    println!("Объявление! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Выше приведена функция `longest` из листинга 10.22, которая возвращает самый длинный строковый срез из двух. Но теперь у нее есть дополнительный параметр с именем `ann` обобщенного типа `T`, который может быть заполнен любым типом, реализующим типаж `Display`, как указано в условии `where`. Этот дополнительный параметр будет напечатан перед тем, как функция сравнит длины строковых срезов, и именно поэтому требуется граница типажа `Display`. Поскольку жизненный цикл является типом обобщения, объявления параметра жизненного цикла `'a` и параметра обобщенного типа `T` помещаются в один и тот же список внутри угловых скобок после имени функции.

Итоги

В этой главе мы узнали много нового! Теперь, когда вам известны параметры обобщенного типа, типажи и границы типажа, а также параметры обобщенных жизненных циклов, вы готовы писать код без повторов, который работает в различных ситуациях. Параметры обобщенного типа позволяют применять код к разным типам. Типажи и границы типажа обеспечивают, чтобы, даже если типы являются обобщенными, они имели свойства, необходимые коду. Вы узнали, как использовать аннотации жизненных циклов, чтобы у гибкого кода не было висячих ссылок. И весь этот анализ происходит во время компиляции, что не влияет на производительность времени выполнения!

Вы не поверите, но нам предстоит еще многое узнать по темам, которые мы обсудили в этой главе: глава 17 посвящена типажным объектам, которые представляют собой еще один способ использования типажей. В главе 19 рассказывается о более сложных сценариях, сопряженных с аннотациями жизненных циклов, а также о нескольких расширенных языковых средствах системы типов. А в следующей главе вы научитесь писать тесты на Rust, чтобы ваш код работал должным образом.

11 Автоматизированные тесты

В своем эссе 1972 года «Смиренный программист» (*The Humble Programmer*) Эдсгер Дейкстра сказал, что «тестирование может показать наличие дефектов в программе, но не доказать их отсутствие». Однако это вовсе не означает, что мы не должны пытаться тестировать как можно больше!

Правильность в программах подразумевает, до какой степени код делает то, что мы от него хотим. При разработке языка Rust большое внимание уделено корректности программ, но корректность — это сложное понятие, ее не так легко доказать. Система типов Rust берет на себя огромную часть этого бремени, но система типов не способна отлавливать все виды некорректной работы. По этой причине Rust включает в себя поддержку написания автоматизированных тестов.

Например, мы пишем функцию `add_two`, которая прибавляет 2 к любому числу, которое ей передается. Сигнатура этой функции на входе принимает целое число в качестве параметра и возвращает целое число на выходе. Когда мы реализуем и компилируем эту функцию, Rust выполняет всю проверку типов и заимствований, которую вы уже изучили, чтобы, например, мы не передавали в эту функцию значение типа `String` или недействительную ссылку. Но Rust не способен проверить, что эта функция будет делать именно то, что мы хотим, а именно возвращать параметр плюс 2, а не, скажем, параметр плюс 10 или параметр минус 50! Вот тутто и вступают в игру тесты.

Мы можем написать тесты, которые проверяют, например, что при передаче в функцию `add_two` числа `3` возвращаемое значение равно числу `5`. Можно выполнять эти тесты всякий раз, когда мы вносим в код изменения, чтобы убедиться, что любое существующее правильное поведение не изменилось.

Тестирование представляет собой сложный навык. Хотя мы не можем охватить в одной главе все правила написания хороших тестов, мы обсудим механику средств тестирования в Rust. Мы поговорим об аннотациях и макрокомандах, имеющихся в вашем распоряжении при написании тестов, поведении по умолчанию и параметрах для выполнения тестов, а также о том, как организовывать модульные и интеграционные тесты.

Как писать тесты

Тесты — это функции, которые проверяют, что не-тестовый код функционирует ожидаемым образом.

Тела тестовых функций обычно выполняют следующие действия:

1. Настраивают необходимые данные или состояние.
2. Выполняют код, который вы хотите протестировать.
3. Подтверждают, что результаты соответствуют ожиданиям.

Давайте рассмотрим специальные средства для написания тестов, выполняющих эти действия, включая атрибут `test`, несколько макрокоманд и атрибут `should_panic`.

Анатомия функции тестирования

В самом простом случае тест — это функция, которая аннотируется атрибутом `test`. Атрибуты — это метаданные о фрагментах кода на языке Rust; одним из примеров является атрибут `derive`, который мы использовали со структурами в главе 5. Для того чтобы превратить функцию в тестовую, добавьте `#[test]` в строке кода перед `fn`. Когда вы выполняете тесты с помощью команды `cargo test`, Rust создает двоичный исполнитель теста, который выполняет функции, помеченные атрибутом `test`, и сообщает результаты прохождения проверки каждой тестовой функцией.

Когда мы создаем новый библиотечный проект с помощью Cargo, автоматически генерируется тестовый модуль с тестовой функцией внутри. Этот модуль помогает писать тесты, благодаря чему вам не придется искать точную структуру и синтаксис тестовых функций всякий раз, когда вы приступаете к новому проекту. Вы можете добавить столько дополнительных тестовых функций и тестовых модулей, сколько захотите!

Мы познакомимся с несколькими аспектами процесса работы тестов, экспериментируя с образцом теста, сгенерированным без фактического тестирования какого-либо кода. Затем мы напишем несколько реальных тестов, которые вызовут некий написанный нами код и подтвердят, что он работает корректно.

Давайте создадим новый библиотечный проект под названием `adder` («сумматор»):

```
$ cargo new adder --lib
    Created library `adder` project
$ cd adder
```

Содержимое файла src/lib.rs в библиотеке `adder` должно выглядеть так, как показано в листинге 11.1.

Листинг 11.1. Тестовый модуль и функция, автоматически генерируемые командой cargo new

src/lib.rs

```
#[cfg(test)]
mod tests {
  ❶ #[test]
    fn it_works() {
      ❷ assert_eq!(2 + 2, 4);
    }
}
```

Пока давайте проигнорируем две верхние строчки и сосредоточимся на функции, чтобы увидеть, как она работает. Обратите внимание на аннотацию `#[test]` ❶: этот атрибут указывает на то, что функция является тестовой, поэтому исполнитель теста знает, что эту функцию нужно рассматривать как тест. В модуле `tests` также могут быть не-тестовые функции, помогающие настраивать общие сценарии или выполнять часто встречающиеся операции. Именно поэтому нам нужно указывать функции, которые являются тестами, используя атрибут `#[test]`.

Тело функции использует макрокоманду `assert_eq!` ❷ для утверждения о том, что 2 + 2 равно 4. Это проверочное утверждение служит примером формата для типичного теста. Давайте выполним его, чтобы увидеть, что этот тест проходит.

Команда `cargo test` выполняет все тесты в проекте, как показано в листинге 11.2.

Листинг 11.2. Данные, полученные в результате выполнения автоматически сгенерированного теста

```
$ cargo test
    Compiling adder v0.1.0 (file:///projects/adder)
     Finished dev [unoptimized + debuginfo] target(s) in 0.22 secs
      Running target/debug/deps/adder-ce99bcc2479f4607

❶ running 1 test
❷ test tests::it_works ... ok

❸ test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

❹ Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Пакетный менеджер Cargo скомпилировал и выполнил тест. После строчек `Compiling`, `Finished` и `Running` идет строчка `running 1 test` ❶. Следующая далее строчка показывает имя сгенерированной тестовой функции под названием `it_works` и результат выполнения этого теста — `ok` ❷. Далее появляется итоговая сводка выполнения тестов. Текст `test result: ok.` ❸ означает, что все тесты прошли, и часть, которая читается как `1 passed; 0 failed`, подытоживает число тестов, которые прошли успешно или не сработали.

Поскольку у нас нет тестов, которые мы пометили как проигнорированные, сводка показывает `0 ignored`. Мы также не фильтровали выполняемые тесты, поэтому в конце сводки показано `0 filtered out`. Мы поговорим об игнорировании и фильтрации тестов в разделе «Контроль выполнения тестов».

Статистический показатель `0 measured` предназначен для сравнительных тестов (эталонных тестов, или бенчмарков), которые измеряют производительность. На момент написания этой статьи сравнительные тесты доступны только в Nightly Rust. За дополнительной информацией по сравнительным тестам обратитесь к соответствующей документации по адресу https://doc.rust-lang.org/nightly/unstable-book/library-features/test.html.

Следующая часть тестовых данных, которая начинается с `Doc-tests adder` ❹, предназначена для результатов любых документационных тестов. У нас пока нет документационных тестов, но Rust может скомпилировать любые примеры кода, которые есть в документации об API. Это языковое средство помогает синхронизировать документы и код. Мы обсудим способ написания документационного теста в разделе «Документационные комментарии в качестве тестов» (с. 345). А пока мы проигнорируем данные `Doc-tests`.

Давайте изменим имя теста и посмотрим, как это изменит тестовые данные. Измените имя функции `it_works` на другое, к примеру, на `exploration`:

src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }
}
```

Затем снова выполните `cargo test`. Данные теперь показывают `exploration` вместо `it_works`:

```
running 1 test
test tests::exploration ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
```

Давайте добавим еще один тест, но на этот раз мы сделаем тот, который не сработает! Тесты неуспешны, когда что-то в тестовой функции поднимает панику. Каждый тест выполняется в новом потоке исполнения, и когда главный поток видит, что тестовый поток остановился, тест помечается как неуспешный. В главе 9 мы говорили о самом простом способе вызвать панику, который предусматривает вызов макрокоманды `panic!`. Введите новый тест, `another`, в результате чего файл src/lib.rs должен выглядеть, как в листинге 11.3.

Листинг 11.3. Добавление второго теста, который провалится, потому что мы вызываем макрокоманду panic!

src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }
    #[test]
    fn another() {
        panic!("Сделать этот тест неуспешным");
    }
}
```

Снова выполните тесты с помощью `cargo test`. Данные должны выглядеть, как в листинге 11.4, они показывают, что тест `exploration` оказался успешным, а тест `another` — нет.

Листинг 11.4. Результаты тестирования: один тест успешен, другой — нет

```
running 2 tests
test tests::exploration ... ok
❶ test tests::another ... FAILED

❷ failures:

---- tests::another stdout ----
    thread 'tests::another' panicked at 'Сделать этот тест неуспешным',
            src/lib.rs:10:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

❸ failures:
      tests::another

❹ test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out

error: test failed
```

Вместо `ok` строчка `test tests::another` показывает `FAILED` ❶. Между отдельными результатами и сводкой появляются два новых раздела: первый раздел ❷ показывает подробную причину провала каждого теста. В данном случае `another` не сработал, потому что он поднял панику в точке `'Сделать этот тест неуспешным'`, что произошло в строке 10 кода в файле src/lib.rs. В следующем разделе ❸ перечислены только названия всех неуспешных тестов, что полезно, когда тестов много и много подробных данных неуспешных тестов. Мы можем использовать название неуспешного теста, чтобы выполнить только этот тест и с легкостью отладить его. Подробнее о способах выполнения тестов мы поговорим в разделе «Контроль выполнения тестов».

Итоговая строчка выводится в конце ❹: в целом результат теста был неуспешным, FAILED. У нас был один тест, который завершился успешно, и один тест, который не сработал.

Теперь, когда вы увидели, как выглядят результаты теста в разных сценариях, давайте посмотрим на несколько макрокоманд, помимо `panic!`, которые широко используются в тестах.

Проверка результатов с помощью макрокоманды assert!

Макрокоманда `assert!`, предусмотренная стандартной библиотекой, полезна, когда вы хотите убедиться, что какое-то условие в тесте принимает значение `true`. Мы даем макрокоманде `assert!` аргумент, который вычисляется как булев. Если значение является истинным, то макрокоманда `assert!` ничего не делает, и тест успешен. Если значение является ложным, то макрокоманда `assert!` вызывает макрокоманду `panic!`, которая приводит к провалу теста. Использование макрокоманды `assert!` помогает нам проверить, что код функционирует именно так, как мы предполагаем.

В листинге 5.15 мы использовали структуру `Rectangle` и метод `can_hold`, которые приводятся повторно в листинге 11.5. Давайте поместим этот код в файл src/lib.rs и напишем для него несколько тестов с использованием макрокоманды `assert!`.

Листинг 11.5. Использование структуры Rectangle и ее метода can_hold из главы 5

src/lib.rs

```
#[derive(Debug)]
pub struct Rectangle {
    length: u32,
    width: u32,
}

impl Rectangle {
    pub fn can_hold(&self, other: &Rectangle) -> bool {
        self.length > other.length && self.width > other.width
    }
}
```

Метод `can_hold` возвращает булев тип — это значит, что такой вариант использования идеален для макрокоманды `assert!`. В листинге 11.6 мы пишем тест, который использует метод `can_hold` путем создания экземпляра структуры `Rectangle` с длиной 8 и шириной 7 и утверждения, что он может содержать другой экземпляр структуры `Rectangle` с длиной 5 и шириной 1.

Листинг 11.6. Тест для метода `can_hold`, который проверяет, может ли больший прямоугольник вместить меньший прямоугольник

src/lib.rs

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

    #[test]
  ❷ fn larger_can_hold_smaller() {
```

```
        ❸ let larger = Rectangle { length: 8, width: 7 };
          let smaller = Rectangle { length: 5, width: 1 };

        ❹ assert!(larger.can_hold(&smaller));
      }
}
```

Обратите внимание, что мы добавили новую строку кода внутрь модуля `tests`: `use super::*;` ❶. Модуль `tests` — это обычный модуль, который подчиняется обычным правилам видимости, описанным в разделе «Пути для ссылки на элемент в дереве модулей» (с. 151). Поскольку модуль `tests` внутренний, необходимо ввести тестируемый код во внешнем модуле в область видимости внутреннего модуля. Здесь мы используем оператор glob `*`, поэтому все, что мы определяем во внешнем модуле, доступно этому модулю `tests`.

Мы назвали тест `larger_can_hold_smaller` ❷ и создали два экземпляра структуры `Rectangle`, которые нам нужны ❸. Затем мы вызвали макрокоманду `assert!` и передали ей результат `larger_can_hold_smaller(&smaller)` ❹. Предполагается, что это выражение вернет `true`, поэтому тест должен быть успешным. Давайте выясним это!

```
running 1 test
test tests::larger_can_hold_smaller ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Он действительно успешен! Давайте добавим еще один тест, на этот раз подтверждающий, что меньший прямоугольник не вместит больший прямоугольник:

src/lib.rs

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

    #[test]
    fn larger_can_hold_smaller() {
        // --пропуск--
    }

    #[test]
    fn smaller_cannot_hold_larger() {
        let larger = Rectangle { length: 8, width: 7 };
        let smaller = Rectangle { length: 5, width: 1 };

        assert!(!smaller.can_hold(&larger));
    }
}
```

Поскольку правильный результат функции `can_hold` в этом случае является ложным, мы должны инвертировать этот результат перед его передачей в макрокоманду `assert!`. В результате тест будет успешным, если `can_hold` возвращает `false`:

```
running 2 tests
test tests::smaller_cannot_hold_larger ... ok
test tests::larger_can_hold_smaller ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Оба теста успешны! Теперь давайте посмотрим, что происходит с результатами тестирования, когда мы вводим в код ошибку. Давайте изменим реализацию метода `can_hold`, заменив знак «больше» на «меньше», когда он сравнивает длины:

```
// --пропуск--

impl Rectangle {
    pub fn can_hold(&self, other: &Rectangle) -> bool {
        self.length < other.length && self.width > other.width
    }
}
```

В результате выполнения тестов теперь мы получаем данные, как показано ниже:

```
running 2 tests
test tests::smaller_cannot_hold_larger ... ok
test tests::larger_can_hold_smaller ... FAILED

failures:

---- tests::larger_can_hold_smaller stdout ----
    thread 'tests::larger_can_hold_smaller' panicked at 'assertion failed:
    larger.can_hold(&smaller)', src/lib.rs:22:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::larger_can_hold_smaller

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

Тесты зафиксировали ошибку! Так как `larger.length` равно 8, а `smaller.length` равно 5, сравнение длин в `can_hold` теперь возвращает `false`: `8` не меньше `5`.

Проверка равенства с помощью макрокоманд assert_eq! и assert_ne!

Широко распространенный способ тестирования функциональности — сравнение результата тестируемого кода со значением, которое он должен вернуть, чтобы проверить, что они равны. Вы можете сделать это с помощью макрокоманды `assert!`, передав ей выражения с оператором `==`. Однако этот тест встречается настолько часто, что стандартная библиотека предоставляет пару макрокоманд — `assert_eq!` и `assert_ne!`, — делая выполнение его удобнее. Эти макрокоманды сравнивают два аргумента на предмет равенства или неравенства соответственно. Они также напечатают эти два значения, если утверждение не сработает, что

облегчает понимание того, почему тест завершился провалом. С другой стороны, макрокоманда `assert!` указывает только на то, что она получила значение `false` для выражения `==`, а не значения, которые приводят к `false`.

В листинге 11.7 мы пишем функцию `add_two`, которая прибавляет 2 к своему параметру и возвращает результат. Затем мы тестируем эту функцию с помощью макрокоманды `assert_eq!`.

Листинг 11.7. Тестирование функции `add_two` с помощью макрокоманды `assert_eq!`

src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

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

    #[test]
    fn it_adds_two() {
        assert_eq!(4, add_two(2));
    }
}
```

Давайте проверим, успешен ли он!

```
running 1 test
test tests::it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Первый аргумент, который мы передали макрокоманде `assert_eq!`, `4`, равен результату вызова `add_two(2)`. К этому тесту относится строка `test tests::it_adds_two ... ok`, и текст `ok` указывает на то, что тест успешный!

Давайте внесем в код ошибку и посмотрим, как выглядит результат, когда тест, который использует макрокоманду `assert_eq!`, не срабатывает. Измените реализацию функции `add_two` так, чтобы теперь она прибавляла `3`:

```
pub fn add_two(a: i32) -> i32 {
    a + 3
}
```

Снова выполните тесты:

```
running 1 test
test tests::it_adds_two ... FAILED

failures:

---- tests::it_adds_two stdout ----
```

```
      ❶ thread 'tests::it_adds_two' panicked at 'assertion failed:
`(left == right)`
❷  left: `4`,
❸ right: `5`', src/lib.rs:11:8
  note: Run with `RUST_BACKTRACE=1` for a backtrace.

  failures:
      tests::it_adds_two

  test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

Тест зафиксировал ошибку! Тест `it_adds_two` не сработал, показав сообщение `assertion failed: `(left == right)`` ❶ и указав, что `left` было равно `4` ❷, а `right` было равно `5` ❸. Это полезное сообщение, оно помогает начать отладку: сообщение означает, что левый аргумент макрокоманды `assert_eq!` был равен `4`, а правый аргумент, где у нас было `add_two(2)`, был равен `5`.

Обратите внимание, что в некоторых языках и тестовых каркасах параметры функций, которые утверждают, что два значения равны, называются «ожидаемыми» (`expected`) и «фактическими» (`actual`), и порядок, в котором мы указываем аргументы, имеет значение. Однако в Rust они называются «левыми» (`left`) и «правыми» (`right`), и порядок, в котором мы описываем ожидаемое значение и значение, которое производит тестируемый код, не важен. Мы могли бы написать проверочное утверждение в этом тесте как `assert_eq!(add_two(2), 4)`, что приведет к выводу сообщения `assertion failed: `(left == right)`` и к указанию, что левый аргумент был равен `5`, а правый аргумент был равен `4`.

Макрокоманда `assert_ne!` завершится успешно, если два значения, которые мы ей передаем, не равны, и не сработает, если они равны. Указанная макрокоманда наиболее полезна в тех случаях, когда мы не уверены, каким будет значение, но знаем, каким значение точно не будет, если код функционирует так, как мы предполагаем. Например, если мы тестируем функцию, которая гарантированно каким-либо образом изменит свои данные на входе, но способ их изменения зависит от дня недели, когда выполняются тесты, то лучше всего удостовериться, что выход из функции не равен входу.

Неявно обе макрокоманды — `assert_eq!` и `assert_ne!` — используют операторы `==` и `!=` соответственно. Когда проверочные утверждения не срабатывают, эти макрокоманды печатают аргументы с помощью отладочного форматирования — это означает, что сравниваемые значения должны реализовать типажи `PartialEq` и `Debug`. Все примитивные типы и большинство типов стандартной библиотеки реализуют эти типажи. Для определяемых вами структур и перечислений нужно будет реализовать `PartialEq`, что позволит сравнить, равны или не равны значения этих типов. Для того чтобы печатать значения, когда проверочное утверждение проходит неуспешно, нужно будет реализовать `Debug`. Поскольку оба типажа генерируемы, как указано в листинге 5.12, обычно необходимо добавить простую аннотацию `#[derive(PartialEq, Debug)]` в определение структуры или перечисления. Дополнительные сведения об этих и других генерируемых типажах см. в приложении Б.

Добавление сообщений об ошибках для пользователя

Вы также можете добавлять сообщения для пользователя, которые будут напечатаны вместе с сообщением об ошибке в качестве необязательных аргументов для макрокоманд `assert!`, `assert_eq!` и `assert_ne!`. Любые аргументы, которые указываются после одного обязательного аргумента для `assert!` или двух обязательных аргументов для `assert_eq!` и `assert_ne!`, передаются вместе с макрокомандой `format!` (см. раздел «Конкатенация с помощью оператора + или макрокоманды format!» (с. 177)), так что вы можете передать форматную строку, содержащую заполнители `{}` и значения для вставки внутрь этих заполнителей. Сообщения для пользователя полезны, чтобы фиксировать значение проверочного утверждения. Если тест не сработает, у вас будет более четкое представление, в чем проблема с кодом.

Допустим, у нас есть функция, которая приветствует людей по имени, и мы хотим проверить, что имя, которое мы передаем в функцию, появляется на выходе из нее:

src/lib.rs

```
pub fn greeting(name: &str) -> String {
    format!("Здравствуй {}!", name)
}

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

    #[test]
    fn greeting_contains_name() {
        let result = greeting("Кэрол");
        assert!(result.contains("Кэрол"));
    }
}
```

Требования к этой программе еще не были согласованы, и мы почти уверены, что текст `Здравствуй` в начале функции `greeting` изменится. Мы решили, что не хотим обновлять тест при изменении требований, поэтому вместо проверки точного равенства значению, возвращаемому функцией `greeting`, мы просто будем проверять, что данные на выходе содержат текст параметра на входе.

Давайте внесем в этот код ошибку, изменив функцию `greeting`, не включив `name`, и посмотрим, как не сработает этот тест:

```
pub fn greeting(name: &str) -> String {
    String::from("Здравствуй!")
}
```

Выполнение этого теста приводит к следующему результату:

```
running 1 test
test tests::greeting_contains_name ... FAILED

failures:
```

```
---- tests::greeting_contains_name stdout ----
        thread 'tests::greeting_contains_name' panicked at 'assertion failed:
result.contains("Кэрол")', src/lib.rs:12:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::greeting_contains_name
```

Этот результат просто указывает, что проверочное утверждение не прошло и на какой строке оно находится. Более полезное сообщение об ошибке в этом случае будет печатать значение, которое мы получили из функции `greeting`. Давайте изменим тестовую функцию, дав ей сообщение об ошибке для пользователя, состоящее из форматной строки с заполнителем, имеющим фактическое значение, полученное из функции `greeting`:

```
#[test]
fn greeting_contains_name() {
    let result = greeting("Кэрол");
    assert!(
        result.contains("Кэрол"),
        "Приветствие не содержало имя, предоставлено значение `{}`", result
    );
}
```

Теперь, выполнив этот тест, мы получим более информативное сообщение об ошибке:

```
---- tests::greeting_contains_name stdout ----
        thread 'tests::greeting_contains_name' panicked at 'Приветствие не
содержало имя, предоставлено значение `Здравствуй!`', src/lib.rs:12:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

В тестовых данных мы видим значение, полученное фактически, которое поможет отладить то, что произошло, а не то, что вероятно произойдет.

Проверка на панику с помощью атрибута should_panic

В дополнение к проверке правильности ожидаемых значений, возвращаемых кодом, также важно проверить, что код обрабатывает условия возникновения ошибки, как мы ожидаем. Например, рассмотрим тип `Guess`, который мы создали в листинге 9.10. Другой код, использующий тип `Guess`, зависит от гарантии того, что экземпляры типа `Guess` будут содержать значения в интервале только между 1 и 100. Мы можем написать тест, который обеспечивает, чтобы попытка создать экземпляр типа `Guess` со значением вне этого интервала поднимала панику.

Мы делаем это, добавляя еще один атрибут, `should_panic`, в тестовую функцию. Этот атрибут делает тест успешным, если код внутри функции паникует. Тест не сработает, если код внутри функции не паникует.

Листинг 11.8 показывает тест, который проверяет, что условия возникновения ошибки функции `Guess::new` наступают тогда, когда мы этого ожидаем.

Листинг 11.8. Проверка того, что условие станет причиной для panic!

src/lib.rs

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Значение догадки должно быть между 1 и 100,
                    получено {}.", value);
        }

        Guess {
            value
        }
    }
}

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

    #[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Мы помещаем атрибут `#[should_panic]` после атрибута `#[test]` и перед тестовой функцией, к которой он применяется. Давайте посмотрим на результат, когда этот тест срабатывает:

```
running 1 test
test tests::greater_than_100 ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Выглядит неплохо! Теперь давайте введем в код ошибку, удалив условие, что функция `new` будет паниковать, если значение больше 100:

```
// --пропуск--

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!("Значение догадки должно быть между 1 и 100,
                    получено {}.", value);
```

```
        }

        Guess {
            value
        }
    }
}
```

Когда мы выполним тест из листинга 11.8, он не сработает:

```
running 1 test
test tests::greater_than_100 ... FAILED

failures:

failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

В этом случае мы получаем не очень полезное сообщение, но, глядя на тестовую функцию, видим, что она аннотирована `#[should_panic]`. Эта ошибка означает, что код в тестовой функции не вызвал паники.

Тесты, использующие `should_panic`, бывают неточными, поскольку они только указывают на то, что код вызвал некую панику. `should_panic` сработает, даже если тест запаникует по другой причине, чем та, которую мы ожидали. Для придания тестам `should_panic` большей точности мы можем добавить необязательный параметр `expected` в атрибут `should_panic`. Тестовая оснастка удостоверится, что сообщение об ошибке содержит предоставленный текст. Рассмотрим модифицированный код для типа `Guess` в листинге 11.9, где функция `new` паникует с разными сообщениями в зависимости от того, является ли значение слишком малым либо слишком большим.

Листинг 11.9. Проверка того, что условие вызовет panic! с особым сообщением о панике

src/lib.rs

```
// --пропуск--
impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!("Значение догадки должно быть больше или равно 1,
                    получено {}.", value);
        } else if value > 100 {
            panic!("Значение догадки должно быть меньше или равно 100,
                    получено {}.", value);
        }

        Guess {
            value
        }
    }
```

```
}

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

    #[test]
    #[should_panic(expected = "Значение догадки должно быть меньше
                               или равно 100")]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Этот тест сработает, потому что значение, которое мы помещаем в параметр `expected` атрибута `should_panic`, является подстрокой сообщения, с которым паникует функция `Guess::new`. Мы могли бы указать все ожидаемое нами сообщение о панике целиком, которое в этом случае было бы `Значение догадки должно быть меньше или равно 100, получено 200`. То, что вы решите указать в параметре `expected` для `should_panic`, зависит от того, какая часть сообщения о панике является уникальной или динамической и насколько точным должен быть тест. В данном случае будет достаточно подстроки сообщения о панике, чтобы убедиться в том, что код в тестовой функции исполняет случай `else if value > 100`.

Для того чтобы увидеть, что происходит, когда тест `should_panic` с ожидаемым сообщением `expected` не срабатывает, давайте снова введем в код ошибку, поменяв местами тела блоков `if value < 1` и `else if value > 100`:

```
if value < 1 {
    panic!("Значение догадки должно быть меньше или равно 100, получено {}.",
           value);
} else if value > 100 {
    panic!("Значение догадки должно быть больше или равно 1, получено {}.",
           value);
}
```

На этот раз, когда мы выполняем тест `should_panic`, он не сработает:

```
running 1 test
test tests::greater_than_100 ... FAILED

failures:

---- tests::greater_than_100 stdout ----
        thread 'tests::greater_than_100' panicked at 'Значение догадки должно
быть больше или равно 1, получено 200.', src/lib.rs:11:12
note: Run with `RUST_BACKTRACE=1` for a backtrace.
note: Panic did not include expected string 'Значение догадки должно быть меньше
или равно 100.'

failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

Сообщение об ошибке указывает на то, что этот тест действительно поднял панику, как мы и предполагали, но в сообщении о панике не было ожидаемой строки `'Значение догадки должно быть меньше или равно 100'`. Сообщение о панике, которое мы получили в этом случае, было `Значение догадки должно быть больше или равно 1, получено 200`. Теперь мы можем заняться вопросом, где находится дефект!

Использование типа Result<T, E> в тестах

До сих пор мы писали тесты, которые паникуют, когда не срабатывают. Мы также можем писать тесты, которые используют тип `Result<T, E>`! Вот тест из листинга 11.1, переписанный так, чтобы использовать тип `Result<T, E>` и возвращать `Err` вместо вызова паники:

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() -> Result<(), String> {
        if 2 + 2 == 4 {
            Ok(())
        } else {
            Err(String::from("два плюс два не равно четырем"))
        }
    }
}
```

Функция `it_works` теперь возвращает тип `Result<(), String>`. В теле функции вместо того, чтобы вызывать макрокоманду `assert_eq!`, мы возвращаем `Ok(())`, когда тест успешен, и `Err` с экземпляром типа `String` внутри, когда тест не срабатывает.

Написание тестов так, чтобы они возвращали тип `Result<T, E>`, позволяет использовать оператор вопросительного знака в теле тестов, что бывает удобно в написании тестов, которые должны закончиться неуспешно, если какая-либо операция в них возвращает вариант `Err`.

Вы не можете применять аннотацию `#[should_panic]` для тестов, использующих тип `Result<T, E>`. Вместо этого вы должны возвращать значение `Err` непосредственно, когда тест должен не сработать.

Теперь, когда вы знаете несколько способов написания тестов, давайте посмотрим, что происходит, когда мы выполняем тесты, и изучим разные варианты, которые можно использовать с командой `cargo test`.

Контроль выполнения тестов

Подобно тому как команда `cargo run` компилирует код, а затем выполняет результирующий двоичный файл, команда `cargo test` компилирует код в тестовом режиме и выполняет результирующий двоичный тест. Вы можете указать аргументы

командной строки, и тем самым изменить поведение по умолчанию для команды `cargo test`. Например, поведение по умолчанию двоичного файла, создаваемого командой `cargo test`, состоит в выполнении всех тестов в параллельном режиме и захвате данных, генерируемых во время выполнения тестов. При этом данные не выводятся на экран, и облегчается чтение данных, связанных с результатами теста.

Одни аргументы командной строки входят в команду `cargo test`, а другие — в результирующий тестовый двоичный файл. Для того чтобы разделить эти два типа аргументов, вы перечисляете те аргументы, которые входят в команду `cargo test`, затем разделитель `--`, а потом те, которые входят в тестовый двоичный файл. Выполнив `cargo test --help`, вы увидите аргументы, которые можно использовать с командой `cargo test`, а выполнив `cargo test -- --help`, вы увидите аргументы, которые можно использовать после разделителя `--`.

Параллельное и последовательное выполнение тестов

Когда вы выполняете несколько тестов, по умолчанию они выполняются параллельно с использованием потоков исполнения. Это означает, что тесты завершат работу быстрее, благодаря чему вы скорее получите обратную связь о том, работает ли код. Поскольку тесты выполняются одновременно, убедитесь, что они не зависят друг от друга или от какого-либо совместного состояния, включая совместную среду, например текущий рабочий каталог или переменные среды.

Например, каждый тест выполняет некий код, который создает файл на диске с именем test-output.txt и пишет в этот файл некие данные. Затем каждый тест читает данные из этого файла и выполняет проверочное утверждение о том, что файл содержит некоторое значение, которое отличается в каждом тесте. Поскольку тесты выполняются одновременно, один тест может переписать файл в промежутке между тем, как другой тест пишет и читает файл. Тогда второй тест не сработает, но не потому, что код является неправильным, а потому, что тесты мешали друг другу во время параллельного выполнения. Одно решение состоит в том, чтобы каждый тест писал в другой файл, а другое — чтобы тесты выполнялись по одному за раз.

Если вы не желаете выполнять тесты параллельно или хотите точнее контролировать число используемых потоков исполнения, вы можете отправить флаг `--test-threads` и число потоков исполнения, которые хотите использовать, в тестовый двоичный файл. Взгляните на следующий пример:

```
$ cargo test -- --test-threads=1
```

Мы установили число тестовых потоков исполнения равным 1, сказав программе не использовать никакого параллелизма. Выполнение тестов с одной нитью исполнения займет больше времени, чем их параллельное выполнение, но тесты не будут мешать друг другу, если они делят между собой совместное состояние.

Показ результатов функции

По умолчанию, если тест успешен, библиотека тестов перехватывает все, что выводится в стандартном выводе данных. Например, если мы вызовем макрокоманду `println!` в тесте, а тест выполнится успешно, то мы не увидим данные работы `println!` в терминале — появится только строчка, которая указывает, что тест успешен. Если тест не сработает, то мы увидим все, что было в стандартном выводе данных вместе с остальной частью сообщения об ошибке.

Например, листинг 11.10 содержит глупую функцию, которая выводит значение своего параметра и возвращает 10, а также успешный и неуспешный тесты.

Листинг 11.10. Тесты для функции, которая вызывает макрокоманду println!

src/lib.rs

```
fn prints_and_returns_10(a: i32) -> i32 {
    println!("Я получил значение {}", a);
    10
}

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

    #[test]
    fn this_test_will_pass() {
        let value = prints_and_returns_10(4);
        assert_eq!(10, value);
    }

    #[test]
    fn this_test_will_fail() {
        let value = prints_and_returns_10(8);
        assert_eq!(5, value);
    }
}
```

Когда мы выполним эти тесты с помощью команды `cargo test`, то увидим данные ниже:

```
running 2 tests
test tests::this_test_will_pass ... ok
test tests::this_test_will_fail ... FAILED

failures:

---- tests::this_test_will_fail stdout ----
     ❶ Я получил значение 8
thread 'tests::this_test_will_fail' panicked at 'assertion failed: `(left ==
right)`
   left: `5`,
  right: `10`', src/lib.rs:19:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

```
failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

Обратите внимание, что в этих данных мы не видим `Я получил значение 4`, которое выводится при успешном выполнении теста. Эти данные были перехвачены. Данные из неуспешного теста — `Я получил значение 8` ❶ — появляются в разделе сводных данных тестирования, которые также показывают причину ошибки теста.

Если мы также в напечатанной форме хотим видеть значения, относящиеся к успешным тестам, то можно отключить поведение перехвата данных с помощью флага `--nocapture`:

```
$ cargo test -- --nocapture
```

Снова выполнив тесты из листинга 11.10 с флагом `--nocapture`, мы увидим следующие данные:

```
running 2 tests
I got the value 4
I got the value 8
test tests::this_test_will_pass ... ok
thread 'tests::this_test_will_fail' panicked at 'assertion failed: `(left ==
right)`
  left: `5`,
 right: `10`', src/lib.rs:19:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.
test tests::this_test_will_fail ... FAILED

failures:

failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

Обратите внимание, что данные тестов и результаты тестирования чередуются, — причина в том, что тесты работают параллельно, как мы уже говорили в предыдущем разделе. Попробуйте применить аргумент `--test-threads=1` и флаг `--nocapture` и посмотрите, как будут выглядеть данные!

Выполнение подмножества тестов по имени

Иногда выполнение полного набора тестов занимает много времени. Если вы работаете с кодом в определенном участке, вам, возможно, потребуется выполнить только те тесты, которые относятся к этому коду. Вы можете выбрать подлежащие выполнению тесты, передав команде `cargo test` имя или имена тестов, которые вы хотите выполнить.

В качестве демонстрации того, как выполнять подмножество тестов, мы создадим три теста для функции `add_two`, как показано в листинге 11.11, и выберем, какие из них нужно выполнить.

Листинг 11.11. Три теста с тремя разными именами

src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

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

    #[test]
    fn add_two_and_two() {
        assert_eq!(4, add_two(2));
    }

    #[test]
    fn add_three_and_two() {
        assert_eq!(5, add_two(3));
    }

    #[test]
    fn one_hundred() {
        assert_eq!(102, add_two(100));
    }
}
```

Если мы выполним тесты без передачи каких-либо аргументов, как мы видели ранее, то все тесты будут выполняться параллельно:

```
running 3 tests
test tests::add_two_and_two ... ok
test tests::add_three_and_two ... ok
test tests::one_hundred ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Выполнение одиночных тестов

Мы можем передать имя любой тестовой функции в команду `cargo test` для выполнения только этого теста:

```
$ cargo test one_hundred
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/deps/adder-06a75b4a1f2515e9

running 1 test
test tests::one_hundred ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out
```

Выполнился только тест с именем `one_hundred`, имена двух других тестов не совпали с этим именем. Данные теста сообщают о том, что у нас было больше тестов, чем то число, которое выполнила команда, показав `2 filtered out` («2 отфильтровано») в конце строки сводки.

Мы не можем указывать имена многочисленных тестов в такой форме, будет использовано только первое значение, переданное в команду `cargo test`. Но есть способ, который позволяет выполнять многочисленные тесты.

Фильтрация с целью выполнения многочисленных тестов

Мы можем указать часть имени теста, и будет выполнен любой тест, имя которого совпадает с этим значением. Поскольку имена двух наших тестов содержат `add`, мы можем их выполнить, запустив команду `cargo test add`:

```
$ cargo test add
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/deps/adder-06a75b4a1f2515e9

running 2 tests
test tests::add_two_and_two ... ok
test tests::add_three_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out
```

Указанная команда выполнила все тесты с `add` в имени и отфильтровала тест с именем `one_hundred`. Также обратите внимание, что модуль, в котором находятся тесты, становится частью имени теста, поэтому мы можем выполнять все тесты в модуле, фильтруя по имени модуля.

Игнорирование нескольких тестов, только если не запрошено иное

Иногда на выполнение нескольких тестов требуется очень много времени, поэтому вы, возможно, захотите исключить их, когда выполняется команда `cargo test`. Не перечисляя в качестве аргументов все тесты, которые вы хотите выполнить, можно аннотировать времязатратные тесты, используя атрибут `ignore` для их исключения, как показано ниже:

src/lib.rs

```
#[test]
fn it_works() {
    assert_eq!(2 + 2, 4);
}

#[test]
#[ignore]
fn expensive_test() {
    // код, выполнение которого занимает один час
}
```

После `#[test]` мы добавляем строку `#[ignore]` в тест, который хотим исключить. Теперь, когда мы запускаем тесты, `it_works` выполняется, а `expensive_test` — нет:

```
$ cargo test
    Compiling adder v0.1.0 (file:///projects/adder)
     Finished dev [unoptimized + debuginfo] target(s) in 0.24 secs
      Running target/debug/deps/adder-ce99bcc2479f4607

running 2 tests
test expensive_test ... ignored
test it_works ... ok

test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out
```

Функция `expensive_test` указана как игнорируемая. Если мы хотим выполнять только игнорируемые тесты, можно применить команду `cargo test -- --ignored`:

```
$ cargo test -- --ignored
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/deps/adder-ce99bcc2479f4607

running 1 test
test expensive_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out
```

Контролируя то, какие тесты выполняются, вы можете сделать так, чтобы результаты команды `cargo test` были быстрыми. Когда вы находитесь в точке, где имеет смысл проверить результаты проигнорированных тестов и у вас есть время на то, чтобы дождаться результатов, то вместо этого вы можете выполнить команду `cargo test -- --ignored`.

Организация тестов

Как уже упоминалось в начале главы, тестирование представляет собой сложную дисциплину и разные люди используют разную терминологию и организацию. Сообщество Rust рассматривает тесты с точки зрения двух главных категорий: модульных тестов и интеграционных тестов. Модульные тесты малы и более сфокусированы на проверке одного модуля в отдельности, они могут тестировать приватные интерфейсы. Интеграционные тесты полностью внешние по отношению к библиотеке и используют ваш код так же, как и любой другой внешний код, применяя только публичный интерфейс и потенциально проверяя несколько модулей в одном тесте.

Важно писать оба вида теста, чтобы разделы библиотеки делали то, что вы хотите, вместе и по отдельности.

Модульные тесты

Назначение модульных тестов (юнит-тестов) — проверять каждую единицу кода отдельно от остальной части кода, чтобы оперативно находить места, где код рабо-

тает, а где не работает как должно. Вы будете размещать модульные тесты в каталоге src в каждом файле вместе с кодом, который они тестируют. В каждом файле принято создавать модуль с именем `tests`, содержащим тестовые функции, и аннотировать модуль с помощью `cfg(test)`.

Модуль tests и #[cfg(test)]

Аннотация `#[cfg(test)]` для модуля `tests` говорит компилятору выполнить тестовый код, только при запуске команды `cargo test`, а не при запуске `cargo build`. Это экономит время компиляции, когда вы хотите только собрать библиотеку и сэкономить место в итоговом скомпилированном артефакте, так как тесты не включены. Вы увидите, что, поскольку интеграционные тесты располагаются в другом каталоге, они не нуждаются в аннотации `#[cfg(test)]`. Но поскольку модульные тесты находятся в тех же файлах, что и код, вы будете использовать `#[cfg(test)]`, чтобы указать, что их не следует включать в скомпилированный результат.

Напомним, что, когда мы создавали новый проект `adder` в первом разделе этой главы, Cargo генерировал код ниже за нас:

src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

Этот код автоматически генерируется тестовым модулем. Атрибут `cfg` расшифровывается как *конфигурация* и сообщает компилятору, что следующий элемент должен быть включен только при заданном варианте конфигурации. В данном случае вариантом конфигурации является `test`, предусмотренный языком для компилирования и выполнения тестов. Используя атрибут `cfg`, Cargo компилирует тестовый код только в том случае, если мы активно выполняем тесты с помощью команды `cargo test`. Сюда входят любые вспомогательные функции, которые могут быть в этом модуле, в дополнение к функциям, аннотированным атрибутом `#[test]`.

Тестирование приватных функций

В сообществе тестирования ведутся споры о том, следует ли тестировать приватные функции напрямую, тогда как другие языки затрудняют или делают невозможным тестирование приватных функций. Независимо от того, какой идеологии тестирования вы придерживаетесь, правила конфиденциальности Rust позволяют тестировать приватные функции. Рассмотрим код в листинге 11.12 с приватной функцией `internal_adder`.

Листинг 11.12. Тестирование приватной функции

src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    internal_adder(a, 2)
}

fn internal_adder(a: i32, b: i32) -> i32 {
    a + b
}

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

    #[test]
    fn internal() {
        assert_eq!(4, internal_adder(2, 2));
    }
}
```

Обратите внимание, что функция `internal_adder` не помечена как `pub`, но, поскольку тесты представляют собой простой код Rust, а модуль `tests` — просто еще один модуль, вы можете ввести `internal_adder` в область видимости теста и вызвать его. Если вы не считаете, что приватные функции должны тестироваться, ничто в Rust не заставит вас это сделать.

Интеграционные тесты

В Rust интеграционные тесты являются полностью внешними по отношению к библиотеке. Они используют библиотеку так же, как и любой другой код, то есть они вызывают только те функции, которые являются частью публичного API вашей библиотеки. Их цель — протестировать правильность работы многих частей библиотеки. Единицы кода, которые работают правильно в отдельности, могут иметь проблемы во время интеграции, поэтому тестовое покрытие интегрированного кода так же важно. Для создания интеграционных тестов сначала необходимо создать каталог tests.

Каталог tests

Мы создаем каталог tests на верхнем уровне каталога нашего проекта, рядом с src. Cargo знает, что нужно искать файлы интеграционных тестов в этом каталоге. Затем мы можем создать в этом каталоге столько тестовых файлов, сколько захотим, и каждый из них будет скомпилирован как отдельная упаковка.

Давайте создадим интеграционный тест. Если код из листинга 11.12 по-прежнему находится в файле src/lib.rs, то создайте каталог tests и новый файл с именем tests/integration_test.rs и введите код из листинга 11.13.

Листинг 11.13. Интеграционный тест функции в упаковке `adder`

tests/integration_test.rs

```
use adder;

#[test]
fn it_adds_two() {
    assert_eq!(4, adder::add_two(2));
}
```

Мы добавили строку `use adder` в верхней части кода, которая не нужна в модульных тестах. Причина в том, что каждый тест в каталоге tests — это отдельная упаковка, поэтому нужно ввести библиотеку в область видимости каждой тестовой упаковки.

Не нужно аннотировать какой-либо код в tests/integration_test.rs с помощью `#[cfg(test)]`. Cargo обрабатывает каталог тестов особым образом и компилирует файлы в этом каталоге только при выполнении команды `cargo test`. Теперь выполните команду `cargo test`:

```
$ cargo test
    Compiling adder v0.1.0 (file:///projects/adder)
     Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
      Running target/debug/deps/adder-abcabcabc

❶ running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   ❷ Running target/debug/deps/integration_test-ce99bcc2479f4607

running 1 test
❸ test it_adds_two ... ok

❹ test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

    Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Три раздела данных включают модульные тесты, интеграционный тест и документационные тесты. Первый раздел модульных тестов ❶ совпадает с тем, что мы уже видели: одна строка для каждого модульного теста (эта строка называется `internal`, мы добавили ее в листинге 11.12), а затем итоговая строка для модульных тестов.

Раздел интеграционных тестов начинается со строки `Running target/debug/deps/integration_test-ce99bcc2479f4607` ❷ (хеш-код в конце вывода будет другим).

Далее идет строка для каждой тестовой функции в этом интеграционном тесте ❸ и итоговая строка для результатов интеграционного теста ❹ непосредственно перед началом раздела `Doc-tests adder`.

Добавление большего числа тестовых функций в файл интеграционного теста прибавляет больше строк результатов в этот раздел файла интеграционного теста аналогично тому, как добавление большего числа функций модульного теста прибавляет больше строк результатов в этот раздел модульных тестов. У каждого файла интеграционного теста есть собственный раздел, поэтому если мы добавим больше файлов в каталог тестов, то будет больше разделов интеграционного теста.

Мы по-прежнему можем выполнить отдельно взятую функцию интеграционного теста, указав имя тестовой функции в качестве аргумента для команды `cargo test`. Для выполнения всех тестов в конкретном файле интеграционного теста используйте аргумент `--test` команды `cargo test`, с последующим именем файла:

```
$ cargo test --test integration_test
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/integration_test-952a27e0126bb565

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Эта команда выполняет тесты только в файле tests/integration_test.rs.

Подмодули в интеграционных тестах

По мере добавления новых интеграционных тестов вы, возможно, захотите создать более одного файла в каталоге tests, чтобы организовать их. Например, вы можете сгруппировать тестовые функции по функциональности, которую они тестируют. Как уже упоминалось ранее, каждый файл в каталоге tests компилируется как отдельная упаковка.

Рассматривая каждый файл интеграционного теста как его собственную упаковку, полезно создавать отдельные области видимости — это больше похоже на то, как конечные пользователи будут использовать вашу упаковку. Однако это означает, что файлы в каталоге tests ведут себя не так, как файлы в src, о чем вы узнали в главе 7, посвященной разделению кода на модули и файлы.

Другое поведение файлов в каталоге tests наиболее заметно, когда у вас есть набор вспомогательных функций, которые были бы полезны в нескольких файлах интеграционных тестов, и вы пытаетесь выполнить действия, описанные в разделе «Разделение модулей на разные файлы» (с. 165), чтобы извлечь их в общий модуль. Например, если мы создадим tests/common.rs и поместим в него функцию

`setup`, то сможем добавить в `setup` код, который хотим вызывать из нескольких тестовых функций в нескольких тестовых файлах:

tests/common.rs

```
pub fn setup() {
    // код функции setup, характерный для тестов вашей библиотеки,
    // будет находиться здесь
}
```

Снова выполнив тесты, мы увидим новый раздел в данных теста для файла common.rs, хотя этот файл не содержит никаких тестовых функций и мы не вызывали функцию `setup`:

```
running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

    Running target/debug/deps/common-b8b07b6f1be2db70

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

    Running target/debug/deps/integration_test-d993c68b431d39df

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

    Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Появление `common` в результатах тестирования с сообщением `running 0 tests` — это не совсем то, чего мы хотели. Мы просто хотели поделиться кодом с другими файлами интеграционных тестов.

Чтобы избежать `common` в данных теста, вместо tests/common.rs мы создадим tests/common/mod.rs. Это альтернативное соглашение об именовании, которое язык Rust также понимает. Такое имя файла говорит компилятору не рассматривать модуль `common` как файл интеграционного теста. Когда мы переместим код функции `setup` в файл tests/common/mod.rs и удалим файл tests/common.rs, раздел в данных теста больше не будет появляться. Файлы в подкаталогах каталога tests не компилируются как отдельные упаковки и не имеют разделов в данных теста.

После того как мы создали файл tests/common/mod.rs, можно использовать его из любого файла интеграционного теста в качестве модуля. Вот пример вызова функции `setup` из теста `it_adds_two` в файле tests/integration_test.rs:

tests/integration_test.rs

```
use adder;

mod common;

#[test]
fn it_adds_two() {
    common::setup();
    assert_eq!(4, adder::add_two(2));
}
```

Обратите внимание, что объявление `mod common;` совпадает с объявлениями модулей из листинга 7.21. Затем в тестовой функции можно вызвать функцию `common::setup()`.

Интеграционные тесты для двоичных упаковок

Если проект представляет собой двоичную упаковку, содержащую только файл src/main.rs, и в нем нет файла src/lib.rs, то мы не можем создавать интеграционные тесты в каталоге tests и вводить функции, определенные в файле src/main.rs, в область видимости с помощью инструкции `use`. Только в библиотечных упаковках имеются функции, которые могут использоваться другими упаковками; двоичные упаковки задуманы для того, чтобы они выполнялись самостоятельно.

Это одна из причин, по которой в проектах Rust, предоставляющих двоичный файл, есть простой файл src/main.rs, который вызывает алгоритм, расположенный в файле src/lib.rs. Используя эту структуру, интеграционные тесты могут проверять библиотечную упаковку с помощью `use`, испытывая важное свойство. Если это свойство работает, то небольшой объем кода в файле src/main.rs тоже будет работать, и его не нужно тестировать.

Итоги

Средства тестирования Rust дают возможность уточнить, как должен функционировать код, чтобы бесперебойная работа продолжалась даже при внесении изменений. В модульных тестах испытываются разные части библиотеки по отдельности, а также приватные детали реализации. В интеграционных тестах выполняется проверка правильности совместной работы многочисленных частей библиотеки и используется публичный API библиотеки, чтобы протестировать код таким же образом, каким его будет использовать внешний код. Несмотря на то что система типов и правила владения в языке Rust помогают предотвращать некоторые виды ошибок, тесты по-прежнему важны для уменьшения числа логических ошибок, связанных с ожидаемым поведением кода.

Давайте объединим знания, которые вы получили в этой главе и в предыдущих, чтобы поработать над проектом!

12 Проект ввода-вывода: сборка программы командной строки

Эта глава представляет собой сводку многих навыков, которые вы уже усвоили. В ней пойдет речь еще о нескольких средствах стандартной библиотеки. Мы создадим инструмент командной строки, который взаимодействует с файловым и консольным вводом-выводом, чтобы на практике изучить некоторые понятия языка Rust, которые теперь есть в вашем арсенале.

Скорость, безопасность, единый двоичный вывод и кросс-платформенная поддержка Rust делают его идеальным для создания инструментов командной строки, поэтому для проекта мы создадим собственную версию классического инструмента командной строки `grep` (*globally search a regular expression and print*, то есть «глобальный поиск по регулярному выражению и вывод»). В простейшем случае grep выполняет поиск заданного строкового значения в заданном файле. Для этого grep берет в качестве аргументов имя файла и строковое значение. Затем он читает файл, находит в нем строчки, которые содержат строковый аргумент, и выводит их.

Попутно мы покажем, как сделать так, чтобы инструмент командной строки использовал средства терминала, применяемые во многих инструментах командной строки. Мы будем читать значение переменной среды, чтобы позволить пользователю настроить свойства инструмента. Мы также будем печатать в поток стандартного вывода ошибок (`stderr`) вместо стандартного вывода данных (`stdout`), благодаря чему, к примеру, пользователь сможет перенаправлять вывод успешных данных в файл, при этом по-прежнему видя сообщения об ошибках на экране.

Один из членов сообщества Эндрю Галлант (Andrew Gallant) уже создал полнофункциональную, очень быструю версию `grep` под названием `ripgrep`. Для сравнения, наша версия `grep` будет довольно простой, но, изучив эту главу, вы получите базовые знания, необходимые для понимания реального проекта, такого как `ripgrep`.

Наш проект `grep` объединит ряд идей, которые вы уже усвоили:

- Организация кода (модули, глава 7).
- Использование векторов и строк (коллекции, глава 8).

- Обработка ошибок (глава 9).
- Использование типажей и жизненных циклов (глава 10).
- Написание тестов (глава 11).

Мы также кратко представим замыкания, итераторы и типажные объекты, которые будут подробно рассмотрены в главах 13 и 17.

Принятие аргументов командной строки

Давайте создадим новый проект с помощью команды `cargo new`. Назовем его `minigrep`, чтобы отличать от инструмента `grep`, который, возможно, уже есть в вашей системе.

```
$ cargo new minigrep
    Created binary (application) `minigrep` project
$ cd minigrep
```

Первая задача — сделать так, чтобы `minigrep` принимал два аргумента командной строки: имя файла и искомое строковое значение. То есть мы хотим иметь возможность запускать программу, используя команду `cargo run`, поисковое значение и путь к файлу, где будет осуществляться поиск, таким образом:

```
$ cargo run поисковое-значение образец-файла.txt
```

Сейчас программа, сгенерированная командой `cargo new`, не способна обрабатывать аргументы, которые мы ей передаем. Некоторые существующие библиотеки на https://crates.io/ помогут в написании программы, которая принимает аргументы командной строки, но поскольку вы только начали осваивать это понятие, давайте реализуем эту способность сами.

Чтение значений аргументов

Для того чтобы позволить программе `minigrep` читать значения аргументов командной строки, которые мы ей передаем, нам понадобится функция `std::env::args` из стандартной библиотеки. Эта функция возвращает итератор аргументов командной строки, которые были переданы в `minigrep`. Мы подробно рассмотрим итераторы в главе 13. А пока что вам нужно знать об итераторах только две вещи: итераторы производят серию значений, и мы можем вызывать метод `collect` для итератора, превращая его в коллекцию, такую как вектор, содержащий все элементы, которые производятся итератором.

Используйте код из листинга 12.1, чтобы позволить программе `minigrep` читать любые передаваемые ей аргументы командной строки, а затем собирать значения в вектор.

Листинг 12.1. Сборка аргументов командной строки в вектор и их печать

src/main.rs

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    println!("{:?}", args);
}
```

Сначала мы вводим модуль `std::env` в область видимости с помощью инструкции `use`, в результате чего можно использовать его функцию `args`.

Обратите внимание, что функция `std::env::args` вложена в два уровня модулей. Как мы уже говорили в главе 7, в тех случаях, когда желаемая функция вложена в несколько модулей, обычно в область видимости вводится родительский модуль, а не сама функция. Благодаря этому мы можем легко использовать другие функции из `std::env`. Это также менее двусмысленно, чем добавление `use std::env::args`, а затем вызов функции с указанием только `args`, потому что функцию `args` легко по ошибке принять за функцию, определенную в текущем модуле.

> **ФУНКЦИЯ ARGS И НЕДЕЙСТВИТЕЛЬНЫЙ ЮНИКОД**
>
> Обратите внимание, что `std::env::args` будет паниковать, если какой-либо аргумент содержит недействительный Юникод. Если программе нужно принимать аргументы, содержащие недействительный Юникод, используйте функцию `std::env::args_os`. Эта функция возвращает итератор, который вместо значений типа `String` создает значения типа `OsString`. Мы решили использовать `std::env::args` здесь для простоты, потому что значения типа `OsString` отличаются от платформы к платформе и сложнее для работы, чем значения типа `String`.

В первой строке функции `main` мы вызываем функцию `env::args` и сразу же применяем функцию `collect`, чтобы превратить итератор в вектор, содержащий все значения, полученные итератором. Можно использовать функцию `collect` для создания многих видов коллекций, поэтому мы аннотируем тип `args`, указав, что нам нужен вектор строк. Хотя в Rust аннотировать типы приходится очень редко, функция `collect` является одной из тех, которую часто нужно аннотировать, потому что компилятор не может определить тип требуемой коллекции.

Наконец, мы выводим вектор с помощью отладочного средства форматирования `:?`. Давайте попробуем выполнить этот код сначала без аргументов, а затем с двумя аргументами:

```
$ cargo run
--пропуск--
["target/debug/minigrep"]

$ cargo run needle haystack
```

```
--пропуск--
["target/debug/minigrep", "needle", "haystack"]
```

Обратите внимание, что первым значением в векторе является `"target/debug/minigrep"`, то есть имя двоичного файла. Это соответствует поведению списка аргументов в C, позволяя программам использовать имя, по которому они были вызваны при исполнении. Часто бывает удобно иметь доступ к имени программы, в случае если вы хотите напечатать его в сообщениях или изменить поведение программы, основываясь на том, какой псевдоним командной строки был использован для вызова программы. Но в данной главе мы это проигнорируем и сохраним только два нужных нам аргумента.

Сохранение значений аргументов в переменных

Вывод значения вектора аргументов иллюстрирует, что программа способна обращаться к значениям, указанным в качестве аргументов командной строки. Теперь нужно сохранить значения двух аргументов в переменных, чтобы мы могли использовать их в остальной части программы. Мы делаем это в листинге 12.2.

Листинг 12.2. Создание переменных для хранения аргумента с искомым значением и аргумента с именем файла

src/main.rs

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    let query = &args[1];
    let filename = &args[2];

    println!("Поиск {}", query);
    println!("В файле {}", filename);
}
```

Как мы видели, когда выводили вектор, имя программы занимает первое значение вектора в `args[0]`, поэтому мы начинаем с индекса 1. Первым аргументом, который берет программа `minigrep`, является строковое значение, которое мы ищем, поэтому мы помещаем ссылку на первый аргумент в переменную `query`. Вторым аргументом будет имя файла, поэтому мы помещаем ссылку на второй аргумент в переменную `filename`.

Мы временно выводим значения этих переменных, чтобы доказать, что код работает так, как мы предполагаем. Давайте снова запустим эту программу с аргументами test и sample.txt:

```
$ cargo run test sample.txt
    Compiling minigrep v0.1.0 (file:///projects/minigrep)
     Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
```

```
        Running `target/debug/minigrep test sample.txt`
Searching for test
In file sample.txt
```

Отлично, программа работает! Значения необходимых аргументов сохраняются в правильных переменных. Позже мы добавим небольшую обработку ошибок, чтобы учесть потенциальные ошибочные ситуации, например, когда пользователь не предоставляет никаких аргументов. Пока что мы проигнорируем эту ситуацию и поработаем над добавлением способности читать файлы.

Чтение файла

Теперь мы добавим функциональность чтения файла, указываемого в аргументе командной строки `filename`. Сначала нам нужен образец файла для тестирования. Для проверки работы программы `minigrep` самый лучший файл — это тот, в котором мало текста на нескольких строках с несколькими повторяющимися словами. В листинге 12.3 представлено стихотворение Эмили Дикинсон (Emily Dickinson), которое нам подходит. Создайте файл под названием poem.txt на корневом уровне проекта и введите стихотворение *I'm Nobody! Who are you?* («Я — никто! А ты кто такой?»).

Листинг 12.3. Стихотворение Эмили Дикинсон — хороший пример для теста

poem.txt

```
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us — don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

Когда текст будет готов, отредактируйте src/main.rs и добавьте код для чтения файла, как показано в листинге 12.4.

Листинг 12.4. Чтение содержимого файла, указанного вторым аргументом

src/main.rs

```
    use std::env;
❶   use std::fs;

    fn main() {
        // --пропуск--
        println!("В файле {}", filename);

      ❷ let contents = fs::read_to_string(filename)
            .expect("Что-то пошло не так при чтении файла");

      ❸ println!("С текстом:\n{}", contents);
    }
```

Сначала мы добавляем еще одну инструкцию `use`, чтобы ввести соответствующую часть стандартной библиотеки: нам нужна функция `std::fs` для обработки файлов ❶.

В функцию `main` мы добавили новую инструкцию: функция `fs::read_to_string` берет имя файла, открывает его и возвращает `Result<String>` содержимого файла ❷.

После этой инструкции мы снова добавили временную инструкцию `println!`, которая выводит значение `contents` после прочтения файла, благодаря чему можно проверить, что программа пока что работает ❸.

Давайте выполним этот код с любым строковым значением в качестве первого аргумента командной строки (потому что мы еще не реализовали поисковую часть) и файлом poem.txt в качестве второго аргумента:

```
$ cargo run the poem.txt
    Compiling minigrep v0.1.0 (file:///projects/minigrep)
     Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
      Running `target/debug/minigrep the poem.txt`
Searching for test
In file sample.txt
С текстом:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us — don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

Отлично! Код прочитал, а затем вывел содержимое файла. Но у этого кода есть несколько недостатков. У функции `main` много обязанностей: в общем, функции более понятны и их проще технически обслуживать, если каждая функция отвечает только за одну идею. Другая проблема заключается в том, что мы не обрабатываем ошибки, как должно, на все сто. Программа пока еще небольшая, так что эти недостатки — небольшая проблема, но по мере расширения программы будет сложнее устранять их безупречно. Рекомендуется начинать рефакторинг на ранних стадиях разработки программы, поскольку гораздо проще рефакторить меньшие объемы кода. Мы сделаем это позже.

Рефакторинг с целью улучшения модульности и обработки ошибок

С целью улучшения программы мы устраним четыре проблемы, связанные со структурой программы и с тем, как она обрабатывает потенциальные ошибки.

Прежде всего, функция `main` теперь выполняет две задачи: проводит разбор аргументов и читает файлы. Для такой маленькой функции это не является серьезной проблемой. Однако если мы продолжим развивать программу внутри функции `main`, то число отдельных задач, которые обрабатывает функция `main`, будет увеличиваться. По мере того как расширяется круг ответственности функции, об этой функции становится труднее рассуждать, ее становится труднее тестировать и труднее вносить в нее изменения, не нарушая ни одной из ее частей. Лучше всего разделить функциональность так, чтобы каждая функция отвечала за одну задачу.

Эта трудность связана со второй проблемой: хотя переменные `query` и `filename` являются конфигурационными для программы, такие переменные, как `contents`, используются для выполнения алгоритма программы. Чем длиннее функция `main`, тем больше переменных нужно ввести в область видимости; чем больше переменных в области видимости, тем сложнее отслеживать цель каждой из них. Лучше всего сгруппировать конфигурационные переменные в одну структуру и тем самым сделать ясным их назначение.

Третья проблема заключается в том, что мы использовали метод `expect` для печати сообщения об ошибке, когда не получается прочитать файл, а сообщение об ошибке просто выводит `Что-то пошло не так при чтении файла`. Чтение файла может не получиться в нескольких случаях: например, файл отсутствует либо у нас нет разрешения на его открытие. Сейчас, независимо от этой ситуации, мы выведем сообщение об ошибке `Что-то пошло не так при чтении файла`, которое не даст пользователю никакой информации!

Четвертая проблема: мы используем метод `expect` многократно для обработки разных ошибок, и если пользователь запускает программу, не указывая достаточное число аргументов, он видит ошибку выхода индекса за пределы границ (*index out of bounds*), которая неявно объясняет проблему. Было бы лучше, если бы весь код обработки ошибок был в одном месте, — благодаря этому будущие разработчики будут консультироваться только в одном месте кода, если потребуется внести изменение в алгоритм обработки ошибок. Наличие всего кода обработки ошибок в одном месте также позволит печатать сообщения, которые будут иметь смысл для конечных пользователей.

Давайте решим эти четыре проблемы путем рефакторинга проекта.

Разделение обязанностей в двоичных проектах

Функция `main` отвечает за многочисленные задачи — эта организационная проблема часто встречается во многих двоичных проектах. В результате сообщество языка Rust разработало процесс, используемый в качестве руководящего принципа для распределения обязанностей в двоичной программе, когда `main` расширяется. Этот процесс состоит из следующих шагов:

- Разбить программу на main.rs и lib.rs и переместить алгоритм программы в lib.rs.

- Пока алгоритм разбора командной строки небольшой, он может оставаться в main.rs.
- Когда алгоритм разбора командной строки начнет усложняться, извлечь его из main.rs и переместить в lib.rs.

После этого процесса функция `main` должна отвечать за следующее:

- Вызов алгоритма разбора командной строки со значениями аргументов.
- Настройка любой другой конфигурации.
- Вызов функции `run` в lib.rs.
- Обработка ошибки, если `run` возвращает ошибку.

Этот паттерн касается разделения обязанностей: main.rs выполняет программу, а lib.rs занимается полным алгоритмом поставленной задачи. Поскольку вы не можете проверить функцию `main` напрямую, эта структура позволяет тестировать весь алгоритм программы, переместив его в функции внутри lib.rs. Единственный код, который остается в main.rs, будет достаточно мал, и его правильность можно легко проверить, прочтя его. Давайте переделаем программу, следуя этому процессу.

Извлечение парсера аргументов

Мы извлекаем свойство разбора аргументов в функцию, которая будет вызываться функцией `main`, для подготовки к перемещению алгоритма разбора командной строки в src/lib.rs. Листинг 12.5 показывает новое начало функции `main`, вызывающей новую функцию `parse_config`, которую мы пока обозначим в src/main.rs.

Листинг 12.5. Извлечение функции parse_config из main

src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let (query, filename) = parse_config(&args);

    // --пропуск--
}

fn parse_config(args: &[String]) -> (&str, &str) {
    let query = &args[1];
    let filename = &args[2];

    (query, filename)
}
```

Мы по-прежнему собираем аргументы командной строки в вектор, но вместо того, чтобы передавать значение аргумента в индексе 1 переменной `query`, а значение аргумента в индексе 2 переменной `filename` внутри функции `main`, мы передаем весь вектор функции `parse_config`. Тогда функция `parse_config` содержит

алгоритм, который выясняет, какие аргументы входят в переменные, и передает значения обратно в функцию `main`. Мы по-прежнему создаем переменные `query` и `filename` в функции `main`, но `main` больше не отвечает за то, как соотносятся аргументы и переменные командной строки.

С учетом того, что программа небольшая, эта переделка, возможно, покажется излишней, но мы проводим рефакторинг понемногу, поступательными шагами. После внесения этого изменения выполните программу еще раз, чтобы убедиться, что разбор аргументов по-прежнему работает. Полезно часто проверять, есть ли улучшения. Это помогает выявлять причину проблем, когда они возникают.

Группировка конфигурационных значений

Мы можем сделать еще один небольшой шаг для дальнейшего улучшения функции `parse_config`. В данный момент мы возвращаем кортеж, но затем мы немедленно снова разбиваем его на отдельные части. Это признак того, что правильной абстракции, скорее всего, еще нет.

Еще одним индикатором, который говорит о том, что возможны улучшения, является конфигурационная часть `parse_config`. Из нее следует, что два возвращаемых значения связаны и являются частью одного конфигурационного значения. В настоящее время мы не передаем этот смысл в структуре данных иначе, чем группируя два значения в кортеж; мы могли бы поместить два значения в одну структуру и дать каждому полю структуры осмысленное имя. Так будущим разработчикам кода будет проще понять, как разные значения соотносятся друг с другом и какова их цель.

ПРИМЕЧАНИЕ

Использование примитивных значений, когда более уместен сложный тип, — это антипаттерн, который называется «одержимость примитивами».

Листинг 12.6 показывает улучшения функции `parse_config`.

Листинг 12.6. Рефакторинг функции `parse_config` для возвращения экземпляра структуры `Config`

src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

  ❶ let config = parse_config(&args);

    println!("Поиск {}", config.query❷);
    println!("В файле {}", config.filename❸);

    let contents = fs::read_to_string(config.filename❹)
        .expect("Что-то пошло не так при чтении файла");
    // --пропуск--
```

```
    }

❺ struct Config {
        query: String,
        filename: String,
    }

❻ fn parse_config(args: &[String]) -> Config {
    ❼ let query = args[1].clone();
    ❽ let filename = args[2].clone();

        Config { query, filename }
    }
```

Мы добавили структуру с именем `Config`, определенную для полей `query` и `filename` ❺. Сигнатура `parse_config` теперь указывает на то, что она возвращает значение типа `Config` ❻. В теле `parse_config`, где мы обычно возвращали строковые срезы, ссылающиеся на значения типа `String` в `args`, теперь мы определяем структуру `Config`, которая содержит обладаемые значения типа `String`. Переменная `args` в функции `main` является владельцем значений аргументов и позволяет функции `parse_config` их заимствовать. Это означает, что мы бы нарушили правила заимствования языка Rust, если бы структура `Config` попыталась взять значения в `args` во владение.

Мы могли бы управлять данными `String` несколькими разными способами, но самый простой маршрут, хотя и несколько неэффективный, — вызывать метод `clone` для значений ❼❽. В результате этого появится полная копия данных, дав возможность экземпляру `Config` владеть, что займет больше времени и памяти, чем хранение ссылки на строковые данные. Однако клонирование данных также делает код очень простым, потому что не нужно управлять жизненными циклами ссылок. В этих обстоятельствах отказ от небольшой производительности ради достижения простоты — ценный компромисс.

> **КОМПРОМИССЫ ИСПОЛЬЗОВАНИЯ МЕТОДА CLONE**
>
> Многие растиане избегают метод `clone` для решения проблем владения в связи с затратностью времени его выполнения. В главе 13 вы узнаете, как в подобных ситуациях использовать более эффективные методы. Но сейчас можно скопировать несколько строк, чтобы продолжить, так как вы сделаете эти копии только один раз, а имя файла и строка запроса очень малы. Лучше иметь программу, которая работает, но при этом несколько неэффективна, чем пытаться улучшить код сверх меры на первом этапе. Когда вы наберетесь опыта, вам будет легче начать с самого эффективного решения, однако пока вполне приемлемо вызывать `clone`.

Мы обновили функцию `main` так, что она поместила экземпляр структуры `Config`, возвращаемый функцией `parse_config`, в переменную с именем `config` ❶. Мы

также обновили код, который ранее использовал отдельные переменные `query` и `filename`, поэтому теперь он использует поля в структуре `Config` ❷❸❹.

Теперь код гораздо четче передает идею о том, что `query` и `filename` взаимосвязаны и что их цель — настроить порядок работы программы. Любой код, использующий эти значения, знает, что их можно найти в экземпляре структуры `Config` в полях, названных по их назначению.

Создание конструктора для Config

Пока что мы извлекли алгоритм, который отвечает за проведение разбора аргументов командной строки, из функции `main`, и поместили его в функцию `parse_config`. Это помогло нам увидеть, что значения `query` и `filename` взаимосвязаны, и смысл этой связи должен быть передан в коде. Затем мы добавили структуру `Config`, чтобы обозначить взаимосвязанную цель значений `query` и `filename` и иметь возможность возвращать имена этих значений в качестве имен полей структуры из функции `parse_config`.

Поэтому теперь, когда цель функции `parse_config` — создание экземпляра структуры `Config`, мы можем поменять обыкновенную функцию `parse_config` на функцию с именем `new`, которая связана со структурой `Config`. Внесение этого изменения сделает код выразительнее. Мы можем создавать экземпляры типов стандартной библиотеки, таких как `String`, вызывая `String::new`. Схожим образом, поменяв `parse_config` на функцию `new`, связанную со структурой `Config`, мы сможем создавать экземпляры `Config`, вызывая `Config::new`. Листинг 12.7 показывает, какие изменения необходимо внести.

Листинг 12.7. Замена `parse_config` на `Config::new`

src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

 ❶ let config = Config::new(&args);

    // --пропуск--
}

// --пропуск--

❷ impl Config {
  ❸ fn new(args: &[String]) -> Config {
        let query = args[1].clone();
        let filename = args[2].clone();

        Config { query, filename }
    }
}
```

Мы обновили функцию `main` там, где вызывали `parse_config`, чтобы вызывать `Config::new` ❶. Мы изменили имя `parse_config` на `new` ❸ и переместили ее в блок

`impl` ❷, который связывает функцию `new` со структурой `Config`. Попробуйте еще раз скомпилировать этот код, чтобы убедиться, что он работает.

Исправление обработки ошибок

Теперь мы поработаем над исправлением обработки ошибок. Напомним, что попытка обратиться к значениям в векторе `args` с индексом 1 или 2 вызовет панику программы, если вектор содержит менее трех элементов. Попробуйте выполнить программу без аргументов. Она будет выглядеть так:

```
$ cargo run
    Compiling minigrep v0.1.0 (file:///projects/minigrep)
     Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
      Running `target/debug/minigrep`
thread 'main' panicked at 'index out of bounds: the len is 1
but the index is 1', src/main.rs:25:21
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

Строчка с сообщением об ошибке `index out of bounds: the len is 1 but the index is 1` («индекс за пределами границ: длина равна 1, но индекс равен 1») предназначена разработчикам. Указанное сообщение не поможет конечным пользователям понять, что произошло и что они должны делать. Давайте это исправим.

Улучшение сообщения об ошибке

В листинг 12.8 мы добавим проверку функции `new`, которая удостоверится, что срез является достаточно длинным, перед тем как обращаться к индексам 1 и 2. Если срез недостаточно длинный, то программа поднимает панику и показывает более качественное сообщение об ошибке, чем сообщение о выходе индекса за пределы.

Листинг 12.8. Добавление проверки числа аргументов

src/main.rs

```
// --пропуск--
fn new(args: &[String]) -> Config {
    if args.len() < 3 {
        panic!("недостаточно аргументов");
    }
    // --пропуск--
```

Этот код похож на функцию `Guess::new` из листинга 9.10, где мы вызывали `panic!`, когда аргумент `value` находился вне интервала допустимых значений. Вместо проверки интервала значений здесь мы проверяем, что длина `args` не менее 3, а остальная часть функции может работать, исходя из предположения, что это условие соблюдено. Если в `args` менее трех элементов, то это условие будет истинным и мы вызовем макрокоманду `panic!`, которая немедленно завершит программу.

Добавив несколько строк кода в `new`, снова выполним программу без аргументов и посмотрим, как ошибка выглядит на этот раз:

```
$ cargo run
    Compiling minigrep v0.1.0 (file:///projects/minigrep)
     Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
      Running `target/debug/minigrep`
thread 'main' panicked at 'недостаточно аргументов', src/main.rs:26:13
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

Эти данные смотрятся лучше: теперь у нас есть разумное сообщение об ошибке. Однако у нас также есть посторонняя информация, которую мы не хотим сообщать пользователям. Пожалуй, использование здесь приема из листинга 9.10 не является наилучшим вариантом: вызов макрокоманды `panic!` больше подходит для задач по программированию, чем для прикладных задач, как описано в главе 9. Вместо этого можно применить другой прием, о котором вы узнали в главе 9, — возвращение типа `Result`, который указывает либо на успех, либо на ошибку.

Возвращение типа Result из функции new вместо вызова макрокоманды panic!

Вместо этого мы можем возвращать значение типа `Result`, которое в случае успеха будет содержать экземпляр структуры `Config`, а в случае ошибки опишет проблему. Когда функция `Config::new` сообщает что-либо функции `main`, мы можем использовать тип `Result`, сигнализируя о том, что возникла проблема. Затем можно изменить функцию `main` так, чтобы она конвертировала вариант `Err` в более практичную для пользователей ошибку без окружающего текста о потоке 'main' (`thread 'main'`) и обратной трассировке (`RUST_BACKTRACE`), поводом для которых является вызов `panic!`.

Листинг 12.9 показывает изменения, которые нужно внести в значение, возвращаемое из `Config::new`, и тело функции, необходимые для того, чтобы возвращать `Result`. Обратите внимание, что этот код не будет компилироваться до тех пор, пока мы не обновим функцию `main`, что мы и сделаем в следующем листинге.

Листинг 12.9. Возвращение типа `Result` из `Config::new`

src/main.rs

```
impl Config {
    fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("недостаточно аргументов");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        Ok(Config { query, filename })
    }
}
```

Функция `new` теперь возвращает `Result` с экземпляром структуры `Config` в случае успеха и `&'static str` в случае ошибки. Вспомните из раздела «Статический жиз-

ненный цикл», что `&'static str` является типом строковых литералов, который пока что является типом сообщения об ошибке.

Мы сделали два изменения в теле функции `new`: вместо вызова `panic!`, когда пользователь не передает достаточного числа аргументов, мы теперь возвращаем значение `Err`, и мы завернули в `Ok` значение, возвращаемое из структуры `Config`. Эти изменения подчиняют функцию новой сигнатуре типа.

Возвращение значения `Err` из функции `Config::new` позволяет функции `main` обрабатывать значение `Result`, возвращаемое из функции `new`, и лучше завершать работу в случае ошибки.

Вызов функции Config::new и обработка ошибок

Для того чтобы обработать ошибку и напечатать удобное для пользователя сообщение, нужно обновить функцию `main` в части обработки `Result`, возвращаемого из функции `Config::new`, как показано в листинге 12.10. Мы также займемся выходом из программы командной строки с ненулевым кодом ошибки из макрокоманды `panic!` и реализуем его вручную. Ненулевой статус завершения работы традиционно сигнализирует процессу, вызвавшему программу, о том, что программа завершила работу с ошибкой.

Листинг 12.10. Завершение работы с кодом ошибки, если не получается создать новый экземпляр структуры `Config`

src/main.rs

```
❶ use std::process;

  fn main() {
      let args: Vec<String> = env::args().collect();

    ❷ let config = Config::new(&args).unwrap_or_else(❸|err❹| {
        ❺ println!("Проблема при разборе аргументов: {}", err);
        ❻ process::exit(1);
      });

      // --пропуск--
```

В этом листинге мы использовали метод, который ранее не рассматривали: `unwrap_or_else`, определенный стандартной библиотекой для `Result<T, E>` ❷. Использование метода `unwrap_or_else` позволяет определять настраиваемую обработку ошибок без вызова макрокоманды `panic!`. Если `Result` равен значению `Ok`, то поведение этого метода аналогично методу `unwrap`: он возвращает внутреннее значение, обернутое `Ok`. Однако если это значение равно `Err`, то метод вызывает код в замыкании, то есть анонимную функцию, которую мы определяем и передаем в качестве аргумента в методе `unwrap_or_else` ❸. Мы рассмотрим замыкание подробнее в главе 13. А пока вам просто нужно знать, что метод `unwrap_or_else` передаст внутреннее значение `Err`, которое в данном случае является статической строкой `недостаточно аргументов`, добавленной в листинг 12.9. Она находится

в замыкании в аргументе `err`, расположенном между вертикальными чертами ❹. Код в замыкании может затем использовать значение `err` во время работы.

Мы добавили новую строку кода `use`, введя `process` из стандартной библиотеки в область видимости ❶. Код в замыкании, который будет выполняться в случае ошибки, состоит всего из двух строк: мы печатаем значение `err` ❺, а затем вызываем `process::exit` ❻. Функция `process::exit` немедленно остановит программу и вернет число, которое было передано в качестве кода завершения работы. Это похоже на `panic!`-ориентированную обработку, которую мы использовали в листинге 12.8, но мы больше не получаем лишних данных. Давайте попробуем это сделать:

```
$ cargo run
    Compiling minigrep v0.1.0 (file:///projects/minigrep)
     Finished dev [unoptimized + debuginfo] target(s) in 0.48 secs
      Running `target/debug/minigrep`
Проблема с разбором аргументов: недостаточно аргументов
```

Великолепно! Пользоваться этими данными гораздо удобнее.

Извлечение алгоритма из функции main

Теперь, когда мы закончили рефакторинг конфигурационного разбора, давайте обратимся к алгоритму программы. Как мы уже говорили в разделе «Разделение обязанностей в двоичных проектах», мы извлекаем функцию с именем `run`, которая будет содержать весь алгоритм, пока что находящийся в функции `main`, которая не связана с настройкой конфигурации или обработкой ошибок. Когда мы закончим, функция `main` будет краткой, ее легко будет проверить и мы сможем писать тесты для остального алгоритма.

Листинг 12.11 показывает извлеченную функцию `run`. Пока мы делаем лишь небольшое, поступательное улучшение процедуры извлечения функции. Мы попрежнему определяем указанную функцию в src/main.rs.

Листинг 12.11. Извлечение функции `run`, содержащей остальной алгоритм

src/main.rs

```
fn main() {
    // --пропуск--

    println!("Поиск {}", config.query);
    println!("В файле {}", config.filename);

    run(config);
}

fn run(config: Config) {
    let contents = fs::read_to_string(config.filename)
        .expect("Что-то пошло не так при чтении файла");
```

```
        println!("С текстом:\n{}", contents);
    }
    // --пропуск--
```

Функция `run` теперь содержит остальной алгоритм функции `main`, начиная с чтения файла. Функция `run` принимает экземпляр структуры `Config` в качестве аргумента.

Возвращение ошибок из функции run

Теперь, когда остальной алгоритм программы выделен в функцию `run`, мы можем улучшить обработку ошибок, как это было сделано с функцией `Config::new` в листинге 12.9. Вместо того чтобы давать программе паниковать, вызывая метод `expect`, функция `run` будет возвращать `Result<T, E>`, когда что-то пойдет не так. Это позволит еще больше консолидировать алгоритм вокруг обработки ошибок в функции `main` в удобном для пользователя виде. В листинге 12.12 показаны изменения, которые мы должны внести в сигнатуру и тело функции `run`.

Листинг 12.12. Изменение функции run в части возвращения экземпляра типа Result

src/main.rs

```
❶ use std::error::Error;

    // --пропуск--

❷ fn run(config: Config) -> Result<(), Box<dyn Error>> {
        let contents = fs::read_to_string(config.filename)?❸;

        println!("С текстом:\n{}", contents);

      ❹ Ok(())
    }
```

Мы внесли три существенных изменения. Во-первых, мы изменили тип значения, возвращаемого из функции `run`, на `Result<(), Box<dyn Error>>` ❷. Эта функция ранее возвращала пустой тип `()`; мы оставляем его как значение, возвращаемое в случае `Ok`.

Для типа ошибки мы использовали типажный объект `Box<dyn Error>` (мы ввели `std::error::Error` в область видимости с помощью инструкции `use` вверху ❶). Мы рассмотрим типажные объекты в главе 17. А пока просто знайте, `Box<dyn Error>` означает, что функция будет возвращать тип, который реализует типаж `Error`, но нам не нужно уточнять, каким именно типом будет возвращаемое значение. Это дает возможность гибко возвращать значения ошибок, которые могут иметь разные типы в разных случаях. Ключевое слово `dyn` расшифровывается как «динамический».

Во-вторых, мы удалили вызов метода `expect` в пользу оператора `?` ❸, как мы уже говорили в главе 9. Вместо того чтобы вызывать `panic!` в случае ошибки, опера-

тор `?` будет возвращать значение ошибки из текущей функции, чтобы вызывающий код обработал ее.

В-третьих, функция `run` теперь возвращает значение `Ok` в случае успеха ❹. Мы объявили тип успеха функции `run` как `()` в сигнатуре — это означает, что нужно обернуть значение пустого типа в значение `Ok`. Синтаксис `Ok(())` на первый взгляд может показаться немного странным, но такое использование `()` — это идиоматический способ, указывающий, что мы вызываем функцию `run` только ради ее побочных эффектов. Она не возвращает нужное значение.

Когда вы выполните этот код, он скомпилируется, но выдаст предупреждение[1]:

```
warning: unused `std::result::Result` that must be used
 --> src/main.rs:17:5
   |
17 |     run(config);
   |     ^^^^^^^^^^^^
   |
   = note: #[warn(unused_must_use)] on by default
   = note: this `Result` may be an `Err` variant, which should be handled
```

Код проигнорировал значение `Result`, а значение `Result`, возможно, указывает, что произошла ошибка. Но мы не проверяем, была ли ошибка, и компилятор напоминает нам о том, что мы, вероятно, хотели бы иметь здесь некий код обработки ошибок! Давайте пофиксим.

Обработка ошибок в функции main, возвращаемых из функции run

Мы проверим, есть ли ошибки, и обработаем их с помощью приема, аналогичного тому, который мы использовали с `Config::new` в листинге 12.10, но с небольшим отличием:

src/main.rs

```
fn main() {
    // --пропуск--

    println!("Поиск {}", config.query);
    println!("В файле {}", config.filename);

    if let Err(e) = run(config) {
        println!("Ошибка в приложении: {}", e);

        process::exit(1);
    }
}
```

Мы используем `if let` вместо метода `unwrap_or_else` для проверки, что функция `run` возвращает значение `Err`, и вызываем `process::exit(1)`, если это так.

[1] `неиспользуемый `std::result::Result`, который должен быть использован`

Функция `run` не возвращает значение, которое требуется развернуть (методом `unwrap`) так, как это делает функция `Config::new`, которая возвращает экземпляр структуры `Config`. Поскольку `run` в случае успеха возвращает `()`, нас интересует только обнаружение ошибки, поэтому нам не нужно выполнять метод `unwrap_or_else`, чтобы вернуть развернутое значение, потому что оно будет исключительно `()`.

Тела блока `if let` и метода `unwrap_or_else` в обоих случаях одинаковы: мы печатаем ошибку и завершаем работу.

Разбивка кода в библиотечную упаковку

Наш проект `minigrep` пока что выглядит неплохо! Теперь мы разобьем файл src/main.rs и поместим некоторую часть кода в файл src/lib.rs, чтобы протестировать его и сократить число обязанностей файла src/main.rs.

Давайте переместим весь код, который не является функцией `main`, из src/main.rs в src/lib.rs:

- Определение функции `run`.
- Релевантные инструкции `use`.
- Определение структуры `Config`.
- Определение функции `Config::new`.

Содержимое src/lib.rs должно иметь сигнатуры, показанные в листинге 12.13 (для краткости тела функций опущены). Обратите внимание, что этот код не будет компилироваться до тех пор, пока мы не изменим src/main.rs в листинге 12.14.

Листинг 12.13. Перемещение определений структуры `Config` и функции `run` в src/lib.rs

src/lib.rs

```
use std::error::Error;
use std::fs;

pub struct Config {
    pub query: String,
    pub filename: String,
}

impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        // --пропуск--
    }
}

pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    // --пропуск--
}
```

Мы свободно использовали ключевое слово pub: со структурой Config, с ее полями и методом new, а также с функцией run. Теперь у нас есть библиотечная упаковка с публичным API, который мы можем протестировать!

Сейчас нужно ввести код, который мы перенесли в src/lib.rs, в область видимости двоичной упаковки в src/main.rs, как показано в листинге 12.14.

Листинг 12.14. Использование упаковки minigrep в src/main.rs

src/main.rs

```
use std::env;
use std::process;

use minigrep::Config;

fn main() {
    // --пропуск--
    if let Err(e) = minigrep::run(config) {
        // --пропуск--
    }
}
```

Мы добавляем строку кода `use minigrep::Config`, чтобы ввести тип `Config` из библиотечной упаковки в область видимости двоичной упаковки, и предваряем функцию `run` префиксом с именем нашей упаковки. Теперь все свойства должны работать. Запустите программу с помощью команды `cargo run` и убедитесь, что все работает правильно.

Ух ты! Мы проделали большую работу и настроились на будущий успех. Теперь обрабатывать ошибки стало гораздо проще, и мы сделали код более модульным. С этого момента почти вся работа будет вестись в src/lib.rs.

Давайте воспользуемся этой модульностью и сделаем то, что было бы трудно осуществить со старым кодом, но легко с новым — напишем несколько тестов!

Развитие функциональности библиотеки с помощью методики разработки на основе тестов

Теперь, когда мы извлекли алгоритм, перенеся его в src/lib.rs, и оставили сбор аргументов и обработку ошибок в src/main.rs, стало гораздо проще писать тесты для основной функциональности кода. Можно вызывать функции с различными аргументами напрямую и проверять возвращаемые значения, не вызывая двоичный код из командной строки. Напишите несколько тестов для функциональности в функциях `Config::new` и `run` самостоятельно.

В этом разделе мы добавим в программу `minigrep` поисковый алгоритм с помощью методики разработки на основе тестирования (TDD от англ. *test-driven development*). Методика разработки ПО следующая:

1. Написать тест, который не срабатывает, и выполнить его, чтобы убедиться, что он не срабатывает по известной вам причине.
2. Написать или модифицировать объем кода, достаточный для того, чтобы новый тест проходил успешно.
3. Провести рефакторинг кода, который вы только что добавили или модифицировали, и убедиться, что тесты продолжают проходить успешно.
4. Повторить, начиная с пункта 1!

Этот процесс является лишь одним из многих способов написания ПО, но разработка на основе тестирования также помогает направлять процесс планирования кода. Написание теста перед написанием кода, который приводит к успешному прохождению теста, помогает поддерживать высокий уровень тестового покрытия на протяжении всего процесса.

Мы проверим реализацию функциональности, которая будет фактически искать строку запроса в содержимом файла и создавать список строк файла, соответствующих запросу. Мы добавим это свойство в функцию под названием `search`.

Написание провального теста

Поскольку инструкции `println!`, которые мы использовали для проверки поведения программы, больше не нужны, давайте уберем их из src/lib.rs и src/main.rs. Затем в src/lib.rs мы добавим модуль `tests` с тестовой функцией, как это было сделано в главе 11. Тестовая функция определяет необходимые свойства функции `search`: она будет принимать запрос и текст для поиска запроса. Эта функция будет возвращать только те строки из текста, которые содержат запрос. В листинге 12.15 показан этот тест, он пока что не компилируется.

Листинг 12.15. Создание провального теста для функции search, которую мы хотели бы иметь

src/lib.rs

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

    #[test]
    fn one_result() {
        let query = "duct";
        let contents = "\

Rust:
safe, fast, productive.
Pick three.";

        assert_eq!(
            vec!["safe, fast, productive."],
```

```
            search(query, contents)
        );
    }
}
```

Этот тест выполняет поиск строкового значения `"duct"` («провод»). Текст, в котором мы производим поиск, состоит из трех строк, и только одна из них содержит слово `"duct"`. Мы подтверждаем (с помощью `assert_eq!`), что значение, возвращаемое функцией `search`, содержит только ту строку текста, которую мы ожидаем.

Мы не можем выполнить этот тест, потому что тест даже не компилируется: функции `search` еще не существует! Поэтому теперь добавим объем кода, достаточный для того, чтобы тест можно было скомпилировать и выполнить, добавив определение функции `search`, которое всегда возвращает пустой вектор, как показано в листинге 12.16. Тогда тест должен скомпилироваться и не сработать, потому что пустой вектор не совпадает с вектором, содержащим строку текста `"safe, fast, productive"` («безопасно, быстро, продуктивно»).

Листинг 12.16. Определение функции `search` в объеме, достаточном для компиляции кода

src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    vec![]
}
```

Обратите внимание, что требуется явно выраженный жизненный цикл `'a`, определенный в сигнатуре функции `search` и используемый с аргументом `contents` и возвращаемым из функции значением. Вспомните, что в главе 10 параметры жизненных циклов описывают то, какой аргумент жизненного цикла соединен с жизненным циклом возвращаемого значения. В этом случае мы указываем, что возвращаемый вектор должен содержать строковые срезы, которые ссылаются на срезы аргумента `contents` (а не аргумента `query`).

Другими словами, мы говорим компилятору, что данные, возвращаемые функцией `search`, будут жить столько же, сколько данные, передаваемые в функцию `search` в аргументе `contents`. Это очень важно! Чтобы ссылка была действительной, данные, на которые ссылается срез, должны быть действительными; если компилятор допускает, что мы делаем строковые срезы аргумента `query`, а не аргумента `contents`, то он будет выполнять проверку безопасности неправильно.

Если мы забудем об аннотациях жизненных циклов и попытаемся скомпилировать эту функцию, то получим такую ошибку:

```
error[E0106]: missing lifetime specifier
 --> src/lib.rs:5:51
  |
5 | pub fn search(query: &str, contents: &str) -> Vec<&str> {
  |                                                   ^ expected lifetime
parameter
  |
```

```
= help: this function's return type contains a borrowed value, but the
signature does not say whether it is borrowed from `query` or `contents`
```

Компилятор не может знать, какой из двух аргументов нам нужен, поэтому мы должны сообщить об этом. Поскольку аргумент `contents` содержит весь текст, а мы хотим вернуть совпадающие части этого текста, мы знаем, что `contents` — именно тот аргумент, который нужно соединить с возвращаемым значением, используя синтаксис жизненного цикла.

В других языках программирования не требуется соединять аргументы с возвращаемыми значениями в сигнатуре. Хотя на первый взгляд, возможно, это покажется странным, но со временем все станет проще. Возможно, вы захотите сравнить этот пример с примером из раздела «Проверка ссылок с помощью жизненных циклов» (с. 235).

А теперь давайте выполним тест:

```
$ cargo test
    Compiling minigrep v0.1.0 (file:///projects/minigrep)
--warnings--
     Finished dev [unoptimized + debuginfo] target(s) in 0.43 secs
      Running target/debug/deps/minigrep-abcabcabc

running 1 test
test tests::one_result ... FAILED

failures:

---- tests::one_result stdout ----
        thread 'tests::one_result' panicked at 'assertion failed: `(left ==
right)`
left: `["safe, fast, productive."]`,
right: `[]`)', src/lib.rs:48:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::one_result

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out

error: test failed, to rerun pass '--lib'
```

Великолепно! Тест не сработал, как мы и ожидали. Давайте сделаем так, чтобы тест завершался успешно!

Написание кода для успешного завершения теста

В настоящее время наш тест не срабатывает, потому что мы всегда возвращаем пустой вектор. Для того чтобы это исправить и реализовать функцию `search`, программа должна выполнить следующие действия:

1. Перебрать каждую строку содержимого.
2. Проверить, содержит ли строка содержимого строку запроса.

3. Если содержит, то добавить эту строку в список возвращаемых значений.
4. Если не содержит, то ничего не делать.
5. Вернуть список совпавших результатов.

Давайте проработаем каждый шаг, начиная с перебора строк содержимого.

Перебор строк содержимого с помощью метода lines

В Rust есть полезный метод для работы с построчным перебором значений типа `String`, удобно названный `lines`, который работает, как показано в листинге 12.17. Обратите внимание, что этот код пока не компилируется.

Листинг 12.17. Перебор каждой строки содержимого в contents

src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        // сделать что-то со строкой текста
    }
}
```

Метод `lines` возвращает итератор. Мы подробно поговорим об итераторах в главе 13, но вспомните, что вы видели этот способ применения итератора в листинге 3.5, где мы использовали цикл `for` с итератором, чтобы выполнить код для каждого элемента в коллекции.

Поиск запроса в каждой строке

Далее мы проверим наличие строки запроса в текущей строке текста. К счастью, в типе `String` имеется полезный метод `contains`, который делает это за нас! Добавьте вызов метода `contains` в функцию `search`, как показано в листинге 12.18. Обратите внимание, что этот код по-прежнему не компилируется.

Листинг 12.18. Добавление функциональности для проверки, содержит ли строка текста запрос

src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        if line.contains(query) {
            // сделать что-то со строкой текста
        }
    }
}
```

Хранение совпавших строк текста

Нам также нужен способ хранения строк текста, содержащих строку запроса. Для этого мы создаем изменяемый вектор перед циклом `for` и вызываем метод `push`,

чтобы хранить строки текста в этом векторе. После цикла `for` мы возвращаем вектор, как показано в листинге 12.19.

Листинг 12.19. Хранение совпадающих строк с целью их последующего возвращения

src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
```

Теперь функция `search` должна возвращать только те строки текста, которые содержат `query`, и тест должен завершиться успешно. Давайте выполним тест:

```
$ cargo test
--пропуск--
running 1 test
test tests::one_result ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

На данном этапе мы могли бы рассмотреть возможности, чтобы рефакторить реализацию функции `search`, сохраняя при этом успешное прохождение тестов с целью поддержания той же функциональности. Код в функции `search` не так уж и плох, но в нем не использованы некоторые полезные свойства итераторов. Мы вернемся к этому примеру в главе 13, где подробно изучим итераторы и рассмотрим, как улучшить данный пример.

Использование функции search в функции run

Теперь, когда проверенная функция `search` работает, нужно вызвать `search` из функции `run`. Требуется передать значение `config.query` и содержимое `contents`, которое функция `run` читает из файла, в функцию `search`. Затем функция `run` выводит каждую строку текста, возвращаемую из функции `search`:

src/lib.rs

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.filename)?;

    for line in search(&config.query, &contents) {
        println!("{}", line);
    }

    Ok(())
}
```

Чтобы вернуть каждую строку из функции `search` и напечатать ее, мы по-прежнему используем цикл `for`.

Теперь вся программа должна работать! Давайте испытаем ее сначала со словом, которое должно вернуть ровно одну строчку из стихотворения Эмили Дикинсон — *frog* («лягушка»):

```
$ cargo run frog poem.txt
    Compiling minigrep v0.1.0 (file:///projects/minigrep)
     Finished dev [unoptimized + debuginfo] target(s) in 0.38 secs
      Running `target/debug/minigrep frog poem.txt`
How public, like a frog
```

Круто! Теперь давайте попробуем слово, которое совпадет с несколькими строками текста, к примеру, *body* («тело»):

```
$ cargo run body poem.txt
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/minigrep body poem.txt`
I'm nobody! Who are you?
Are you nobody, too?
How dreary to be somebody!
```

И наконец, давайте убедимся, что мы не получаем никаких строк, когда ищем слово, которого нет в стихотворении, к примеру, *monomorphization* («мономорфизация»):

```
$ cargo run monomorphization poem.txt
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/minigrep monomorphization poem.txt`
```

Превосходно! Мы создали свою мини-версию классического инструмента и многое узнали о том, как структурировать приложения. Мы также немного узнали о файловом вводе и выводе, жизненных циклах, тестировании и разборе командной строки.

В завершение этого проекта мы кратко продемонстрируем, как работать с переменными среды и как печатать в стандартный вывод ошибок — и то и другое полезно при написании программ командной строки.

Работа с переменными среды

Мы улучшим программу `minigrep`, добавив дополнительное средство — возможность поиска без учета регистра, которую пользователь активирует через переменную среды. Мы могли бы сделать это средство аргументом командной строки и требовать от пользователей, чтобы они вводили его всякий раз, когда хотят применить, но вместо этого воспользуемся переменной среды. Это позволит пользователям устанавливать переменную среды один раз, что сделает поиск нечувствительным к регистру в конкретном сеансе терминала.

Написание провального теста для функции search, нечувствительной к регистру

Мы хотим добавить новую функцию `search_case_insensitive` («нечувствительный к регистру поиск»), которую будем вызывать, когда переменная среды активирована. Мы продолжим следовать методике разработки на основе тестов, поэтому первым шагом снова будет написание провального теста. Мы добавим новый тест для новой функции `search_case_insensitive` и переименуем старый тест из `one_result` в `case_sensitive`, чтобы прояснить различия между этими тестами, как показано в листинге 12.20.

Листинг 12.20. Добавление нового провального теста для нечувствительной к регистру функции, которую мы намереваемся добавить

src/lib.rs

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

    #[test]
    fn case_sensitive() {
        let query = "duct";
        let contents = "\

Rust:
safe, fast, productive.
Pick three.
Duct tape.";

        assert_eq!(
            vec!["safe, fast, productive."],
            search(query, contents)
        );
    }

    #[test]
    fn case_insensitive() {
        let query = "rUsT";
        let contents = "\

Rust:
safe, fast, productive.
Pick three.
Trust me.";

        assert_eq!(
            vec!["Rust:", "Trust me."],
            search_case_insensitive(query, contents)
        );
    }
}
```

Обратите внимание, мы также отредактировали содержимое (`contents`) старого теста. Мы добавили новую строку с текстом `"Duct tape."` («клейкая лента»), используя заглавную букву D, которая не должна совпасть с искомым словом `"duct"`, когда мы ищем в режиме, чувствительном к регистру. Такое изменение старого теста нужно, чтобы мы ненароком не нарушили функциональность чувствительного к регистру поиска, которую мы уже реализовали. Этот тест должен завершиться успешно сейчас и в дальнейшем, по мере того как мы будем работать над поиском, нечувствительным к регистру.

Новый тест для нечувствительного к регистру поиска использует `"rUsT"` в качестве запроса. В функции `search_case_insensitive`, которую мы намерены добавить, запрос `"rUsT"` должен совпасть со строкой текста, содержащей `"Rust:"` с большой буквы R, а также совпасть со строкой текста `"Trust me."`, даже несмотря на то, что регистр обеих строк отличается от запроса. Это провальный тест, он не компилируется, потому что мы еще не определили функцию `search_case_insensitive`. Добавьте скелетную реализацию, которая всегда возвращает пустой вектор, подобно тому как мы сделали для функции `search` в листинге 12.16, чтобы удостовериться — тест компилируется и не срабатывает.

Реализация функции search_case_insensitive

Функция `search_case_insensitive` в листинге 12.21 будет почти такой же, как и функция `search`. Единственное отличие состоит в том, что мы будем переводить `query` и каждую строку текста в нижний регистр, поэтому независимо от регистра входных аргументов они будут в одинаковом регистре, когда мы проверяем, содержит ли строка текста запрос.

Листинг 12.21. Определение функции `search_case_insensitive` с переводом запроса и строки текста в нижний регистр перед их сравнением

src/lib.rs

```
pub fn search_case_insensitive<'a>(query: &str, contents: &'a str) -> Vec<&'a
str> {
 ❶ let query = query.to_lowercase();
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.to_lowercase()❷.contains(&query❸) {
            results.push(line);
        }
    }

    results
}
```

Сначала мы переводим строку запроса `query` в нижний регистр и сохраняем ее в затененной переменной с таким же именем ❶. Вызов `to_lowercase` для запроса необходим, чтобы независимо от того, каким является запрос со стороны пользо-

вателя — "rust", "RUST", "Rust" и "rUsT" — этот запрос рассматривался так, как если бы это был "rust", нечувствительный к регистру.

Обратите внимание, что query теперь относится к типу String, а не к строковому срезу, поскольку вызов метода to_lowercase создает новые данные, а не ссылается на существующие. Допустим, что запросом, к примеру, является "rUsT": этот строковый срез не содержит строчной буквы u или t, поэтому мы должны выделить новый экземпляр типа String, содержащий "rust". Когда мы теперь передаем query в качестве аргумента в метод contains, нам нужно добавить амперсанд ❸, потому что сигнатура метода contains по определению берет строковый срез.

Далее мы добавляем вызов метода to_lowercase для каждой line, который переводит все символы строки текста в нижний регистр, перед проверкой на наличие запроса query ❷. Теперь, конвертировав line и query в нижний регистр, мы найдем совпадения независимо от регистра запроса.

Давайте посмотрим, пройдет ли эта реализация проверку:

```
running 2 tests
test tests::case_insensitive ... ok
test tests::case_sensitive ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Отлично! Проверка пройдена. Теперь давайте вызовем новую функцию search_case_insensitive из функции run. Сначала мы добавим вариант конфигурации в структуру Config, чтобы переключать поиск с чувствительного к регистру на нечувствительный и наоборот. Добавление этого поля вызовет ошибки компилятора, так как мы еще нигде не инициализируем это поле:

src/lib.rs

```
pub struct Config {
    pub query: String,
    pub filename: String,
    pub case_sensitive: bool,
}
```

Обратите внимание, что мы добавили поле case_sensitive, которое содержит булево значение. Далее нужно, чтобы функция run проверяла значение поля case_sensitive и использовала его для принятия решения о том, какую функцию вызывать — search или search_case_insensitive, как показано в листинге 12.22. Обратите внимание, что этот код пока не компилируется.

Листинг 12.22. Вызов функции search либо search_case_insensitive в зависимости от значения в config.case_sensitive

src/lib.rs

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.filename)?;
```

```
    let results = if config.case_sensitive {
        search(&config.query, &contents)
    } else {
        search_case_insensitive(&config.query, &contents)
    };

    for line in results {
        println!("{}", line);
    }

    Ok(())
}
```

Наконец, нужно проверить переменную среды. Функции для работы с переменными среды находятся в модуле `env` стандартной библиотеки, поэтому мы хотим ввести этот модуль в область видимости с помощью строки `use std::env;` в верхней части src/lib.rs. Затем мы воспользуемся функцией `var` из модуля `env` для проверки переменной среды с именем `CASE_INSENSITIVE`, как показано в листинге 12.23.

Листинг 12.23. Проверка переменной среды с именем `CASE_INSENSITIVE`

src/lib.rs

```
use std::env;
// --пропуск--

impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("недостаточно аргументов");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        let case_sensitive = env::var("CASE_INSENSITIVE").is_err();

        Ok(Config { query, filename, case_sensitive })
    }
}
```

Здесь мы создаем новую переменную `case_sensitive`. Для того чтобы задать ее значение, мы вызываем функцию `env::var` и передаем ей имя переменной среды `CASE_INSENSITIVE`. Функция `env::var` возвращает экземпляр типа `Result`, который будет равен успешному варианту `Ok` и будет содержать значение переменной среды, если переменная среды задана. Она вернет вариант `Err`, если переменная среды не задана.

Мы используем метод `is_err` для экземпляра типа `Result`, чтобы проверить, имеется ли ошибка и задан ли он. Это значит, что программа должна выполнить поиск, чувствительный к регистру. Если переменная среды `CASE_INSENSITIVE` задана и имеет любое значение, то `is_err` вернет `false` и программа выполнит поиск, не-

чувствительный к регистру. Нас интересует не само значение переменной среды, а только то, что оно задано, поэтому мы проверяем `is_err`, не используя методы `unwrap`, `expect` или любые другие, которые мы видели в типе `Result`.

Мы передаем значение переменной `case_sensitive` в экземпляр структуры `Config`, благодаря чему функция `run` может прочитать это значение и принять решение о том, какую функцию следует вызывать — `search` или `search_case_insensitive`, как это было реализовано в листинге 12.22.

Давайте попробуем это сделать! Сначала выполним программу без заданной переменной среды и с запросом `to`, который должен совпасть с любой строкой текста со словом *to* в нижнем регистре:

```
$ cargo run to poem.txt
    Compiling minigrep v0.1.0 (file:///projects/minigrep)
     Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
      Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
```

Похоже, пока что все работает! Теперь давайте выполним программу с использованием переменной среды `CASE_INSENSITIVE`, равной 1, но с тем же запросом `to`.

Если вы используете PowerShell, то нужно будет установить переменную среды и выполнить программу двумя командами, а не одной:

```
$ $env:CASE_INSENSITIVE=1
$ cargo run to poem.txt
```

Мы должны получить строки со словом *to*, которые могут иметь буквы в верхнем регистре:

```
$ CASE_INSENSITIVE=1 cargo run to poem.txt
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
To tell your name the livelong day
To an admiring bog!
```

Отлично! У нас также есть строки текста, содержащие *To*! Программа `minigrep` теперь может выполнять нечувствительный к регистру поиск под управлением переменной среды. Таким образом, вы знаете, как управлять вариантами, заданными с аргументами командной строки и переменными среды.

Некоторые программы одновременно допускают и аргументы, и переменные среды для одной и той же конфигурации. В этих случаях принимается решение о приоритете одного либо другого. В качестве еще одного самостоятельного упражнения попробуйте поиграть с нечувствительностью к регистру посредством аргумента командной строки либо переменной среды. Примите решение о прио-

ритете аргумента командной строки или переменной среды, если программа выполняется, когда одно из двух задано как чувствительное к регистру, а другое — как нечувствительное к регистру.

Модуль `std::env` содержит много дополнительных полезных функций для работы с переменными среды. Обратитесь к его документации и посмотрите, что в нем есть.

Запись сообщений об ошибках в стандартный вывод ошибок вместо стандартного вывода данных

Пока что мы записываем все данные в терминал с помощью макрокоманды `println!`. Большинство терминалов обеспечивают два вида вывода: стандартный вывод данных (`stdout`) для общих сведений и стандартный вывод ошибок (`stderr`) для сообщений об ошибках. Это различие позволяет пользователям направлять успешный вывод программы в файл, но при этом выводить на экран сообщения об ошибках.

Макрокоманда `println!` предназначена только для стандартного вывода данных, поэтому мы должны использовать что-то еще для вывода в стандартный вывод ошибок.

Проверка места, куда записываются ошибки

Сначала давайте посмотрим, как содержимое, печатаемое программой `minigrep`, сейчас записывается в стандартный вывод данных, включая любые сообщения об ошибках, которые нужно записывать в стандартный вывод ошибок. Мы сделаем это, перенаправив поток стандартного вывода данных в файл, а также намеренно вызвав ошибку. Мы не будем перенаправлять поток стандартного вывода ошибок, поэтому любое содержимое, отправляемое в стандартный вывод ошибок, будет по-прежнему отображаться на экране.

Предполагается, что программы командной строки будут отправлять сообщения об ошибках в поток стандартного вывода ошибок, поэтому, даже если мы перенаправляем поток стандартного вывода данных в файл, сообщения об ошибках попрежнему видны на экране. Наша программа пока не очень хорошо работает, но мы скоро увидим, что она записывает вывод сообщений об ошибках в файл!

Можно продемонстрировать это, если выполнить программу с `>` и именем файла output.txt, в который мы хотим перенаправить поток стандартного вывода данных. Мы не станем передавать никаких аргументов, и это должно стать причиной ошибки:

```
$ cargo run > output.txt
```

Синтаксис > говорит, чтобы оболочка записала содержимое стандартного вывода данных не на экран, а в output.txt. Мы не увидели сообщения об ошибке на экране, как ожидали, а значит, оно должно было оказаться в файле. Вот что содержит файл output.txt:

```
Problem parsing arguments: not enough arguments
```

Точно, сообщение об ошибке выводится в стандартный вывод данных. Гораздо полезнее, чтобы сообщения об ошибках вроде этого выводились в стандартный вывод ошибок, вследствие чего в файл попадут только данные при успешном выполнении. Мы это изменим.

Запись сообщения об ошибках в стандартный вывод ошибок

Мы используем код из листинга 12.24, чтобы изменить способ вывода сообщений об ошибках. Из-за рефакторинга, проделанного ранее в этой главе, весь код, который выводит сообщения об ошибках, находится в одной функции — `main`. Стандартная библиотека предоставляет макрокоманду `eprintln!`, которая выполняет вывод в поток стандартного вывода ошибок. Поэтому давайте изменим два места, в которых для вывода ошибок мы вызывали макрокоманду `println!`, вместо этого воспользовавшись макрокомандой `eprintln!`.

Листинг 12.24. Запись сообщения об ошибках в стандартный вывод ошибок вместо стандартного вывода данных с использованием макрокоманды eprintln!

src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args).unwrap_or_else(|err| {
        eprintln!("Проблема с разбором аргументов: {}", err);
        process::exit(1);
    });

    if let Err(e) = minigrep::run(config) {
        eprintln!("Ошибка в приложении: {}", e);

        process::exit(1);
    }
}
```

После замены `println!` на `eprintln!` давайте снова выполним программу таким же образом, без аргументов, перенаправляя стандартный вывод данных с помощью >:

```
$ cargo run > output.txt
Problem parsing arguments: not enough arguments
```

Теперь мы видим ошибку на экране, а файл output.txt ничего не содержит, что и является ожидаемым поведением программ командной строки.

Давайте снова выполним программу, теперь уже с аргументами, которые не являются причиной ошибки, но все-таки перенаправляют стандартный вывод данных в файл, как тут:

```
$ cargo run to poem.txt > output.txt
```

Мы не увидим данных в терминале, а файл output.txt будет содержать результаты:

output.txt

```
Are you nobody, too?
How dreary to be somebody!
```

Это показывает, что теперь мы используем стандартный вывод данных для вывода успешных данных и стандартный вывод ошибок для вывода ошибок сообразно обстоятельствам.

Итоги

Мы свели воедино основные уже изученные идеи, а также всесторонне описали принцип работы операций ввода-вывода, часто встречающихся в Rust. Применяя аргументы командной строки, файлы, переменные среды и макрокоманду `eprintln!` для вывода ошибок, теперь вы готовы писать приложения командной строки. Используя идеи, описанные в предыдущих главах, вы сможете хорошо организовать код, проверить его, эффективно хранить данные в соответствующих структурах данных и обрабатывать ошибки.

Далее мы рассмотрим некоторые средства Rust, на которые оказали влияние функциональные языки — замыкания и итераторы.

13 Функциональные средства языка: итераторы и замыкания

Дизайн Rust был вдохновлен многими существующими языками и парадигмами, и одним из важных факторов является парадигма функционального программирования. Программирование в функциональном стиле часто предусматривает использование функций в качестве значений, передавая их в аргументах, возвращая их из других функций, присваивая их переменным для последующего исполнения и т. п.

В этой главе мы не будем говорить о том, что такое функциональное программирование, а обсудим некоторые средства Rust, похожие на средства многих языков, которые часто называются функциональными.

Мы рассмотрим:

- Замыкания — похожая на функцию конструкция, которую можно хранить в переменной.
- Итераторы — способ обработки серии элементов.
- Как использовать эти языковые средства для улучшения проекта ввода-вывода из главы 12.
- Производительность этих языковых средств (спойлер: они быстрее, чем вы думаете!).

Другие средства Rust, такие как сопоставление с паттернами и перечисления, рассмотренные нами в других главах, также находятся под влиянием функционального стиля. Освоение замыканий и итераторов — это важная составляющая написания выразительного и быстрого кода Rust, поэтому мы посвятим им всю главу.

Замыкание: анонимные функции, которые могут захватывать среду

Замыкания языка Rust — это анонимные функции, которые можно сохранять в переменной или передавать в качестве аргументов другим функциям. Вы можете

создавать замыкание в одном месте, а затем вызвать его для оценки в другом контексте. В отличие от функций, замыкания могут захватывать значения из области видимости, в которой они определены. Мы продемонстрируем, как эти средства обеспечивают повторное использование кода и индивидуализацию поведения.

Создание абстракции поведения с помощью замыканий

Давайте поработаем над примером ситуации, в которой полезно хранить замыкание для последующего исполнения. Попутно мы поговорим о синтаксисе замыканий, логическом выводе типов и типажах.

Рассмотрим гипотетическую ситуацию: мы работаем в стартапе, где создается приложение для генерирования индивидуальных планов тренировок. Бэкенд написан на Rust, и алгоритм, который генерирует план тренировок, берет в расчет многие факторы, такие как возраст пользователя приложения, индекс массы тела, предпочтения в упражнениях, последние тренировки и интенсивность, задаваемую пользователем. Фактический алгоритм, используемый в примере, не важен. Важно то, что этот расчет занимает несколько секунд. Мы хотим вызывать этот алгоритм, только когда нужно и лишь один раз, чтобы не заставлять пользователя ждать больше, чем необходимо.

Мы смоделируем вызов этого гипотетического алгоритма с помощью функции `simulated_expensive_calculation`, показанной в листинге 13.1, которая будет выводить сообщение

```
вычисляется медленно...
```

ждать 2 секунды, а затем возвращать число, которое мы передали.

Листинг 13.1. Функция, замещающая гипотетическое вычисление, которое занимает около 2 секунд

src/main.rs

```
use std::thread;
use std::time::Duration;

fn simulated_expensive_calculation(intensity: u32) -> u32 {
    println!("вычисляется медленно...");
    thread::sleep(Duration::from_secs(2));
    intensity
}
```

Далее идет функция `main`, содержащая важные для данного примера части приложения плана тренировок. Эта функция представляет собой код, который приложение будет вызывать, когда пользователь запрашивает план тренировок. Поскольку взаимодействие с фронтэндом приложения не имеет отношения к использованию замыканий, мы будем жестко встраивать в код все значения, представляющие программе данные на входе, и выведем данные на выходе.

Обязательные входные данные:

- Число интенсивности, заданное пользователем, конкретизируется, когда он запрашивает тренировку, и указывает, какая тренировка необходима — низкоинтенсивная или высокоинтенсивная.
- Случайное число, которое будет генерировать разнообразие в планах тренировки.

Результатом будет рекомендованный план тренировок. В листинге 13.2 показана функция `main`, которую мы будем использовать.

Листинг 13.2. Функция main с жестко заданными значениями для моделирования ввода пользователем данных и генерации случайных чисел

src/main.rs

```
fn main() {
    let simulated_user_specified_value = 10;
    let simulated_random_number = 7;

    generate_workout(
        simulated_user_specified_value,
        simulated_random_number
    );
}
```

Для простоты мы жестко закодировали переменную `simulated_user_specified_value` как 10, а переменную `simulated_random_number` как 7. В реальной программе мы бы получили число интенсивности из фронтэнда приложения и использовали бы упаковку `rand` для генерации случайного числа, как в игре из главы 2. Функция `main` вызывает функцию `generate_workout` с условными входными значениями.

Теперь, когда у нас есть контекст, давайте перейдем к алгоритму. Функция `generate_workout` в листинге 13.3 содержит логику функционирования приложения, которая в этом примере интересует нас больше всего. Остальные изменения в коде этого примера будут вноситься именно в эту функцию.

Листинг 13.3. Логика функционирования, которая выводит планы тренировок на основе входных данных и вызывает функцию simulated_expensive_calculation

src/main.rs

```
fn generate_workout(intensity: u32, random_number: u32) {
  ❶ if intensity < 25 {
        println!(
            "Сегодня сделайте {} отжиманий!",
            simulated_expensive_calculation(intensity)
        );
        println!(
            "Далее, сделайте {} приседаний!",
            simulated_expensive_calculation(intensity)
        );
    } else {
      ❷ if random_number == 3 {
```

```
            println!("Сделайте сегодня перерыв! Пейте больше воды!");
        ❸ } else {
            println!(
                "Сегодня пробежка {} минут!",
                simulated_expensive_calculation(intensity)
            );
        }
    }
}
```

Код в листинге 13.3 несколько раз вызывает функцию медленного вычисления. Первый блок `if` ❶ вызывает функцию `simulated_expensive_calculation` дважды, `if` внутри внешнего `else` ❷ не вызывает ее вообще, а код внутри второго `else case` ❸ вызывает ее один раз.

Желаемое поведение функции `generate_workout` состоит в том, чтобы сначала проверить, с какой интенсивностью пользователь хочет тренироваться — с низкой (обозначается числом меньше 25) или с высокой (число 25 или больше).

Планы тренировок с низкой интенсивностью будут рекомендовать ряд отжиманий и приседаний, основанных на сложном алгоритме, который мы моделируем.

Если пользователь хочет тренироваться с высокой интенсивностью, то существует дополнительный алгоритм: если значение случайного числа, генерируемого приложением, равно 3, то приложение посоветует передохнуть и попить воды. Если нет, то пользователь получит несколько минут пробежки на основе сложного алгоритма.

Этот код работает так, как этого хочет бизнес сейчас, но допустим, что команда исследователей данных решит, что в будущем нужно будет внести некоторые изменения в способ вызова функции `simulated_expensive_calculation`. Чтобы упростить обновления, когда эти изменения наступят, мы хотим провести рефакторинг этого кода так, чтобы он вызывал функцию `simulated_expensive_calculation` только один раз. Мы также хотим устранить то место, где сейчас без необходимости вызываем функцию дважды, в процессе не добавляя других вызовов этой функции. То есть мы не хотим вызывать ее, если результат не нужен, и все же хотим вызывать функцию только один раз.

Рефакторинг с использованием функций

Мы можем реструктурировать программу тренировок по-разному. Сначала мы попытаемся извлечь повторный вызов функции `simulated_expensive_calculation` в переменную, как показано в листинге 13.4.

Листинг 13.4. Извлечение вызовов функции simulated_expensive_calculation в одно место и сохранение результата в переменной expensive_result

src/main.rs

```
fn generate_workout(intensity: u32, random_number: u32) {
    let expensive_result = simulated_expensive_calculation(intensity);
```

```
    if intensity < 25 {
        println!(
            "Сегодня сделайте {} отжиманий!",
            expensive_result
        );
        println!(
            "Далее, сделайте {} приседаний!",
            expensive_result
        );
    } else {
        if random_number == 3 {
            println!("Сделайте сегодня перерыв! Пейте больше воды!");
        } else {
            println!(
                "Сегодня пробежка {} минут!",
                expensive_result
            );
        }
    }
}
```

Это изменение унифицирует все вызовы функции `simulated_expensive_calculation` и решает проблему, когда первый блок `if` без необходимости вызывает функцию дважды. Но, к сожалению, теперь мы вызываем эту функцию и ждем результата во всех случаях, включая внутренний блок `if`, который вообще не использует значение результата.

Мы хотим определить код в одном месте программы, но выполнить его только там, где действительно нужен результат. Этот случай хорошо подходит для использования замыканий!

Рефакторинг с замыканиями для сохранения кода в переменной

Вместо того чтобы всегда вызывать функцию `simulated_expensive_calculation` перед блоками `if`, мы можем определить замыкание и сохранить его в переменной, а не сохранять результат вызова функции, как показано в листинге 13.5. На самом деле, можно переместить все тело функции `simulated_expensive_calculation` внутрь замыкания, которое мы вводим ниже.

Листинг 13.5. Определение замыкания и сохранение его в переменной expensive_closure

src/main.rs

```
let expensive_closure = |num| {
    println!("вычисляется медленно...");
    thread::sleep(Duration::from_secs(2));
    num
};
```

Определение замыкания следует за оператором `=`, который назначает его переменной `expensive_closure`. Для того чтобы определить замыкание, мы начинаем

с пары вертикальных черт (`|`), внутри которых задаем параметры замыкания. Этот синтаксис был выбран из-за его сходства с определениями замыкания в Smalltalk и Ruby. Это замыкание имеет один параметр с именем `num`: если бы у нас было больше одного параметра, то мы бы разделили их запятыми, как `|param1, param2|`.

После параметров мы помещаем фигурные скобки, которые содержат тело замыкания, — они являются необязательными, если тело замыкания представлено одним выражением. В конце замыкания, после фигурных скобок, требуется точка с запятой в завершение инструкции `let`. Значение, возвращаемое из последней строки в теле замыкания (`num`), будет значением, возвращаемым из замыкания при его вызове, потому что эта строка кода не заканчивается точкой с запятой, как в телах функций.

Обратите внимание, инструкция `let` означает, что переменная `expensive_closure` содержит определение анонимной функции, а не результирующее значение вызова анонимной функции. Вспомните, что мы используем замыкание, потому что хотим определить некий код для того, чтобы вызвать его в одном месте, сохранить и вызвать его позже в другом месте. Код, который мы хотим вызвать, теперь хранится в переменной `expensive_closure`.

После определения замыкания мы можем изменить код в блоках `if` так, чтобы вызывать замыкание для выполнения кода и получать результирующее значение. Мы вызываем замыкание так же, как и функцию: указывая имя переменной, содержащей определение замыкания, и круглые скобки, включающие значения аргументов, которые мы хотим использовать, как показано в листинге 13.6.

Листинг 13.6. Вызов замыкания expensive_closure, которое мы определили

src/main.rs

```
fn generate_workout(intensity: u32, random_number: u32) {
    let expensive_closure = |num| {
        println!("вычисляется медленно...");
        thread::sleep(Duration::from_secs(2));
        num
    };

    if intensity < 25 {
        println!(
            "Сегодня сделайте {} отжиманий!",
            expensive_closure(intensity)
        );
        println!(
            "Далее, сделайте {} приседаний!",
            expensive_closure(intensity)
        );
    } else {
        if random_number == 3 {
            println!("Сделайте сегодня перерыв! Пейте больше воды!");
        } else {
            println!(
                "Сегодня пробежка {} минут!",
```

```
                expensive_closure(intensity)
            );
        }
    }
}
```

Теперь затратное вычисление вызывается только в одном месте, и мы выполняем этот код только там, где нужны результаты.

Однако мы вновь ввели одну из проблем из листинга 13.3: мы по-прежнему вызываем замыкание дважды в первом блоке `if`, который вызовет затратный код дважды и будет заставлять пользователя ждать вдвое дольше, чем нужно. Мы могли бы решить эту проблему, создав локальную для блока `if` переменную, которая будет содержать результат вызова замыкания. Но замыкание предоставляет нам другое решение. Мы вскоре поговорим о нем. Но сначала давайте разберемся, почему нет аннотаций типов в определении замыкания и типажей, связанных с замыканиями.

Логический вывод типа и аннотация замыкания

Замыкания не требуют аннотирования типов параметров или возвращаемого значения, как это делают функции `fn`. Аннотации типов являются обязательными для функций, потому что это часть интерфейса, демонстрируемого пользователям. Строгое определение этого интерфейса важно для проверки, что все согласны с тем, какие типы значений функция использует и возвращает. Но замыкания не применяются в таком публичном интерфейсе, как этот: они хранятся в переменных и используются без именования и демонстрации пользователям библиотеки.

Замыкания обычно бывают короткими и релевантными только в узком контексте, а не в любом произвольном сценарии. Внутри таких лимитированных контекстов компилятор способен надежно логически выводить типы параметров и возвращаемый тип, подобно тому, как он может определять типы большинства переменных.

Аннотировать типы в этих малых анонимных функциях будет для программистов надоедливой работой и в значительной степени избыточной с учетом информации, которая у компилятора уже есть.

Как и в случае с переменными, мы можем добавлять аннотации типов, если хотим повысить явную выраженность и ясность за счет большей детализации, чем это необходимо. Аннотирование типов для замыкания, которое мы определили в листинге 13.5, будет выглядеть так же, как определение в листинге 13.7.

Листинг 13.7. Добавление необязательных аннотаций типов параметров и возвращаемых значений в замыкание

src/main.rs

```
let expensive_closure = |num: u32| -> u32 {
    println!("вычисляется медленно...");
    thread::sleep(Duration::from_secs(2));
    num
};
```

С аннотациями типов синтаксис замыканий более похож на синтаксис функций. Ниже приводится вертикальное сравнение синтаксиса определения функции, прибавляющей 1 в свой параметр, и замыкание, которое имеет такое же поведение. Мы добавили несколько пробелов, чтобы выровнять соответствующие части. Этот пример иллюстрирует, что синтаксис замыкания похож на синтаксис функции, за исключением вертикальных черт и объема синтаксиса, который является необязательным:

```
fn add_one_v1    (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1 ;
```

В первой строке показано определение функции, а во второй — полностью аннотированное определение замыкания. Третья строка удаляет аннотации типов из определения замыкания, а четвертая — удаляет скобки, которые являются необязательными, поскольку тело замыкания имеет только одно выражение. Все это — допустимые определения, которые будут вести себя одинаково при вызове.

Определения замыканий будут иметь один конкретный тип, логически выводимый для каждого из параметров и возвращаемого значения. Например, в листинге 13.8 показано определение краткого замыкания, которое просто возвращает значение, полученное в качестве параметра. Это замыкание не очень полезно, если не считать целей данного примера. Обратите внимание, что мы не добавили никаких аннотаций типов в определение: если потом мы попытаемся вызвать замыкание дважды, используя тип `String` в качестве аргумента в первый раз, а тип `u32` — во второй, то произойдет ошибка.

Листинг 13.8. Попытка вызвать замыкание, типы которого логически выводятся из двух разных типов

src/main.rs

```
let example_closure = |x| x;

let s = example_closure(String::from("Привет"));
let n = example_closure(5);
```

Компилятор выдает такую ошибку:

```
error[E0308]: mismatched types
 --> src/main.rs
  |
  | let n = example_closure(5);
  |                         ^ expected struct `std::string::String`, found
integral variable
  |
  = note: expected type `std::string::String`
             found type `{integer}`
```

В первый раз, когда мы вызываем `example_closure` со значением типа `String`, компилятор логически выводит тип параметра `x` и тип, возвращаемый из замыкания,

как `String`. Эти типы затем запираются внутри замыкания в `example_closure`, и мы получаем ошибку типа, если пытаемся использовать другой тип с тем же замыканием.

Хранение замыканий с использованием обобщенных параметров и типажей Fn

Давайте вернемся к приложению, генерирующему тренировки. В листинге 13.6 код все так же вызывал замыкание затратного вычисления чаще, чем это было необходимо. Один из вариантов решения этой проблемы — сохранение результата затратного замыкания в переменной для повторного использования и применение переменной в каждом месте, где нужен результат, вместо повторного вызова замыкания. Однако этот метод приводит к большому объему повторяющегося кода.

К счастью, есть еще одно решение. Мы можем создать структуру, которая будет содержать замыкание и результирующее значение вызова замыкания. Структура будет исполнять замыкание только в том случае, если понадобится результирующее значение. Она будет кэшировать результирующее значение, благодаря чему остальной части кода не придется отвечать за сохранение и повторное использование результата. Вы, возможно, знаете этот паттерн как мемоизацию, или ленивое вычисление.

Для того чтобы собрать структуру, содержащую замыкание, необходимо указать для замыкания его тип, поскольку определение структуры требует типы для каждого поля. У каждого экземпляра замыкания есть уникальный анонимный тип: то есть, даже если два замыкания имеют одинаковую сигнатуру, их типы все равно считаются разными. Для определения структур, перечислений или параметров функций, использующих замыкания, мы используем обобщения и границы типажа, как мы обсуждали в главе 10.

Типажи `Fn` предусмотрены стандартной библиотекой. Все замыкания реализуют по крайней мере один из трех типажей: `Fn`, `FnMut` или `FnOnce`. Мы обсудим разницу между ними в разделе «Захватывание среды с помощью замыканий». В данном примере можно использовать типаж `Fn`.

Мы добавляем типы в границу типажа `Fn`, указывая типы параметров и возвращаемых значений, которые замыкания должны иметь, чтобы совпасть с этой границей типажа. В данном случае замыкание имеет параметр типа `u32` и возвращает тип `u32`, поэтому описываемая граница типажа равна `Fn(u32) -> u32`.

Листинг 13.9 показывает определение структуры `Cacher`, содержащей замыкание и необязательное значение результата.

Листинг 13.9. Определение структуры Cacher, содержащей замыкание в calculation и необязательный результат в value

src/main.rs

```
struct Cacher<T>
    where T: Fn(u32) -> u32
{
```

```
    calculation: T,
    value: Option<u32>,
}
```

Структура `Cacher` имеет поле `calculation` обобщенного типа `T`. Границы типажа для `T` уточняют, что оно является замыканием с типажом `Fn`. Любое замыкание, которое мы хотим сохранить в поле `calculation`, должно иметь один параметр типа `u32` (уточняемый внутри скобок после `Fn`) и возвращать тип `u32` (уточняемый после `->`).

ПРИМЕЧАНИЕ

Функции тоже могут реализовывать все три типажа `Fn`. Если то, что мы хотим сделать, не требует захватывания значения из среды, то можно использовать не замыкание, а функцию, где нам нужно что-то, что реализует типаж `Fn`.

Поле `value` имеет тип `Option<u32>`. Перед тем как мы исполним замыкание, `value` будет `None`. В момент, когда код, использующий структуру `Cacher`, запрашивает результат замыкания, `Cacher` исполнит замыкание и сохранит результат внутри варианта `Some` в поле `value`. Затем, если код опять запросит результат замыкания, то, не исполняя замыкание снова, `Cacher` вернет результат, содержащийся в варианте `Some`.

Алгоритм, связанный с только что описанным полем `value`, определен в листинге 13.10.

Листинг 13.10. Алгоритм кэширования структуры `Cacher`

src/main.rs

```
impl<T> Cacher<T>
 ❶ where T: Fn(u32) -> u32
{
 ❷ fn new(calculation: T) -> Cacher<T> {
     ❸ Cacher {
            calculation,
            value: None,
        }
    }

 ❹ fn value(&mut self, arg: u32) -> u32 {
        match self.value {
         ❺ Some(v) => v,
         ❻ None => {
               let v = (self.calculation)(arg);
               self.value = Some(v);
               v
            },
        }
    }
}
```

Мы хотим, чтобы структура `Cacher` управляла значениями своих полей, не давая вызывающему коду потенциально изменять значения в этих полях напрямую, поэтому поля являются приватными.

Функция `Cacher::new` берет обобщенный параметр `T` ❷, который мы определили как имеющий тот же типаж, что и структура `Cacher` ❶. Затем `Cacher::new` возвращает экземпляр структуры `Cacher` ❸, содержащий замыкание, указанное в поле `calculation`, и значение `None` в поле `value`, потому что замыкание еще не было исполнено.

Когда вызывающему коду требуется результат оценки замыкания, вместо прямого вызова замыкания он вызовет метод `value` ❹. Этот метод проверяет, есть ли у нас результирующее значение в `self.value` внутри `Some`. Если оно там есть, то он возвращает значение внутри `Some`, не исполняя замыкание снова ❺.

Если `self.value` равно `None`, то код вызывает замыкание, хранящееся в `self.calculation`, сохраняет результат `self.value` для будущего использования, а также возвращает значение ❻.

Листинг 13.11 показывает, как использовать структуру `Cacher` в функции `generate_workout` из листинга 13.6.

Листинг 13.11. Использование структуры Cacher в функции generate_workout, чтобы абстрагировать алгоритм кэширования

src/main.rs

```
fn generate_workout(intensity: u32, random_number: u32) {
  ❶ let mut expensive_result = Cacher::new(|num| {
        println!("вычисляется медленно...");
        thread::sleep(Duration::from_secs(2));
        num
    });

    if intensity < 25 {
        println!(
            "Сегодня сделайте {} отжиманий!",
          ❷ expensive_result.value(intensity)
        );
        println!(
            "Далее, сделайте {} приседаний!",
          ❸ expensive_result.value(intensity)
        );
    } else {
        if random_number == 3 {
            println!("Сделайте сегодня перерыв! Пейте больше воды!");
        } else {
            println!(
                "Сегодня пробежка {} минут!",
              ❹ expensive_result.value(intensity)
            );
        }
    }
}
```

Не сохраняя замыкание непосредственно в переменной, мы сохраняем новый экземпляр структуры `Cacher`, который содержит замыкание ❶. Затем в каждом месте, где мы хотим получить результат ❷❸❹, вызываем метод `value` для экземпляра `Cacher`. Можно вызывать метод `value` сколько угодно или не вызывать его вообще, и затратное вычисление будет исполнено максимум один раз.

Попробуйте выполнить эту программу с помощью функции `main` из листинга 13.2. Измените значения в переменных `simulated_user_specified_value` и `simulated_random_number`, чтобы проверить, что во всех случаях в различных блоках `if` и `else` сообщение `вычисляется медленно...` появляется только один раз и только при необходимости. `Cacher` отвечает за алгоритм, необходимый для того, чтобы не вызывать затратное вычисление чаще, чем нужно, благодаря чему функция `generate_workout` может сосредоточиться на логике функционирования.

Ограничения в реализации структуры Cacher

Кэширование значений в общем-то является полезным, его можно использовать в других частях кода с разными замыканиями. Однако в текущей реализации структуры `Cacher` есть две проблемы, которые затрудняют ее повторное использование в разных контекстах.

Первая проблема заключается в том, что для экземпляра структуры `Cacher` предполагается, что он всегда будет получать одно и то же значение для параметра `arg` в методе `value`. То есть этот тест структуры `Cacher` не сработает:

```
#[test]
fn call_with_different_values() {
    let mut c = Cacher::new(|a| a);

    let v1 = c.value(1);
    let v2 = c.value(2);

    assert_eq!(v2, 2);
}
```

Этот тест создает новый экземпляр структуры `Cacher` с замыканием, которое возвращает переданное в него значение. Мы вызываем метод `value` для этого экземпляра структуры `Cacher` со значением `arg`, равным 1, а затем значением `arg`, равным 2, и ожидаем, что вызов метода `value` со значением `arg`, равным 2, вернет 2.

Выполните этот тест, реализовав структуру `Cacher` из листингов 13.9 и 13.10, и тест не сработает на `assert_eq!` с таким сообщением:

```
thread 'call_with_different_values' panicked at 'assertion failed: `(left ==
right)`
  left: `1`,
 right: `2`', src/main.rs
```

Проблема состоит в том, что в первый раз, когда мы вызвали `c.value` с 1, экземпляр структуры `Cacher` сохранил `Some(1)` в `self.value`. После этого, независимо от того, что мы передаем в метод `value`, он всегда будет возвращать 1.

Попробуйте модифицировать структуру `Cacher` так, чтобы она содержала не одно значение, а хеш-отображение. Ключами хеш-отображения будут передаваемые значения `arg`, а значения хеш-отображения будут результатом вызова замыкания по этому ключу. Вместо того чтобы смотреть, есть ли в `self.value` напрямую значение `Some` или `None`, функция `value` будет искать `arg` в хеш-отображении и возвращать значение, если оно присутствует. Если его нет, то `Cacher` вызовет замыкание и сохранит полученное значение в хеш-отображении, связанном со значением его `arg`.

Вторая проблема в текущей реализации структуры `Cacher` заключается в том, что она принимает только те замыкания, которые берут один параметр типа `u32` и возвращают тип `u32`. Например, мы могли бы кэшировать результаты замыканий, которые принимают строковый срез и возвращают значения типа `usize`. В целях устранения этой проблемы попробуйте ввести более обобщенные параметры для повышения гибкости в функциональности структуры `Cacher`.

Захватывание среды с помощью замыканий

В примере генератора тренировок мы использовали замыкания только как встроенные анонимные функции. Однако у замыканий есть дополнительная возможность, которой нет у функций: они могут захватывать свою среду и обращаться к переменным из области видимости, в которой они определены.

В листинге 13.12 приведен пример замыкания, хранящегося в переменной `equal_to_x`, которая использует переменную `x` из среды замыкания.

Листинг 13.12. Пример замыкания, ссылающегося на переменную в окаймляющей ее области видимости

src/main.rs

```
fn main() {
    let x = 4;

    let equal_to_x = |z| z == x;

    let y = 4;

    assert!(equal_to_x(y));
}
```

Здесь, даже если `x` не является одним из параметров замыкания `equal_to_x`, указанному замыканию разрешено использовать переменную `x`, которая определена в той же области видимости, в которой определено замыкание `equal_to_x`.

Мы не можем сделать то же самое с функциями. Если мы попробуем это сделать в следующем примере, то код компилироваться не будет:

src/main.rs

```
fn main() {
    let x = 4;

    fn equal_to_x(z: i32) -> bool { z == x }

    let y = 4;

    assert!(equal_to_x(y));
}
```

Мы получаем ошибку[1]:

```
error[E0434]: can't capture dynamic environment in a fn item; use the || { ...
} closure form instead
 --> src/main.rs
  |
4 |     fn equal_to_x(z: i32) -> bool { z == x }
  |
```

Компилятор даже напоминает нам о том, что это работает только с замыканиями!

Когда замыкание захватывает значение из своей среды, оно использует память, сохраняя значения в теле замыкания. Такое использование памяти является теми накладными расходами, которые мы не хотим оплачивать в более общих случаях, когда нужно выполнить код, который не захватывает свою среду. Поскольку функции не позволяют захватывать среду, определение и использование функций не повлечет за собой таких накладных расходов.

Замыкания захватывают значения из своей среды тремя способами — они непосредственно соотносятся с тремя способами, которыми функция может брать параметр: беря во владение, заимствуя изменяемо и заимствуя неизменяемо. Они кодируются в трех типажах `Fn` следующим образом:

- `FnOnce` использует переменные, которые он захватывает из окаймляющей его области видимости, именуемой средой замыкания. Для того чтобы потребить захваченные переменные, замыкание должно брать их во владение и перемещать в замыкание, когда оно определено. Часть имени `Once` означает, что замыкание не может владеть одними и теми же переменными более одного раза, поэтому оно может быть вызвано только один раз.
- `FnMut` может изменять среду, потому что он заимствует значения изменяемо.
- `Fn` заимствует значения из среды неизменяемо.

Когда вы создаете замыкание, язык Rust логически выводит типаж, подлежащий использованию, основываясь на том, как замыкание использует значения из среды. Абсолютно все замыкания реализуют `FnOnce`, потому что все они могут быть

[1] `ошибка[E0434]: не удается захватить динамическую среду в элементе fn; вместо этого используйте форму замыкания || { ... }`

вызваны хотя бы один раз. Замыкания, которые не перемещают захваченные переменные, также реализуют `FnMut`, а замыкания, которым не нужен изменяемый доступ к захваченным переменным, также реализуют `Fn`. В листинге 13.12 замыкание `equal_to_x` неизменяемо заимствует `x` (поэтому `equal_to_x` имеет типаж `Fn`), поскольку телу замыкания нужно только считывать значение в `x`.

Если вы хотите, чтобы замыкание брало во владение значения, которые оно использует в среде, вы можете применить ключевое слово `move` перед списком параметров. Этот технический прием главным образом полезен при передаче замыкания нового потока, — он позволяет перемещать данные так, чтобы ими владел новый поток.

У нас будет больше примеров замыканий с ключевым словом `move` в главе 16, когда мы будем говорить о конкурентности. А пока — вот код из листинга 13.12 с ключевым словом `move`, добавленным в определение замыкания и использующим векторы вместо целых чисел, потому что целые числа можно копировать, а не перемещать. Обратите внимание, что этот код пока не компилируется.

src/main.rs

```
fn main() {
    let x = vec![1, 2, 3];

    let equal_to_x = move |z| z == x;

    println!("невозможно использовать x здесь: {:?}", x);

    let y = vec![1, 2, 3];

    assert!(equal_to_x(y));
}
```

Мы получаем следующую ошибку[1]:

```
error[E0382]: use of moved value: `x`
 --> src/main.rs:6:40
  |
4 |     let equal_to_x = move |z| z == x;
  |                      -------- value moved (into closure) here
5 |
6 |     println!("невозможно использовать x здесь: {:?}", x);
  |                                                ^ value used here
                                                   after move
  |
  = note: move occurs because `x` has type `std::vec::Vec<i32>`, which does
not implement the `Copy` trait
```

Значение `x` перемещается внутрь замыкания, когда замыкание определено, потому что мы добавили ключевое слово `move`. Затем замыкание берет `x` во владение,

[1] `ошибка[E0382]: использование перемещенного значения `x``

и функции `main` больше не разрешено использовать `x` в макрокоманде `println!`. Удаление макрокоманды `println!` исправит этот пример.

Во время указания одной из границ типажа `Fn` в большинстве случаев вы можете начать с `Fn`, а компилятор сообщит о необходимости иметь `FnMut` или `FnOnce`, основываясь на том, что происходит в теле замыкания.

В качестве примера, когда замыкания, которые захватывают свою среду, полезны в виде параметров функций, давайте перейдем к следующей теме — итераторам.

Обработка серии элементов с помощью итераторов

Итераторный паттерн позволяет поочередно выполнять работу с последовательностью элементов. Итератор отвечает за логику перебора каждого элемента и определяет, когда эта последовательность закончилась. Когда вы используете итераторы, вам не нужно повторно реализовывать эту логику самим.

В Rust итераторы являются ленивыми, то есть не действуют, пока вы не вызовете методы, которые потребляют итератор до его исчерпания. Например, код в листинге 13.13 создает итератор над элементами вектора `v1`, вызывая метод `iter`, определенный для `Vec<T>`. Этот код сам по себе не делает ничего полезного.

Листинг 13.13. Создание итератора

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();
```

После создания итератора можно использовать его различными способами. В листинге 3.5 мы применили итераторы с циклами `for`, чтобы исполнить код для каждого элемента, хотя тогда мы сознательно умолчали о том, что делает вызов метода `iter`, отложив это до настоящего момента.

Пример в листинге 13.14 отделяет создание итератора от его использования в цикле `for`. Итератор хранится в переменной `v1_iter`, и тогда не происходит никаких итераций. При вызове цикла `for` с применением итератора в `v1_iter` каждый элемент в итераторе используется в одной итерации цикла, которая выводит каждое значение.

Листинг 13.14. Использование итератора в цикле `for`

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();

for val in v1_iter {
    println!("Получено: {}", val);
}
```

В языках, в которых итераторы не предусмотрены стандартными библиотеками, вы, вероятно, пишете эту же функциональность, начав переменную в индексе 0, используя ее для доступа внутри вектора по индексу, чтобы получить значение, и увеличивая значение переменной в цикле до тех пор, пока оно не достигнет суммарного числа элементов в векторе.

Итераторы обрабатывают этот алгоритм за вас, сокращая повторяющийся код, который вы потенциально можете испортить. Итераторы дают больше гибкости при использовании одного и того же алгоритма с несколькими разными типами последовательностей, а не только с теми структурами данных, которые вы можете индексировать как векторы. Давайте посмотрим, как итераторы это делают.

Типаж Iterator и метод next

Все итераторы реализуют типаж `Iterator`, определенный в стандартной библиотеке. Определение этого типажа выглядит следующим образом:

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    // методы с реализациями по умолчанию опускаются
}
```

Обратите внимание, что в данном определении используется новый синтаксис: `type Item` и `Self::Item`, который определяет тип, *связанный* с этим типажом. Мы подробно поговорим о связанных типах в главе 19. А пока вам нужно знать лишь одно: этот код говорит о том, что реализация типажа `Iterator` требует, чтобы вы также определили тип `Item`, который используется в качестве типа, возвращаемого из метода `next`. Другими словами, `Item` будет типом, возвращаемым из итератора.

Типаж `Iterator` требует от разработчиков определения только одного метода — `next`, который возвращает по одному элементу итератора, завернутому в `Some`, за раз, а когда перебор завершается, он возвращает `None`.

Мы можем вызывать метод `next` непосредственно для итераторов. Листинг 13.15 демонстрирует, какие значения возвращаются при повторных вызовах метода `next` для итератора, созданного из вектора.

Листинг 13.15. Вызов метода `next` для итератора

src/lib.rs

```
#[test]
fn iterator_demonstration() {
    let v1 = vec![1, 2, 3];

    let mut v1_iter = v1.iter();

    assert_eq!(v1_iter.next(), Some(&1));
```

```
    assert_eq!(v1_iter.next(), Some(&2));
    assert_eq!(v1_iter.next(), Some(&3));
    assert_eq!(v1_iter.next(), None);
}
```

Обратите внимание, что нам нужно было сделать `v1_iter` изменяемым: вызов метода `next` для итератора изменяет внутреннее состояние, которое итератор использует для отслеживания своей позиции в последовательности. Другими словами, этот код потребляет, или расходует, итератор. Каждый вызов метода `next` съедает элемент из итератора. Когда мы использовали цикл `for`, нам не нужно было делать `v1_iter` изменяемым, потому что указанный цикл брал `v1_iter` во владение и делал его изменяемым неявно.

Также обратите внимание на то, что значения, которые мы получаем от вызовов метода `next`, являются неизменяемыми ссылками на значения в векторе. Метод `iter` создает итератор для неизменяемых ссылок. Если мы хотим создать итератор, который берет `v1` во владение и возвращает обладаемые значения, то мы вызываем `into_iter` вместо `iter`. Схожим образом, если мы хотим перебрать изменяемые ссылки, то мы вызываем `iter_mut` вместо `iter`.

Методы, которые потребляют итератор

В типаже `Iterator` имеется ряд разных методов с реализациями по умолчанию, предусмотренными стандартной библиотекой. Вы найдете сведения об этих методах, если заглянете в документацию API стандартной библиотеки о типаже `Iterator`. Некоторые из них вызывают метод `next` в своем определении, и именно поэтому требуется реализовать метод `next` во время реализации типажа `Iterator`.

Методы, которые вызывают `next`, называются «потребляющие адаптеры», потому что их вызов расходует итератор. Одним из примеров является метод `sum`, который берет итератор во владение и перебирает элементы, повторно вызывая `next`, тем самым потребляя итератор. По мере перебора он добавляет каждый элемент в промежуточную сумму и возвращает его по завершении итераций. В листинге 13.16 приводится тест, иллюстрирующий использование метода `sum`.

Листинг 13.16. Вызов метода sum для получения суммы всех элементов в итераторе

src/lib.rs

```
#[test]
fn iterator_sum() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum();

    assert_eq!(total, 6);
}
```

Не разрешается использовать `v1_iter` после вызова `sum`, потому что `sum` берет во владение итератор, для которого мы его вызываем.

Методы, которые производят другие итераторы

Другие методы, определенные для типажа `Iterator`, — «итераторные адаптеры» — позволяют переделывать итераторы в разные виды. Вы можете выстроить несколько вызовов итераторных адаптеров в цепь, выполняя сложные действия в удобном виде. Но поскольку все итераторы являются ленивыми, то для того чтобы получить результаты вызовов итераторных адаптеров, вам придется вызвать один из методов потребляющего адаптера.

В листинге 13.17 показан пример вызова метода `map` итераторного адаптера, который берет замыкание вызова для каждого элемента, производя новый итератор. Замыкание здесь создает новый итератор, в котором каждый элемент из вектора был увеличен на 1. Однако этот код выдает предупреждение.

Листинг 13.17. Вызов метода map итераторного адаптера для создания нового итератора

src/main.rs

```
let v1: Vec<i32> = vec![1, 2, 3];

v1.iter().map(|x| x + 1);
```

Мы получаем такое предупреждение[1]:

```
warning: unused `std::iter::Map` which must be used: iterator adaptors are
lazy and do nothing unless consumed
 --> src/main.rs:4:5
  |
4 |     v1.iter().map(|x| x + 1);
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: #[warn(unused_must_use)] on by default
```

Код в листинге 13.17 ничего не делает, указанное нами замыкание никогда не вызывается. Это предупреждение напоминает, почему итераторные адаптеры являются ленивыми, и здесь нам нужно употребить итератор.

Для того чтобы это исправить и употребить итератор, мы применим метод `collect`, использованный с `env:args` в листинге 12.1. Указанный метод потребляет итератор и собирает результирующие значения в коллекционный тип данных.

В листинге 13.18 мы собираем результаты перебора итератора, возвращаемые после вызова метода `map`, в вектор. Этот вектор в конечном итоге будет содержать каждый элемент из исходного вектора, увеличенный на 1.

---

[1] предупреждение: неиспользуемый 'std::iter::Map', который должен быть использован: итераторные адаптеры являются ленивыми и ничего не делают, если они не потребляются

Листинг 13.18. Вызов метода map для создания нового итератора и последующий вызов метода collect для потребления нового итератора и создания вектора

src/main.rs

```
let v1: Vec<i32> = vec![1, 2, 3];

let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();

assert_eq!(v2, vec![2, 3, 4]);
```

Поскольку метод `map` берет замыкание, мы можем указать любую операцию, которую хотим выполнять с каждым элементом. Это отличный пример того, как замыкания позволяют настраивать поведение под свои нужды, при этом повторно используя итерационное поведение, которое обеспечивается типажом `Iterator`.

Использование замыканий, которые захватывают свою среду

Теперь, когда мы познакомились с итераторами, можно продемонстрировать часто встречающееся использование замыканий, которые захватывают свою среду, применяя итераторный адаптер `filter`. Метод `filter` для итератора берет замыкание, которое в свою очередь берет каждый элемент из итератора и возвращает булево значение. Если замыкание возвращает `true`, то значение будет включено в результирующий итератор, созданный методом `filter`. Если замыкание возвращает `false`, то значение в этот итератор включено не будет.

В листинге 13.19 мы используем метод `filter` с замыканием, захватывающим переменную `shoe_size` из своей среды, чтобы выполнить перебор коллекции экземпляров структуры `Shoe`. Он вернет только ту обувь, которая имеет заданный размер.

Листинг 13.19. Использование метода filter с замыканием, которое захватывает shoe_size

src/lib.rs

```
#[derive(PartialEq, Debug)]
struct Shoe {
    size: u32,
    style: String,
}

❶ fn shoes_in_my_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe> {
  ❷ shoes.into_iter()
      ❸ .filter(|s| s.size == shoe_size)
      ❹ .collect()
    }

#[test]
fn filters_by_size() {
    let shoes = vec![
```

```
            Shoe { size: 10, style: String::from("кроссовки") },
            Shoe { size: 13, style: String::from("сандалии") },
            Shoe { size: 10, style: String::from("ботинки") },
        ];

        let in_my_size = shoes_in_my_size(shoes, 10);

        assert_eq!(
            in_my_size,
            vec![
                Shoe { size: 10, style: String::from("кроссовки") },
                Shoe { size: 10, style: String::from("ботинки") },
            ]
        );
    }
```

Функция `shoes_in_my_size` берет во владение вектор и размер обуви в качестве параметров ❶. Она возвращает вектор, содержащий обувь только заданного размера.

В теле функции `shoes_in_my_size` мы вызываем метод `into_iter` для создания итератора, который берет вектор во владение ❷. Затем мы вызываем метод `filter`, чтобы переделать этот итератор в новый, содержащий только те элементы, для которых замыкание возвращает `true` ❸.

Замыкание захватывает параметр `shoe_size` из среды и сравнивает его значение с размером каждого вида обуви, сохраняя только обувь заданного размера. Наконец, вызов метода `collect` собирает значения, возвращенные переделанным итератором, в вектор, возвращаемый функцией ❹.

Тест показывает, что, когда мы вызываем функцию `shoes_in_my_size`, мы получаем обратно только ту обувь, которая имеет тот же размер, что и заданное значение.

Создание собственных итераторов с помощью типажа Iterator

Мы показали, что вы можете создавать итератор, вызывая `iter`, `into_iter` или `iter_mut` для вектора. Итераторы можно создавать из других типов коллекций стандартной библиотеки, таких как хеш-отображение. Вы также можете создавать итераторы, которые делают все, что вы хотите, путем реализации типажа `Iterator` в собственных типах. Как уже упоминалось ранее, метод `next` — единственный, для которого требуется определение. После того как вы его предоставите, вы сможете использовать все другие методы с реализациями по умолчанию, предусмотренные типажом `Iterator`!

В качестве примера давайте создадим итератор, который будет всегда считать только от 1 до 5. Сначала мы сделаем структуру для хранения нескольких значений. Затем мы превратим ее в итератор, реализовав типаж `Iterator` и используя эти значения в указанной реализации.

Листинг 13.20 содержит определение структуры `Counter` и связанную с ней функцию `new` для создания экземпляров структуры `Counter`.

Листинг 13.20. Определение структуры Counter и функции new, создающей экземпляры структуры Counter с начальным значением поля count, равным 0

src/lib.rs

```
struct Counter {
    count: u32,
}

impl Counter {
    fn new() -> Counter {
        Counter { count: 0 }
    }
}
```

Структура `Counter` имеет одно поле с именем `count`. Это поле содержит значение типа `u32`, которое будет отслеживать, где мы находимся в процессе перебора значений от 1 до 5. Поле `count` является приватным, потому что мы хотим, чтобы реализация структуры `Counter` управляла своим значением. Функция `new` обеспечивает поддержку поведения, состоящего в том, что новые экземпляры всегда начинаются со значения 0 в поле `count`.

Далее мы выполним реализацию типажа `Iterator` для типа `Counter` путем определения тела метода `next`, чтобы уточнить, что должно происходить, когда этот итератор используется, как показано в листинге 13.21.

Листинг 13.21. Реализация типажа Iterator в структуре Counter

src/lib.rs

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        self.count += 1;

        if self.count < 6 {
            Some(self.count)
        } else {
            None
        }
    }
}
```

В итераторе мы устанавливаем связанный тип `Item` равным `u32`, то есть итератор будет возвращать значения типа `u32`. Опять-таки, пока не переживайте насчет связанных типов, мы рассмотрим их в главе 19.

Мы хотим, чтобы итератор прибавлял к текущему состоянию 1, поэтому мы инициализировали `count` со значением 0, в результате чего он сначала будет возвращать 1. Если значение `count` меньше 6, то метод `next` будет возвращать текущее

значение, завернутое в `Some`, но если `count` равен 6 или выше, то итератор будет возвращать `None`.

Использование метода next итератора структуры Counter

После того как мы реализовали типаж `Iterator`, у нас появился итератор! В листинге 13.22 показан тест, демонстрирующий, что мы можем использовать итераторную функциональность структуры `Counter`, напрямую вызывая метод `next` точно так же, как мы это делали с итератором, созданным из вектора в листинге 13.15.

Листинг 13.22. Тестирование функциональности реализации метода next

src/lib.rs

```
#[test]
fn calling_next_directly() {
    let mut counter = Counter::new();

    assert_eq!(counter.next(), Some(1));
    assert_eq!(counter.next(), Some(2));
    assert_eq!(counter.next(), Some(3));
    assert_eq!(counter.next(), Some(4));
    assert_eq!(counter.next(), Some(5));
    assert_eq!(counter.next(), None);
}
```

Этот тест создает в переменной `counter` новый экземпляр структуры `Counter`, а затем повторно вызывает метод `next`, проверяя, что мы реализовали нужное поведение итератора, а именно возвращение значений от 1 до 5.

Использование других методов типажа Iterator

Мы реализовали типаж `Iterator` путем определения метода `next`, поэтому теперь мы можем использовать реализации по умолчанию любого метода типажа `Iterator`, определенного в стандартной библиотеке, поскольку все они используют функциональность метода `next`.

Например, если по какой-то причине мы хотим взять значения, произведенные экземпляром структуры `Counter`, соединить их в пары со значениями, произведенными еще одним экземпляром структуры `Counter` после пропуска первого значения, перемножить пары между собой, сохранить только те результаты, которые делятся на 3, и сложить все полученные значения вместе, то мы можем сделать это, как показано в тесте из листинга 13.23.

Листинг 13.23. Использование различных методов типажа Iterator для итератора структуры Counter

src/lib.rs

```
#[test]
fn using_other_iterator_trait_methods() {
    let sum: u32 = Counter::new().zip(Counter::new().skip(1))
                                 .map(|(a, b)| a * b)
```

```
                                    .filter(|x| x % 3 == 0)
                                    .sum();
        assert_eq!(18, sum);
    }
```

Обратите внимание, что метод `zip` производит только четыре пары. Теоретически возможная пятая пара `(5, None)` никогда не производится, потому что `zip` возвращает `None`, когда любой из его входных итераторов возвращает `None`.

Вызовы всех этих методов становятся возможными благодаря тому, что мы прописали, как работает метод `next`, а стандартная библиотека предоставляет реализации по умолчанию для других методов, которые вызывают `next`.

Улучшение проекта ввода-вывода

С учетом новых знаний об итераторах мы можем улучшить проект ввода-вывода из главы 12 путем применения итераторов, сделав места в коде более четкими и лаконичными. Давайте посмотрим, как итераторы улучшат реализацию функций `Config::new` и `search`.

Удаление метода clone с помощью Iterator

В листинг 12.6 мы добавили код, который брал срез значений типа `String` и создавал экземпляр структуры `Config` путем индексирования внутри этого среза и клонирования значений, позволяя структуре `Config` владеть этими значениями. В листинге 13.24 мы воспроизвели реализацию функции `Config::new`, как это было в листинге 12.23.

Листинг 13.24. Воспроизведение функции Config::new из листинга 12.23

src/lib.rs

```
impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("недостаточно аргументов");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        let case_sensitive = env::var("CASE_INSENSITIVE").is_err();

        Ok(Config { query, filename, case_sensitive })
    }
}
```

В то время мы отметили, что не стоит беспокоиться о неэффективных вызовах метода `clone`, потому что мы удалим их в будущем. Ну что ж, время пришло!

Здесь мы нуждались в методе `clone` потому, что в параметре `args` у нас срез с элементами типа `String`, но функция `new` параметром `args` не владеет. Для того чтобы возвращать владение экземплярам структуры `Config`, нам пришлось клонировать значения из полей `query` и `filename` структуры `Config`, и благодаря этому экземпляр структуры `Config` может владеть своими значениями.

С учетом новых знаний об итераторах мы можем изменить функцию `new` так, чтобы в качестве своего аргумента вместо заимствования среза она брала во владение итератор. Мы будем использовать итераторную функциональность вместо кода, который проверяет длину среза и выполняет индексирование в определенные места. Это сделает яснее принцип работы функции `Config::new`, поскольку итератор будет обращаться к значениям.

После того как `Config::new` берет итератор во владение и перестает использовать заимствующие индексные операции, мы можем больше не вызывать метод `clone` и не выделять новое пространство памяти, а переместить значения типа `String` из итератора в структуру `Config`.

Использование возвращаемого итератора напрямую

Откройте файл src/main.rs проекта ввода-вывода, который должен выглядеть следующим образом:

src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args).unwrap_or_else(|err| {
        eprintln!("Проблема с разбором аргументов: {}", err);
        process::exit(1);
    });

    // --пропуск--
}
```

Мы изменим начало функции `main`, которое было в листинге 12.24, на код из листинга 13.25. Этот код не будет компилироваться до тех пор, пока мы также не обновим `Config::new`.

Листинг 13.25. Передача возвращаемого значения из функции `env::args` в функцию `Config::new`

src/main.rs

```
fn main() {
    let config = Config::new(env::args()).unwrap_or_else(|err| {
        eprintln!("Проблема с разбором аргументов: {}", err);
        process::exit(1);
    });

    // --пропуск--
}
```

Функция `env::args` возвращает итератор! Вместо того чтобы собирать итераторные значения в вектор и затем передавать срез в метод `Config::new`, теперь мы напрямую передаем владение итератора, возвращаемого из функции `env::args`, в функцию `Config:: new`.

Далее нужно обновить определение `Config::new`. В файле src/lib.rs проекта ввода-вывода давайте изменим сигнатуру функции `Config::new` так, чтобы она выглядела как в листинге 13.26. Этот код пока не компилируется, потому что нужно обновить тело функции.

Листинг 13.26. Обновление сигнатуры функции `Config::new`, в результате которого ожидается итератор

src/lib.rs

```
impl Config {
    pub fn new(mut args: std::env::Args) -> Result<Config, &'static str> {
        // --пропуск--
```

Документация стандартной библиотеки для функции `env::args` показывает, что тип возвращаемого ею итератора равен `std::env::Args`. Мы обновили сигнатуру функции `Config::new`, поэтому параметр `args` вместо типа `&[String]` теперь имеет тип `std::env::Args`. Поскольку мы берем параметр `args` во владение и будем изменять `args` путем перебора, мы можем добавить в описание параметра `args` ключевое слово `mut`, сделав его изменяемым.

Использование методов типажа Iterator вместо индексирования

Далее мы исправим тело метода `Config::new`. В документации стандартной библиотеки также упоминается, что `std::env::Args` реализует типаж `Iterator`, поэтому мы знаем, что можем вызывать для него метод `next`! В листинге 13.27 обновлен код из листинга 12.23 посредством использования метода `next`.

Листинг 13.27. Изменение тела функции Config::new с целью использования итераторных методов

src/lib.rs

```
impl Config {
    pub fn new(mut args: std::env::Args) -> Result<Config, &'static str> {
        args.next();

        let query = match args.next() {
            Some(arg) => arg,
            None => return Err("Не получена строка запроса"),
        };

        let filename = match args.next() {
            Some(arg) => arg,
            None => return Err("Не получено имя файла"),
        };
```

```
        let case_sensitive = env::var("CASE_INSENSITIVE").is_err();

        Ok(Config { query, filename, case_sensitive })
    }
}
```

Как вы помните, первым значением в возвращаемом значении `env::args` является имя программы. Мы хотим его проигнорировать и перейти к следующему значению, поэтому сначала мы вызываем `next` и ничего с возвращенным значением не делаем. Далее мы вызываем `next` и получаем значение, которое хотим поместить в поле `query` структуры `Config`. Если `next` возвращает значение `Some`, то мы используем выражение `match` для извлечения значения. Если он возвращает `None`, то это означает, что было дано недостаточно аргументов, и мы возвращаемся досрочно со значением `Err`. Мы делаем то же самое для значения `filename`.

Написание более ясного кода с помощью итераторных адаптеров

Мы можем воспользоваться преимуществами итераторов и в функции `search` проекта ввода-вывода, которая воспроизводится в листинге 13.28 так, как это было в листинге 12.19.

Листинг 13.28. Реализация функции search из листинга 12.19

src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
```

Мы можем написать этот код в более сжатом виде, используя методы итераторных адаптеров. Они также позволяют избежать изменяемого вектора промежуточных результатов `results`. Функциональный стиль программирования предпочитает минимизировать объем изменяемых состояний, чтобы сделать код понятнее. Удаление изменяемого состояния может обеспечить возможность для будущих улучшений, которые позволят выполнять процедуру поиска в параллельном режиме, поскольку нам не придется управлять конкурентным доступом к вектору `results`. Это изменение показано в листинге 13.29.

Листинг 13.29. Использование методов итераторных адаптеров в реализации функции search

src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    contents.lines()
        .filter(|line| line.contains(query))
        .collect()
}
```

Напомним, что функция `search` предназначена для возвращения в `contents` всех строк текста, которые содержат запрос `query`. Как и в примере с `filter` из листинга 13.19, этот код использует адаптер `filter`, который оставляет только те строки, для которых `line.contains(query)` возвращает `true`. Затем мы собираем совпадающие строки текста в еще один вектор с помощью метода `collect`. Гораздо проще! Вы можете свободно внести такое же изменение с использованием итераторных методов и в функцию `search_case_insensitive`.

Следующий логический вопрос заключается в том, какой стиль вы должны выбрать в коде и почему: исходную реализацию в листинге 13.28 или версию с использованием итераторов в листинге 13.29. Большинство программистов Rust предпочитают использовать итераторный стиль. Поначалу к нему трудновато привыкнуть, но после того, как вы опробуете различные итераторные адаптеры и их действия, итераторы станут понятными. Вместо того чтобы возиться с различными частями цикла и собирать новые векторы, код сосредоточен на высокоуровневой цели цикла. Он абстрагируется от обыкновенного кода, и поэтому становится легче увидеть уникальные для этого кода понятия, такие как условие фильтрации, которое каждый элемент итератора должен пройти.

Но действительно ли эти две реализации эквивалентны? Интуитивно можно допустить, что более низкоуровневый цикл будет быстрее. Давайте поговорим о производительности.

Сравнение производительности: циклы против итераторов

Для того чтобы определиться с тем, что использовать — циклы или итераторы, — необходимо знать, какая версия функции `search` работает быстрее: версия с явно выраженным циклом `for` или версия с итераторами.

Мы провели сравнительный анализ, загрузив все содержимое «Приключений Шерлока Холмса» сэра Артура Конан Дойла в экземпляр типа `String` и проведя поиск слова *the* в содержимом. Вот результаты сравнительного тестирования версии функции `search` с циклом `for` и с итераторами:

```
test bench_search_for ... bench: 19,620,300 ns/iter (+/- 915,700)
test bench_search_iter ... bench: 19,234,900 ns/iter (+/- 657,200)
```

Итераторная версия оказалась чуть-чуть быстрее! Мы не будем здесь объяснять исходный код, потому что речь не о том, чтобы доказать, что эти две версии эквивалентны, а о том, чтобы получить общее представление, как эти реализации соотносятся с точки зрения производительности.

В целях всестороннего сравнительного тестирования вы должны провести проверку, используя тексты разных размеров в качестве содержимого и слова разной длины в качестве запросов, а также другие всевозможные вариации. Суть в следующем: итераторы, хотя и являются высокоуровневой абстракцией, компилируются примерно до такого же кода, как если бы вы сами написали код более низкого уровня. Итераторы в языке Rust представляют собой одну из абстракций с нулевой стоимостью, под которой мы подразумеваем, что использование абстракции не накладывает никакого расхода времени выполнения. Это аналогично тому, как Бьерн Страуструп, первый дизайнер и разработчик C++, определяет нулевые накладные расходы в «Основах C++» (2012):

> *В общем, реализации C++ подчиняются принципу нулевых накладных расходов: вы не платите за то, что не используете. И далее: а с тем, что вы применяете, лучшего кода вручную вы и написать не сможете.*

В качестве еще одного примера можно привести следующий код, взятый из аудиодекодера. Алгоритм декодирования использует математическую операцию линейного предсказания, чтобы оценивать будущие значения на основе линейной функции предыдущих выборок. Этот код использует итераторную цепочку для выполнения математических вычислений с тремя переменными в области видимости: со срезом данных `buffer`, массивом `coefficients` из 12 коэффициентов и суммой, на которую данные должны быть сдвинуты в `qlp_shift`. В этом примере мы объявили переменные, но не дали им никаких значений. Хотя этот код не очень важен вне своего контекста, это все равно краткий, реальный пример того, как язык Rust транслирует высокоуровневые идеи в низкоуровневый код.

```
let buffer: &mut [i32];
let coefficients: [i64; 12];
let qlp_shift: i16;

for i in 12..buffer.len() {
    let prediction = coefficients.iter()
                                 .zip(&buffer[i - 12..i])
                                 .map(|(&c, &s)| c * s as i64)
                                 .sum::<i64>() >> qlp_shift;
    let delta = buffer[i];
    buffer[i] = prediction as i32 + delta;
}
```

Для того чтобы рассчитать значение переменной `prediction`, этот код перебирает каждое из 12 значений в `coefficients` и использует метод `zip` для соединения значений коэффициентов в пары с предыдущими 12 значениями в `buffer`. Затем

мы умножаем значения каждой пары, суммируем все результаты и сдвигаем биты в сумме бит `qlp_shift` вправо.

В вычислениях в таких приложениях, как аудиодекодеры, часто отдают наиболее высокий приоритет производительности. Здесь мы создаем итератор, используя два адаптера, а затем потребляем значение. В какой ассемблерный код будет компилироваться этот код на языке Rust? Скажем так, на момент написания этой книги он компилируется до той же сборки, которую вы бы написали от руки. Там вообще нет цикла, соответствующего перебору значений `coefficients`: язык Rust знает, что имеется 12 итераций, и поэтому он «разворачивает» цикл. Разворачивание представляет собой оптимизацию, которая устраняет накладные расходы по управлению циклом в коде и вместо этого генерирует повторяющийся код для каждой итерации цикла.

Все коэффициенты хранятся в регистрах — это значит, что доступ к значениям происходит очень быстро. Нет никаких проверок границ доступа к массиву во время выполнения. Все эти оптимизации, которые язык Rust способен применять, делают результирующий код чрезвычайно эффективным. Теперь, зная это, вы можете использовать итераторы и замыкания без страха! Они делают код внешне более высокоуровневым, но при этом не налагают за это штраф в виде снижения производительности времени выполнения.

Итоги

Замыкания и итераторы — это средства Rust, вдохновленные идеями языков функционального программирования. Они помогаюти Rust четко выражать высокоуровневые идеи при производительности, характерной для низкого уровня. Реализации замыканий и итераторов таковы, что производительность времени выполнения не страдает. Отчасти это способствует тому, чтобы достичь абстракций с нулевой стоимостью.

Теперь, улучшив выразительность проекта ввода-вывода, давайте рассмотрим еще несколько средств консольной команды `cargo`, которые помогут нам поделиться этим проектом со всем миром.

14 Подробнее о Cargo и Crates.io

До сих пор мы использовали только базовые средства пакетного менеджера Cargo, предназначенные для создания, выполнения и тестирования кода, но данный инструмент может делать гораздо больше. В этой главе мы обсудим более продвинутые его средства и покажем, как делать следующее:

- Настраивать сборки с помощью релизных профилей.
- Публиковать библиотеки в https://crates.io/.
- Организовывать крупные проекты с рабочими пространствами.
- Инсталлировать двоичные файлы из https://crates.io/.
- Расширять Cargo с помощью индивидуальных команд.

Cargo может делать даже больше, чем то, что мы рассматриваем в этой главе, поэтому полное объяснение всех его средств смотрите в документации по адресу https://doc.rust-lang.org/cargo/.

Собственная настройка сборок с помощью релизных профилей

В языке Rust релизные профили — это предопределенные и собственно настраиваемые профили с разными конфигурациями, которые позволяют программисту иметь бóльший контроль над различными вариантами компиляции кода. Каждый профиль настраивается независимо от других.

Cargo состоит из двух основных профилей: профиля `dev`, который Cargo использует при выполнении команды `cargo build`, и профиля `release`, используемый Cargo при выполнении команды `cargo build --release`. Профиль `dev` определяется со значениями по умолчанию, пригодными для разработки, а профиль `release` имеет значения по умолчанию, пригодные для релизных сборок.

Возможно, эти имена профилей знакомы вам по выходным данным сборок:

```
$ cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
$ cargo build --release
    Finished release [optimized] target(s) in 0.0 secs
```

Профили `dev` и `release`, показанные в этих данных сборки, говорят о том, что компилятор использует разные профили.

У Cargo есть настройки по умолчанию для каждого профиля, который применяется, когда в файле Cargo.toml нет разделов `[profile.*]`. Добавляя разделы `[profile.*]` для любого профиля, который вы хотите настроить, вы можете переопределить любое подмножество настроек по умолчанию. Например, вот значения по умолчанию параметра `opt-level` для профилей `dev` и `release`:

Cargo.toml

```
[profile.dev]
opt-level = 0

[profile.release]
opt-level = 3
```

Параметр `opt-level` управляет числом оптимизаций, которые будут применяться к коду, в интервале от 0 до 3. Применение большего числа оптимизаций увеличивает время компиляции, поэтому, если вы ведете разработку и часто компилируете код, то вам потребуется более быстрая компиляция, даже если полученный код работает медленнее. Именно поэтому уровень `opt-level` по умолчанию равен 0. Когда вы готовы сделать релиз своего кода, то лучше всего потратить больше времени на компиляцию. Вы скомпилируете в релизном режиме лишь один раз, но будете выполнять скомпилированную программу многократно, поэтому в релизном режиме мы жертвуем более длительной компиляцией ради кода, который работает быстрее. Вот почему `opt-level` по умолчанию для профиля `release` равен 3.

Вы можете переопределить любую настройку по умолчанию, добавив для нее другое значение в файл Cargo.toml. Например, если мы хотим использовать `opt-level`, равный 1, в профиле `release`, то можно добавить следующие две строчки в файл Cargo.toml нашего проекта:

Cargo.toml

```
[profile.dev]
opt-level = 1
```

Этот код переопределяет значение по умолчанию, равное 0. Теперь, когда мы выполняем команду `cargo build`, Cargo будет использовать значения по умолчанию для профиля `dev` плюс нашу индивидуальную настройку для `opt-level`. Поскольку мы установили `opt-level` равным 1, Cargo будет применять больше оптимизаций, чем с настройками по умолчанию, но не так много, как в релизной сборке.

Полный список вариантов конфигурации и значений по умолчанию для каждого профиля смотрите в документации Cargo https://doc.rust-lang.org/cargo/.

Публикация упаковки для Crates.io

Мы использовали пакеты из https://crates.io/ как зависимости проекта, но вы также можете поделиться своим кодом с другими людьми, опубликовав собственные пакеты. Реестр упаковок по адресу https://crates.io/ распространяет исходный код ваших пакетов, и поэтому он содержит по преимуществу открытый исходный код.

В Rust и Cargo есть средства, которые в первую очередь помогают упростить использование и поиск опубликованного пакета для других людей. Далее мы поговорим о некоторых из этих средств, а затем объясним процедуру публикации пакета.

Внесение полезных документационных комментариев

Точное документирование пакетов поможет другим пользователям быть в курсе того, как и когда их использовать, поэтому стоит потратить время на написание документации. В главе 3 мы обсуждали, как комментировать код Rust, используя две косые черты `//`. В Rust также имеется особый вид комментария для документирования, у него удобное название — «документационный комментарий». Он будет генерировать документацию в формате HTML. HTML показывает содержимое документационных комментариев элементов публичного API, предназначенных для программистов, которые хотят знать, как использовать вашу упаковку, а не как она реализована.

Документационные комментарии вместо двух косых черт используют три косые черты `///` и для форматирования текста поддерживают нотацию упрощенной разметки (Markdown). Помещайте документационные комментарии непосредственно перед элементом, который они документируют. В листинге 14.1 показаны документационные комментарии для функции `add_one` в упаковке с именем `my_crate`.

Листинг 14.1. Документационный комментарий для функции

src/lib.rs

```
/// Добавляет единицу к заданному числу.
///
/// # Примеры
///
/// ```
/// let arg = 5;
/// let answer = my_crate::add_one(arg);
///
/// assert_eq!(6, answer);
```

```
/// ```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

Здесь мы описываем, что делает функция `add_one`, начинаем раздел с заголовка `Примеры`, а затем предоставляем код, который демонстрирует применение функции `add_one`. Мы можем сгенерировать HTML-документацию из этого документационного комментария, выполнив команду `cargo doc`. Эта команда запускает инструмент `rustdoc`, распространяемый вместе с языком Rust, и помещает сгенерированную HTML-документацию в каталог target/doc.

Для удобства выполнение команды `cargo doc --open` построит HTML для документации к текущей упаковке (а также для документации ко всем ее зависимостям) и откроет результат в веб-браузере. Перейдите к функции `add_one`, и вы увидите, как в документационных комментариях воспроизводится текст (рис. 14.1).

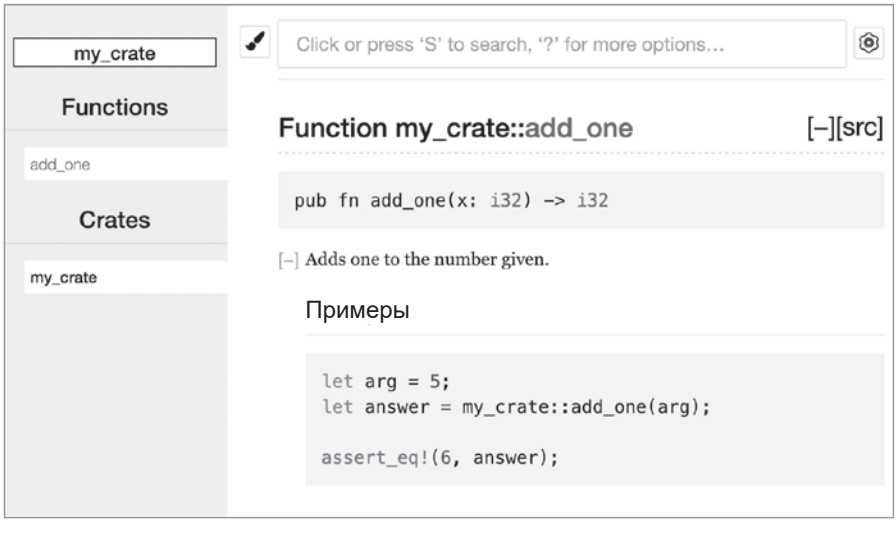

Рис. 14.1. HTML-документация для функции add_one

Часто используемые разделы

Мы использовали заголовок в формате упрощенной разметки `# Примеры` в листинге 14.1 для создания раздела в HTML с заголовком «Примеры». Вот некоторые другие разделы, которые авторы упаковок часто используют в документации:

Паники

Сценарии, в которых документируемая функция может поднимать панику. Пользователи функции, которые не хотят, чтобы их программы паниковали, должны следить за тем, чтобы они не вызывали функцию в таких ситуациях.

Ошибки

Если функция возвращает `Result`, то описание видов ошибок, которые могут возникнуть, и условий, которые могут привести к возвращению этих ошибок, может оказаться полезным для ее пользователей при написании своего кода по обработке разных видов ошибок разными способами.

Безопасность

Если функция помечена как небезопасная для вызова (мы обсудим небезопасность, `unsafe`, в главе 19), то должен быть раздел, объясняющий, почему функция небезопасна. В нем также должны быть описаны инварианты, которые функция ожидает от вызывающих ее пользователей.

Большинство документационных комментариев не нуждаются в этих разделах, но приведенный выше перечень — это хороший контрольный список, служащий напоминанием о тех аспектах кода, которые будут интересны людям, его вызывающим.

Документационные комментарии в качестве тестов

Добавление демонстрационных блоков кода в документационные комментарии помогает проиллюстрировать способы применения библиотеки, причем, делая так, вы получаете дополнительный бонус: выполнение `cargo test` запустит примеры кода вашей документации в качестве тестов! Нет ничего лучше документирования с примерами. Но нет ничего хуже примеров, которые не работают, потому что код изменился с момента написания документации. Если мы выполним тест с документацией для функции `add_one` из листинга 14.1, то увидим раздел в результатах теста, подобный этому:

```
   Doc-tests my_crate

running 1 test
test src/lib.rs - add_one (line 5) ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Теперь, если мы изменим либо функцию, либо пример так, что `assert_eq!` поднимет панику, и снова выполним команду `cargo test`, то увидим, что документационные тесты фиксируют, что пример и код не синхронизированы друг с другом.

Комментирование внутри элементов

Еще один стиль документационного комментария — `//!` — добавляет документацию в элемент, содержащий комментарии, а не в элементы, которые следуют за комментариями. Обычно мы используем эти документационные комментарии внутри корневого файла упаковки (по общему правилу в src/lib.rs) или внутри модуля для документирования упаковки или модуля в целом.

Например, если мы хотим добавить документацию, которая описывает назначение упаковки `my_crate`, содержащей функцию `add_one`, то в самом верху файла src/lib.rs можно добавить документационные комментарии, которые начинаются с `//!`, как показано в листинге 14.2.

Листинг 14.2. Документация для всей упаковки `my_crate` в целом

src/lib.rs

```
//! # Моя упаковка
//!
//! `my_crate` - это коллекция утилит, предназначенная для того, чтобы сделать
//! выполнение определенных вычислений удобнее.

/// Добавить единицу к заданному числу.
// --пропуск--
```

Обратите внимание, что после последней строки, начинающейся с `//!`, нет никакого кода. Так как мы начали комментарии с `//!`, а не с `///`, мы документируем элемент, содержащий этот комментарий, а не элемент, следующий за этим комментарием. В данном случае элементом, содержащим этот комментарий, является файл src/lib.rs, он же корень упаковки. Эти комментарии описывают всю упаковку в целом.

Когда мы будем выполнять команду `cargo doc --open`, эти комментарии будут показываться на первой странице документации для `my_crate` над списком публичных элементов в упаковке, как показано на рис. 14.2.

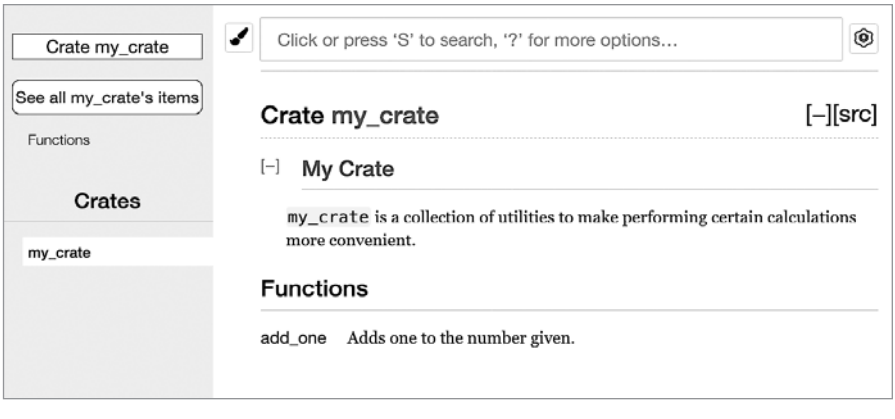

Рис. 14.2. Воспроизведение документации для my_crate, включая комментарий, описывающий всю упаковку в целом

Документационные комментарии внутри элементов особенно полезны для описания упаковок и модулей. Используйте их, чтобы объяснить общее назначение контейнера и помочь пользователям понять организацию упаковки.

Экспорт удобного публичного API с использованием pub

В главе 7 мы рассмотрели использование ключевых слов `mod`, `pub` и `use`, способы организации кода в модули с помощью `mod`, а также то, как делать элементы публичными с помощью `pub` и как вводить их в область видимости с помощью `use`. Однако структура, понятная вам, пока вы разрабатываете упаковку, возможно, будет не очень удобной для пользователей. Вы, возможно, захотите организовать свои структуры в иерархию, содержащую несколько уровней, но тогда тем, кто захочет использовать тип, размещенный вами глубоко в иерархии, может быть трудно обнаружить, что он существует. Возможно, их также будет раздражать необходимость вводить `use my_crate::некий_модуль::еще_один_модуль::ПолезныйТип;` вместо того, чтобы использовать `my_crate::ПолезныйТип;`.

Структура публичного API — важный фактор при публикации упаковки. Люди, использующие вашу упаковку, мало знакомы с ее структурой, в отличие от вас, и им может быть трудно найти нужные элементы, если в упаковке большая иерархия модулей.

Хорошая новость заключается в том, что если структурой неудобно пользоваться другим людям из другой библиотеки, то вам не нужно перестраивать свою внутреннюю организацию. Вместо этого вы можете реэкспортировать элементы, чтобы создать публичную структуру, отличную от вашей конфиденциальной структуры, с помощью `pub use`. Реэкспорт берет публичный элемент в одном месте и делает его публичным в другом месте, как если бы он и был там определен.

Допустим, мы создали библиотеку под названием `art` для моделирования художественных концепций. В этой библиотеке есть два модуля: `kinds`, содержащий два перечисления с именами `PrimaryColor` и `SecondaryColor`, и `utils`, содержащий функцию `mix`, как показано в листинге 14.3.

Листинг 14.3. Художественная библиотека с элементами, организованными в модули `kinds` и `utils`

src/lib.rs

```
//! # Художественная живопись
//!
//! Библиотека для моделирования художественных концепций.

pub mod kinds {
    /// Первичные цвета согласно цветовой модели RYB.
    pub enum PrimaryColor {
        Red,
        Yellow,
        Blue,
    }

    /// Вторичные цвета согласно цветовой модели RYB.
    pub enum SecondaryColor {
        Orange,
        Green,
```

```
        Purple,
    }
}

pub mod utils {
    use crate::kinds::*;

    /// Комбинирует два первичных цвета в одинаковых объемах для создания
    /// вторичного цвета.
    pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {
        // --пропуск--
    }
}
```

На рис. 14.3 приведена первая страница документации, сгенерированная командой `cargo doc`, по этой упаковке

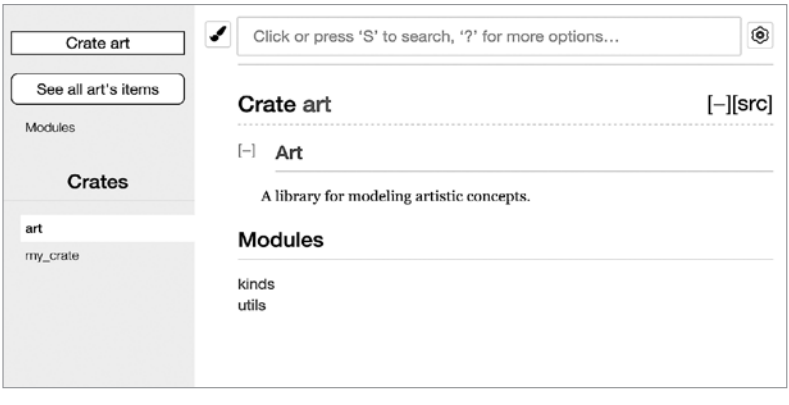

Рис. 14.3. Первая страница документации по упаковке art, на которой перечислены модули kinds и utils

Обратите внимание, что типы `PrimaryColor` и `SecondaryColor` не перечислены на первой странице, как и функция `mix`, и чтобы их увидеть, мы должны нажать `kinds` и `utils`.

Еще одной упаковке, зависящей от этой библиотеки, потребовались бы инструкции `use`, вводящие в область видимости элементы из `art`, уточняющие структуру модуля, которая в настоящее время определена. В листинге 14.4 показан пример упаковки, в которой используются элементы `PrimaryColor` и `mix` из упаковки `art`.

Листинг 14.4. Упаковка, использующая элементы упаковки art с экспортированной внутренней структурой

src/main.rs

```
use art::kinds::PrimaryColor;
use art::utils::mix;

fn main() {
```

```
    let red = PrimaryColor::Red;
    let yellow = PrimaryColor::Yellow;
    mix(red, yellow);
}
```

Автору кода в листинге 14.4, который использует упаковку `art`, приходится выяснять, что `PrimaryColor` находится в модуле `kinds`, а `mix` — в модуле `utils`. Модульная структура упаковки `art` более релевантна для разработчиков, работающих над упаковкой `art`, чем для разработчиков, ее использующих. Внутренняя структура, которая организует части упаковки в модули `kinds` и `utils`, не содержит никакой полезной информации для тех, кто пытается понять, как использовать упаковку `art`.

Напротив, модульная структура упаковки `art` служит причиной путаницы, потому что разработчикам приходится выяснять, где найти что-либо. К тому же структура неудобная, потому что разработчики должны указывать имена модулей в инструкциях `use`.

Чтобы удалить внутреннюю организацию из публичного API, мы можем модифицировать код упаковки `art` в листинге 14.3, добавив инструкции `pub use` для реэкспорта элементов на верхнем уровне, как показано в листинге 14.5.

Листинг 14.5. Добавление инструкций `pub use` для реэкспорта элементов

src/lib.rs

```
//! # Художественная живопись
//!
//! Библиотека для моделирования художественных концепций.

pub use self::kinds::PrimaryColor;
pub use self::kinds::SecondaryColor;
pub use self::utils::mix;

pub mod kinds {
    // --пропуск--
}

pub mod utils {
    // --пропуск--
}
```

Документация API, которую генерирует команда `cargo doc` для этой упаковки, теперь будет перечислять реэкспортированные элементы и отсылать к ним на первой странице, как показано на рис. 14.4, облегчая поиск типов `PrimaryColor` и `SecondaryColor` и их функции `mix`.

Пользователи упаковки `art` по-прежнему могут видеть и использовать внутреннюю структуру из листинга 14.3, как продемонстрировано в листинге 14.4, либо они могут воспользоваться более удобной структурой из листинга 14.5, как показано в листинге 14.6.

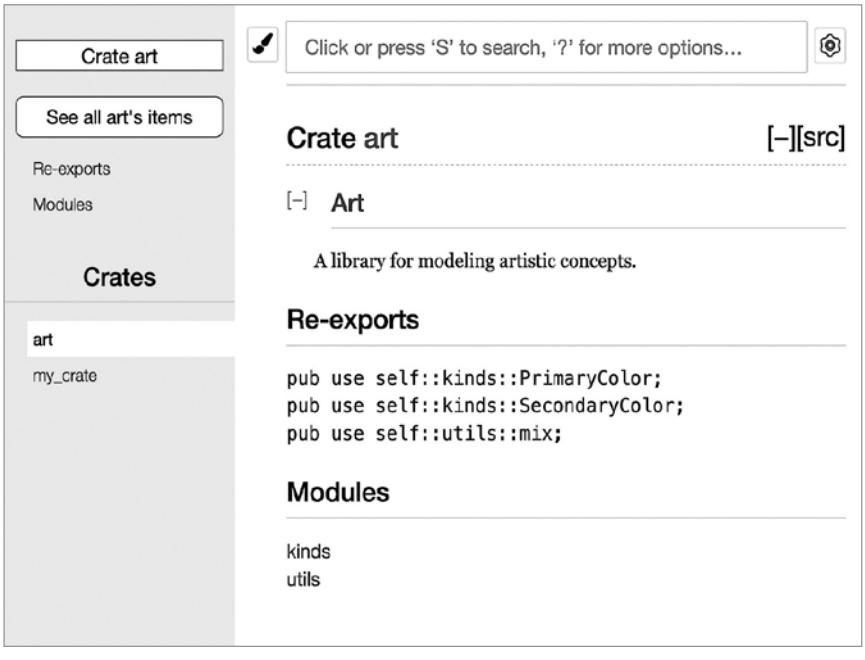

Рис. 14.4. Первая страница документации по упаковке art, на которой перечислены реэкспортированные элементы

Листинг 14.6. Программа, использующая реэкспортированные элементы из упаковки `art`

src/main.rs

```
use art::PrimaryColor;
use art::mix;

fn main() {
    // --пропуск--
}
```

В тех случаях, когда имеется много вложенных модулей, реэкспорт типов на верхнем уровне с помощью `pub` может внести существенные изменения в опыт людей, которые используют упаковку.

Создание полезной публичной структуры API — скорее искусство, чем наука, и вы можете предпринять несколько попыток, прежде чем найдете API, который лучше всего работает для ваших пользователей. Выбор `pub use` дает вам гибкость в том, как организовать внутреннюю структуру упаковки. `pub use` также отделяет эту внутреннюю структуру от того, что вы представляете пользователям. Посмотрите на код установленных вами упаковок, чтобы проверить, отличается ли их внутренняя структура от их публичного API.

Настройка учетной записи Crates.io

Прежде чем публиковать какие-либо упаковки, вам необходимо создать учетную запись на https://crates.io/ и получить токен API. Для этого посетите домашнюю страницу по адресу https://crates.io/ и войдите в систему через учетную запись GitHub. (Учетная запись GitHub в настоящее время является обязательным требованием, но указанный веб-сайт, возможно, будет поддерживать другие способы создания учетной записи в будущем.) После входа в систему зайдите в настройки своей учетной записи по адресу https://crates.io/me/ и получите ключ API. Затем выполните команду `cargo login` с помощью этого ключа, как показано ниже:

```
$ cargo login abcdefghijklmnopqrstuvwxyz012345
```

Эта команда проинформирует Cargo о вашем токене API и сохранит его локально в ~/.cargo/credentials. Обратите внимание, что это секретный токен: не делитесь им ни с кем. Если по какой-то причине вы им поделитесь с кем-либо, то его нужно будет отозвать и сгенерировать новый на https://crates.io/.

Добавление метаданных в новую упаковку

Теперь, когда у вас есть учетная запись, предположим, что у вас есть упаковка, которую вы хотите опубликовать. Перед публикацией нужно будет добавить в упаковку некоторые метаданные, добавив их в раздел `[package]` файла Cargo.toml упаковки.

Вашей упаковке понадобится уникальное имя. Пока вы работаете над упаковкой локально, вы можете называть ее так, как вам нравится. Однако имена упаковок на https://crates.io/ выделяются авторам в порядке «первым пришел, первым обслужен». После того как имя упаковки принято, больше никто не может опубликовать упаковку с этим именем. Прежде чем публиковать упаковку, выполните поиск имени, которое вы хотите использовать на веб-сайте. Если имя уже занято другой упаковкой, то вам нужно будет найти другое имя и отредактировать поле `name` в файле Cargo.toml в разделе `[package]`, чтобы использовать новое имя для публикации, как тут:

Cargo.toml

```
[package]
name = "guessing_game"
```

Даже если вы выбрали уникальное имя, все равно во время выполнения команды `cargo publish` для публикации упаковки в этом месте вы получите предупреждение, а затем ошибку:

```
$ cargo publish
    Updating registry `https://github.com/rust-lang/crates.io-index`
warning: manifest has no description, license, license-file, documentation,
homepage or repository.
--пропуск--
error: api errors: missing or empty metadata fields: description, license.
```

Причина состоит в том, что вы упускаете некоторые важные сведения: описание и лицензия обязательны. Они нужны для того, чтобы люди знали, что ваша упаковка делает и на каких условиях они могут ее использовать. Для исправления этой ошибки необходимо включить эту информацию в файл Cargo.toml.

Добавьте небольшое описание, состоящее из одного-двух предложений, — оно появится вместе с вашей упаковкой в результатах поиска. Для поля `license` необходимо указать значение лицензионного идентификатора. Открытый стандарт по обмену перечнями спецификаций по пакетам ПО Linux Foundation Data Exchange (SPDX) по адресу http://spdx.org/licenses/ перечисляет идентификаторы, которые можно использовать для этого значения. Например, для того чтобы указать, что вы лицензировали свою упаковку, используя лицензию MIT, добавьте идентификатор `MIT`:

Cargo.toml

```
[package]
name = "guessing_game"
license = "MIT"
```

Если вы хотите использовать лицензию, которая не представлена в SPDX, то вам нужно поместить текст этой лицензии в файл, включить файл в свой проект, а затем воспользоваться файлом `license`, указав имя этого файла вместо лицензионного ключа.

Руководство по выбору подходящей лицензии для проекта выходит за рамки темы данной книги. Многие люди из сообщества Rust лицензируют свои проекты так же, как и сам язык Rust, используя двойную лицензию MIT или Apache-2.0. Эта практика показывает, что вы можете указывать несколько лицензионных идентификаторов, разделенных ключевым словом OR, имея для проекта несколько лицензий.

С уникальным именем, версией, сведениями об авторе, которые добавила команда `cargo new`, когда вы создали упаковку, с описанием и добавленной лицензией, файл Cargo.toml готового к публикации проекта может выглядеть следующим образом:

Cargo.toml

```
[package]
name = "guessing_game"
version = "0.1.0"
authors = ["Ваше имя <you@example.com>"]
description = "Веселая игра, в которой вы угадываете, какое число выбрал
компьютер."
license = "MIT OR Apache-2.0"
edition = "2018"

[dependencies]
```

Документация Cargo по адресу https://doc.rust-lang.org/cargo/ описывает другие метаданные, которые вы можете указать, чтобы другие пользователи легче могли находить и использовать вашу упаковку.

Публикация в Crates.io

Теперь, когда вы создали учетную запись, сохранили токен API, выбрали для своей упаковки имя и указали обязательные метаданные, вы готовы к публикации! Публикация упаковки закачивает заданную версию на https://crates.io/ для использования другими.

Будьте осторожны при публикации упаковки, потому что публикация является перманентной. Версию нельзя перезаписать, а код нельзя удалить. Одна из главных целей https://crates.io/ — выступать в качестве перманентного архива кода, тем самым обеспечивая непрерывную работу сборок всех проектов, которые зависят от упаковок из https://crates.io/. Разрешение удалять версии сделало бы достижение этой цели невозможным. Однако число версий упаковок, которые вы можете публиковать, не ограничено.

Снова выполните команду `cargo publish`. Теперь она должна быть успешной:

```
$ cargo publish
 Updating registry `https://github.com/rust-lang/crates.io-index`
Packaging guessing_game v0.1.0 (file:///projects/guessing_game)
Verifying guessing_game v0.1.0 (file:///projects/guessing_game)
Compiling guessing_game v0.1.0
(file:///projects/guessing_game/target/package/guessing_game-0.1.0)
 Finished dev [unoptimized + debuginfo] target(s) in 0.19 secs
Uploading guessing_game v0.1.0 (file:///projects/guessing_game)
```

Примите поздравления! Теперь вы поделились своим кодом с сообществом Rust, и любой желающий может легко добавить вашу упаковку в качестве зависимости своего проекта.

Публикация новой версии существующей упаковки

Когда вы внесли изменения в свою упаковку и готовы выпустить новую версию, вы изменяете значение `version`, указанное в файле Cargo.toml, и публикуете ее повторно. Используйте правила семантического управления версиями по адресу http://semver.org/ для подбора подходящего номера следующей версии, основываясь на внесенных вами изменениях. Затем выполните команду `cargo publish`, в результате которой новая версия будет закачана на https://crates.io/.

Удаление версий из Crates.io с помощью команды cargo yank

Хотя нельзя удалить предыдущие версии упаковки, вы можете запретить любым будущим проектам добавлять их в качестве новой зависимости. Это полезно, когда упаковка в какой-то версии по той или иной причине нарушена. В таких ситуациях Cargo поддерживает «отзыв» (yank) версии упаковки.

Отзыв версии не дает новым проектам зависеть от этой версии, при этом позволяя всем существующим проектам, зависящим от нее, продолжать скачивать эту версию и зависеть от нее. По сути, отзыв означает, что все проекты с Cargo.lock не будут нарушены, а любые файлы Cargo.lock, генерируемые в будущем, не будут использовать отозванную версию.

Для того чтобы отозвать версию упаковки, выполните команду `cargo yank` и укажите номер версии, подлежащей отзыву:

```
$ cargo yank --vers 1.0.1
```

Добавив в эту команду `--undo`, вы также можете отменить отзыв и снова разрешить проектам зависеть от версии:

```
$ cargo yank --vers 1.0.1 --undo
```

Отзыв не удаляет код. Например, это средство не предназначено для удаления нечаянно загруженной секретной информации. Если это произойдет, то вы должны немедленно сбросить эту информацию.

Рабочие пространства Cargo

В главе 12 мы построили пакет, который включал двоичную и библиотечную упаковки. По мере развития проекта вы можете обнаружить, что библиотечная упаковка продолжает увеличиваться и вы хотите разделить пакет еще на несколько библиотечных упаковок. В этой ситуации Cargo предлагает средство под названием «рабочие пространства», которое помогает управлять несколькими взаимосвязанными пакетами, разрабатываемыми совместно.

Создание рабочего пространства

Рабочее пространство — это набор пакетов, которые совместно используют один и тот же файл Cargo.lock и выходной каталог. Давайте создадим проект, используя рабочее пространство. Мы применим простой код и благодаря этому сможем сосредоточиться на структуре рабочего пространства. Существует несколько способов его структурировать. Мы покажем один способ, который встречается наиболее часто. У нас будет рабочее пространство, содержащее двоичный файл и две библиотеки. Двоичный файл, который обеспечит главную функциональность, будет зависеть от двух библиотек. Первая библиотека будет предоставлять функцию `add_one`, а вторая — функцию `add_two`. Эти три упаковки станут частью одного рабочего пространства. Мы начнем с создания нового каталога для рабочего пространства:

```
$ mkdir add
$ cd add
```

Далее в каталоге add мы создаем файл Cargo.toml, который будет настраивать все рабочее пространство. В этом файле не будет раздела [package] или метаданных, которые мы видели в других файлах Cargo.toml. Вместо этого он будет начинаться с раздела [workspace], который позволит добавлять члены в рабочее пространство за счет указания пути к двоичной упаковке. В данном случае этим путем является adder:

Cargo.toml

```
[workspace]

members = [
    "adder",
]
```

Далее мы создадим двоичную упаковку adder, выполнив команду cargo new в каталоге add:

```
$ cargo new adder
    Created binary (application) `adder` project
```

На данном этапе мы можем построить рабочее пространство, выполнив команду cargo build. Файлы в каталоге add должны выглядеть следующим образом:

```
├── Cargo.lock
├── Cargo.toml
├── adder
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└── target
```

В рабочем пространстве есть один целевой каталог на верхнем уровне для размещения в нем скомпилированных артефактов. В упаковке adder нет своего целевого каталога. Даже если бы мы попробовали выполнить команду cargo build изнутри каталога adder, то скомпилированные артефакты все равно оказались бы в add/target, а не в add/adder/target. Cargo структурирует целевой каталог в рабочем пространстве таким образом, потому что упаковки в рабочем пространстве должны зависеть друг от друга. Если бы в каждой упаковке был собственный целевой каталог, то каждой упаковке пришлось бы перекомпилировать все другие упаковки в рабочем пространстве, чтобы иметь артефакты в своем целевом каталоге. При совместном использовании одного целевого каталога упаковки способны избежать ненужного перестроения.

Создание второй упаковки в рабочем пространстве

Далее давайте создадим еще одну упаковку-член в рабочем пространстве и назовем ее add-one. Измените верхнеуровневый Cargo.toml, задав путь add-one в списке members:

Cargo.toml

```
[workspace]

members = [
    "adder",
    "add-one",
]
```

Затем сгенерируйте новую библиотечную упаковку с именем `add-one`:

```
$ cargo new add-one --lib
    Created library `add-one` project
```

Теперь в каталоге `add` должны быть эти каталоги и файлы:

```
├── Cargo.lock
├── Cargo.toml
├── add-one
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
├── adder
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└── target
```

В файл `add-one/src/lib.rs` давайте добавим функцию `add_one`:

add-one/src/lib.rs

```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

Теперь, когда в рабочем пространстве есть библиотечная упаковка, двоичная упаковка `adder` может зависеть от библиотечной упаковки `add-one`. Сначала нужно добавить в `adder/Cargo.toml` зависимость от пути `add-one`.

adder/Cargo.toml

```
[dependencies]

add-one = { path = "../add-one" }
```

Использование Cargo не подразумевает, что упаковки в рабочем пространстве будут зависеть друг от друга, поэтому нам нужно четко определить отношения зависимости между упаковками.

Далее давайте применим функцию `add_one` из упаковки `add-one` в упаковке `adder`. Откройте файл `adder/src/main.rs` и добавьте вверху строку `use`, чтобы ввести новую библиотечную упаковку `add-one` в область видимости. Затем измените функцию `main`, чтобы вызвать функцию `add_one`, как в листинге 14.7.

Листинг 14.7. Использование библиотечной упаковки add-one из упаковки adder

adder/src/main.rs

```
use add_one;

fn main() {
    let num = 10;
    println!("Здравствуй, Мир! {} плюс один равно {}!", num,
             add_one::add_one(num));
}
```

Давайте построим рабочее пространство, выполнив команду `cargo build` в верхнеуровневом каталоге add!

```
$ cargo build
    Compiling add-one v0.1.0 (file:///projects/add/add-one)
    Compiling adder v0.1.0 (file:///projects/add/adder)
     Finished dev [unoptimized + debuginfo] target(s) in 0.68 secs
```

Для того чтобы выполнить двоичную упаковку из каталога add, нужно указать имя пакета из рабочего пространства, который мы хотим использовать, включив аргумент `-p` и имя пакета в команду `cargo run`:

```
$ cargo run -p adder
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/adder`
Здравствуй, Мир! 10 плюс один равно 11!
```

Эта команда выполняет код в adder/src/main.rs, который зависит от упаковки `add-one`.

Зависимость от внешней упаковки в рабочем пространстве

Обратите внимание, вместо того чтобы иметь файл Cargo.lock в каталоге каждой упаковки, рабочее пространство имеет только один файл Cargo.lock на верхнем уровне рабочего пространства. Этим обеспечивается, чтобы все упаковки использовали одну и ту же версию всех зависимостей. Если мы добавим упаковку `rand` в файлы adder/Cargo.toml и add-one/Cargo.toml, то Cargo сведет их к одной версии `rand` и запишет в один файл Cargo.lock. Если все упаковки в рабочем пространстве используют одинаковые зависимости, то это означает, что упаковки в рабочем пространстве всегда будут совместимы друг с другом. Давайте добавим упаковку `rand` в раздел `[dependencies]` файла add-one/Cargo.toml, чтобы иметь возможность использовать упаковку `rand` в упаковке `add-one`:

add-one/Cargo.toml

```
[dependencies]

rand = "0.3.14"
```

Теперь мы можем добавить `use rand;` в файл add-one/src/lib.rs, и построение всего рабочего пространства путем выполнения команды `cargo build` в каталоге add введет и скомпилирует упаковку `rand`:

```
$ cargo build
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading rand v0.3.14
   --пропуск--
   Compiling rand v0.3.14
   Compiling add-one v0.1.0 (file:///projects/add/add-one)
   Compiling adder v0.1.0 (file:///projects/add/adder)
    Finished dev [unoptimized + debuginfo] target(s) in 10.18 secs
```

Верхнеуровневый файл Cargo.lock теперь содержит информацию о зависимости `add-one` от `rand`. Однако, даже если упаковка `rand` используется где-то в рабочем пространстве, мы не можем применять ее в других упаковках рабочего пространства, если не добавим `rand` и в их файлы Cargo.toml тоже. Например, если мы разместим `use rand;` в файле adder/src/main.rs для упаковки `adder`, то получим ошибку[1]:

```
$ cargo build
    Compiling adder v0.1.0 (file:///projects/add/adder)
error: use of unstable library feature 'rand': use `rand` from crates.io (see
issue #27703)
 --> adder/src/main.rs:1:1
  |
1 | use rand;
```

Для ее устранения отредактируйте файл Cargo.toml для упаковки `adder` и укажите, что `rand` является зависимостью и для этой упаковки тоже. Построение упаковки `adder` добавит `rand` в список зависимостей для `adder` в файле Cargo.lock, но никакие дополнительные копии `rand` не будут скачиваться. Cargo обеспечил, чтобы каждая упаковка в рабочем пространстве, использующая упаковку `rand`, применяла одну и ту же версию. Использование одной и той же версии `rand` в рабочем пространстве экономит место, поскольку у нас не будет нескольких копий, и обеспечивает, чтобы упаковки в рабочем пространстве были совместимы друг с другом.

Добавление теста в рабочее пространство

В качестве еще одного улучшения давайте добавим тест функции `add_one::add_one` внутрь упаковки `add_one`:

add-one/src/lib.rs

```
pub fn add_one(x: i32) -> i32 {
    x + 1
}

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

    #[test]
```

[1] ошибка: использование нестабильного библиотечного средства 'rand': используйте `rand' из crates.io (см. вопрос #27703)

```
    fn it_works() {
        assert_eq!(3, add_one(2));
    }
}
```

Теперь выполните команду `cargo test` в верхнеуровневом каталоге add:

```
$ cargo test
    Compiling add-one v0.1.0 (file:///projects/add/add-one)
    Compiling adder v0.1.0 (file:///projects/add/adder)
     Finished dev [unoptimized + debuginfo] target(s) in 0.27 secs
      Running target/debug/deps/add_one-f0253159197f7841

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

    Running target/debug/deps/adder-f88af9d2cc175a5e

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

  Doc-tests add-one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Первая часть данных показывает, что тест `it_works` в упаковке `add-one` пройден. Следующая часть говорит о том, что в упаковке `adder` было найдено ноль тестов, и затем последняя часть показывает, что в упаковке `add-one` было найдено ноль документационных тестов. Команда `cargo test` в рабочем пространстве, структурированная подобным образом, будет выполнять тесты для всех упаковок в рабочем пространстве.

Мы также можем выполнить тесты для одной конкретной упаковки в рабочем пространстве из верхнеуровневого каталога, применив флаг `-p` и указав имя упаковки, которую мы хотим протестировать:

```
$ cargo test -p add-one
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/deps/add_one-b3235fea9a156f74

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

  Doc-tests add-one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Эти данные показывают, что команда `cargo test` выполнила тесты только для упаковки `add-one` и не выполнила тесты для упаковки `adder`.

Если вы публикуете упаковки из рабочего пространства в https://crates.io/, то каждая упаковка в рабочем пространстве должна быть опубликована отдельно. Команда `cargo publish` не имеет флага `--all` или флага `-p`, поэтому для публикации упаковок вы должны перейти в каталог каждой упаковки и выполнить команду `cargo publish` для каждой упаковки в рабочем пространстве.

В качестве дополнительного упражнения добавьте упаковку `add-two` в это рабочее пространство таким же образом, как и упаковку `add-one`! По мере развития проекта подумайте об использовании рабочего пространства: легче понимать маленькие, отдельные компоненты, чем один большой код. Кроме того, хранение упаковок в рабочем пространстве облегчает координацию между ними, если они часто изменяются одновременно.

Установка двоичных файлов из Crates.io с помощью команды cargo install

Команда `cargo install` позволяет устанавливать и использовать бинарные упаковки локально. Она не предназначена для замены системных пакетов, а задумана как удобное средство для разработчиков на языке Rust по установке инструментов, которыми поделились другие на https://crates.io/. Обратите внимание, вы можете устанавливать только те пакеты, которые имеют двоичные цели. Двоичная цель (*binary target*) — это выполняемая программа, которая создается, если упаковка имеет файл src/main.rs или другой файл, указанный как двоичный, в отличие от библиотечной цели, которая не может выполняться в отдельности, но подходит для включения в другие программы. Обычно упаковки содержат информацию о том, является ли упаковка библиотекой, имеет ли она двоичную цель или и то и другое, в файле README.

Все двоичные файлы, устанавливаемые с помощью команды `cargo install`, хранятся в папке bin установочного корневого каталога. Если вы инсталлировали Rust с помощью `rustup` и не имеете никаких собственных конфигураций, то этот каталог будет $HOME/.cargo/bin. Каталог должен находиться в `$PATH` — это даст возможность выполнять программы, инсталлируемые с помощью команды `cargo install`.

Например, в главе 12 мы упоминали, что существует реализация на языке Rust инструмента `grep`, именуемая `ripgrep`, для поиска файлов. Если мы хотим установить `ripgrep`, мы можем выполнить следующие действия:

```
$ cargo install ripgrep
Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading ripgrep v0.3.2
```

```
--пропуск--
  Compiling ripgrep v0.3.2
   Finished release [optimized + debuginfo] target(s) in 97.91 secs
Installing ~/.cargo/bin/rg
```

Последняя строка данных показывает расположение и имя инсталлированного двоичного файла, которым в случае `ripgrep` является `rg`. При условии, что каталог установки находится в `$PATH`, как упоминалось ранее, вы можете выполнять `rg --help` и начать использовать более быстрый, более «растианский» инструмент для поиска файлов!

Расширение Cargo с помощью индивидуальных команд

Cargo спроектирован таким образом, что вы можете расширять его с помощью новых подкоманд, без модификации Cargo. Если двоичный файл в `$PATH` называется `cargo-нечто`, то вы можете выполнить его, как если бы это была подкоманда Cargo, запустив команду `cargo-something`. Индивидуально настроенные команды, подобные этой, также перечисляются при выполнении команды `cargo --list`. Возможность использовать команды `cargo install` для установки расширений, а затем выполнять их так же, как и встроенные инструменты Cargo, — это суперудобное преимущество дизайна пакетного менеджера Cargo!

Итоги

Совместное использование кода с помощью пакетного менеджера Cargo и https://crates.io/ делает экосистему Rust полезной для большого числа разнообразных задач. Стандартная библиотека небольшая и стабильная, но упаковки легко распространять, использовать и улучшать за другой временной промежуток, чем при использовании языка. Делитесь кодом, который полезен вам, на https://crates.io/ — вполне вероятно, что он пригодится и кому-то еще!

15 Умные указатели

Указатель — это общее понятие для переменной, которая содержит адрес в памяти. Этот адрес ссылается, или «указывает», на некие другие данные. Самым часто встречающимся видом указателя в Rust является ссылка, о которой вы узнали в главе 4. Ссылки обозначаются символом `&` и заимствуют значение, на которое указывают. У них нет никаких особых возможностей, кроме обращения к данным. Кроме того, они не затратны, и мы чаще всего используем именно их.

С другой стороны, умные указатели (*Smart pointers*) — это структуры данных, которые не только действуют как указатель, но и обладают дополнительными метаданными и способностями. Понятие умных указателей не является уникальным для Rust: умные указатели возникли в C++ и существуют также в других языках. В Rust разные умные указатели, определенные в стандартной библиотеке, обеспечивают функциональность, выходящую за рамки той, которая предоставляется ссылками. В этой главе мы познакомимся с умным указателем с подсчетом числа ссылок. Этот указатель позволяет иметь несколько владельцев данных, отслеживать их число и, когда владельцев не остается, очищать данные.

В Rust, который использует понятие владения и заимствования, дополнительное различие между ссылками и умными указателями заключается в том, что ссылки — это указатели, которые только заимствуют данные. В отличие от них, умные указатели зачастую владеют данными, на которые они указывают.

Мы уже сталкивались с несколькими умными указателями в этой книге, такими как `String` и `Vec<T>` в главе 8, хотя в то время мы не называли их умными указателями. Оба типа считаются умными указателями, потому что они владеют некоторой памятью и позволяют ею манипулировать. Они также имеют метаданные (например, емкость) и дополнительные способности или гарантии (например, в случае с типом `String` они обеспечивает, чтобы его данные всегда были действительными и кодировались в UTF-8).

Умные указатели обычно реализуются с помощью структур. Особенность, отличающая умный указатель от обычной структуры, заключается в том, что умные

указатели реализуют типажи `Deref` и `Drop`. Типаж `Deref` позволяет экземпляру структуры умного указателя вести себя как ссылка, поэтому вы можете писать код, который работает либо со ссылками, либо с умными указателями. Типаж `Drop` позволяет настраивать код, который выполняется, когда экземпляр умного указателя выходит из области видимости. В этой главе мы обсудим оба типажа и объясним, почему они важны для умных указателей.

С учетом того, что паттерн умного указателя — это общий паттерн проектирования, часто используемый в Rust, эта глава не охватит каждый существующий умный указатель. Многие библиотеки имеют свои умные указатели, и вы даже можете написать собственный указатель. Мы рассмотрим умные указатели, наиболее часто встречающиеся в стандартной библиотеке:

- Умный указатель `Box<T>` для размещения значений в куче.
- Умный указатель `Rc<T>` — тип подсчета числа ссылок, который обеспечивает множественное владение.
- Умные указатели `Ref<T>` и `RefMut<T>` с доступом через `RefCell<T>` — тип, который обеспечивает соблюдение правил заимствования во время выполнения, а не во время компиляции.

Кроме того, мы рассмотрим паттерн внутренней изменяемости, в котором неизменяемый тип предоставляет API для изменения внутреннего значения. Мы также обсудим тему циклов в переходах по ссылкам: как они приводят к утечке памяти и как их предотвращать.

Давайте приступим!

Использование Box<T> для указания на данные в куче

Наиболее простым умным указателем является `Box`, тип которого записывается как `Box<T>`. Умные указатели `Box` позволяют хранить данные не в стеке, а в куче. В стеке же остается указатель на данные кучи. Обратитесь к главе 4, чтобы вспомнить разницу между стеком и кучей.

Помимо хранения своих данных в куче, а не в стеке, умные указатели `Box` не производят трудоемких операций, сказывающихся на производительности. Но у них, впрочем, не так много дополнительных способностей. Вы будете использовать их чаще всего в следующих ситуациях:

- У вас есть тип, размер которого нельзя узнать во время компиляции, и вы хотите использовать значение этого типа в контексте, требующем точного размера.
- У вас крупный объем данных, и вы хотите передать владение так, чтобы при этом данные не были скопированы.

- Вы хотите владеть значением, и нужно, чтобы оно имело не конкретный тип, а тип, который реализует некоторый типаж.

Мы продемонстрируем первую ситуацию в разделе «Применение рекурсивных типов с помощью умных указателей Box». Во втором случае передача владения крупного объема данных может занимать много времени, поскольку данные копируются из всего стека. В целях повышения производительности в этой ситуации мы можем хранить крупный объем данных кучи в умном указателе `Box`. Тогда только небольшой объем данных указателя копируется из стека, в то время как данные, на которые он ссылается, остаются в одном месте в куче. Третий случай называется типажным объектом, в главе 17 целый раздел («Использование типажных объектов, допускающих значения разных типов») посвящен указанной теме. Поэтому вы снова примените знания из этого раздела в главе 17!

Использование Box<T> для хранения данных в куче

Прежде чем обсудить этот вариант использования `Box<T>`, мы рассмотрим его синтаксис и способы взаимодействия со значениями, хранящимися в указателе `Box<T>`.

Листинг 15.1 показывает использование умного указателя `Box` для хранения значения типа `i32` в куче.

Листинг 15.1. Хранение значения типа i32 в куче с помощью умного указателя Box

src/main.rs

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

Мы определяем переменную `b` как имеющую значение `Box`, которая указывает на значение 5, размещаемое в куче. Эта программа выведет `b = 5`. В данном случае мы можем обратиться к данным в умном указателе `Box` аналогично тому, как мы бы сделали, если бы эти данные были в стеке. Так же, как любое обладаемое значение, когда умный указатель `Box` выходит из области видимости, как это делает переменная `b` в конце функции `main`, она будет высвобождена. Высвобождение происходит для умного указателя `Box` (хранящегося в стеке) и для данных, на которые он указывает (хранящихся в куче).

Размещение одного значения в куче не имеет особых преимуществ, поэтому вы не будете часто использовать умные указатели `Box` по отдельности таким образом. В большинстве ситуаций уместнее иметь значения наподобие одного `i32` в стеке, где они хранятся по умолчанию. Давайте посмотрим, в каком случае умные указатели `Box` позволяют определять типы, которые нельзя было бы определять, если бы не было умных указателей `Box`.

Применение рекурсивных типов с помощью умных указателей Box

Компилятору нужно знать, сколько места занимает тип. Один из типов, размер которого нельзя узнать во время компиляции, — рекурсивный тип, где частью значения может быть еще одно значение того же типа. Поскольку эта вложенность значений теоретически может продолжаться бесконечно, язык Rust не знает, сколько места требуется для значения рекурсивного типа. Однако размер умных указателей `Box` известен, поэтому, вставляя `Box` в определение рекурсивного типа, вы можете иметь рекурсивные типы.

Давайте изучим cons-список, рекурсивный тип данных, часто встречающийся в языках функционального программирования. За исключением рекурсии тип cons-списка, который мы определим, довольно прост, поэтому идеи в примере, который мы рассмотрим, пригодятся при работе с более сложными примерами, где есть рекурсивные типы.

Более подробная информация о cons-списке

Cons-список — это структура данных, которая происходит из языка Lisp и его диалектов. В Lisp функция cons (расшифровывается как «сконструировать функцию») создает новую пару из двух аргументов, обычно это значение и еще одна пара. Эти пары, содержащие пары, образуют список.

Понятие функции cons вошло в более общий жаргон функционального программирования: в английском языке «to cons x on y» («прикрепить x к y») неофициально означает «сконструировать новый экземпляр контейнера, поместив элемент `x` в начало этого нового контейнера, за которым следует контейнер `y`».

Каждый элемент в cons-списке содержит два элемента: значение текущего элемента и значение следующего элемента. Последний элемент в списке содержит только значение `Nil` без следующего элемента. Cons-список создается рекурсивным вызовом функции cons. Каноническим именем для обозначения базового случая рекурсии является `Nil`. Обратите внимание, это не то же самое, что понятие “null” в главе 6, которое является недействительным или отсутствующим значением.

Хотя языки функционального программирования часто используют cons-списки, в Rust cons-список применяется не так широко. В большинстве случаев, когда у вас есть список элементов в языке Rust, лучше всего выбрать `Vec<T>`. Другие, более сложные рекурсивные типы данных бывают полезными в различных ситуациях, но, начав с cons-списка, можно, ни на что не отвлекаясь, узнать, как умные указатели `Box` помогают определять рекурсивный тип данных.

Листинг 15.2 содержит определение перечисления для cons-списка. Обратите внимание, что этот код пока не компилируется, поскольку размер типа `List` неизвестен, что мы и продемонстрируем.

Листинг 15.2. Первая попытка определить перечисление enum для представления структуры данных cons-списка значений типа i32

src/main.rs

```
enum List {
    Cons(i32, List),
    Nil,
}
```

ПРИМЕЧАНИЕ

В этом примере мы реализуем cons-список, который содержит только значения типа `i32`. Как в главе 10, мы могли бы реализовать его, используя обобщения, чтобы определить тип cons-списка, который мог бы хранить значения любого типа.

Использование типа `List` для хранения списка `1, 2, 3` будет выглядеть как код из листинга 15.3.

Листинг 15.3. Использование перечисления `List` для хранения списка `1, 2, 3`

src/main.rs

```
use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1, Cons(2, Cons(3, Nil)));
}
```

Первое значение `Cons` содержит 1 и еще одно значение типа `List`. Данное значение типа `List` — это еще одно значение `Cons`, которое содержит 2 и еще одно значение `List`. Данное значение типа `List` — это очередное значение `Cons`, которое содержит 3 и значение типа `List`, в конечном итоге равное `Nil`, то есть нерекурсивному варианту, который сигнализирует о конце списка.

Если мы попытаемся скомпилировать код листинга 15.3, то получим ошибку, показанную в листинге 15.4[1].

Листинг 15.4. Ошибка при попытке определить рекурсивное перечисление

```
error[E0072]: recursive type `List` has infinite size
 --> src/main.rs:1:1
  |
1 | enum List {
  | ^^^^^^^^^ recursive type has infinite size
2 |     Cons(i32, List),
  |               ----- recursive without indirection
  |
  = help: insert indirection (e.g., a `Box`, `Rc`, or `&`) at some point to
  make `List` representable
```

[1] `ошибка[E0072]: рекурсивный тип `List` имеет бесконечный размер`

Данная ошибка показывает, что этот тип «имеет бесконечный размер». Причина в том, что мы определили список с рекурсивным вариантом: он непосредственно содержит еще одно значение самого себя. В результате Rust не может выяснить, сколько места нужно для хранения значения списка. Давайте разберемся, почему мы получаем эту ошибку. Сначала посмотрим, как Rust решает, сколько места ему нужно для хранения значения нерекурсивного типа.

Вычисление размера нерекурсивного типа

Вспомните перечисление `Message`, которое мы определили в листинге 6.2, когда обсуждали тему определения перечисления в главе 6:

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

Чтобы определить, какой объем пространства следует выделить для значения `Message`, язык Rust перебирает все варианты и проверяет, какой вариант требует больше всего пространства. Язык Rust видит, что `Message::Quit` не требует никакого пространства, `Message::Move` требует пространство, достаточное для хранения двух значений типа `i32`, и так далее. Поскольку будет использоваться только один вариант, наибольшим пространством для хранения значения `Message` будет пространство, необходимое для хранения его самого большого варианта.

Сравните, что происходит, когда язык Rust пытается выяснить объем пространства для рекурсивного типа, такого как перечисление `List` в листинге 15.2. Компилятор начинает с просмотра варианта `Cons`, который содержит значения типов `i32` и `List`. Следовательно, `Cons` требует объем пространства, равный размеру типов `i32` и `List`. Чтобы выяснить объем памяти, который требуется типу `List`, компилятор просматривает варианты, начиная с варианта `Cons`. Вариант `Cons` содержит значения типов `i32` и `List`, и этот процесс продолжается бесконечно, как показано на рис. 15.1.

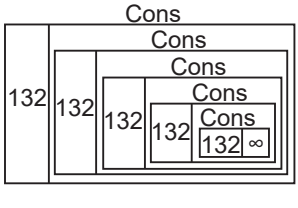

Рис. 15.1. Бесконечный список, состоящий из бесконечных вариантов `Cons`

Использование Box<T> для получения рекурсивного типа с известным размером

Компилятор не может определить объем пространства, который следует выделить для типов, определенных рекурсивно, поэтому выдает ошибку в листинге 15.4. Но ошибка все-таки включает в себя полезную рекомендацию[1]:

```
= help: insert indirection (e.g., a `Box`, `Rc`, or `&`) at some point to
make `List` representable
```

В этой рекомендации косвенное обращение означает, что вместо прямого хранения значения мы изменим структуру данных так, чтобы хранить значение косвенно, сохраняя указатель на значение.

Поскольку `Box<T>` является указателем, Rust всегда знает, сколько места требуется для `Box<T>`: размер указателя не меняется в зависимости от объема данных, на которые он указывает. Это означает, что мы можем поместить указатель `Box<T>` внутрь варианта `Cons` вместо еще одного значения типа `List`. `Box<T>` будет указывать на следующее значение типа `List`, которое будет находиться в куче, а не внутри варианта `Cons`. Концептуально у нас по-прежнему список, созданный с помощью списков, «содержащих» другие списки, но теперь эта реализация больше похожа на размещение элементов рядом друг с другом, а не внутри друг друга.

Мы можем изменить определение перечисления `List` из листинга 15.2 и использование списка в листинге 15.3 на код в листинге 15.5, который будет компилироваться.

Листинг 15.5. Определение списка, который использует умный указатель Box<T>, чтобы иметь известный размер

src/main.rs

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1,
        Box::new(Cons(2,
            Box::new(Cons(3,
                Box::new(Nil))))));
}
```

Вариант `Cons` потребует размер типа `i32` плюс пространство для хранения данных умного указателя `Box`. Вариант `Nil` не сохраняет никаких значений, поэтому ему

[1] `= справка: вставьте косвенное обращение (например, `Box`, `Rc` или `&`) в какой-то точке, сделав `List' представимым`

требуется меньше места, чем варианту `Cons`. Теперь мы знаем, что любое значение списка будет занимать размер типа `i32` плюс размер данных указателя `Box`. Используя умный указатель `Box`, мы разорвали бесконечную рекурсивную цепочку, благодаря чему компилятор может вычислить размер, необходимый для хранения значения типа `List`. На рис. 15.2 показано, как теперь выглядит вариант `Cons`.

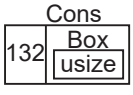

Рис. 15.2. Список, размер которого не бесконечен, так как `Cons` содержит умный указатель `Box`

Умные указатели `Box` обеспечивают только косвенную адресацию и выделение пространства в куче. У них нет специальных свойств, подобных тем, которые мы увидим у других типов умных указателей. На их производительности не сказываются никакие специальные свойства, поэтому они бывают полезными в таких случаях, как cons-список, где нам нужно только косвенное обращение. Мы также рассмотрим другие варианты использования умных указателей `Box` в главе 17.

Тип `Box<T>` является умным указателем, поскольку он реализует типаж `Deref`, который позволяет трактовать значения `Box<T>` как ссылки. Когда значение указателя `Box<T>` выходит из области видимости, данные кучи, на которые указывает `Box`, также очищаются по причине реализации типажа `Drop`. Давайте подробнее изучим эти типажи. Они будут важнее для функциональности, предусмотренной другими типами умных указателей, которые мы обсудим далее в этой главе.

Трактовка умных указателей как обыкновенных ссылок с помощью типажа Deref

Реализация типажа `Deref` позволяет настроить поведение *оператора разыменования* `*` индивидуально (в отличие от оператора умножения или оператора glob). Реализуя типаж `Deref` таким образом, что умный указатель можно трактовать как обыкновенную ссылку, вы можете писать код, который работает со ссылками, и использовать этот же код с умными указателями.

Давайте сначала посмотрим, как оператор разыменования работает с регулярными ссылками. Затем мы попытаемся определить настраиваемый тип, который ведет себя как `Box<T>`, и посмотрим, почему оператор разыменования не работает как ссылка в только что определенном типе. Мы узнаем, как благодаря реализации типажа Deref умные указатели работают по подобию ссылок. Затем мы рассмотрим средство языка Rust — принудительное приведение типа посредством `deref` и то, как оно позволяет работать со ссылками или умными указателями.

ПРИМЕЧАНИЕ

Между типом `MyBox<T>`, который мы собираемся построить, и реальным типом `Box<T>` есть одно большое различие: наша версия не будет хранить свои данные в куче. В этом примере мы фокусируемся на типаже `Deref`, поэтому место, где на самом деле хранятся данные, менее важно, чем поведение, подобное указателю.

Следование по указателю к значению с помощью оператора разыменования

Регулярная ссылка — это тип указателя, и один из способов обозначить указатель — представить стрелку к значению, хранящемуся где-то в другом месте. В листинге 15.6 мы создаем ссылку на значение типа `i32` и затем используем оператор разыменования, чтобы пройти к данным по ссылке.

Листинг 15.6. Использование оператора разыменования, чтобы пройти к значению типа `i32` по ссылке

src/main.rs

```
fn main() {
 ❶ let x = 5;
 ❷ let y = &x;

 ❸ assert_eq!(5, x);
 ❹ assert_eq!(5, *y);
}
```

Переменная `x` содержит значение типа `i32`, равное `5` ❶. Мы устанавливаем значение переменной `y` равным ссылке на переменную `x` ❷. С помощью макрокоманды `assert_eq!` мы можем подтвердить, что переменная `x` равна `5` ❸. Однако, если мы хотим выполнить проверочное утверждение значения в переменной `y`, то нужно использовать синтаксис `*y`, чтобы пройти по ссылке к значению, на которое она указывает ❹. Как только мы применили оператор разыменования к переменной `y`, мы получаем доступ к целочисленному значению, на которое указывает переменная `y` и которое можно сравнить с `5`.

Если бы вместо этого мы попытались написать `assert_eq!(5, y);`, то получили бы вот такую ошибку компиляции[1]:

```
error[E0277]: can't compare `{integer}` with `&{integer}`
 --> src/main.rs:6:5
  |
6 |     assert_eq!(5, y);
  |     ^^^^^^^^^^^^^^^^ no implementation for `{integer} == &{integer}`
  |
  = help: the trait `std::cmp::PartialEq<&{integer}>` is not implemented for
  `{integer}`
```

[1] ошибка[E0277]: не получается сравнить `{целое число} ‘ с ‘ &{целое число}`

Сравнение числа и ссылки на это число не допускается, поскольку они относятся к разным типам. Мы должны использовать оператор разыменования, пройдя по ссылке на значение, на которое она указывает.

Использование Box<T> в качестве ссылки

Мы можем переписать код листинга 15.6 так, чтобы использовать умный указатель `Box<T>` вместо ссылки. Оператор разыменования будет работать, как показано в листинге 15.7.

Листинг 15.7. Использование оператора разыменования в умном указателе `Box<i32>`

src/main.rs

```
fn main() {
    let x = 5;
 ❶ let y = Box::new(x);
    assert_eq!(5, x);
 ❷ assert_eq!(5, *y);
}
```

Единственное различие между листингом 15.7 и листингом 15.6 состоит в том, что здесь мы устанавливаем переменную `y` как экземпляр типа `Box`, указывающего на значение в переменной `x`, а не как ссылку, указывающую на значение переменной `x` ❶. В последнем проверочном утверждении ❷ мы можем использовать оператор разыменования, чтобы пройти по указателю `Box` таким же образом, как мы делали, когда переменная `y` была ссылкой. Далее мы узнаем характерные особенности типа `Box<T>`, благодаря которым можно использовать оператор разыменования путем определения собственного типа `Box`.

Определение собственного умного указателя

Давайте построим умный указатель, подобный типу `Box<T>` из стандартной библиотеки, чтобы увидеть на практике, чем по умолчанию отличаются умные указатели от ссылок. Затем мы посмотрим, как добавлять возможность использовать оператор разыменования.

Тип `Box<T>` в конечном счете определяется как кортежная структура с одним элементом, поэтому листинг 15.8 определяет тип `MyBox<T>` таким же образом. Мы также определим функцию `new`, которая будет соответствовать функции `new`, определенной для `Box<T>`.

Листинг 15.8. Определение типа `MyBox<T>`

src/main.rs

```
❶ struct MyBox<T>(T);

impl<T> MyBox<T> {
```

```
    ❷ fn new(x: T) -> MyBox<T> {
        ❸ MyBox(x)
      }
}
```

Мы определяем структуру с именем `MyBox` и объявляем обобщенный параметр `T` ❶, поскольку хотим, чтобы наш тип содержал значения любого типа. Тип `MyBox` представляет собой кортежную структуру с одним элементом типа `T`. Функция `MyBox::new` берет один параметр типа `T` ❷ и возвращает экземпляр структуры `MyBox`, содержащий значение, переданное внутрь ❸.

Давайте попробуем добавить функцию `main` из листинга 15.7 в листинг 15.8 и изменить ее на тип `MyBox<T>`, который мы определили вместо `Box<T>`. Код в листинге 15.9 не компилируется, потому что язык Rust не знает, как пройти по указателю `MyBox` к значению с помощью оператора разыменования.

Листинг 15.9. Попытка использовать `MyBox<T>` таким же образом, как ссылки и умный указатель `Box<T>`

src/main.rs

```
fn main() {
    let x = 5;
    let y = MyBox::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

Ошибка в результате компиляции[1]:

```
error[E0614]: type `MyBox<{integer}>` cannot be dereferenced
 --> src/main.rs:14:19
   |
14 |     assert_eq!(5, *y);
   |                   ^^
```

У нас не получается пройти по указателю `MyBox<T>` к значению с помощью оператора разыменования, потому что мы не реализовали эту способность в нашем типе. Чтобы иметь возможность пройти по указателю к значению с помощью оператора `*`, мы реализуем типаж `Deref`.

Трактовка типа как ссылки путем реализации типажа Deref

Как обсуждалось в главе 10, для того чтобы реализовать типаж, нужно предоставить реализации для обязательных методов типажа. Типаж `Deref`, предусмо-

[1] `ошибка[E0614]: не получается пройти к существующему значению по указателю типа `MyBox<{integer}>``

тренный стандартной библиотекой, требует реализации одного метода с именем `deref`, который заимствует `self` и возвращает ссылку на внутренние данные. Листинг 15.10 содержит реализацию типажа `Deref`, добавляемую в определение типа `MyBox`.

Листинг 15.10. Реализация типажа `Deref` в типе `MyBox<T>`

src/main.rs

```
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
  ❶ type Target = T;

    fn deref(&self) -> &T {
     ❷ &self.0
    }
}
```

Синтаксис `type Target = T;` ❶ определяет связанный тип, который используется типажом `Deref`. Связанные типы — это несколько иной способ объявления обобщенного параметра, но сейчас вам не нужно думать о них, мы подробнее рассмотрим эти типы в главе 19.

Мы заполняем тело метода `deref`, используя `&self.0`, в результате чего `deref` возвращает ссылку на значение, к которому мы хотим обратиться с помощью оператора `*` ❷. Функция `main` в листинге 15.9, которая вызывает `*` для значения типа `MyBox<T>`, теперь компилируется, и проверочные утверждения проходят успешно!

Без типажа `Deref` компилятор может применять оператор разыменования `*` только к ссылкам `&`. Метод `deref` наделяет компилятор способностью брать значение любого типа, который реализует типаж `Deref`, и вызывать метод `deref`, чтобы получать ссылку `&`, по которой можно пройти к существующему значению.

Когда мы ввели `*y` в листинге 15.9, компилятор фактически выполнил этот код:

```
*(y.deref())
```

Язык Rust заменяет оператор `*` вызовом метода `deref`, а затем просто следует по ссылке к существующему значению, и поэтому нам не приходится вызывать метод `deref`. Это средство языка Rust позволяет писать код, который функционирует одинаково независимо от того, что у нас есть: регулярная ссылка либо тип, который реализует `Deref`.

Из-за системы владения метод `deref` возвращает ссылку на значение, и попрежнему нужно просто идти по ссылке к существующему значению вне скобок в `*(y.deref())`. Если бы метод `deref` возвращал вместо ссылки на значение непосредственно само значение, то это значение было бы перемещено из `self`. Мы не хотим брать во владение внутреннее значение `MyBox<T>` в данном случае, да и в большинстве случаев, когда используется оператор разыменования `*`.

Обратите внимание, что оператор * заменяется вызовом метода `deref`, а затем вызовом оператора * всего один раз, когда мы используем * в коде. Поскольку замена оператора * не входит в бесконечную рекурсию, мы в итоге имеем данные типа `i32`, которые совпадают с `5` в инструкции `assert_eq!` из листинга 15.9.

Скрытые принудительные приведения типов посредством deref с функциями и методами

Принудительное приведение типа посредством `deref` — это вспомогательная операция, которую Rust выполняет с аргументами функций и методов. Указанная операция преобразует ссылку на тип, который реализует типаж `Deref`, в ссылку на тип, в который типаж `Deref` может преобразовать исходный тип. Принудительное приведение типа посредством `deref` происходит автоматически, когда мы передаем ссылку на значение некоторого типа в качестве аргумента функции или метода, который не совпадает с типом параметра в определении функции или метода. Последовательность вызовов метода `deref` преобразует предоставленный нами тип в тип, необходимый параметру.

Принудительное приведение типа посредством `deref` было добавлено в Rust, чтобы программистам, пишущим вызовы функций и методов, не приходилось добавлять так много явных операций референции и разыменования с использованием `&` и *. Средство принудительного приведения типа посредством `deref` также позволяет писать код, который может работать как для ссылок, так и для умных указателей.

Для того чтобы увидеть принудительное приведение типа посредством `deref` в действии, давайте применим тип `MyBox<T>`, который мы определили в листинге 15.8, а также реализацию `Deref`, добавленную в листинг 15.10. В листинге 15.11 показано определение функции, имеющей параметр с типом строкового среза.

Листинг 15.11. Функция `hello`, которая имеет параметр `name` типа `&str`

src/main.rs

```
fn hello(name: &str) {
    println!("Здравствуй, {}!", name);
}
```

Мы можем вызвать функцию `hello` со строковым срезом в качестве аргумента, к примеру, `hello("Rust");`. Принудительное приведение типа посредством `deref` позволяет вызывать `hello` со ссылкой на значение типа `MyBox<String>`, как показано в листинге 15.12.

Листинг 15.12. Вызов функции `hello` со ссылкой на значение `MyBox<String>`, которое работает из-за принудительного приведения типа посредством `deref`

src/main.rs

```
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&m);
}
```

Здесь мы вызываем функцию `hello` с аргументом `&m`, то есть ссылкой на значение `MyBox<String>`. Поскольку в листинге 15.10 мы реализовали типаж `Deref` в типе `MyBox<T>`, компилятор может превратить `&MyBox<String>` в `&String` путем вызова `deref`. Стандартная библиотека предоставляет реализацию типажа `Deref` в типе `String`, которая возвращает строковый срез, и эта информация есть в документации API о типаже `Deref`. Язык Rust снова вызывает `deref`, превращая `&String` в `&str`, что совпадает с определением функции `hello`.

Если бы принудительное приведение типа посредством `deref` не было реализовано в Rust, то для вызова функции `hello` со значением типа `&MyBox<String>` вместо кода из листинга 15.12 нам пришлось бы писать код из листинга 15.13.

Листинг 15.13. Код, который мы должны были бы написать, если бы принудительного приведения типа посредством deref не было

src/main.rs

```
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&(*m)[..]);
}
```

Выражение `(*m)` дереференцирует[1] тип `MyBox<String>` в тип `String`. Затем `&` и `[..]` берут строковый срез экземпляра типа `String`, который равен всему строковому значению, чтобы совпасть с сигнатурой функции `hello`. Код без принудительного приведения типа посредством `deref` труднее читать, писать и понимать со всеми этими символами. Принудительное приведение типа посредством `deref` позволяет компилятору обрабатывать эти конверсии автоматически.

Когда типаж `Deref` будет определен для участвующих типов, язык Rust будет анализировать типы и использовать метод `Deref::deref` столько раз, сколько необходимо, чтобы получить ссылку, совпадающую с типом параметра. Число раз, когда нужно вставить вызов метода `Deref::deref`, определяется во время компиляции, поэтому время выполнения не увеличивается из-за принудительного приведения типа посредством `deref`!

Как принудительное приведение типа посредством deref взаимодействует с изменяемостью

Вы можете использовать типаж `DerefMut` для переопределения оператора * на изменяемых ссылках подобно тому, как вы используете типаж `Deref` для переопределения оператора * на неизменяемых ссылках.

Язык Rust выполняет принудительное приведение типа посредством `deref`, когда он находит типы и реализации типажей в трех случаях:

---

[1] То есть переходит по указателю MyBox к значению типа String с помощью оператора *.

- Из `&T` в `&U`, когда `T: Deref<Target=U>`.
- Из `&mut T` в `&mut U`, когда `T: DerefMut<Target=U>`.
- Из `&mut T` в `&U`, когда `T: Deref<Target=U>`.

Первые два случая одинаковы, за исключением изменяемости. Первый случай констатирует, что если у вас есть ссылка `&T`, а тип `T` реализует типаж `Deref` для некоторого типа `U`, то вы можете получить ссылку `&U` прозрачно. Второй случай констатирует, что то же самое принудительное приведение типа посредством `deref` происходит для изменяемых ссылок.

Третий случай хитрее: Rust также выполняет приведение изменяемой ссылки к неизменяемой. Но обратное невозможно: неизменяемые ссылки никогда не будут приводиться к изменяемым ссылкам. По причине правил заимствования, если у вас есть изменяемая ссылка, то эта изменяемая ссылка должна быть единственной ссылкой на эти данные (в противном случае программа не компилируется). Конвертирование одной изменяемой ссылки в одну неизменяемую ссылку никогда не нарушит правила заимствования. Конвертирование неизменяемой ссылки в изменяемую ссылку потребует наличия только одной неизменяемой ссылки на эти данные, а правила заимствования этого не гарантируют. Следовательно, в Rust нельзя предположить, что конвертирование неизменяемой ссылки в изменяемую возможно.

Выполнение кода при очистке с помощью типажа Drop

Второй типаж, важный для паттерна умного указателя, — это `Drop`, который позволяет индивидуально настраивать то, что происходит, когда значение вот-вот выйдет из области видимости. Вы можете реализовать типаж `Drop` для любого типа, а указанный код может использоваться для высвобождения ресурсов, таких как файлы или сетевые подключения. Мы вводим `Drop` в контексте умных указателей, потому что функциональность типажа `Drop` почти всегда используется во время реализации умного указателя. Например, умный указатель `Box<T>` настраивает типаж `Drop` индивидуально для отмены выделения пространства в куче, на которую указывает тип `Box`.

В некоторых языках программист должен вызывать код для высвобождения памяти или ресурсов всякий раз, когда он заканчивает использовать экземпляр умного указателя. Если он об этом забудет, то система может оказаться перегруженной и рухнуть. В Rust вы можете указать, что некоторая часть кода будет выполняться всякий раз, когда значение выходит из области видимости, и компилятор вставит этот код автоматически. В результате вам не нужно размещать код очистки в программе там, где завершался экземпляр некоторого типа, — у вас не будет утечки ресурсов!

Задайте код, который будет выполняться, когда значение выходит из области видимости, путем реализации типажа `Drop`. Типаж `Drop` требует реализации одного метода с именем `drop`, который принимает изменяемую ссылку на `self`. Для того чтобы увидеть, когда Rust вызывает метод `drop`, давайте пока реализуем метод `drop` с инструкциями `println!`.

Листинг 15.14 показывает структуру `CustomSmartPointer`, единственная настраиваемая функциональность которой заключается в том, что она выведет

```
Dropping CustomSmartPointer!
```

когда экземпляр выходит из области видимости. Этот пример показывает, когда язык Rust выполняет метод `drop`.

Листинг 15.14. Структура CustomSmartPointer, реализующая типаж Drop там, где мы разместим код очистки

src/main.rs

```
struct CustomSmartPointer {
    data: String,
}

❶ impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
      ❷ println!("Отбрасывается CustomSmartPointer с данными `{}`!", self.data);
    }
}

fn main() {
  ❸ let c = CustomSmartPointer { data: String::from("мои вещи") };
  ❹ let d = CustomSmartPointer { data: String::from("чужие вещи") };
  ❺ println!("Экземпляры CustomSmartPointer созданы.");
❻ }
```

Типаж `Drop` включен в прелюдию, поэтому нам не нужно вводить его в область видимости. Мы реализуем типаж `Drop` в типе `CustomSmartPointer` ❶ и предоставляем реализацию метода `drop`, который вызывает макрокоманду `println!` ❷. В теле метода `drop` вы будете размещать любую логику, которую хотите выполнять, когда экземпляр вашего типа выходит из области видимости. Здесь мы печатаем текст, чтобы показать, когда Rust вызывает метод `drop`.

В функции `main` мы создаем два экземпляра типа `CustomSmartPointer` ❸❹, а затем печатаем сообщение `Экземпляры CustomSmartPointer созданы.` ❺ В конце функции `main` ❻ экземпляры типа `CustomSmartPointer` выйдут из области видимости, и Rust вызовет код, который мы поместим в метод `drop`❷, печатающий окончательное сообщение. Обратите внимание, нам не нужно было вызывать метод `drop` явным образом.

Когда мы выполним эту программу, то увидим следующие данные:

```
Экземпляры CustomSmartPointer созданы.
Отбрасывается CustomSmartPointer с данными `чужие вещи`!
Отбрасывается CustomSmartPointer с данными `мои вещи`!
```

Rust автоматически вызывал метод `drop`, когда экземпляры выходили из области видимости, вызывая заданный нами код. Переменные отбрасываются в обратном порядке их создания, поэтому переменная `d` была отброшена до переменной `c`.

Этот пример дает вам наглядное руководство по работе метода `drop`. Обычно вместо печатаемого сообщения вы задаете код очистки, который должен выполняться типом.

Досрочное отбрасывание значения с помощью std::mem::drop

К сожалению, отключить автоматический метод `drop` не так просто. Отключать метод `drop` обычно не требуется, смысл типажа `Drop` в том, что он обрабатывается автоматически. Время от времени, однако, вы, возможно, захотите очищать значение досрочно. Один из примеров — использование умных указателей, управляющих замками: вы, возможно, захотите выполнить метод `drop` принудительно, освободив замок, чтобы другой код в той же области видимости смог получить этот замок. Rust не дает вызывать метод `drop` типажа `Drop` вручную. Если вы захотите принудительно отбросить значение до конца его области видимости, то должны вызвать функцию `std::mem::drop`, предусмотренную стандартной библиотекой.

Если мы попытаемся вызвать метод `drop` типажа `Drop` вручную путем модификации функции `main` в листинге 15.14, как показано в листинге 15.15, то произойдет ошибка компилятора.

Листинг 15.15. Попытка вызвать метод drop из типажа Drop вручную для досрочной очистки

src/main.rs

```
fn main() {
    let c = CustomSmartPointer { data: String::from("некие данные") };
    println!("Экземпляр CustomSmartPointer создан.");
    c.drop();
    println!("CustomSmartPointer отброшен до конца функции main.");
}
```

Когда мы попытаемся скомпилировать этот код, то получим ошибку[1]:

```
error[E0040]: explicit use of destructor method
 --> src/main.rs:14:7
   |
14 |    c.drop();
   |      ^^^^ explicit destructor calls not allowed
```

[1] `ошибка[E0040]: явное использование метода деструктора`

Это сообщение об ошибке указывает, что нельзя вызывать метод `drop` явным образом. В указанном сообщении используется общий термин программирования «деструктор», обозначающий функцию, которая очищает экземпляр. Деструктор аналогичен конструктору, который создает экземпляр. Функция `drop` языка Rust представляет собой деструктор.

Язык Rust не позволяет вызывать метод `drop` явным образом, потому что он будет вызывать метод `drop` автоматически для значения в конце функции `main`. Иначе это привело бы к ошибке двойного высвобождения, потому что язык Rust пытался бы очистить одно и то же значение дважды.

Нельзя отключить автоматическую вставку `drop`, когда значение выходит из области видимости, также нельзя вызывать метод `drop` явным образом. Поэтому, если нам нужно принудительно очистить значение досрочно, то мы можем применить функцию `std::mem::drop`.

Функция `std:: mem:: drop` отличается от метода `drop` из типажа `Drop`. Мы вызываем ее, передавая в качестве аргумента значение, которое нужно принудительно отбросить досрочно. Указанная функция находится в прелюдии, поэтому мы можем изменить функцию `main` из листинга 15.15 так, чтобы та вызывала функцию `drop`, как показано в листинге 15.16.

Листинг 15.16. Вызов функции `std::mem::drop` для явного отбрасывания значения, прежде чем оно выйдет из области видимости

src/main.rs

```
fn main() {
    let c = CustomSmartPointer { data: String::from("некие данные") };
    println!("Экземпляр CustomSmartPointer создан.");
    drop(c);
    println!("CustomSmartPointer отброшен до конца функции main.");
}
```

При выполнении этого кода будет выведено следующее:

```
CustomSmartPointer created.
Dropping CustomSmartPointer with data `some data`!
CustomSmartPointer dropped before the end of main.
```

Текст `Dropping CustomSmartPointer with data `some data`!` выводится между текстом `CustomSmartPointer created.` и текстом `CustomSmartPointer dropped before the end of main.`, показывая, что код функции `drop` вызывается для отбрасывания переменной `c` в этой точке.

Вы можете использовать код, задаваемый в реализации типажа `Drop`, многими способами, делая очистку удобной и безопасной. Например, можно использовать его для создания собственного средства выделения памяти! С типажом `Drop` и системой владения Rust вам не нужно помнить об очистке, потому что компилятор делает ее автоматически.

Вам также не нужно беспокоиться о проблемах, возникающих в результате нечаянной очистки используемых значений: благодаря системе владения ссылки всегда действительны, а метод `drop` вызывается только один раз, когда значение больше не используется.

Теперь, рассмотрев `Box<T>` и некоторые характеристики умных указателей, давайте обратимся к другим умным указателям из стандартной библиотеки.

Rc<T> — умный указатель подсчета ссылок

В большинстве случаев владение понятно: вы точно знаете, какая переменная владеет тем или иным значением. Однако бывают случаи, когда у одного значения может быть несколько владельцев. Например, в графовых структурах данных несколько ребер могут указывать на один и тот же узел, и этот узел концептуально находится во владении всех ребер, которые на него указывают. Узел не должен очищаться, если только на него не указывает ни одно ребро.

Для множественного владения в языке Rust имеется тип с именем `Rc<T>` — это аббревиатура словосочетания *reference counting*, то есть «подсчет ссылок». Тип `Rc<T>` отслеживает число ссылок на значение и позволяет выяснить, используется ли значение по-прежнему. Если имеется ноль ссылок на значение, то это значение можно очистить, при этом ссылки не становятся недействительными.

Представьте себе `Rc<T>` как телевизор в гостиной. Когда один человек приходит посмотреть телевизор, он его включает. Другие тоже могут прийти в комнату посмотреть телевизор. Когда последний человек выходит из комнаты, он выключает телевизор, потому что его больше никто не смотрит. Если кто-то выключит телевизор, когда другие его смотрят, то оставшиеся телезрители будут недовольны!

Мы используем тип `Rc<T>`, когда хотим разместить некоторые данные в куче, чтобы их читали несколько частей программы, и мы не можем выяснить во время компиляции, какая часть последней завершит использование данных. Если бы мы знали, какая часть будет последней, то мы бы просто могли сделать указанную часть владельцем этих данных, и в силу вступили бы обычные правила владения, применяемые во время компиляции.

Обратите внимание, что тип `Rc<T>` предназначен только для использования в однопоточных сценариях. При рассмотрении конкурентности в главе 16 мы изучим, как считать ссылки в многопоточных программах.

Применение Rc<T> для совместного использования данных

Давайте вернемся к примеру с cons-списком в листинге 15.5. Напомним, что мы определили его с помощью умного указателя `Box<T>`. На этот раз мы создадим два

списка, которые будут совместно владеть третьим списком. Концептуально это выглядит примерно так же, как на рис. 15.3.

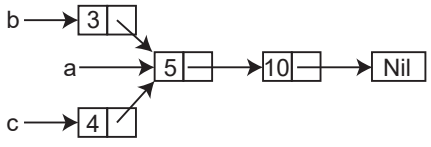

Рис. 15.3. Два списка — b и c — совместно владеющие третьим списком — a

Мы создадим список `a`, который содержит 5, а затем 10. Затем мы составим еще два списка: `b`, который начинается с 3, и `c`, который начинается с 4. Затем оба списка, `b` и `c`, перейдут к первому списку, `a`, содержащему 5 и 10. Другими словами, оба списка будут совместно использовать первый список, содержащий 5 и 10.

Попытка реализовать этот сценарий с определением перечисления `List` и умным указателем `Box<T>` не сработает, как показано в листинге 15.17.

Листинг 15.17. Нельзя иметь два списка, использующих умный указатель Box<T>, которые пытаются совместно владеть третьим списком

src/main. rs

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let a = Cons(5,
        Box::new(Cons(10,
            Box::new(Nil))));
 ❶ let b = Cons(3, Box::new(a));
 ❷ let c = Cons(4, Box::new(a));
}
```

Когда мы компилируем этот код, то получаем ошибку[1]:

```
error[E0382]: use of moved value: `a`
 --> src/main.rs:13:30
   |
12 |     let b = Cons(3, Box::new(a));
   |                              - value moved here
13 |     let c = Cons(4, Box::new(a));
   |                              ^ value used here after move
   |
   = note: move occurs because `a` has type `List`, which does not implement
   the `Copy` trait
```

[1] ошибка[E0382]: использование перемещенного значения `a`

Варианты `Cons` владеют данными, которые они содержат, поэтому, когда мы создаем список `b` ❶, список `a` перемещается в `b`, и `b` владеет `a`. Затем, когда мы пытаемся использовать список `a` снова при создании списка `c` ❷, это не разрешается, потому что `a` был перемещен.

Мы могли бы изменить определение `Cons` так, чтобы содержались ссылки, но тогда пришлось бы задавать параметры жизненных циклов. Задавая параметры жизненных циклов, мы бы описывали, что каждый элемент в списке будет жить по крайней мере столько же, сколько и весь список. Например, контролер заимствования не позволит нам скомпилировать инструкцию `let a = Cons(10, &Nil);`, потому что временное значение `Nil` будет отброшено до того, как `a` сможет взять ссылку на него.

Мы изменим определение типа `List` так, чтобы использовать умный указатель `Rc<T>` вместо умного указателя `Box<T>`, как показано в листинге 15.18. Каждый вариант `Cons` теперь будет содержать значение и умный указатель `Rc<T>`, указывающий на `List`. Когда мы создаем `b`, вместо того чтобы брать `a` во владение, мы будем клонировать умный указатель `Rc<List>`, который содержит `a`, тем самым увеличивая число ссылок с одной до двух и позволяя `a` и `b` совместно владеть данными в этом умном указателе `Rc<List>`. Мы также будем клонировать `a` при создании `c`, увеличивая число ссылок с двух до трех. Каждый раз, когда мы вызываем `Rc::clone`, число ссылок на данные в умном указателе `Rc<List>` увеличивается, и данные не очищаются, если только на них нет ни одной ссылки.

Листинг 15.18. Определение типа List, который использует умный указатель Rc<T>

src/main.rs

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}
use crate::List::{Cons, Nil};
❶ use std::rc::Rc;

fn main() {
  ❷ let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
  ❸ let b = Cons(3, Rc::clone(&a));
  ❹ let c = Cons(4, Rc::clone(&a));
}
```

Нужно добавить инструкцию `use` для введения `Rc<T>` в область видимости ❶, потому что его нет в прелюдии. В функции `main` мы создаем список, содержащий 5 и 10, и сохраняем его в новом умном указателе `Rc<List>` в `a` ❷. Затем, создавая `b` ❸ и `c`❹, мы вызываем функцию `Rc::clone` и передаем внутрь ссылку на `Rc<List>` в качестве аргумента.

Вместо функции `Rc::clone(&a)` мы могли бы вызвать метод `a.clone()`, но исходя из принятых соглашений в данном случае используется функция `Rc::clone`. Реализация функции `Rc::clone` не делает глубокую копию всех данных, как это

делают реализации метода `clone` в большинстве типов. Вызов функции `Rc::clone` только наращивает число ссылок, это недолгий процесс. Глубокое копирование данных занимает много времени. Используя функцию `Rc::clone` для подсчета ссылок, мы можем видеть различие между видами клонирований с глубоким копированием и с увеличением числа ссылок. При поиске проблем с производительностью в коде мы должны учитывать только клонирования с глубоким копированием и не обращать внимания на вызовы функции `Rc::clone`.

Клонирование Rc<T> увеличивает число ссылок

Давайте изменим пример из листинга 15.18 и посмотрим, как изменяется число ссылок, когда мы создаем и отбрасываем ссылки на `Rc<List>` в `a`.

В листинге 15.19 мы изменим функцию `main`, чтобы у нее была внутренняя область вокруг списка `c`. Тогда мы сможем увидеть, как изменяется число ссылок, когда `c` выходит из указанной области.

Листинг 15.19. Вывод числа ссылок

src/main.rs

```
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("число после создания a = {}", Rc::strong_count(&a));
    let b = Cons(3, Rc::clone(&a));
    println!("число после создания b = {}", Rc::strong_count(&a));
    {
        let c = Cons(4, Rc::clone(&a));
        println!("число после создания c = {}", Rc::strong_count(&a));
    }
    println!("число после выхода c из области видимости = {}",
             Rc::strong_count(&a));
}
```

В каждой точке программы, где число ссылок изменяется, мы печатаем число ссылок, которое можно получить путем вызова функции `Rc::strong_count`. Эта функция названа `strong_count` («подсчет числа сильных ссылок») вместо `count`, потому что тип `Rc<T>` также имеет функцию `weak_count` («подсчет числа слабых ссылок»). Мы увидим, для чего используется функция `weak_count`, в разделе «Предотвращение циклов в переходах по ссылкам: превращение Rc<T> в Weak<T>» (с. 397).

Этот код выводит следующее:

```
число после создания a = 1
число после создания b = 2
число после создания c = 3
число после выхода c из области видимости = 2
```

Мы видим, что изначально `Rc<List>` имеет в `a` число ссылок, равное 1. Затем каждый раз, когда мы вызываем `clone`, это число увеличивается на 1. Когда `c` выхо-

дит из области видимости, число уменьшается на 1. Нам не приходится вызывать функцию, чтобы уменьшить число ссылок, в отличие от ситуации, когда нужно вызывать функцию `Rc::clone`, чтобы увеличить число ссылок. Реализация типажа `Drop` уменьшает число ссылок автоматически, когда значение `Rc<T>` выходит из области видимости.

Чего мы не увидим в этом примере, так это то, что когда `b` и затем `a` выходят из области видимости в конце функции `main`, то число становится равным 0, и в этот момент `Rc<List>` полностью очищается. Использование `Rc<T>` позволяет одному значению иметь несколько владельцев, а подсчет ссылок обеспечивает, чтобы значение оставалось действительным до тех пор, пока любой из владельцев существует.

Посредством неизменяемых ссылок умный указатель `Rc<T>` позволяет делиться данными только для чтения между несколькими частями программы. Если бы помимо этого умный указатель `Rc<T>` разрешил иметь несколько изменяемых ссылок, то вы бы могли нарушить одно из правил заимствования, описанных в главе 4: несколько изменяемых заимствований в одном и том же месте могут стать причиной гонки данных и несогласованности. Но изменять данные — очень полезная возможность! В следующем разделе мы обсудим паттерн внутренней изменяемости и тип `RefCell<T>`, который вы можете использовать в сочетании с умным указателем `Rc<T>` для работы с ограничением неизменяемости.

RefCell<T> и паттерн внутренней изменяемости

Внутренняя изменяемость — это паттерн проектирования, который позволяет изменять данные, даже когда есть неизменяемые ссылки на эти данные. Обычно это действие запрещено правилами заимствования. Для того чтобы изменить данные, указанный паттерн использует небезопасный (`unsafe`) код внутри структуры данных, чтобы обойти обычные правила Rust, которые управляют изменением и заимствованием. Мы рассмотрим небезопасный код в главе 19. Можно использовать типы, использующие паттерн внутренней изменяемости, когда мы можем обеспечить, чтобы правила заимствования соблюдались во время выполнения, даже если компилятор этого не гарантирует. В этом случае небезопасный код обертывается в безопасный API, и внешний тип остается неизменяемым.

Давайте узнаем, как это работает, обратившись к типу `RefCell<T>`, который подчиняется паттерну внутренней изменяемости.

Соблюдение правил заимствования во время выполнения с помощью RefCell<T>

В отличие от `Rc<T>`, тип `RefCell<T>` представляет одинарное владение данными, которые он содержит. Что отличает `RefCell<T>` от типа `Box<T>`? Вспомните правила заимствования, с которыми вы познакомились в главе 4:

- В любой момент у вас может быть одно из следующих значений, но не оба: одна изменяемая ссылка или любое число неизменямых ссылок.
- Ссылки всегда должны быть действительными.

В случае со ссылками и умным указателем `Box<T>` инварианты правил заимствования соблюдаются во время компиляции. В случае с умным указателем `RefCell<T>` эти инварианты соблюдаются во время выполнения. В случае со ссылками, если вы нарушите эти правила, то произойдет ошибка компилятора. В случае с `RefCell<T>`, если вы их нарушите, то программа поднимет панику и завершится.

Преимущества проверки правил заимствования во время компиляции заключаются в том, что ошибки выявляются на раннем этапе в процессе разработки, и ничто не влияет на производительность времени выполнения, поскольку весь анализ выполняется заранее. Поэтому проверка правил заимствования во время компиляции — чаще всего лучший вариант и выполняется в Rust по умолчанию.

Преимущество проверки правил заимствования во время выполнения заключается в том, что в этом случае разрешаются некоторые безопасные для памяти сценарии, тогда как они запрещаются проверками во время компиляции. Статический анализ, как и компилятор, по своей сути является консервативным. Некоторые свойства кода невозможно обнаружить путем анализа: самый известный пример — это проблема остановки, рассмотрение которой выходит за рамки темы данной книги, но это интересный вопрос для исследования.

Поскольку анализ невозможен, если компилятор не способен проверить, что код согласуется с правилами владения, он может отклонить правильную программу. В этом смысле он консервативен. Если бы Rust принял неправильную программу, то пользователи не смогли бы доверять возможностям этого языка. Но если Rust будет отклонять правильную программу, то программисту будет не очень удобно работать, но ничего страшного не произойдет. Тип `RefCell<T>` полезен, когда вы уверены, что код следует правилам заимствования, но компилятор не может понять и гарантировать это.

Как и `Rc<T>`, тип `RefCell<T>` предназначен для использования только в однопоточных сценариях и будет выдавать ошибку времени компиляции, если вы попытаетесь применить его в многопоточном контексте. О том, как получить функциональность типа `RefCell<T>` в многопоточной программе, мы поговорим в главе 16.

Кратко скажем, почему нужно выбирать типы `Box<T>`, `Rc<T>` или `RefCell<T>`:

- Тип `Rc<T>` дает возможность иметь более одного владельца одних и тех же данных. У типов `Box<T>` и `RefCell<T>` единственный владелец.
- Тип `Box<T>` допускает, чтобы неизменяемые или изменяемые заимствования проверялись во время компиляции. Тип `Rc<T>` допускает, чтобы только неизменяемые заимствования проверялись во время компиляции. Тип `RefCell<T>` допускает, чтобы неизменяемые либо изменяемые заимствования проверялись во время выполнения.

- Так как тип `RefCell<T>` допускает, чтобы изменяемые заимствования проверялись во время выполнения, вы можете изменять значение внутри типа `RefCell<T>`, даже когда тип `RefCell<T>` неизменяем.

Изменение значения внутри неизменяемого значения — это паттерн внутренней изменяемости. Давайте посмотрим, когда внутренняя изменяемость бывает полезна, и узнаем, как это работает.

Внутренняя изменяемость: изменяемое заимствование неизменяемого значения

Из правил заимствования следует, что, когда у вас есть неизменяемое значение, вы не можете заимствовать его изменяемо. Например, этот код не компилируется:

```
fn main() {
    let x = 5;
    let y = &mut x;
}
```

Если вы попытаетесь скомпилировать этот код, то получите ошибку[1]:

```
error[E0596]: cannot borrow immutable local variable 'x' as mutable
 --> src/main.rs:3:18
  |
2 |     let x = 5;
  |         - consider changing this to 'mut x'
3 |     let y = &mut x;
  |                  ^ cannot borrow mutably
```

Однако есть ситуации, в которых полезно, чтобы значение само изменялось в своих методах, но было неизменяемым для другого кода. Код, находящийся вне методов такого значения, не сможет изменить это значение. Использование типа `RefCell<T>` — это один из способов получить внутреннюю изменяемость. Но тип `RefCell<T>` не обходит правила заимствования полностью: контролер заимствования в компиляторе допускает эту внутреннюю изменяемость, а правила заимствования проверяются во время выполнения. Нарушив эти правила, вместо ошибки компилятора вы получите `panic!`.

Давайте на практике посмотрим, где можно использовать тип `RefCell<T>` для изменения неизменяемого значения, и узнаем, почему это полезно.

Вариант использования внутренней изменяемости: mock-объекты

Тестовый двойник — это часто встречающееся понятие программирования, когда во время тестирования один тип используется вместо другого. Объекты-пустыш-

[1] `ошибка[E0596]: невозможно заимствовать неизменяемую локальную переменную 'x' как изменяемую`

ки, или mock-объекты, — это специфические типы тестовых двойников, которые регистрируют происходящее во время теста, что позволяет проверить правильность предпринятых действий.

В Rust нет объектов в том же смысле, что и в других языках, а в стандартную библиотеку Rust не встроена функциональность имитационных объектов, как в некоторых других языках. Однако вы можете создать структуру, подобную имитационному объекту.

Вот сценарий, который мы будем тестировать: мы создадим библиотеку, которая отслеживает значение по отношению к максимальному значению и отправляет сообщения о том, насколько близко текущее значение к максимальному. Эта библиотека может использоваться, например, для отслеживания квоты пользователя на число вызовов API, которые ему разрешено выполнять.

В нашей библиотеке будет только одна возможность: отследить, насколько близко значение находится к максимальному, а также какие сообщения необходимы и в какое время. Приложения, которые используют библиотеку, должны обеспечить механизм отправки сообщений: приложение могло бы помещать сообщение в приложение, отправлять электронное письмо, текстовое сообщение или что-то еще. Библиотека не должна знать эту деталь. Все, что ей нужно, — это нечто, что реализует типаж под названием `Messenger`. Листинг 15.20 показывает код библиотеки.

Листинг 15.20. Библиотека, позволяющая отслеживать близость значения к максимальному и предупреждать, когда это значение находится на неких уровнях

src/lib.rs

```
pub trait Messenger {
  ❶ fn send(&self, msg: &str);
}

pub struct LimitTracker<'a, T: 'a + Messenger> {
    messenger: &'a T,
    value: usize,
    max: usize,
}

impl<'a, T> LimitTracker<'a, T>
    where T: Messenger {
    pub fn new(messenger: &T, max: usize) -> LimitTracker<T> {
        LimitTracker {
            messenger,
            value: 0,
            max,
        }
    }

  ❷ pub fn set_value(&mut self, value: usize) {
        self.value = value;
```

```
        let percentage_of_max = self.value as f64 / self.max as f64;

        if percentage_of_max >= 1.0 {
            self.messenger.send("Ошибка: Вы превысили свою квоту!");
        } else if percentage_of_max >= 0.9 {
            self.messenger.send("Срочное предупреждение: Вы израсходовали
    свыше 90% своей квоты!");
        } else if percentage_of_max >= 0.75 {
            self.messenger.send("Предупреждение: Вы израсходовали свыше 75%
                                 своей квоты!");
        }
    }
}
```

Одна из важных частей этого кода заключается в том, что типаж `Messenger` имеет один метод под названием `send`, который берет неизменяемую ссылку на `self` и текст сообщения ❶. Это тот интерфейс, который должен быть у имитационного объекта. Другая важная часть заключается в том, что мы хотим проверить поведение метода `set_value` в структуре `LimitTracker`❷. Мы можем изменить то, что передаем внутрь для параметра `value`, но метод `set_value` не возвращает ничего, что бы мы могли использовать для проверочных утверждений. Мы хотим иметь возможность сказать, что если мы создадим структуру `LimitTracker` с чем-то, что реализует типаж `Messenger`, и конкретным значением для поля `max`, то, когда мы передаем разные числа для поля `value`, мессенджер должен отправить соответствующие сообщения.

Нам нужен mock-объект, который вместо отправки электронной почты или текстового сообщения при вызове метода `send` будет отслеживать только те сообщения, которые должен. Мы можем создать новый экземпляр имитационного объекта и структуру `LimitTracker`, которая использует этот имитационный объект, вызвать метод `set_value` для `LimitTracker`, а затем проверить, что у имитационного объекта нужные сообщения. Листинг 15.21 показывает попытку реализовать имитационный объект, чтобы выполнить поставленную задачу, но контролер заимствования не разрешает это сделать.

Листинг 15.21. Попытка реализовать структуру MockMessenger, которая запрещена контролером заимствования

src/lib.rs

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

 ❶ struct MockMessenger {
    ❷ sent_messages: Vec<String>,
    }

    impl MockMessenger {
    ❸ fn new() -> MockMessenger {
            MockMessenger { sent_messages: vec![] }
        }
```

```
    }

❹ impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
          ❺ self.sent_messages.push(String::from(message));
        }
    }

    #[test]
❻ fn it_sends_an_over_75_percent_warning_message() {
        let mock_messenger = MockMessenger::new();
        let mut limit_tracker = LimitTracker::new(&mock_messenger, 100);

        limit_tracker.set_value(80);

        assert_eq!(mock_messenger.sent_messages.len(), 1);
    }
}
```

Этот тестовый код определяет структуру `MockMessenger` ❶, которая имеет поле `sent_messages` с вектором значений типа `String` ❷, чтобы отслеживать сообщения, которые она должна отправлять. Мы также определяем связанную функцию `new` ❸ для удобства в создании новых значений типа `MockMessenger`, которые начинают с пустого списка сообщений. Затем мы реализуем типаж `Messenger` для `MockMessenger` ❹, благодаря чему можно передавать структуру `MockMessenger` структуре `LimitTracker`. В определении метода `send` ❺ мы берем сообщение, переданное внутрь в качестве параметра, и сохраняем его в поле `sent_messages` списка отправленных сообщений, состоящего из структур `MockMessenger`.

В тесте мы проверяем, что происходит, когда `LimitTracker` поручает установить значение равным чему-то, что составляет более 75% от максимального значения ❻. Сначала мы создаем новый экземпляр структуры `MockMessenger`, который будет начинаться с пустого списка сообщений. Затем мы создаем новый экземпляр структуры `LimitTracker` и даем ему ссылку на новый `MockMessenger` и максимальное значение 100. Мы вызываем метод `set_value` для `LimitTracker` со значением 80, что составляет более 75% от 100. Затем мы выполняем проверочное утверждение, что список сообщений, который отслеживается структурой `MockMessenger`, теперь должен содержать одно сообщение.

Однако с этим тестом есть одна проблема, как показано ниже[1]:

```
error[E0596]: cannot borrow immutable field 'self.sent_messages' as mutable
 --> src/lib.rs:52:13
   |
51 |         fn send(&self, message: &str) {
   |                 ----- use '&mut self' here to make mutable
52 |             self.sent_messages.push(String::from(message));
   |             ^^^^^^^^^^^^^^^^^^ cannot mutably borrow immutable field
```

[1] ошибка[E0596]: не получается заимствовать неизменяемое поле 'self.sent_messages' как изменяемое

Мы не можем модифицировать `MockMessenger`, чтобы отслеживать сообщения, потому что метод `send` берет неизменяемую ссылку на `self`. Мы также не можем принять рекомендацию из текста ошибки использовать `&mut self`, потому что тогда сигнатура метода `send` не будет совпадать с сигнатурой в определении типажа `Messenger` (попробуйте это сделать и посмотрите, какое сообщение об ошибке вы получите).

Это именно та ситуация, в которой помогает внутренняя изменяемость! Мы сохраним поле `sent_messages` внутри умного указателя `RefCell<T>`. Затем сообщение из метода `send` сможет модифицировать поле `sent_messages`, чтобы сохранить сообщения, которые мы видели. Листинг 15.22 показывает, как это выглядит.

Листинг 15.22. Использование умного указателя `RefCell<T>` для изменения внутреннего значения, тогда как внешнее значение считается неизменяемым

src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;
    use std::cell::RefCell;

    struct MockMessenger {
     ❶ sent_messages: RefCell<Vec<String>>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
         ❷ MockMessenger { sent_messages: RefCell::new(vec![]) }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
         ❸ self.sent_messages.borrow_mut().push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        // --пропуск--

     ❹ assert_eq!(mock_messenger.sent_messages.borrow().len(), 1);
    }
}
```

Поле `sent_messages` теперь имеет тип `RefCell<Vec<String>>` ❶ вместо `Vec<String>`. В функции `new` мы создаем новый экземпляр `RefCell<Vec<String>>` вокруг пустого вектора ❷.

Для реализации метода `send` первый параметр по-прежнему является неизменяемым заимствованием `self`, которое совпадает с определением типажа. Мы вызыва-

ем `borrow_mut` для `RefCell<Vec<String>>` в `self.sent_messages` ❸, чтобы получить изменяемую ссылку, указывающую на значение внутри `RefCell<Vec<String>>`, которое является вектором. Затем мы можем вызвать `push` для изменяемой ссылки, указывающей на вектор, чтобы отслеживать сообщения, отправляемые во время теста.

Последнее изменение, которое мы должны внести, расположено в проверочном утверждении: чтобы увидеть число элементов во внутреннем векторе, мы вызываем метод `borrow` для `RefCell<Vec<String>>`, получая неизменяемую ссылку на вектор ❹.

Теперь вы знаете, как использовать умный указатель `RefCell<T>`. Давайте углубимся в принцип его работы!

Отслеживание заимствований во время выполнения с помощью RefCell<T>

При создании неизменяемых и изменяемых ссылок мы используем синтаксис `&` и `&mut` соответственно. С умным указателем `RefCell<T>` мы используем методы заимствования `borrow` и `borrow_mut`, являющиеся частью безопасного API, который принадлежит умному указателю `RefCell<T>`. Метод `borrow` возвращает тип умного указателя `Ref<T>`, а `borrow_mut` возвращает тип умного указателя `RefMut<T>`. Оба типа реализуют типаж `Deref`, поэтому мы можем рассматривать их как регулярные ссылки.

Умный указатель `RefCell<T>` отслеживает, сколько умных указателей `Ref<T>` и `RefMut<T>` активны в настоящий момент. Всякий раз, когда мы вызываем метод `borrow`, умный указатель `RefCell<T>` увеличивает число активных неизменяемых заимствований. Когда значение умного указателя `Ref<T>` выходит из области видимости, число неизменяемых заимствований уменьшается на единицу. Как и правила заимствования времени компиляции, умный указатель `RefCell<T>` позволяет в любой момент иметь много неизменяемых заимствований либо одно изменяемое заимствование.

Если мы попытаемся эти правила нарушить, то вместо ошибки компилятора, как это было бы со ссылками, реализация умного указателя `RefCell<T>` поднимет панику во время выполнения. Листинг 15.23 показывает модификацию реализации метода `send` из листинга 15.22. Мы намеренно пытаемся создать два изменяемых заимствования, активных для одной и той же области видимости, чтобы проиллюстрировать, что умный указатель `RefCell<T>` не позволяет нам делать это во время выполнения.

Листинг 15.23. Создание двух изменяемых ссылок в одной области видимости как иллюстрация того, что умный указатель RefCell<T> будет паниковать

src/lib.rs

```
impl Messenger for MockMessenger {
    fn send(&self, message: &str) {
```

```
            let mut one_borrow = self.sent_messages.borrow_mut();
            let mut two_borrow = self.sent_messages.borrow_mut();

            one_borrow.push(String::from(message));
            two_borrow.push(String::from(message));
        }
    }
```

Мы создаем переменную `one_borrow` для умного указателя `RefMut<T>`, возвращаемого из функции `loan_mut`. Затем мы создаем еще одно изменяемое заимствование таким же образом в переменной `two_borrow`. В результате в одной области видимости создаются две изменяемые ссылки, что недопустимо. Когда мы выполним тесты для библиотеки, код в листинге 15.23 будет скомпилирован без ошибок, но тест не сработает:

```
---- tests::it_sends_an_over_75_percent_warning_message stdout ----
        thread 'tests::it_sends_an_over_75_percent_warning_message' panicked
at 'already borrowed: BorrowMutError', src/libcore/result.rs:906:4
note: Run with 'RUST_BACKTRACE=1' for a backtrace.
```

Обратите внимание, что код поднял панику с сообщением `already borrowed: BorrowMutError` («уже заимствовано»). Именно так умный указатель `RefCell<T>` обрабатывает нарушения правил заимствования во время выполнения.

Фиксация ошибок заимствования во время выполнения, а не во время компиляции означает, что в процессе разработки вы либо поздно найдете ошибку в коде, либо она обнаружится, лишь когда код будет запущен в производство. Кроме того, из-за отслеживания заимствований во время выполнения, а не во время компиляции произойдет небольшое снижение производительности времени выполнения кода. Однако благодаря умному указателю `RefCell<T>` можно написать имитационный объект, способный модифицировать себя, чтобы отслеживать сообщения, которые он видел, пока вы используете его в контексте, где разрешены только неизменяемые значения. Вы можете использовать умный указатель `RefCell<T>` несмотря на его компромиссы, чтобы воспользоваться бóльшими преимуществами, чем дают регулярные ссылки.

Наличие нескольких владельцев изменяемых данных путем сочетания Rc<T> и RefCell<T>

Умный указатель `RefCell<T>` часто используется в сочетании с умным указателем `Rc<T>`. Напомним, что `Rc<T>` позволяет иметь несколько владельцев данных, но он дает только неизменяемый доступ к этим данным. Если у вас есть умный указатель `Rc<T>`, содержащий `RefCell<T>`, то можно получить значение, способное иметь нескольких владельцев, которое можно изменять!

Вспомните пример cons-списка из листинга 15.18, где мы использовали умный указатель `Rc<T>`, чтобы разрешить нескольким спискам совместно владеть дру-

гим списком. Поскольку умный указатель `Rc<T>` содержит только неизменяемые значения, создав значения, мы не можем изменить ни одно из них. Давайте добавим умный указатель `RefCell<T>`, чтобы изменять значения в списках. Листинг 15.24 показывает, что при использовании умного указателя `RefCell<T>` в определении `Cons` мы можем модифицировать значение, хранящееся во всех списках.

Листинг 15.24. Использование умного указателя Rc<RefCell<i32>> для создания экземпляра перечисления List, который мы можем изменять

src/main.rs

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}
use crate::List::{Cons, Nil};
use std::rc::Rc;
use std::cell::RefCell;

fn main() {
  ❶ let value = Rc::new(RefCell::new(5));

  ❷ let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(6)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(10)), Rc::clone(&a));

  ❸ *value.borrow_mut() += 10;

    println!("a после = {:?}", a);
    println!("b после = {:?}", b);
    println!("c после = {:?}", c);
}
```

Мы создаем значение, которое является экземпляром `Rc<RefCell<i32>>`, и сохраняем его в переменной с именем `value` ❶, чтобы впоследствии обращаться к нему напрямую. Затем мы создаем `List` в `a` с вариантом `Cons`, который содержит `value` ❷. Нам нужно клонировать `value` так, чтобы и `a`, и `value` были владельцами внутреннего значения 5, а не передавали владение от `value` к `a` или заимствовали у `value`.

Мы обертываем список `a` в умный указатель `Rc<T>`, чтобы при создании списков `b` и `c` они оба могли ссылаться на `a`, что мы и сделали в листинге 15.18.

После того как мы создали списки в `a`, `b` и `c`, мы прибавляем 10 к значению в `value` ❸. Мы делаем это, вызывая функцию `loan_mut` для `value`, которая использует средство автоматической разыменования, описанное в главе 5 (см. врезку на с. 128 «Где оператор ->?») для того, чтобы проследовать по указателю `Rc<T>` к внутреннему значению `RefCell<T>`. Метод `borrow_mut` возвращает умный указатель

`RefMut<T>`, и мы используем для него оператор разыменования и изменяем внутреннее значение.

Когда мы печатаем `a`, `b` и `c`, мы видим, что все они имеют модифицированное значение 15, а не 5:

```
a после = Cons(RefCell { value: 15 }, Nil)
b после = Cons(RefCell { value: 6 }, Cons(RefCell { value: 15 }, Nil))
c после = Cons(RefCell { value: 10 }, Cons(RefCell { value: 15 }, Nil))
```

Этот технический прием довольно хорош! Используя умный указатель `RefCell<T>`, мы получаем внешне неизменяемое значение типа `List`. Но с умным указателем `RefCell<T>` мы можем использовать методы, которые обеспечивают доступ к его внутренней изменяемости, благодаря чему при необходимости можно модифицировать данные. Проверки правил заимствования во время выполнения защищают от гонки данных, и иногда есть смысл пожертвовать скоростью ради гибкости в структурах данных.

Стандартная библиотека имеет другие типы, которые обеспечивают внутреннюю изменяемость. Например, похожий тип `Cell<T>`, но в нем нет ссылки на внутреннее значение, значение копируется в `Cell<T>` и из него. Существует также тип `Mutex<T>`, обеспечивающий внутреннюю изменяемость, безопасную для использования между потоками. Мы обсудим этот тип в главе 16. Дополнительные сведения о различиях между этими типами смотрите в документации стандартной библиотеки.

Циклы в переходах по ссылкам приводят к утечке памяти

В языке Rust из-за гарантий безопасности памяти непросто, но в вместе с тем возможно случайно создать пространство памяти, которое никогда не очищается (так называемая «утечка памяти»). Всеобъемлющее предотвращение утечки памяти — это гарантия Rust, так же как и запрещение гонки данных во время компиляции. То есть в Rust утечка памяти безопасна. Мы видим, что Rust допускает утечку памяти путем использования умных указателей `Rc<T>` и `RefCell<T>`: можно создавать ссылки, где элементы ссылаются друг на друга в цикле. Это приводит к утечке памяти, потому что число ссылок для каждого элемента в цикле никогда не достигнет 0, а значения никогда не будут отброшены.

Создание цикла в переходах по ссылкам

Давайте посмотрим, как происходит цикл в переходах по ссылкам (так называемый «референсный цикл») и как его предотвращать, начав с определения перечисления `List` и метода `tail` в листинге 15.25.

Листинг 15.25. Определение cons-списка, содержащее умный указатель RefCell<T>, чтобы модифицировать то, на что ссылается вариант Cons

src/main.rs

```
use std::rc::Rc;
use std::cell::RefCell;
use crate::List::{Cons, Nil};

#[derive(Debug)]
enum List {
  ❶ Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
  ❷ fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match self {
            Cons(_, item) => Some(item),
            Nil => None,
        }
    }
}
```

Мы используем еще одну вариацию определения перечисления `List` из листинга 15.5. Второй элемент в варианте `Cons` теперь равен умному указателю `RefCell<Rc<List>>`❶, то есть вместо способности изменять значение типа `i32`, как мы делали в листинге 15.24, мы хотим модифицировать некое значение типа `List`, на которое указывает вариант `Cons`. Мы также добавляем метод `tail` ❷, чтобы сделать удобным доступ ко второму элементу, если у нас есть вариант `Cons`.

В листинге 15.26 мы добавляем функцию `main`, которая использует определения из листинга 15.25. Этот код создает список в переменной `a` и список в переменной `b`, которая указывает на список в `a`. Затем он модифицирует список в `a`, указывая на `b` и создавая цикл в переходах по ссылкам. В коде есть инструкции `println!`, которые показывают количества ссылок в различных точках этого процесса.

Листинг 15.26. Создание цикла в переходах по ссылкам из двух значений типа List, которые указывают друг на друга

src/main.rs

```
fn main() {
  ❶ let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    println!("a начальное число rc = {}", Rc::strong_count(&a));
    println!("a следующий элемент = {:?}", a.tail());

  ❷ let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    println!("a число rc после создания b = {}", Rc::strong_count(&a));
    println!("b начальное число rc = {}", Rc::strong_count(&b));
    println!("b следующий элемент = {:?}", b.tail());
```

```
    ❸ if let Some(link) = a.tail() {
        ❹ *link.borrow_mut() = Rc::clone(&b);
      }

      println!("b число rc после изменения a = {}", Rc::strong_count(&b));
      println!("a число rc после изменения a = {}", Rc::strong_count(&a));

      // Раскомментируйте следующую строку кода, и вы увидите, что у нас цикл;
      // он переполнит стек.
      // println!("a следующий элемент = {:?}", a.tail());
}
```

Мы создаем экземпляр `Rc<List>`, содержащий значение типа `List` в переменной `a` с начальным списком, состоящим из `5, Nil` ❶. Затем мы создаем экземпляр `Rc<List>`, содержащий еще одно значение типа `List` в переменной `b`, которая содержит значение `10` и указывает на список в `a` ❷.

Мы модифицируем переменную `a` так, чтобы она указывала на `b` вместо `Nil`, создавая цикл. Мы делаем это с помощью метода `tail`, получающего ссылку на `RefCell<Rc<List>>` в `a`, которую мы помещаем в переменную `link` ❸. Затем мы применяем метод `borrow_mut` для `RefCell<Rc<List>>`, чтобы изменить значение внутри с `Rc<List>`, который содержит значение `Nil`, на `Rc<List>` в `b` ❹.

Когда мы выполним этот код, оставив последнюю инструкцию `println!` закомментированной, мы получим такие данные:

```
a начальное число rc = 1
a следующий элемент = Some(RefCell { value: Nil })
a число rc после создания b = 2
b начальное число rc = 1
b следующий элемент = Some(RefCell { value: Cons(5, RefCell { value: Nil }) })
b число rc после изменения a = 2
a число rc после изменения a = 2
```

Число ссылок экземпляров умного указателя `Rc<List>` в `a` и `b` равно 2 после того, как мы изменим список в `a`, чтобы он указывал на `b`. В конце функции `main` компилятор попытается сначала удалить `b`, что уменьшит число экземпляров умного указателя `Rc<List>` в `b` на 1.

Однако, поскольку `a` по-прежнему ссылается на умный указатель `Rc<List>`, который был в `b`, этот `Rc<List>` имеет число 1, а не 0, поэтому память для умного указателя `Rc<List>` в куче удалена не будет. Память просто останется там навсегда с числом 1. Для визуализации этого цикла в переходах по ссылкам мы создали диаграмму на рис. 15.4.

Если вы раскомментируете последнюю инструкцию `println!` и выполните программу, Rust попытается напечатать этот цикл, в котором `a` указывает на `b`, который указывает на `a`, и так далее, пока он не переполнит стек.

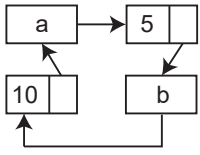

Рис. 15.4. Цикл в переходах по ссылкам в списках a и b, которые указывают друг на друга

В этом случае сразу после того, как мы создадим цикл в переходах по ссылкам, программа завершится. Последствия этого цикла не очень страшны. Однако, если бы более сложная программа выделяла много памяти в цикле и удерживала ее в течение длительного времени, то программа использовала бы больше памяти, чем нужно, и могла бы переполнить систему, приведя к исчерпанию доступной памяти.

Создавать циклы в переходах по ссылкам не так просто, но реально. Если у вас есть значения умного указателя `RefCell<T>`, которые содержат значения `Rc<T>` или схожие вложенные сочетания типов с внутренней изменяемостью и подсчетом числа ссылок, то вы должны убедиться, что не создаете циклы. Нельзя рассчитывать, что компилятор их выявит. Создание цикла в переходах по ссылкам в программе было бы логической ошибкой, которую следует минимизировать с помощью автотестов, ревизий кода и других практик разработки ПО.

Можно избежать циклов в переходах по ссылкам, если реорганизовать структуры данных таким образом, чтобы некоторые ссылки выражали владение, а некоторые — нет. Как результат, циклы будут состоять из нескольких связей владения и нескольких связей отсутствия владения. Только связи владения влияют на то, может ли значение быть отброшено. В листинге 15.25 мы хотим, чтобы варианты `Cons` всегда владели своим списком, и поэтому реорганизация структуры данных невозможна. Давайте рассмотрим пример с графами, состоящими из родительских и дочерних узлов, и выясним, когда связи отсутствия владения подходят для предотвращения циклов в переходах по ссылкам.

Предотвращение циклов в переходах по ссылкам: превращение Rc<T> в Weak<T>

Вы видели, что вызов функции `Rc::clone` увеличивает число `strong_count` экземпляра `Rc<T>`, а экземпляр `Rc<T>` очищается только в том случае, если его число `strong_count` равно 0. Вы также можете создавать слабую ссылку на значение внутри экземпляра `Rc<T>` путем вызова функции `Rc::downgrade` и передачи ссылки на `Rc<T>`. Когда вы вызываете функцию `Rc::downgrade`, вы получаете умный указатель типа `Weak<T>`. Не увеличивая число `strong_count` в экземпляре `Rc<T>` на 1, вызов функции `Rc::downgrade` увеличивает на 1 число `weak_count`. Тип `Rc<T>` ис-

пользует число `weak_count` в целях отслеживания числа ссылок `Weak<T>`, подобно числу `strong_count`. Разница в том, что число `weak_count` не должно равняться 0, чтобы экземпляр тира `Rc<T>` был очищен.

Сильные ссылки — это то, как вы делитесь владением экземпляром `Rc<T>`. Слабые ссылки не выражают связи владения. Они не станут причиной цикла в переходах по ссылкам, потому что любой цикл с участием слабых ссылок будет разрушен, как только число участвующих сильных ссылок будет равно 0.

Так как значение, на которое ссылается `Weak<T>`, возможно, было отброшено, чтобы сделать что-либо со значением, на которое `Weak<T>` продолжает указывать, вы должны убедиться, что значение все еще существует. Делайте это, вызывая метод `upgrade` для экземпляра `Weak<T>`, который будет возвращать `Option<Rc<T>>`. Вы будете получать результат `Some`, если значение `Rc<T>` еще не отброшено, и результат `None`, если значение `Rc<T>` отброшено. Поскольку метод `upgrade` возвращает `Option<T>`, язык Rust сделает так, чтобы обрабатывались случаи `Some` и `None` и не было недействительного указателя.

В качестве примера вместо списка, элементы которого знают только о следующем элементе, мы создадим дерево, элементы которого знают о своих дочерних и родительских элементах.

Создание древовидной структуры данных: узел с дочерними узлами

Для начала мы построим дерево с узлами, которые знают о своих дочерних узлах. Мы создадим структуру с именем `Node` («узел»), которая содержит собственное значение типа `i32`, а также ссылки на значения своих дочерних структур `Node`:

src/main.rs

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

#[derive(Debug)]
struct Node {
    value: i32,
    children: RefCell<Vec<Rc<Node>>>,
}
```

Мы хотим, чтобы структура `Node` владела своими дочерними структурами `Node`, и хотим делиться этим владением с переменными, чтобы можно было обращаться к каждому узлу `Node` в дереве напрямую. Для этого мы определяем элементы типа `Vec<T>` как значения типа `Rc<Node>`. Мы также хотим изменить то, какие узлы являются дочерними другому узлу, поэтому в поле `children` свойство `Vec<Rc<Node>>` оборачивается в умный указатель `RefCell<T>`.

Далее мы воспользуемся определением и создадим один экземпляр структуры `Node` с именем `leaf` со значением 3 без дочерних узлов и еще один экземпляр

с именем `branch` со значением 5 и с `leaf` в качестве одного из его дочерних узлов, как показано в листинге 15.27.

Листинг 15.27. Создание узла leaf без дочерних узлов и узла branch с leaf в качестве одного из его дочерних узлов

src/main.rs

```
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        children: RefCell::new(vec![]),
    });

    let branch = Rc::new(Node {
        value: 5,
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });
}
```

Мы клонируем умный указатель `Rc<Node>` из узла `leaf` и сохраняем его в узле `branch`, то есть экземпляр структуры `Node` в узле `leaf` теперь имеет двух владельцев: `leaf` и `branch`. Мы можем добраться из `branch` в `leaf` через `branch.children`, но путь из `leaf` в `branch` отсутствует. Причина в том, что `leaf` не имеет ссылки на `branch` и не знает, что они связаны. Мы хотим, чтобы узел `leaf` знал, что узел `branch` — его родитель. Мы сделаем это далее.

Добавление ссылки из дочернего узла на родительский

Для того чтобы дочерний узел знал о своем родителе, нужно добавить поле `parent` в определение структуры `Node`. Трудно решить, каким должен быть тип поля `parent`. Мы знаем, что оно не может содержать умный указатель `Rc<T>`, потому что это создаст цикл в переходах по ссылкам, где `leaf.parent` будет указывать на `branch`, а `branch.children` будет указывать на `leaf`, поэтому значения их чисел `strong_count` никогда не будут равны 0.

Если рассмотреть связи с другой стороны, родительский узел должен владеть своими дочерними узлами: если родительский узел отброшен, то и его дочерние узлы тоже должны быть отброшены. Однако дочерний узел не должен владеть своим родителем: если мы отбросим дочерний узел, то родитель все равно должен существовать. Это как раз тот случай, который относится к слабым ссылкам!

Таким образом, вместо умного указателя `Rc<T>` мы сделаем так, чтобы тип родителя использовал тип `Weak<T>`, а точнее — умный указатель `RefCell<Weak<Node>>`. Теперь определение структуры `Node` выглядит следующим образом:

src/main.rs

```
use std::rc::{Rc, Weak};
use std::cell::RefCell;

#[derive(Debug)]
```

```
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}
```

Узел сможет ссылаться на родительский узел, но не будет владеть своим родителем. В листинге 15.28 мы обновляем функцию `main`, чтобы теперь она использовала новое определение, благодаря которому узел `leaf` сможет ссылаться на своего родителя `branch`.

Листинг 15.28. Узел leaf со слабой ссылкой на родительский узел branch

src/main.rs

```
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
     ❶ parent: RefCell::new(Weak::new()),
          children: RefCell::new(vec![]),
    });

  ❷ println!("родительский узел leaf = {:?}", leaf.parent.borrow().upgrade());

    let branch = Rc::new(Node {
        value: 5,
     ❸ parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });

  ❹ *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

  ❺ println!("родительский узел leaf = {:?}", leaf.parent.borrow().upgrade());
}
```

Создание узла `leaf` выглядит так же, как оно выглядело в листинге 15.27, за исключением поля `parent`: узел `leaf` начинается без родителя, поэтому мы создаем новый, пустой экземпляр ссылки `Weak<Node>` ❶.

В этом месте, когда мы пытаемся получить ссылку на родителя узла `leaf` с помощью метода `upgrade`, мы получаем значение `None`. Мы видим это в данных первой инструкции `println!` ❷:

```
leaf parent = None
```

Когда мы создаем узел `branch`, у него также есть новая ссылка `Weak<Node>` в поле `parent` ❸, потому что узел `branch` не имеет родительского узла. У нас еще есть узел `leaf` как один из дочерних узлов `branch`. Как только в узле `branch` появляется экземпляр структуры `Node`, мы можем модифицировать узел `leaf`, чтобы дать ему ссылку `Weak<Node>` на его родителя ❹. Мы используем метод `loan_mut` для

`RefCell<Weak<Node>>` в поле `parent` узла `leaf`, а затем функцию `Rc::downgrade` для создания ссылки `Weak<Node>` на узел `branch` из умного указателя `Rc<Node>` в узле `branch`.

Когда мы снова выведем родителя узла `leaf` ❺, то на этот раз получим вариант `Some`, содержащий узел `branch`: теперь узел `leaf` может обратиться к своему родителю! Когда мы печатаем узел `leaf`, мы также избегаем цикла, который в конечном итоге закончился бы переполнением стека, как это было в листинге 15.26. Ссылки `Weak<Node>` печатаются как `(Weak)`:

```
leaf parent = Some(Node { value: 5, parent: RefCell { value: (Weak) },
children: RefCell { value: [Node { value: 3, parent: RefCell { value: (Weak)
}, children: RefCell { value: [] } }] } })
```

Отсутствие бесконечного вывода данных показывает, что код не создал цикла в переходах по ссылкам. Это подтверждается и значениями, которые мы получаем из вызовов функции — `Rc::strong_count` и `Rc::weak_count`.

Визуализация изменений, вносимых в strong_count и weak_count

Давайте посмотрим, как изменяются значения `strong_count` и `weak_count` экземпляров `Rc<Node>`, создав новую внутреннюю область видимости и переместив узел `branch` в эту область. Сделав это, мы увидим, что происходит, когда узел `branch` создается, а затем отбрасывается, выходя из области видимости. Эти изменения показаны в листинге 15.29.

Листинг 15.29. Создание узла branch во внутренней области видимости и проверка числа сильных и слабых ссылок

src/main.rs

```
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

❶ println!(
        "сильные ссылки leaf = {}, слабые ссылки = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );

❷ {
        let branch = Rc::new(Node {
            value: 5,
            parent: RefCell::new(Weak::new()),
            children: RefCell::new(vec![Rc::clone(&leaf)]),
        });
```

```
        *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

    ❸ println!(
            "сильные ссылки branch = {}, слабые ссылки = {}",
            Rc::strong_count(&branch),
            Rc::weak_count(&branch),
        );

    ❹ println!(
            "сильные ссылки leaf = {}, слабые ссылки = {}",
            Rc::strong_count(&leaf),
            Rc::weak_count(&leaf),
        );
❺ }

❻ println!("родительский узел leaf = {:?}", leaf.parent.borrow().upgrade());
❼ println!(
        "сильные ссылки leaf = {}, слабые ссылки = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );
}
```

После того как узел `leaf` создан, его умный указатель `Rc<Node>` имеет сильное число 1 и слабое число 0 ❶. Во внутренней области ❷ мы создаем узел `branch` и связываем его с узлом `leaf`, и в момент, когда мы выведем числа ❸, умный указатель `Rc<Node>` в узле `branch` будет иметь сильное число 1 и слабое число 1 (для `leaf.parent`, указывающей на `branch` с `Weak<Node>`). Когда мы выведем числа в узле `leaf` ❹, мы увидим, что у него будет сильное число 2, потому что узел `branch` теперь имеет клон умного указателя `Rc<Node>` узла `leaf`, хранящегося в `branch.children`, но по-прежнему будет иметь слабое число 0.

Когда внутренняя область видимости заканчивается ❺, узел `branch` выходит из области и сильное число умного указателя `Rc<Node>` уменьшается до 0, поэтому его экземпляр структуры `Node` отбрасывается. Слабое число 1 из `leaf.parent` не имеет отношения к тому, будет ли этот `Node` отброшен, поэтому нет никаких утечек памяти!

Если мы попытаемся обратиться к родительскому узлу `leaf` после окончания области видимости, то мы снова получим `None` ❻. В конце программы ❼ умный указатель `Rc<Node>` в узле `leaf` имеет сильное число 1 и слабое число 0, потому что узел `leaf` теперь снова является единственной ссылкой на `Rc<Node>`.

Весь алгоритм, управляющий подсчетами и отбрасыванием значений, встроен в указатели `Rc<T>` и `Weak<T>`, а также в их реализации типажа `Drop`. Описав в определении структуры `Node`, что связь между дочерним узлом и его родителем должна быть ссылкой `Weak<T>`, вы можете сделать так, чтобы родительские узлы указывали на дочерние и наоборот, не создавая цикла в переходах по ссылкам и утечек памяти.

Итоги

В этой главе мы рассмотрели, как использовать умные указатели для создания различных преимуществ, отличных от тех, которые язык Rust предоставляет по умолчанию вместе с регулярными ссылками. Тип `Box<T>` имеет известный размер и указывает на данные, выделенные в куче. Тип `Rc<T>` отслеживает число ссылок на данные в куче, благодаря чему данные могут иметь несколько владельцев. `RefCell<T>` с его внутренней изменяемостью дает нам тип, который можно использовать, когда нужен неизменяемый тип, но требуется изменить его внутреннее значение. Он также обеспечивает соблюдение правил заимствования во время выполнения, а не во время компиляции.

Мы также обсудили типажи `Deref` и `Drop`, которые обеспечивают функциональность умных указателей. Мы изучили проблему циклов в переходах по ссылкам, которые служат причиной утечки памяти, и познакомились со способами ее предотвращения при помощи ссылки `Weak<T>`.

Если эта глава вызвала у вас интерес и вы хотите реализовать собственные умные указатели, ознакомьтесь с «Растономиконом» https://doc.rust-lang.org/stable/nomicon/ для получения дополнительной информации.

Далее мы поговорим о параллелизме в языке Rust. Вы также узнаете о новых умных указателях.

16 Конкурентность без страха

Безопасная и эффективная работа с конкурентностью в программировании — это одна из главных целей языка Rust. *Конкурентное программирование*, когда разные части программы исполняются независимо, и *параллельное программирование*, когда разные части программы исполняются одновременно, становятся все более важными, поскольку все больше компьютеров задействуют преимущества своих многочисленных процессоров. Исторически заниматься программированием в этих контекстах было сложно и допускалось много ошибок. Разработчики языка Rust надеются это изменить.

Изначально создатели Rust считали, что обеспечение безопасности памяти и предотвращение проблем конкурентности — это отдельные задачи, которые должны решаться по-разному. Со временем они обнаружили, что владение и система типов представляют собой мощный набор инструментов для управления безопасностью памяти и проблемами конкурентности. Благодаря проверке владения и типов многие ошибки конкурентности становятся ошибками времени компиляции, а не времени выполнения. Следовательно, вам не придется тратить время на попытки воспроизвести точные обстоятельства, при которых дефект параллелизма возникает во время выполнения, — неправильный код сам откажется компилироваться и представит ошибку, объясняющую проблему. В результате этого вы можете исправлять код во время работы над ним, а не после его отправки в производство. Мы назвали этот аспект языка Rust «конкурентность без страха» или «безбоязненная конкурентность». Благодаря безбоязненной конкурентности можно писать код, свободный от скрытых ошибок, который легко подвергается рефакторингу.

ПРИМЕЧАНИЕ

Для простоты мы будем называть большинство задач конкурентными, а не конкурентными и/или параллельными, как было бы точнее. Если бы эта книга была о конкурентности и/или параллелизме, то мы бы выражались конкретнее. В этой главе, когда мы используем термин «конкурентный», просим мысленно добавлять «конкурентный и/или параллельный».

Многие языки строги к техническим решениям, которые они предлагают для работы с конкурентными задачами. Например, у Erlang есть элегантная функ-

циональность для конкурентной передачи сообщений, но невнятные способы совместного использования состояния между потоками. Для языков высокого уровня поддерживать только подмножества возможных решений разумно, поскольку в таких языках можно получить преимущества, если отказаться от некоторого контроля в пользу абстракций. Однако предполагается, что языки низкого уровня должны обеспечивать решение с наилучшей производительностью в любой ситуации и иметь меньше абстракций по сравнению с аппаратным обеспечением. Следовательно, язык Rust предлагает разнообразие инструментов для моделирования задач любым способом, подходящим для вашей ситуации и технических требований.

Вот темы, которые мы рассмотрим в этой главе:

- Как создавать потоки, которые выполняют многочисленные части кода одновременно.
- Конкурентность передачи сообщений, при которой каналы посылают сообщения между потоками.
- Конкурентность совместного состояния, когда несколько потоков исполнения имеют доступ к некоторой части данных.
- Типажи `Sync` и `Send`, которые расширяют гарантии конкурентности языка Rust для типов, определяемых пользователем, а также типов, предусмотренных стандартной библиотекой.

Использование потоков исполнения для одновременного выполнения кода

В большинстве современных операционных систем код работающей программы исполняется в процессе, а операционная система управляет несколькими процессами сразу. Внутри программы вы также можете иметь независимые части, которые работают одновременно. Средства, которые управляют этими независимыми частями, называются потоками (*thread*).

Разделение вычислений в программе на несколько потоков исполнения повышает производительность, поскольку программа работает с несколькими задачами в одно и то же время, но это также добавляет сложности. Поскольку потоки работают одновременно, нет никакой гарантии порядка, в котором части кода будут выполняться в разных потоках. Это приводит к таким проблемам, как:

- Состояние гонки, когда потоки обращаются к данным или ресурсам в несогласованном порядке.
- Взаимоблокировки, при которых поток ждет, когда другой поток закончит использование ресурса, принадлежавшего первому. В итоге это препятствует продолжению работы обоих потоков.

- Дефекты, возможные только в некоторых ситуациях, которые трудно воспроизвести и устранить.

Rust пытается смягчить негативные последствия использования потоков. Но чтобы программировать в многопоточном контексте, нужно тщательно все обдумать и разработать структуру кода, отличную от структуры кода в программах, работающих в одном потоке.

Языки программирования реализуют потоки несколькими способами. Многие ОС предусматривают API для создания новых потоков. Модель, в которой язык вызывает API для создания потоков, иногда называют 1:1, имея в виду один поток ОС на один поток языка.

Многие языки программирования предоставляют специальные реализации потоков. Потоки, предоставляемые языком программирования, называют также зелеными. Языки, использующие зеленые потоки, будут выполнять их в контексте другого числа потоков ОС. По этой причине модель с зеленым потоком называется моделью *M:N*:, существует *M* зеленых потоков на *N* потоков ОС, где *M* и *N* — не обязательно одно и то же число.

Каждая модель имеет свои преимущества и компромиссы, и самый важный компромисс для Rust — это поддержка времени выполнения. Термин «время выполнения» (*runtime*) запутанный и в разных контекстах может иметь разные значения[1].

В данном контексте под временем выполнения мы подразумеваем код, который включается языком в каждый двоичный файл. Этот код может быть крупным или малым в зависимости от языка, но каждый язык, не являющийся ассемблером, будет иметь некий объем кода времени выполнения. Когда говорят, что у языка «нет времени выполнения», часто имеют в виду «малый объем кода времени выполнения». Меньшие объемы кода времени выполнения имеют меньшее число функциональных средств, но у них есть преимущество: в результате получаются меньшие по объему двоичные файлы, которые облегчают объединение языка с другими языками в большем количестве контекстов. Хотя многие языки устроены так, что в них увеличено время выполнения в обмен на дополнительные функциональные средства, в Rust почти не должно быть времени выполнения, и в нем не может быть компромисса с возможностью вызова C для поддержания производительности.

Модель обработки на основе зеленых потоков *M:N* требует более крупной языковой среды времени выполнения, чтобы управлять потоками. По этой причине стандартная библиотека Rust предусматривает реализацию только потоков 1:1. Ввиду того что язык Rust является весьма низкоуровневым языком, существуют упаковки, которые реализуют поточную обработку *M:N*, если вам нужен больший контроль того, какие потоки и когда выполняются и как снизить трудоемкость переключения контекстов.

---

[1] В контексте данного раздела термин «runtime» скорее обозначает *среду* времени выполнения.

Теперь, когда мы дали определение понятию потока в языке Rust, давайте узнаем, как использовать предусмотренный стандартной библиотекой API, связанный с потоками исполнения.

Создание нового потока с помощью spawn

Для того чтобы создать новый поток, мы вызываем функцию `thread::spawn` и передаем ей замыкание (мы говорили о замыканиях в главе 13), содержащее код, который мы хотим выполнить в новом потоке исполнения. Пример из листинга 16.1 выводит некий текст из главного потока исполнения и другой текст из нового потока.

Листинг 16.1. Создание нового потока для вывода чего-либо, в то время как главный поток выводит что-то другое

src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("привет, число {} из порожденного потока!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("привет, число {} из главного потока!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

Обратите внимание, что с помощью этой функции новый поток остановится, когда закончится главный, независимо от того, выполнился он или нет. Данные этой программы могут каждый раз немного отличаться, но они будут выглядеть примерно так:

```
привет, число 1 из главного потока!
привет, число 1 из порожденного потока!
привет, число 2 из главного потока!
привет, число 2 из порожденного потока!
привет, число 3 из главного потока!
привет, число 3 из порожденного потока!
привет, число 4 из главного потока!
привет, число 4 из порожденного потока!
привет, число 5 из порожденного потока!
```

Вызовы функции `thread::sleep` заставляют поток останавливать свое исполнение на короткое время, позволяя другому потоку работать. Потоки, вероятно, будут чередоваться, но это не гарантировано: это зависит от того, как ваша операцион-

ная система планирует потоки. В этом прогоне главный поток исполнения выводится первым, даже если инструкция вывода из порожденного потока появляется в коде первой. И хотя перед порожденным потоком стояла задача выводиться до тех пор, пока переменная `i` равна 9, он добрался только до 5, прежде чем главный поток исполнения закрылся.

Если, выполняя этот код, вы видите данные только из главного потока или не видите никаких данных других потоков, попробуйте увеличить числа в интервалах, чтобы создать больше возможностей для переключения операционной системы между потоками.

Ожидание завершения работы всех потоков с использованием дескрипторов join

Код в листинге 16.1 не только чаще всего преждевременно останавливает порожденный поток из-за окончания работы главного потока, но и не может гарантировать, что порожденный поток вообще будет запущен. Причина в том, что нет гарантии, в каком порядке потоки будут выполняться!

Мы можем решить проблему незапуска или неполной работы порожденного потока исполнения, сохранив значение, возвращаемое из функции `thread::spawn`, в переменной. Функция `thread::spawn` возвращает тип `JoinHandle`. Тип `JoinHandle` — это обладаемое значение, которое, когда мы вызываем для него метод `join`, будет ждать до тех пор, пока его поток не завершится. Листинг 16.2 показывает, как использовать дескриптор `JoinHandle` потока, который мы создали в листинге 16.1, и как вызывать `join`, чтобы порожденный поток завершался до выхода `main`.

Листинг 16.2. Сохранение дескриптора JoinHandle из функции thread::spawn, чтобы поток завершился полностью

src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("привет, число {} из порожденного потока!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("привет, число {} из главного потока!", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```

Вызов `join` для дескриптора блокирует поток, работающий в настоящее время, до тех пор, пока поток дескриптора не завершится. Блокирование потока означает, что поток не может выполнять работу или выходить. Поскольку мы поместили вызов `join` после цикла `for` главного потока, выполнение листинга 16.2 должно привести к результату, аналогичному тому, который приведен ниже:

```
привет, число 1 из главного потока!
привет, число 2 из главного потока!
привет, число 1 из порожденного потока!
привет, число 3 из главного потока!
привет, число 2 из порожденного потока!
привет, число 4 из главного потока!
привет, число 3 из порожденного потока!
привет, число 4 из порожденного потока!
привет, число 5 из порожденного потока!
привет, число 6 из порожденного потока!
привет, число 7 из порожденного потока!
привет, число 8 из порожденного потока!
привет, число 9 из порожденного потока!
```

Оба потока продолжают чередоваться, но главный поток ждет из-за вызова `handle.join()` и не завершается до тех пор, пока не завершен порожденный поток.

Но давайте посмотрим, что произойдет, когда мы переместим вызов `handle.join()` и поставим его перед циклом `for` в функции `main`, как показано ниже:

src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("привет, число {} из порожденного потока!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("привет, число {} из главного потока!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

Главный поток будет ждать завершения порожденного потока, а затем выполнит свой цикл `for`, поэтому результаты больше не будут чередоваться, как показано ниже:

```
привет, число 1 из порожденного потока!
привет, число 2 из порожденного потока!
привет, число 3 из порожденного потока!
привет, число 4 из порожденного потока!
привет, число 5 из порожденного потока!
привет, число 6 из порожденного потока!
привет, число 7 из порожденного потока!
привет, число 8 из порожденного потока!
привет, число 9 из порожденного потока!
привет, число 1 из главного потока!
привет, число 2 из главного потока!
привет, число 3 из главного потока!
привет, число 4 из главного потока!
```

Небольшие детали, такие как место вызова `join`, могут повлиять на то, будут ли потоки выполняться в одно и то же время.

Использование замыкания move с потоками

Замыкание `move` часто используется вместе с функцией `thread::spawn`, поскольку позволяет применять данные из одного потока в другой.

В главе 13 мы упоминали о том, что можно использовать ключевое слово `move` перед списком параметров замыкания, чтобы сделать замыкание владельцем значений, которые оно использует в среде. Этот технический прием особенно полезен при создании новых потоков исполнения, чтобы передавать владение значениями из одного потока в другой.

Обратите внимание, что в листинге 16.1 замыкание, которое мы передаем в функцию `thread::spawn`, не берет никаких аргументов: мы не используем данные из главного потока исполнения в коде порожденного потока. Для того чтобы использовать данные из главного потока в порожденный, замыкание порожденного потока должно захватывать необходимые ему значения. Листинг 16.3 показывает попытку создать вектор в главном потоке исполнения и использовать его в порожденном потоке. Но как вы вскоре увидите, это пока не работает.

Листинг 16.3. Попытка использовать вектор, созданный главным потоком, в другом потоке

src/main.rs

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Вот вектор: {:?}", v);
    });

    handle.join().unwrap();
}
```

Замыкание использует переменную v, поэтому оно захватит v и сделает ее частью среды замыкания. Поскольку функция `thread::spawn` выполняет это замыкание в новом потоке исполнения, мы должны иметь возможность обращаться к переменной v внутри этого нового потока. Но когда мы компилируем этот пример, возникает следующая ошибка[1]:

```
error[E0373]: closure may outlive the current function, but it borrows `v`,
which is owned by the current function
 --> src/main.rs:6:32
  |
6 |     let handle = thread::spawn(|| {
  |                                ^^ may outlive borrowed value `v`
7 |     println!("Вот вектор: {:?}", v);
  |                                    - `v` is borrowed here
  |
help: to force the closure to take ownership of `v` (and any other referenced
  variables), use the `move` keyword
  |
6 |     let handle = thread::spawn(move || {
  |                                ^^^^^^^
```

Rust логически выводит то, как следует захватывать переменную v, а поскольку макрокоманде `println!` требуется только ссылка на v, замыкание пытается заимствовать v. Однако существует одна проблема: нельзя сказать, как долго будет работать порожденный поток, поэтому неизвестно, будет ли ссылка на v всегда действительной.

Листинг 16.4 содержит сценарий, в котором ссылка на переменную v, скорее всего, не будет действительной.

Листинг 16.4. Поток с замыканием, пытающийся захватить ссылку на переменную v из главного потока, который отбрасывает переменную v

src/main.rs

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Вот вектор: {:?}", v);
    });

    drop(v); // о нет!

    handle.join().unwrap();
}
```

[1] ошибка[E0373]: замыкание может пережить текущую функцию, но оно заимствует переменную `v`, которая принадлежит текущей функции

Если бы нам было разрешено выполнить этот код, то, вероятно, порожденный поток был бы немедленно помещен на задний план вообще без выполнения. Порожденный поток имеет ссылку на переменную `v` внутри, но главный поток исполнения немедленно отбрасывает `v`, используя метод `drop`, который мы обсуждали в главе 15. Затем, когда порожденный поток начинает выполняться, переменная `v` больше недействительна, поэтому ссылка на нее тоже недействительна. О нет!

Для того чтобы исправить ошибку компилятора в листинге 16.3, мы можем воспользоваться советом, содержащимся в сообщении об ошибке[1]:

```
help: to force the closure to take ownership of `v` (and any other referenced
variables), use the `move` keyword
  |
6 |     let handle = thread::spawn(move || {
  |                                ^^^^^^^
```

Добавляя ключевое слово `move` перед замыканием, мы делаем замыкание владельцем значений, которые оно использует, то есть языку Rust не нужно выводить логически, что требуется заимствовать эти значения. Модификация листинга 16.3, показанная в листинге 16.5, будет компилироваться и выполняться так, как мы хотим.

Листинг 16.5. Ключевое слово move делает замыкание владельцем значений, которые оно использует

src/main.rs

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Вот вектор: {:?}", v);
    });

    handle.join().unwrap();
}
```

Что произошло бы с кодом в листинге 16.4, где главный поток исполнения вызывал `drop`, если бы мы использовали замыкание с ключевым словом `move`? Может ли `move` исправить это дело? К сожалению, нет. Возникнет другая ошибка, поскольку то, что пытается сделать листинг 16.4, не допускается по другой причине. Если мы добавим ключевое слово `move` в замыкание, то переместим переменную `v` в среду замыкания и больше не сможем вызывать для него `drop` в главный поток исполнения. Вместо этого мы получим такую ошибку компилятора[2]:

---

[1] `справка: для того чтобы заставить замыкание стать владельцем переменной` `v` `(и любых других ссылочных переменных), используйте ключевое слово` `move`

[2] `ошибка[E0382]: использование перемещенного значения` `v`

```
error[E0382]: use of moved value: `v`
  --> src/main.rs:10:10
   |
6  |     let handle = thread::spawn(move || {
   |                                ------- value moved (into closure) here
...
10 |     drop(v); // о нет!
   |          ^ value used here after move
   |
   = note: move occurs because `v` has type `std::vec::Vec<i32>`, which does
    not implement the `Copy` trait
```

Правила владения снова нас спасли! Мы получили ошибку из кода в листинге 16.3, потому что Rust был строг и позаимствовал переменную `v` только для этого потока, то есть главный поток исполнения теоретически мог аннулировать ссылку порожденного потока. Говоря компилятору переместить владение переменной `v` в порожденный поток, мы гарантируем, что главный поток больше не будет использовать `v`. Если мы изменяем листинг 16.4 таким же образом, то нарушаем правила владения, когда пытаемся использовать `v` в главном потоке. Ключевое слово `move` отменяет консервативное поведение заимствования, присущее языку Rust по умолчанию, поэтому мы не нарушаем правила владения.

Изучив основы потоков исполнения и их API, давайте посмотрим, что можно с ними делать.

Использование передачи сообщений для пересылки данных между потоками

Для безопасной конкурентности все популярнее становится передача сообщений, когда потоки или акторы общаются, отправляя друг другу сообщения, содержащие данные. Вот как эта идея выражена в слогане из документации языка Go (http://golang.org/doc/effective_go.html): «Не общайтесь, используя одну и ту же память. Делитесь памятью, общаясь».

Один из основных инструментов языка Rust, предназначенный для достижения параллелизма отправки сообщений, — это понятие под названием «канал», реализация которого предусматривается стандартной библиотекой. Вы можете представить себе канал в программировании как канал с водой, такой как ручей или река. Если вы положите что-то вроде резиновой утки или лодки в поток воды, то она будет двигаться вниз по течению до конца водного пути.

Канал в программировании состоит из двух элементов: передатчика и приемника. Передатчик — это место выше по течению, где вы кладете резиновых уток в реку, а приемник — это место, где резиновая утка заканчивает движение вниз по течению. Одна часть кода вызывает методы для передатчика с данными, которые вы хотите отправить, а другая — проверяет принимающий конец на наличие посту-

пающих сообщений. Канал считается закрытым, если отброшена половина либо передатчика, либо приемника.

Здесь мы поработаем над программой, которая имеет один поток для генерирования значений и отправки их по каналу, а другой поток для получения значений и их вывода. Чтобы показать, как это работает, будем отправлять простые значения между потоками, используя канал. Как только вы ознакомитесь с этим техническим решением, то сможете использовать каналы для реализации чат-системы или системы, в которой много потоков исполнения выполняют части вычисления и отправляют их в один поток, который агрегирует результаты.

Сначала в листинге 16.6 мы создадим канал, но ничего с ним делать не будем. Обратите внимание, что этот код пока не компилируется, потому что компилятору не известно, какой тип значений мы хотим отправить по каналу.

Листинг 16.6. Создание канала и присвоение значений `tx` и `rx` его элементам

src/main. rs

```
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();
}
```

Мы создаем новый канал, используя функцию `mpsc::channel`, `mpsc` расшифровывается как «много производителей, один потребитель» от англ. *multiple producer, single consumer*. Одним словом, то, как стандартная библиотека языка Rust реализует каналы, означает, что канал может иметь несколько отправляющих концов, которые производят значения, и только один принимающий конец, который потребляет эти значения. Представьте себе, что несколько потоков сливаются в одну большую реку: все, что отправлено вниз по любому из потоков, в конце концов окажется в одной реке. Сейчас мы начнем с одного производителя, но, когда этот пример будет работать, мы добавим несколько производителей.

Функция `mpsc::channel` возвращает кортеж, первый элемент которого является отправляющим, а второй — принимающим. Аббревиатуры `tx` и `rx` традиционно используются во многих областях для обозначения передатчика и приемника соответственно, поэтому мы и называем так переменные, чтобы указать каждый конец. Мы используем инструкцию `let` с паттерном, который деструктурирует кортежи. Мы обсудим использование паттернов в инструкциях `let` и деструктурирование в главе 18. Используя инструкцию `let` таким образом, вы можете легко извлекать части кортежа, возвращаемого функцией `mpsc::channel`.

Давайте переместим передающий конец в порожденный поток и дадим ему отправить одно строковое значение, чтобы порожденный поток общался с главным потоком, как показано в листинге 16.7. Все равно, что положить резиновую утку в реку в верхнем течении или отправить сообщение в чате из одной ветви в другую.

Листинг 16.7. Перемещение переменной tx в порожденный поток и отправка сообщения «привет»

src/main.rs

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("привет");
        tx.send(val).unwrap();
    });
}
```

Опять же мы используем функцию `thread::spawn`, чтобы создать новый поток, а затем применяем `move` для перемещения переменной `tx` в замыкание, чтобы порожденный поток стал владельцем переменной `tx`. Порожденный поток должен владеть передающим концом канала, чтобы иметь возможность отправлять сообщения.

Передающий конец имеет метод `send`, берущий значение, которое мы хотим отправить. Метод `send` возвращает тип `Result<T, E>`, поэтому, если принимающий конец уже отброшен и послать значение некуда, то операция `send` будет возвращать ошибку. В данном примере мы вызываем метод `unwrap`, чтобы поднимать панику в случае ошибки. Но в реальном приложении мы бы обрабатывали это надлежащим образом. Вернитесь к главе 9, чтобы освежить в памяти стратегии надлежащей обработки ошибок.

В листинге 16.8 мы получим значение от принимающего конца канала в главный поток исполнения.

Листинг 16.8. Получение значения «привет» в главном потоке исполнения и его вывод

src/main.rs

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("привет");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Получено: {}", received);
}
```

Принимающий конец канала имеет два полезных метода: `recv` и `try_recv`. Мы используем метод `recv`, сокращенно от англ. *receive*, который блокирует выполнение главного потока исполнения и ждет до тех пор, пока значение не будет отправлено вниз по каналу. Как только значение будет отправлено, метод `recv` вернет его в экземпляре типа `Result<T, E>`. Когда передающий конец канала закроется, метод `recv` вернет ошибку, сигнализирующую о том, что больше никаких значений не будет.

Метод `try_recv` не блокирует, а вместо этого сразу возвращает экземпляр типа `Result<T, E>`: значение `Ok`, содержащее сообщение, если оно имеется, и значение `Err`, если на этот раз никаких сообщений нет. Метод `try_recv` полезен, если у потока во время ожидания сообщений есть другие задачи. Мы могли бы написать цикл, который регулярно вызывает метод `try_recv`, обрабатывает сообщение, если оно имеется, а в противном случае некоторое время выполняет другую работу до тех пор, пока не проверит снова.

В данном примере мы использовали метод `recv`, поскольку он простой; главный поток должен только ожидать сообщения, поэтому блокировка главного потока исполнения уместна.

Когда мы выполним код из листинга 16.8, то увидим значение, выведенное из главного потока исполнения:

```
Получено: привет
```

Идеально!

Каналы и передача владения

Правила владения играют важную роль в отправке сообщений, поскольку помогают писать безопасный конкурентный код. Благодаря владению во всех программах на языке Rust у вас есть преимущество — вы можете предотвращать ошибки в конкурентном программировании. Давайте проведем эксперимент, чтобы показать, как каналы и владение работают вместе и предотвращают проблемы. Мы попробуем использовать значение переменной `val` в порожденном потоке исполнения после того, как отправили его по каналу. Попробуйте скомпилировать код из листинга 16.9, чтобы увидеть, почему этот код не выполняется.

Листинг 16.9. Попытка использовать переменную `val` после того, как мы отправили ее по каналу

src/main.rs

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
```

```
        let val = String::from("привет");
        tx.send(val).unwrap();
        println!("val равна {}", val);
    });

    let received = rx.recv().unwrap();
    println!("Получено: {}", received);
}
```

Здесь мы пытаемся напечатать переменную `val` после того, как отправили ее по каналу через `tx.send`. Разрешить это — плохая идея: как только значение отправлено в другой поток, она может его изменить или отбросить, прежде чем мы попытаемся использовать значение снова. Модификации другого потока потенциально могут приводить к ошибкам или неожиданным результатам из-за несогласованных или несуществующих данных. Однако компилятор выдает ошибку, если мы пытаемся скомпилировать код в листинге 16.9.

```
error[E0382]: use of moved value: `val`
  --> src/main.rs:10:31
   |
9  |     tx.send(val).unwrap();
   |             --- value moved here
10 |     println!("val is {}", val);
   |                           ^^^ value used here after move
   |
   = note: move occurs because `val` has type `std::string::String`, which
   does not implement the `Copy` trait
```

Неточность конкурентности стала причиной ошибки времени компиляции. Функция `send` берет свой параметр во владение, а когда значение перемещается, получатель становится его владельцем. Благодаря этому мы не можем случайно использовать значение снова после его отправки. Система владения проверяет, что все в порядке.

Отправка нескольких значений и ожидание приемника

Код в листинге 16.8 скомпилирован и выполнен, но он не совсем ясно показал, что два отдельных потока разговаривают друг с другом по каналу. В листинг 16.10 мы внесли несколько модификаций, которые докажут, что код в листинге 16.8 выполняется параллельно: порожденный поток теперь будет отправлять несколько сообщений и делать секундную паузу между каждым сообщением.

Листинг 16.10. Отправка нескольких сообщений и пауза между ними

src/main.rs

```
use std::thread;
use std::sync::mpsc;
use std::time::Duration;

fn main() {
```

```
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let vals = vec![
            String::from("привет"),
            String::from("из"),
            String::from("потока"),
            String::from("исполнения"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Получено: {}", received);
    }
}
```

На этот раз порожденный поток имеет вектор строковых значений, которые мы хотим отправить в главный поток. Мы перебираем их, посылая каждое по отдельности, и делаем паузу между ними, вызывая функцию `thread::sleep` со значением `Duration`, равным 1 секунде.

В главном потоке исполнения мы больше не вызываем функцию `recv` явно: вместо этого мы трактуем `rx` как итератор. Мы выводим каждое полученное значение. Когда канал будет закрыт, итерация закончится.

При выполнении этого кода из листинга 16.10 вы должны увидеть следующие данные с секундной паузой между каждой строкой результата:

```
Получено: привет
Получено: из
Получено: потока
Получено: исполнения
```

Поскольку у нас нет кода, который приостанавливает или задерживает цикл `for` в главном потоке исполнения, можно сказать, что главный поток ожидает значения из порожденного потока.

Создание нескольких производителей путем клонирования передатчика

Ранее мы уже упоминали, что `mpsc` — это сокращение для обозначения нескольких производителей и одного потребителя. Давайте применим `mpsc` и расширим код из листинга 16.10 так, чтобы создать несколько потоков исполнения, которые посылают значения одному и тому же приемнику. Мы можем сделать это, клонировав передающую половину канала, как показано в листинге 16.11.

Листинг 16.11. Отправка нескольких сообщений от нескольких производителей
src/main.rs

```
// --пропуск--

let (tx, rx) = mpsc::channel();

let tx1 = mpsc::Sender::clone(&tx);
thread::spawn(move || {
    let vals = vec![
        String::from("привет"),
        String::from("из"),
        String::from("потока"),
        String::from("исполнения"),
    ];

    for val in vals {
        tx1.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

thread::spawn(move || {
    let vals = vec![
        String::from("еще"),
        String::from("сообщения"),
        String::from("для"),
        String::from("вас"),
    ];

    for val in vals {
        tx.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

for received in rx {
    println!("Получено: {}", received);
}

// --пропуск--
```

На этот раз, прежде чем мы создадим первый порожденный поток, мы вызываем `clone` на отправляющем конце канала. Это дает новый отправляющий дескриптор, который мы можем передать первому порожденному потоку исполнения. Мы передаем исходный отправляющий конец канала второму порожденному потоку исполнения. Это дает нам два потока, каждый из которых отправляет разные сообщения в принимающий конец канала.

Когда вы выполните этот код, данные должны выглядеть примерно так:

```
Получено: привет
Получено: еще
Получено: от
```

```
Получено: сообщения
Получено: для
Получено: потока
Получено: исполнения
Получено: вас
```

Возможно, вы увидите значения в другом порядке — результат зависит от вашей системы. Именно это делает конкурентность интересной и одновременно сложной. Если вы поэкспериментируете с функцией `thread::sleep`, давая ей разные значения в разных потоках, то каждый прогон будет менее детерминированным и каждый раз будет создавать разные данные на выходе.

Теперь, познакомившись с работой каналов, давайте посмотрим на другой метод конкурентности.

Конкурентность совместного состояния

Передача сообщений — это прекрасный способ конкурентности, но не единственный. Еще раз обратите внимание на часть слогана из документации языка Go: «Делитесь памятью, общаясь».

Как будет выглядеть общение, при котором делятся памятью? Вдобавок, почему энтузиасты передачи сообщений так не делают, а поступают наоборот?

В некотором смысле каналы в любом языке программирования подобны одинарному владению, потому что после пересылки значения по каналу вы больше не должны использовать это значение. Конкурентность совместной памяти подобна множественному владению: несколько потоков исполнения могут одновременно обращаться к одному и тому же месту в памяти. Как вы видели в главе 15, множественное владение стало возможным благодаря умным указателям. Множественное владение добавляет сложность, потому что разные владельцы нуждаются в управлении. Система типов и правила владения языка Rust значительно обеспечивают надлежащее управление. Для примера давайте рассмотрим мьютексы, один из наиболее часто встречающихся примитивов конкурентности для совместной памяти.

Использование мьютексов для обеспечения доступа к данным из одного потока за один раз

Мьютекс — это сокращение английского термина *mutual exclusion*, «взаимное исключение», то есть мьютекс позволяет обращаться к неким данным в любой момент времени только одному потоку. Для того чтобы обратиться к данным в мьютексе, поток должен сначала подать сигнал о том, что он хочет получить доступ, сделав запрос на получение замка мьютекса. Замок — это структура данных, часть мьютекса, которая отслеживает, кто в настоящее время имеет исключительный

доступ к данным. Следовательно, мьютекс охраняет данные, которые он содержит, посредством системы замков.

Считается, что мьютексы трудно использовать, поэтому важно помнить два правила:

- Вы должны получить замок перед использованием данных.
- Когда вы закончите работу с данными, которые мьютекс охраняет, вы должны снять замок с данных, чтобы другие потоки могли получить этот замок.

В качестве реального примера мьютекса представьте себе коллективное обсуждение на конференции с одним микрофоном. Прежде чем кто-либо из участников дискуссии начнет свое выступление, он должен попросить или подать сигнал, что хочет воспользоваться микрофоном. Когда он получает микрофон, то может говорить сколько угодно, а затем передать микрофон следующему участнику, который просит слова. Если участник дискуссии забудет отдать микрофон, когда закончит свою речь, то никто больше не сможет выступить. Если работа с совместным микрофоном не заладится, то коллективное обсуждение пройдет не так, как планировалось!

Разобраться в управлении мьютексами бывает невероятно сложно, именно поэтому так много людей с энтузиазмом относятся к каналам. Тем не менее благодаря системе типов и правилам владения Rust вы точно разберетесь с установкой и снятием замков.

API Mutex<T>

В качестве примера давайте начнем с использования мьютекса в однопоточном контексте, как показано в листинге 16.12.

Листинг 16.12. Знакомство с API Mutex<T> в однопоточном контексте в качестве простого примера

src/main.rs

```
use std::sync::Mutex;

fn main() {
  ❶ let m = Mutex::new(5);

    {
      ❷ let mut num = m.lock().unwrap();
      ❸ *num = 6;
  ❹ }

  ❺ println!("m = {:?}", m);
}
```

Как и во многих других типах, мы создаем экземпляр типа `Mutex<T>`, используя связанную с ним функцию `new` ❶. Для того чтобы обратиться к данным внутри

мьютекса, мы используем метод `lock` для получения замка ❷. Его вызов заблокирует текущий поток, в результате чего он не сможет ничего сделать до тех пор, пока не наступит наша очередь блокировки.

Вызов метода `lock` не сработает, если еще один поток, удерживающий этот замок, поднимет панику. В этом случае никто не сможет получить этот замок, поэтому мы решили использовать метод `unwrap` и позволить потоку поднимать панику в подобной ситуации.

После получения замка мы можем трактовать возвращаемое значение, в данном случае именуемое `num`, как изменяемую ссылку на данные внутри ❸. Система типов отвечает за то, чтобы мы получали замок перед использованием значения в `m`: `Mutex<i32>` не является типом `i32`, поэтому мы *должны* получить замок, чтобы иметь возможность использовать значение типа `i32`. Мы не забудем это сделать, в противном случае система типов не позволит нам обратиться к внутреннему типу `i32`.

Как вы можете догадаться, тип `Mutex<T>` представляет собой умный указатель. Точнее, вызов метода `lock` возвращает умный указатель под названием `MutexGuard`. Этот умный указатель реализует типаж `Deref` для указания на внутренние данные. У этого умного указателя также есть реализация типажа `Drop`, которая автоматически освобождает замок, когда умный указатель `MutexGuard` выходит из области видимости, что и происходит в конце внутренней области ❹. В результате мы точно не забудем открыть замок и заблокировать мьютекс от использования другими потоками, потому что замок открывается автоматически.

После снятия блокировки мы выводим значение мьютекса и видим, что нам удалось изменить внутренний экземпляр типа `i32` на 6 ❺.

Совместное использование Mutex<T> несколькими потоками

Теперь давайте попробуем поделиться значением между несколькими потоками с помощью умного указателя `Mutex<T>`. Мы раскрутим 10 потоков, каждый из них будет увеличивать значение счетчика на 1, и поэтому счетчик движется от 0 до 10. Обратите внимание, что в следующих примерах будут ошибки компилятора. Мы будем использовать эти ошибки, чтобы узнать больше об умном указателе `Mutex<T>` и о том, как Rust помогает нам реализовать его правильно. Листинг 16.13 показывает первый пример.

Листинг 16.13. Десять потоков исполнения, каждый из которых увеличивает счетчик, охраняемый умным указателем Mutex<T>

src/main.rs

```
use std::sync::Mutex;
use std::thread;

fn main() {
 ❶ let counter = Mutex::new(0);
   let mut handles = vec![];
```

```
    ❷ for _ in 0..10 {
        ❸ let handle = thread::spawn(move || {
            ❹ let mut num = counter.lock().unwrap();

            ❺ *num += 1;
          });
        ❻ handles.push(handle);
      }

      for handle in handles {
        ❼ handle.join().unwrap();
      }

    ❽ println!("Результат: {}", *counter.lock().unwrap());
  }
```

Мы создаем переменную `counter` для хранения типа `i32` внутри `Mutex<T>` ❶, как это было сделано в листинге 16.12. Далее мы создаем 10 потоков исполнения, перебирая диапазон чисел ❷. Мы используем функцию `thread::spawn` и даем всем потокам одинаковое замыкание, которое перемещает переменную `counter` в поток ❸, получает замок для умного указателя `Mutex<T>`, вызывая метод `lock` ❹, а затем прибавляет 1 к значению в мьютексе ❺. Когда поток завершит выполнение своего замыкания, переменная `num` выйдет из области видимости и освободит замок, в результате чего его сможет получить еще один поток.

В главном потоке исполнения мы собираем все дескрипторы ❻, а затем, как и в листинге 16.2, для каждого из них вызываем `join`, тем самым обеспечивая завершение работы всех потоков ❼. В этот момент главный поток получит замок и выведет результат этой программы ❽.

Мы предупредили, что этот пример не будет компилироваться. А теперь давайте выясним почему![1]

```
error[E0382]: capture of moved value: `counter`
  --> src/main.rs:10:27
   |
9  |         let handle = thread::spawn(move || {
   |                                    ------- value moved (into closure) here
10 |             let mut num = counter.lock().unwrap();
   |                           ^^^^^^^ value captured here after move
   |
   = note: move occurs because `counter` has type `std::sync::Mutex<i32>`,
   which does not implement the `Copy` trait

error[E0382]: capture of moved value: `counter`
  --> src/main.rs:21:29
   |
```

[1] ошибка[E0382]: захват перемещенного значения: `counter`
ошибка[E0382]: использование перемещенного значения: `counter`
ошибка: прерывание работы из-за двух предыдущих ошибок

```
9  |         let handle = thread::spawn(move || {
   |                                    ------- value moved (into closure) here
...
21 |      println!("Result: {}", *counter.lock().unwrap());
   |                              ^^^^^^^ value used here after move
   |
   = note: move occurs because `counter` has type `std::sync::Mutex<i32>`,
   which does not implement the `Copy` trait

error: aborting due to 2 previous errors
```

В сообщении об ошибке говорится, что значение переменной `counter` перемещается в замыкание, а затем захватывается, когда мы вызываем `lock`. Мы этого хотели, но это запрещено!

Давайте выясним это, упростив программу. Вместо десяти потоков исполнения в цикле `for` давайте сделаем только два потока без цикла и посмотрим, что получится. Замените первый цикл `for` в листинге 16.13 следующим кодом.

```
let handle = thread::spawn(move || {
    let mut num = counter.lock().unwrap();

    *num += 1;
});
handles.push(handle);

let handle2 = thread::spawn(move || {
    let mut num2 = counter.lock().unwrap();

    *num2 += 1;
});
handles.push(handle2);
```

Мы делаем два потока и меняем имена переменных, используемых со вторым потоком, на `handle2` и `num2`. Когда мы выполняем код в этот раз, компиляция дает следующее:

```
error[E0382]: capture of moved value: `counter`
  --> src/main.rs:16:24
   |
8  |     let handle = thread::spawn(move || {
   |                                ------- value moved (into closure) here
...
16 |     let mut num2 = counter.lock().unwrap();
   |                    ^^^^^^^ value captured here after move
   |
   = note: move occurs because `counter` has type `std::sync::Mutex<i32>`,
   which does not implement the `Copy` trait

  error[E0382]: use of moved value: `counter`
  --> src/main.rs:26:29
   |
```

```
8  |     let handle = thread::spawn(move || {
   |                                ------- value moved (into closure) here
...
26 |     println!("Result: {}", *counter.lock().unwrap());
   |                             ^^^^^^^ value used here after move
   |
   = note: move occurs because `counter` has type `std::sync::Mutex<i32>`,
   which does not implement the `Copy` trait

error: aborting due to 2 previous errors
```

Ага! Первое сообщение об ошибке говорит, что переменная `counter` перемещается внутрь замыкания для потока, связанного с переменной `handle`. Это перемещение мешает захватить переменную `counter`, когда мы пытаемся вызвать для нее метод `lock` и сохранить результат в переменной `num2` во втором потоке! Поэтому Rust говорит нам о том, что мы не можем переместить владение переменной `counter` в несколько потоков. Раньше это было трудно увидеть, потому что наши потоки были в цикле, а Rust не может указывать на разные потоки в разных итерациях цикла. Давайте исправим ошибку компилятора с помощью метода множественного владения.

Множественное владение с несколькими нитями исполнения

В главе 15 мы передавали значение нескольким владельцам, используя умный указатель `Rc<T>`, чтобы создать значение с числом посчитанных ссылок. Давайте сделаем то же самое здесь и посмотрим, что произойдет. В листинге 16.14 мы обернем `Mutex<T>` в умный указатель `Rc<T>` и клонируем `Rc<T>` перед перемещением владения в поток. Теперь, когда мы увидели ошибки, мы также вернемся к циклу `for` и оставим ключевое слово `move` с замыканием.

Листинг 16.14. Попытка использовать умный указатель `Rc<T>`, чтобы разрешить нескольким потокам владеть умным указателем `Mutex<T>`

src/main.rs

```
use std::rc::Rc;
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Rc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Rc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
```

```
        }

        for handle in handles {
            handle.join().unwrap();
        }

        println!("Результат: {}", *counter.lock().unwrap());
    }
```

Мы еще раз компилируем и получаем... другие ошибки! Компилятор многому учит нас [1].

```
❶ error[E0277]: the trait bound `std::rc::Rc<std::sync::Mutex<i32>>:
   std::marker::Send` is not satisfied in `[closure@src/main.rs:11:36: 15:10
   counter:std::rc::Rc<std::sync::Mutex<i32>>]`
     --> src/main.rs:11:22
      |
   11 |             let handle = thread::spawn(move || {
   ❷ |                          ^^^^^^^^^^^^^ `std::rc::Rc<std::sync::Mutex<i32>>`
   cannot be sent between threads safely
      |
      = help: within `[closure@src/main.rs:11:36: 15:10 counter:std::rc::Rc<std::
   sync::Mutex<i32>>]`, the trait `std::marker::Send` is not implemented for `std
   ::rc::Rc<std::sync::Mutex<i32>>`
      = note: required because it appears within the type `[closure@src/main.
   rs:11:36: 15:10 counter:std::rc::Rc<std::sync::Mutex<i32>>]`
      = note: required by `std::thread::spawn`
```

Блеск! Это очень подробное сообщение об ошибке. Вот некоторые важные части, на которые следует обратить внимание: первая внутристрочная ошибка говорит о том, что не получается переслать мьютекс между потоками безопасным образом (`` `std::rc::Rc<std::sync::Mutex<i32>>` cannot be sent between threads safely``) ❷. Причина кроется в следующей важной части, требующей нашего внимания, — самом сообщении об ошибке. Там говорится о том, что граница типажа `'Send'` не удовлетворена (`trait bound `Send` is not satisfied`) ❶. Мы поговорим о `Send` в следующем разделе: это один из типажей, предназначенный для того, чтобы типы, используемые с потоками, применялись в конкурентных ситуациях.

К сожалению, умный указатель `Rc<T>` небезопасен для совместного использования между потоками. Когда `Rc<T>` управляет подсчетом ссылок, он прибавляет к числу при каждом вызове метода `clone` и вычитает из числа, когда каждый клон отбрасывается. Но он не использует никаких параллельных примитивов, чтобы изменения в подсчете не прерывались другим потоком. Это может привести к неправильным подсчетам ссылок — едва уловимым ошибкам, которые могут повлечь за собой утечку памяти или отбрасывание значения, прежде чем мы закончим работу. Нам нужен тип, подобный `Rc<T>`, который вносит изменения в подсчет ссылок безопасным для потоков способом.

---

[1] `ошибка[E0277]: типажная граница `std::rc::Rc<std::sync::Mutex<i32>>: std::marker::Send` не удовлетворена`

Атомарный подсчет ссылок с помощью Arc<T>

К счастью, тип `Arc<T>` подобен `Rc<T>` и безопасен для использования в конкурентных ситуациях. Буква «a» в имени типа расшифровывается как «атомарный», то есть это тип с атомарным подсчетом ссылок. Атомики (atomics) — это дополнительный вид конкурентных примитивов, который здесь не будет рассматриваться подробно: для получения дополнительной информации обратитесь к документации стандартной библиотеки по `std::sync::atomic`. Пока же вам просто нужно знать, что атомики работают как примитивные типы, но ими можно безопасно делиться между несколькими потоками.

Тогда вы можете задаться вопросом, почему все примитивные типы не атомарны и почему типы стандартной библиотеки не реализованы с `Arc<T>` по умолчанию. Причина заключается в том, что безопасность потоков затратна для производительности и применяется только когда это необходимо. Если вы выполняете операции лишь со значениями в одном потоке, то код будет работать быстрее, если ему не нужно соблюдать гарантии, предусмотренные атомиками.

Давайте вернемся к нашему примеру: типы `Arc<T>` и `Rc<T>` имеют один и тот же API, поэтому мы исправляем программу, изменяя строки `use`, вызова функции `new` и вызова метода `clone`. Код в листинге 16.15 будет окончательно скомпилирован и выполнен.

Листинг 16.15. Использование типа `Arc<T>` для обертывания типа `Mutex<T>`, чтобы делиться владением между несколькими потоками

src/main.rs

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

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Результат: {}", *counter.lock().unwrap());
}
```

Этот код выведет следующее:

```
Результат: 10
```

У нас получилось! Мы подсчитали числа от 0 до 10, что само по себе выглядит не очень впечатляюще, но это действительно дало нам много информации о `Mutex<T>` и безопасности потоков. Вы также могли бы использовать структуру этой программы для выполнения более сложных операций, чем просто увеличение подсчета. Используя эту стратегию, вы можете разделить вычисление на независимые части, которые можно разложить по потокам, а затем использовать тип `Mutex<T>` так, чтобы каждый поток обновлял конечный результат своей частью.

Сходства между RefCell<T>/Rc<T> и Mutex<T>/Arc<T>

Возможно, вы заметили, что переменная `counter` неизменяемая, но мы можем получить изменяемую ссылку на значение внутри нее. Это означает, что тип `Mutex<T>` обеспечивает внутреннюю изменяемость, как и семейство `Cell`. Мы используем тип `Mutex<T>`, чтобы изменять содержимое внутри типа `Arc<T>`, точно так же как мы использовали тип `RefCell<T>` в главе 15, который позволял нам изменять содержимое внутри типа `Rc<T>`,.

Следует отметить еще одну деталь — Rust не может защитить вас от всех видов логических ошибок, когда вы используете тип `Mutex<T>`. Вспомните из главы 15, что использование типа `Rc<T>` сопряжено с риском создания циклов в переходах по ссылкам, где два значения типа `Rc<T>` ссылаются друг на друга, вызывая утечку памяти. Схожим образом тип `Mutex<T>` связан с риском создания взаимоблокировок (deadlock). Это происходит, когда операции требуется запереть два ресурса на замок, а оба потока получают один из замков, вследствие чего они ожидают друг друга. Если вас интересуют взаимоблокировки, попробуйте создать программу Rust с взаимоблокировкой. Затем изучите, как избежать взаимоблокировок для мьютексов в любом языке и попробуйте реализовать их в Rust. Документация об API стандартной библиотеки по темам `Mutex<T>` и `MutexGuard` предлагает полезную информацию.

Мы завершим эту главу рассказом о типажах `Send` и `Sync` и о том, как их использовать с настраиваемыми типами.

Расширяемая конкурентность с типажами Send и Sync

Интересно, что в Rust очень мало средств конкурентности. Почти все средства, о которых мы говорили в этой главе, были частью стандартной библиотеки, а не самого языка. Ваши варианты работы с конкурентностью не ограничиваются язы-

ком или стандартной библиотекой; вы можете создать свои инструменты или же использовать те, которые написаны другими.

Тем не менее два понятия конкурентности встроены в язык: маркерные (`std::marker`) типажи `Sync` и `Send`.

Разрешение передавать владение между потоками с помощью Send

Маркерный типаж `Send` указывает на то, что владение типом, реализующим `Send`, может передаваться между потоками. Почти каждый тип языка Rust маркируется как `Send`, но есть некоторые исключения. К ним относится тип `Rc<T>`: он не может реализовать `Send`, потому что, если вы клонировали значение типа `Rc<T>` и попытались передать владение клоном в другом потоке, то оба потока могут обновить счетчик ссылок в одно и то же время. По этой причине тип `Rc<T>` реализован для однопоточных ситуаций, в которых не нужно тратить ресурсы для обеспечения безопасности потока.

Следовательно, благодаря системе типов Rust и границам типажа вы не отправите случайно значение типа `Rc<T>` через потоки в небезопасном виде. Когда мы попытались сделать это в листинге 16.14, возникла ошибка, а именно типаж `Send` не был реализован для `Rc<Mutex<i32>> (the trait Send is not implemented for Rc<Mutex<i32>>)`. Когда мы переключились на `Arc<T>`, который реализует `Send`, код скомпилировался.

Любой тип, целиком состоящий из типажей `Send`, автоматически помечается как `Send`. Почти все примитивные типы маркируются типажом `Send`, за исключением сырых указателей, которые мы обсудим в главе 19.

Разрешение доступа из нескольких потоков исполнения с помощью Sync

Маркерный типаж `Sync` указывает на то, что он безопасен для типа, реализующего `Sync`, на который можно ссылаться из нескольких потоков исполнения. Другими словами, любой тип `T` маркируется типажом `Sync`, если `&T` (ссылка на `T`) маркируется как `Send`, имея в виду, что ссылку можно безопасно отправить в другой поток. Подобно `Send` примитивные типы маркируются как `Sync`, а типы, целиком состоящие из типов, маркируемых как `Sync`, тоже маркируются как `Sync`.

Умный указатель `Rc<T>` не маркируется как `Sync` по тем же причинам, по которым он не маркируется как `Send`. Тип `RefCell<T>` (о котором мы говорили в главе 15) и семейство связанных типов `Cell<T>` не маркируются как `Sync`. Реализация проверки заимствования, которую `RefCell<T>` выполняет во время выполнения, небезопасна для потоков. Умный указатель `Mutex<T>` маркируется как `Sync` и может

использоваться для совместного доступа с несколькими потоками, как вы видели в разделе «Совместное использование Mutex<T> несколькими потоками» (с. 422).

Реализовывать Send и Sync вручную небезопасно

Поскольку типы, состоящие из типажей `Send` и `Sync`, также автоматически маркируются как `Send` и `Sync`, нам не нужно реализовывать их вручную. Как маркерные типажи, они даже не имеют никаких методов, подлежащих реализации. Они просто полезны для соблюдения инвариантов, связанных с конкурентностью.

Ручная реализация этих типажей предусматривает реализацию небезопасного кода Rust. Мы поговорим об использовании небезопасного кода Rust в главе 19. А пока важно знать, что нужно тщательно обдумать новые конкурентные типы, не состоящие из `Send` и `Sync`, чтобы обеспечить безопасность. В «Растономиконе» (https://doc.rust-lang.org/stable/nomicon/) есть дополнительная информация о гарантиях безопасности.

Итоги

В этой книге вы еще столкнетесь с конкурентностью. В главе 20 при работе над проектом будут использованы идеи из этой главы в более реалистичной ситуации, чем те небольшие примеры, которые мы рассмотрели здесь.

Как уже упоминалось ранее, в самом языке мало инструментов конкурентности, поэтому многие решения для задач конкурентности реализованы в качестве упаковок. Они развиваются быстрее, чем стандартная библиотека, поэтому обязательно поищите в интернете актуальные современные упаковки для многопоточных ситуаций.

Стандартная библиотека Rust предоставляет каналы для передачи сообщений и типы умных указателей, такие как `Mutex<T>` и `Arc<T>`, безопасные для использования в конкурентных контекстах. Благодаря системе типов и проверке заимствования в коде, использующем эти решения, в итоге нет гонки данных и недействительных ссылок. Как только ваш код будет компилироваться, будьте уверены, что он будет работать в нескольких потоках без едва заметных ошибок, часто встречающихся в других языках. Больше не следует опасаться конкурентного программирования: смело делайте свои программы конкурентными!

Далее мы поговорим об идиоматических способах моделирования задач и структурирования решений по мере увеличения ваших программ. Кроме того, обсудим, как идиомы Rust соотносятся с теми, которые, возможно, знакомы вам по объектно-ориентированному программированию.

17 Средства объектно-ориентированного программирования

Объектно-ориентированное программирование (ООП) — это способ моделирования программ. Объекты пришли из языка Simula в 1960-е годы. Они повлияли на архитектуру программирования Алана Кея (Alan Kay), в которой объекты передают сообщения друг другу. Он ввел термин «объектно-ориентированное программирование» в 1967 году для описания этой архитектуры. Есть много других определений, описывающих, что такое ООП. Согласно некоторым из них язык Rust классифицируется как объектно-ориентированный, но другие определения не относят Rust к этому типу. В этой главе мы рассмотрим некоторые характеристики, которые, как правило, считаются объектно-ориентированными, и узнаем, как они реализованы в Rust. Затем покажем способ реализации объектно-ориентированного паттерна проектирования и обсудим плюсы и минусы этого способа и применение некоторых сильных сторон языка Rust.

Характеристики объектно-ориентированных языков

В сообществе программистов нет единого мнения о том, какие языковые средства должны быть признаны объектно-ориентированными. Язык Rust находится под влиянием многих парадигм программирования, включая ООП. Например, в главе 13 мы познакомились со средствами, которые пришли из функционального программирования. Можно утверждать, что языки ООП имеют некоторые общие характеристики, а именно объекты, инкапсуляцию и наследование. Давайте посмотрим, что означает каждая из этих характеристик и поддерживает ли их язык Rust.

Объекты содержат данные и поведение

Книга «Паттерны объектно-ориентированного проектирования: паттерны проектирования»[1] Эриха Гаммы, Ричарда Хелма, Ральфа Джонсона и Джона Влиссидеса (*Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides Addison-Wesley Professional, 1994), в народе известных как «Банда четырех», представляет собой каталог паттернов объектно-ориентированного проектирования. В ней ООП определено таким образом:

> *Объектно-ориентированные программы состоят из объектов. Объект сочетает данные и процедуры для их обработки. Такие процедуры обычно называют методами или операциями.*

Если исходить из этого определения, то Rust объектно-ориентированный: структуры и перечисления имеют данные, а блоки `impl` предоставляют методы для этих структур и перечислений. Несмотря на то что структуры и перечисления с методами не называются объектами, они обеспечивают ту же самую функциональность, которая соответствует определению объектов, данному «Бандой четырех».

Инкапсуляция, которая скрывает детали реализации

Еще один аспект, который, как правило, ассоциируют с ООП, — инкапсуляция. Она означает, что детали реализации объекта недоступны коду, использующему этот объект. Следовательно, единственный способ взаимодействия с объектом представлен его публичным API. Код, использующий объект, не должен иметь возможности проникать во внутренние части объекта напрямую и изменять данные или поведение. За счет этого программист может изменять и рефакторизовать внутренние компоненты объекта, не меняя при этом код, использующий этот объект.

Мы обсуждали способы управления инкапсуляцией в главе 7: можно использовать ключевое слово `pub`, чтобы выбрать, какие модули, типы, функции и методы в коде должны быть публичными. Все остальное по умолчанию является приватным. Например, мы можем определить структуру `AveragedCollection` с полем, содержащим вектор значений типа `i32`. Указанная структура также может иметь поле, содержащее среднее арифметическое значений в векторе, то есть среднее арифметическое значение не обязательно должно вычисляться по чьему-либо требованию. Другими словами, `AveragedCollection` будет кэшировать вычисленное среднее значение. В листинге 17.1 дано определение структуры `AveragedCollection`.

---

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — 448 с.

Листинг 17.1. Структура AveragedCollection, которая поддерживает список целых чисел и среднее арифметическое значение элементов в коллекции

src/lib.rs

```
pub struct AveragedCollection {
    list: Vec<i32>,
    average: f64,
}
```

Указанная структура помечена как `pub`, благодаря чему ее может использовать другой код, но поля внутри структуры остаются приватными. Это важно в данном случае потому, что нужно, чтобы всякий раз, когда значение добавляется в список или удаляется из него, среднее значение также обновлялось. Мы делаем это, реализуя методы `add`, `remove` и `average` в структуре, как показано в листинге 17.2.

Листинг 17.2. Реализации публичных методов add, remove и average в структуре AveragedCollection

src/lib.rs

```
impl AveragedCollection {
    pub fn add(&mut self, value: i32) {
        self.list.push(value);
        self.update_average();
    }

    pub fn remove(&mut self) -> Option<i32> {
        let result = self.list.pop();
        match result {
            Some(value) => {
                self.update_average();
                Some(value)
            },
            None => None,
        }
    }

    pub fn average(&self) -> f64 {
        self.average
    }

    fn update_average(&mut self) {
        let total: i32 = self.list.iter().sum();
        self.average = total as f64 / self.list.len() as f64;
    }
}
```

Публичные методы `add`, `remove` и `average` — это единственные способы доступа к данным, а также для их изменения в экземпляре структуры `AveragedCollection`. Когда элемент добавляется в список с помощью метода `add` или удаляется с помощью метода `remove`, реализации каждого из них вызывают приватный метод `update_average`, который также обрабатывает обновление поля `average`.

Мы оставляем поля `list` и `average` приватными, чтобы внешний код не мог добавлять или удалять элементы непосредственно в поле `list`. В противном случае поле `average` может оказаться несинхронизированным при изменении `list`. Метод `average` возвращает значение в поле `average`, позволяя внешнему коду читать `average`, не изменяя его.

Поскольку мы инкапсулировали детали реализации структуры `AveragedCollection`, в будущем можно легко изменять такие аспекты, как структура данных. Например, мы могли бы использовать `HashSet<i32>` вместо `Vec<i32>` для поля `list`. До тех пор пока сигнатуры публичных методов `add`, `remove` и `average` остаются неизменными, код, использующий `AveragedCollection`, не нуждается в изменении. Если бы мы сделали поле `list` публичным, то это не обязательно было бы так: `HashSet<i32>` и `Vec<i32>` имеют разные методы добавления и удаления элементов, поэтому внешний код, вероятно, нужно было бы изменить, если бы он преобразовывал поле `list` напрямую.

Если инкапсуляция — это необходимый аспект для объектно-ориентированного языка, то Rust удовлетворяет этому требованию. Возможность выбирать, когда использовать ключевое слово `pub` для разных частей кода, а когда нет, позволяет инкапсулировать детали реализации.

Наследование как система типов и как совместное использование кода

Наследование — это механизм, посредством которого объект может наследовать определение другого объекта, получая таким образом данные и поведение родителя без необходимости их повторного определения.

Если наследование — это черта объектно-ориентированного языка, то тогда Rust нельзя отнести к этому типу. В нем невозможно определить структуру, которая наследует поля и реализации методов родительской структуры. Но если вы привыкли иметь наследование в своем инструментарии, то вы можете использовать другие решения Rust в зависимости от своих целей.

Вы выбираете наследование по двум главным причинам. Одна — ради повторного использования кода: вы можете реализовать некоторое поведение для одного типа, а наследование позволяет использовать эту реализацию повторно для другого. Вместо этого вы можете делиться кодом Rust с помощью реализаций по умолчанию для типажных методов, которые вы видели в листинге 10.14, когда мы добавили реализацию по умолчанию метода `summarize` для типажа `Summary`. Любой тип, реализующий типаж `Summary`, будет иметь метод `summarize`, доступный для него без какого-либо дополнительного кода. Это похоже на родительский класс с реализацией метода, наследующий классу потомка, также с реализацией этого метода. Кроме того, мы можем переопределить реализацию по умолчанию метода `summarize` во время реализации типажа `Summary`, что аналогично классу потомка, переопределяющему реализацию метода, унаследованного от родительского класса.

Другая причина использования наследования связана с системой типов: оно необходимо для возможности использовать тип потомков в тех же местах, что и родительский тип. Это также называется «полиморфизм», то есть вы можете заменять одни объекты другими во время выполнения, если у них есть некоторые общие характеристики.

ПОЛИМОРФИЗМ

Для многих людей полиморфизм является синонимом наследования. Но на самом деле это более широкое понятие, относящееся к коду, который может работать с данными нескольких типов. Для наследования эти типы обычно являются подклассами.

Вместо этого Rust использует обобщения для абстрагирования от всевозможных типов и границы типажа для ограничения того, что эти типы должны обеспечивать. Этот подход иногда называют ограниченным параметрическим полиморфизмом.

Во многих языках программирования наследование в последнее время стало выходить из употребления как техническое решение для создания программ, потому что из-за него часто есть риск поделиться бóльшим объемом кода, чем необходимо. Подклассы не обязательно будут все время делиться всеми характеристиками своего родительского класса, но будут делать это с наследованием. Вследствие этого проектирование программы может стать менее гибким. Кроме того, из-за этого возникает вероятность бессмысленного вызова методов для подклассов, что может привести к ошибкам, потому что методы не применимы к подклассу. В дополнение к этому, некоторые языки допускают наследование подкласса только от одного класса, что еще больше ограничивает гибкость программы.

По этим причинам в языке Rust есть другой подход — использование типажных объектов вместо наследования. Давайте посмотрим, как типажные объекты обеспечивают полиморфизм в языке Rust.

Использование типажных объектов, допускающих значения разных типов

В главе 8 мы упоминали, что одним из ограничений векторов является то, что они могут хранить элементы только одного типа. В листинге 8.10 мы создали обходной путь, где определили перечисление `SpreadsheetCell`, которое имело варианты для хранения целых чисел, чисел с плавающей точкой и текста. Это означало, что мы могли хранить разные типы данных в каждой ячейке и при этом иметь вектор в виде серии ячеек. Это идеальное решение, когда взаимозаменяемые элементы представляют собой фиксированный набор типов, при которых код компилируется.

Однако иногда мы хотим, чтобы пользователь библиотеки мог расширить набор типов, допустимых в конкретной ситуации. Чтобы показать, как это сделать, мы создадим пример графического пользовательского интерфейса (GUI), который перебирает элементы в списке, вызывая метод `draw` для отрисовки каждого из них на экране, — часто встречающийся технический прием для инструментов GUI. Мы создадим библиотечную упаковку под названием `gui`, которая содержит в себе структуру библиотеки GUI. Эта упаковка включает в себя несколько типов, таких как `Button` или `TextField`. В дополнение к этому пользователи `gui` смогут создавать собственные типы, которые можно нарисовать: например, один программист, возможно, добавит изображение (тип `Image`), а другой — поле выбора (тип `SelectBox`).

В данном примере мы не будем реализовывать полнофункциональную библиотеку GUI, но покажем, как ее части будут сочетаться друг с другом. На момент написания библиотеки мы не можем знать и определить все типы, которые другие программисты, возможно, захотят создать. Но мы знаем, что библиотека `gui` должна отслеживать большое число значений разных типов и вызывать метод `draw` для каждого из этих по-разному типизированных значений. Не нужно знать наверняка, что произойдет, когда мы вызовем метод `draw`; требуется знать только то, что этот метод будет иметься у значения для вызова.

Для того чтобы сделать это в языке с наследованием, мы, возможно, определили бы класс с именем `Component` с методом `draw`. Другие классы, такие как `Button`, `Image` и `SelectBox`, наследовали бы у класса `Component` и поэтому наследовали бы метод `draw`. Каждый из них мог бы переопределить метод `draw`, задав свое индивидуальное поведение, но интерфейсный каркас мог бы трактовать все типы, как если бы они были экземплярами класса `Component`, и вызывать метод `draw` для них. Но поскольку Rust не имеет наследования, нам нужен другой способ структурировать библиотеку `gui`, чтобы позволить пользователям расширять ее новыми типами.

Определение типажа для часто встречающегося поведения

Для того чтобы реализовать поведение, которого мы хотим добиться от `gui`, мы определим типаж с именем `Draw` с одним методом `draw`. Затем мы можем определить вектор, который берет типажный объект. Типажный объект указывает как на экземпляр типа, реализующего указанный нами типаж, так и на таблицу для поиска методов типажа в этом типе во время выполнения. Мы создаем типажный объект, описывая некоторый вид указателя, такой как ссылка `&` или умный указатель `Box<T>`, с последующим ключевым словом `dyn`, а затем указывая соответствующий типаж. (Мы поговорим, почему типажные объекты должны использовать указатель, в разделе «Динамически изменяемые типы и типаж `Sized`».) Мы можем использовать типажные объекты вместо обобщенного или конкретного типа. Везде, где мы используем типажный объект, система типов Rust во время компиляции сделает так, чтобы любое значение в этом контексте реализовывало типаж типаж-

ного объекта. Следовательно, во время компиляции нам не нужно знать все возможные типы.

Мы уже упоминали, что в Rust не принято называть структуры и перечисления объектами, чтобы отличать их от объектов других языков. В структуре либо в перечислении данные в полях структуры и поведение в блоках `impl` отделены, тогда как в других языках данные и поведение объединены в одно понятие, которое часто обозначается термином «объект». Однако типажные объекты больше похожи на объекты в других языках в том смысле, что они объединяют данные и поведение. Но при этом типажные объекты отличаются от традиционных объектов тем, что в типажный объект нельзя добавлять данные. Типажные объекты не так полезны, как объекты в других языках. Их специфическая цель — сделать возможной абстракцию в часто встречающемся поведении.

Листинг 17.3 показывает, как определять типаж с именем `Draw` с одним методом `draw`.

Листинг 17.3. Определение типажа Draw

src/lib.rs

```
pub trait Draw {
    fn draw(&self);
}
```

Вам знаком этот синтаксис из темы определения типажей в главе 10. Далее следует новый синтаксис: листинг 17.4 определяет структуру `Screen`, которая содержит вектор `components`. Этот вектор имеет тип `Box<Dyn Draw>`, который является типажным объектом. Он замещает любой тип внутри `Box`, реализующий типаж `Draw`.

Листинг 17.4. Определение структуры Screen с полем components, содержащим вектор типажных объектов, реализующих типаж Draw

src/lib.rs

```
pub struct Screen {
    pub components: Vec<Box<dyn Draw>>,
}
```

В структуре `Screen` мы определим метод `run`, который будет вызывать метод `draw` для каждого своего компонента, как показано в листинге 17.5.

Листинг 17.5. Метод run в структуре Screen, который вызывает метод draw для каждого компонента

src/lib.rs

```
impl Screen {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

Она работает иначе, чем определение структуры, которое использует параметр обобщенного типа с границами типажа. Параметр обобщенного типа может быть заменен только одним конкретным типом за раз, в то время как типажные объекты обеспечивают возможность заполнять типажный объект несколькими конкретными типами во время выполнения. Например, мы могли бы определить структуру `Screen` с помощью обобщенного типа и границы типажа, как в листинге 17.6.

Листинг 17.6. Альтернативная реализация структуры Screen и ее метода run с использованием обобщений и границ типажа

src/lib.rs

```
pub struct Screen<T: Draw> {
    pub components: Vec<T>,
}

impl<T> Screen<T>
    where T: Draw {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

Это ограничивает нас экземпляром структуры `Screen`, который содержит список всех компонентов типа `Button` либо `TextField`. Если когда-либо у вас будут только однородные коллекции, то предпочтительно использовать обобщения и границы типажа, потому что определения будут мономорфизированы во время компиляции для использования конкретных типов.

С другой стороны, с методом, использующим типажные объекты, один экземпляр структуры `Screen` может содержать `Vec<T>` с `Box<Button>`, а также `Box<TextField>`. Давайте посмотрим, как это работает, а затем поговорим о последствиях для производительности времени выполнения.

Реализация типажа

Теперь мы добавим несколько типов, реализующих типаж `Draw`. Мы предоставим тип `Button`. Опять же, фактическая реализация библиотеки GUI выходит за рамки темы этой книги, поэтому в теле метода `draw` не будет никакой полезной реализации. Чтобы представить себе, как может выглядеть реализация, структура `Button` может иметь поля `width`, `height` и `label`, как показано в листинге 17.7.

Листинг 17.7. Структура Button, реализующая типаж Draw

src/lib.rs

```
pub struct Button {
    pub width: u32,
```

```
    pub height: u32,
    pub label: String,
}

impl Draw for Button {
    fn draw(&self) {
        // код для фактической отрисовки кнопки
    }
}
```

Поля `width`, `height` и `label` в структуре `Button` будут отличаться от полей в других компонентах, таких как тип `TextField`, которые вместо этого могли бы иметь эти же поля плюс поле `placeholder`. Каждый тип, который мы хотим нарисовать на экране, будет реализовывать типаж `Draw`, но использовать другой код в методе `draw` с определением способа отрисовки этого конкретного типа, как здесь у типа `Button` (без фактического кода GUI, который выходит за рамки темы этой главы). Тип `Button`, например, мог бы иметь дополнительный блок `impl`, содержащий методы, связанные с тем, что происходит, когда пользователь нажимает кнопку. Эти методы не будут применимы к таким типам, как `TextField`.

Если пользователь библиотеки решит реализовать структуру `SelectBox`, содержащую поля `width`, `height` и `options`, то он также реализует типаж `Draw` для типа `SelectBox`, как показано в листинге 17.8.

Листинг 17.8. Еще одна упаковка, использующая gui и реализующая типаж Draw в структуре SelectBox

src/main.rs

```
use gui::Draw;

struct SelectBox {
    width: u32,
    height: u32,
    options: Vec<String>,
}

impl Draw for SelectBox {
    fn draw(&self) {
        // код для фактической отрисовки кнопки
    }
}
```

Теперь пользователь библиотеки может написать свою функцию `main` с экземпляром структуры `Screen`. В этот экземпляр он может добавить структуры `SelectBox` и `Button`, поместив каждую в `Box<T>`, которые станут типажным объектом. Затем он может вызвать метод `run` для экземпляра `Screen`, который будет вызывать метод `draw` для каждого компонента. Эта реализация показана в листинге 17.9.

Листинг 17.9. Использование типажных объектов для хранения значений разных типов, реализующих один и тот же типаж

src/main.rs

```
use gui::{Screen, Button};

fn main() {
    let screen = Screen {
        components: vec![
            Box::new(SelectBox {
                width: 75,
                height: 10,
                options: vec![
                    String::from("Да"),
                    String::from("Может быть"),
                    String::from("Нет")
                ],
            }),
            Box::new(Button {
                width: 50,
                height: 10,
                label: String::from("OK"),
            }),
        ],
    };

    screen.run();
}
```

Когда мы писали библиотеку, то не знали, что кто-то может добавить тип `SelectBox`. Но реализация типа `Screen` была способна работать с новым типом и рисовать его, потому что `SelectBox` реализует типаж `Draw`, то есть реализует метод `draw`.

Идея заниматься только сообщениями, на которые откликается значение, а не конкретным типом значения похожа на идею утиной типизации в динамически типизированных языках: если что-то ходит как утка и крякает как утка, то это точно утка! В реализации функции `run` в структуре `Screen` листинга 17.5 не требуется знать конкретный тип каждого компонента. Функция не проверяет, экземпляром какой структуры, `Button` или `SelectBox`, является компонент. Она просто вызывает метод `draw` для компонента. Указав `Box<dyn Draw>` в качестве типа значений в векторе `components`, мы определили, что структуре `Screen` нужны значения, для которых мы можем вызывать метод `draw`.

Преимущество типажных объектов и системы типов Rust для написания кода, похожего на код с утиной типизацией, заключается в том, что нам не нужно проверять во время выполнения, реализует ли значение тот или иной метод. Нам также не нужно беспокоиться о возможных ошибках, если значение не реализует метод, но мы все равно его вызываем. Rust не будет компилировать код, если значения не реализуют типажи, которые нужны типажным объектам.

Например, в листинге 17.10 показано, что произойдет, если мы попытаемся создать структуру `Screen` с типом `String` в качестве компонента.

Листинг 17.10. Попытка использовать тип, который не реализует типаж типажного объекта

src/main.rs

```
use gui::Screen;

fn main() {
    let screen = Screen {
        components: vec![
            Box::new(String::from("Привет")),
        ],
    };

   screen.run();
}
```

Мы получим эту ошибку, потому что тип `String` не реализует типаж `Draw`:

```
error[E0277]: the trait bound `std::string::String: gui::Draw` is not
satisfied
 --> src/main.rs:7:13
  |
7 |              Box::new(String::from("Привет")),
  |              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait gui::Draw is not
  implemented for `std::string::String`
  |
  = note: required for the cast to the object type `gui::Draw`
```

Эта ошибка показывает, что либо в экземпляр структуры `Screen` передается что-то, что мы не задумывали передавать, и нам следует передать другой тип, либо нужно реализовать `Draw` в типе `String`, в результате чего структура `Screen` сможет вызывать для него метод `draw`.

Типажные объекты выполняют динамическую диспетчеризацию

Напомним, что в разделе «Производительность кода с использованием обобщений» мы обсуждали процесс мономорфизации, выполняемый компилятором при использовании границ типажа для обобщений: компилятор генерирует необобщенные реализации функций и методов для каждого конкретного типа, используемые вместо параметра обобщенного типа. Код, полученный в результате мономорфизации, выполняет статическую диспетчеризацию, когда компилятор знает, какой метод вы вызываете во время компиляции. Это противоречит динамической диспетчеризации, когда компилятор не может сказать во время компиляции, какой метод вы вызываете. В случаях динамической диспетчеризации компиля-

тор порождает код, который во время выполнения будет выяснять, какой метод вызывать.

Когда мы используем типажные объекты, Rust должен применять динамическую диспетчеризацию. Компилятор не знает все типы, которые могли бы использоваться с кодом, в котором есть типажные объекты, поэтому ему неизвестно, ни какой метод реализован, ни для какого типа его вызывать. Вместо этого во время выполнения Rust использует указатели внутри типажного объекта, чтобы знать, какой метод вызывать. Когда есть уточняющий запрос, несколько увеличивается время выполнения, чего не происходит со статической диспетчеризацией. Динамическая диспетчеризация также не дает компилятору внедрять код метода, поэтому, в свою очередь, не выполняются некоторые оптимизации. Тем не менее мы получили дополнительную гибкость в коде из листинга 17.5 и смогли поддержать эту гибкость в листинге 17.9, так что стоит это учитывать.

Объектная безопасность необходима для типажных объектов

Можно переделывать в типажные объекты только объектно-безопасные типажи. Несколько сложных правил управляют всеми свойствами, которые делают типажный объект безопасным, но на практике важны только два правила. Типаж объектно-безопасен, если все методы, определенные в нем, имеют следующие свойства:

- Возвращаемый тип не является `Self`.
- Параметры обобщенных типов отсутствуют.

Ключевое слово `Self` — это псевдоним типа, в котором мы реализуем типажи или методы. Типажные объекты должны быть объектно-безопасными, потому что после использования типажного объекта Rust больше не знает конкретный тип, реализующий этот типаж. Если типажный метод возвращает конкретный тип `Self`, а типажный объект забывает точный тип, который `Self` имеет, то нет ни единого способа, которым данный метод сможет использовать оригинальный конкретный тип. То же самое верно и для параметров обобщенного типа, которые заполняются параметрами конкретного типа при использовании типажа: конкретные типы становятся частью типа, реализующего типаж. Когда тип забывается в результате использования типажного объекта, никак нельзя узнать, какими типами заполнять параметры обобщенного типа.

Примером типажа, методы которого не являются объектно-безопасными, является `Clone` стандартной библиотеки. Сигнатура метода `clone` в типаже `Clone` выглядит следующим образом:

```
pub trait Clone {
    fn clone(&self) -> Self;
}
```

Тип `String` реализует типаж `Clone`. Вызывая метод `clone` для экземпляра `String`, мы получаем обратно экземпляр типа `String`. Схожим образом, если мы вызываем метод `clone` для экземпляра типа `Vec<T>`, то получаем обратно экземпляр типа `Vec<T>`. Сигнатура метода `clone` должна знать, какой тип будет использоваться для `Self`, потому что этот тип возвращается.

Компилятор выдаст сообщение, если вы попытаетесь сделать что-то против правил объектной безопасности в отношении типажных объектов. Допустим, мы попытались реализовать структуру `Screen` в листинге 17.4 для хранения типов, реализующих типаж `Clone` вместо типажа `Draw`, как тут:

```
pub struct Screen {
    pub components: Vec<Box<dyn Clone>>,
}
```

получаем такую ошибку[1]:

```
error[E0038]: the trait `std::clone::Clone` cannot be made into an object
 --> src/lib.rs:2:5
  |
2 |     pub components: Vec<Box<dyn Clone>>,
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `std::clone::Clone`
 cannot be made into an object
  |
  = note: the trait cannot require that `Self : Sized`
```

Указанная ошибка означает, что вы не можете использовать этот типаж в качестве типажного объекта в таком виде. Если вас интересует более подробная информация об объектной безопасности, то обратитесь к Rust RFC 255 по адресу https://github.com/rust-lang/rfcs/blob/master/text/0255-object-safety.md/.

Реализация объектно-ориентированного паттерна проектирования

Паттерн переходов между состояниями — это объектно-ориентированный паттерн проектирования. Суть этого паттерна в том, что у значения есть некое внутреннее состояние, представленное совокупностью объектов состояния, и поведение данного значения изменяется в зависимости от этого внутреннего состояния. Объекты состояния имеют общий функционал: в Rust, конечно, мы используем структуры и типажи, а не объекты и наследование. Каждый объект состояния отвечает за свое поведение и за управление тем, когда он должен переходить в другое состояние. Значение, содержащее объект состояния, ничего не знает о разном поведении состояний и о том, когда переходить между состояниями.

[1] `ошибка[E0038]: типаж 'std::clone::Clone' не может быть переделан в объект`

Использование паттерна переходов между состояниями означает, что при изменении бизнес-требований к программе не нужно изменять код значения, содержащего состояние, или код, использующий значение. Нам нужно только обновить код внутри одного из объектов состояния, чтобы изменить его правила или, возможно, добавить больше объектов состояния. Давайте рассмотрим пример паттерна переходов между состояниями и его применение в Rust.

Мы реализуем процесс создания поста в блоге в несколько этапов. Функциональность блога будет выглядеть следующим образом:

1. Пост начинается как пустой черновик.
2. Когда черновик готов, запрашивается его проверка.
3. После того как пост одобрен, он публикуется.
4. Печатаются только опубликованные посты, поэтому неодобренные не могут быть опубликованы случайно.

Другие изменения поста не должны иметь никакого эффекта. Например, если мы попытаемся одобрить черновик поста до того, как запросим проверку, то такой пост должен остаться неопубликованным черновиком.

В листинге 17.11 процесс создания поста показан в коде: он представляет собой пример использования API для реализации в библиотечной упаковке `blog`. Код пока не компилируется, так как мы еще не реализовали упаковку `blog`.

Листинг 17.11. Код, демонстрирующий желаемое поведение упаковки blog

src/main.rs

```
use blog::Post;

fn main() {
 ❶ let mut post = Post::new();

 ❷ post.add_text("Сегодня на обед я ел салат");
 ❸ assert_eq!("", post.content());

 ❹ post.request_review();
 ❺ assert_eq!("", post.content());

 ❻ post.approve();
 ❼ assert_eq!("Сегодня на обед я ел салат", post.content());
}
```

Мы хотим, чтобы пользователь мог создавать новый черновик поста с помощью `Post::new` ❶. Затем мы хотим, чтобы текст добавлялся в пост, пока он находится в черновом состоянии ❷. Если мы попытаемся получить содержимое статьи немедленно, до ее одобрения, то ничего не произойдет, потому что пост пока еще является черновиком. Мы добавили макрокоманду `assert_eq!` в код для демонстрационных целей ❸. Отличным модульным тестом для нее была бы проверка, что черновик возвращает пустую строку из метода `content`, но для данного примера мы писать тесты не будем.

Затем мы хотим обеспечить возможность запроса на рассмотрение поста ❹, чтобы метод `content` возвращал пустую строку, ожидая рассмотрения ❺. Когда пост получает одобрение ❻, он должен быть опубликован, то есть текст сообщения будет возвращен, когда будет вызван метод `content` ❼.

Обратите внимание, что единственный тип из упаковки, с которым мы взаимодействуем, — это `Post`. Этот тип будет использовать паттерн переходов между состояниями и содержать значение — один из трех объектов состояния, в которых может быть статья-черновик, статья, которая ожидает рассмотрения, или опубликованная статья. Переход из одного состояния в другое будет управляться внутри типа `Post`. Состояния изменяются в ответ на методы, вызываемые пользователями библиотеки в экземпляре типа `Post`, но пользователям не приходится управлять изменениями состояний напрямую. Кроме того, пользователи не могут ошибиться в состояниях, например, публикуя пост до того, как он будет рассмотрен.

Определение поста и создание нового экземпляра в состоянии черновика

Давайте приступим к реализации библиотеки! Нам нужна публичная структура `Post`, имеющая некое содержимое, поэтому мы начнем с определения структуры и связанной с ней публичной функции `new` для создания экземпляра структуры `Post`, как показано в листинге 17.12. Мы также создадим приватный типаж `State`. Тогда `Post` будет содержать типажный объект `Box<Dyn State>` внутри `Option<T>` в приватном поле `state`. Чуть позже вы увидите, зачем нужен тип `Option<T>`.

Листинг 17.12. Определение структуры Post и функции new, создающей новый экземпляр структуры Post, а также типаж State и структура Draft

src/lib.rs

```
pub struct Post {
    state: Option<Box<dyn State>>,
    content: String,
}

impl Post {
    pub fn new() -> Post {
        Post {
         ❶ state: Some(Box::new(Draft {})),
         ❷ content: String::new(),
        }
    }
}

trait State {}

struct Draft {}

impl State for Draft {}
```

Типаж `State` определяет совместное поведение разных состояний поста, и все состояния — `Draft`, `PendingReview` и `Published` — будут реализовывать типаж `State`. Пока что у данного типажа нет никаких методов. Мы начнем с определения только `Draft`, потому что именно с этого состояния начинается пост.

Когда мы создаем новый экземпляр структуры `Post`, то устанавливаем его поле `state` равным значению `Some`, в котором содержится умный указатель `Box` ❶. `Box` указывает на новый экземпляр структуры `Post`. Благодаря этому всякий раз, когда мы создаем новый экземпляр структуры `Post`, он будет начинаться как черновик. Поскольку поле `state` экземпляра структуры `Post` приватное, нет никакого способа создать `Post` в любом другом состоянии! В функции `Post::new` мы устанавливаем поле `content` равным новому, пустому экземпляру типа `String` ❷.

Хранение текста поста

Листинг 17.11 показывает, что мы хотим вызывать метод `add_text` и передавать ему строковый срез `&str`, который затем добавляется в текстовое содержимое статьи блога. Мы реализуем его как метод, а не представляем поле `content` как `pub`. Это означает, что позже мы можем реализовать метод, который будет контролировать, как читаются данные поля `content`. Метод `add_text` довольно прост, поэтому давайте добавим его реализацию в блок `impl Post` листинга 17.13.

Листинг 17.13. Реализация метода add_text для добавления текста в содержимое поста

src/lib.rs

```
impl Post {
    // --пропуск--
    pub fn add_text(&mut self, text: &str) {
        self.content.push_str(text);
    }
}
```

Метод `add_text` берет изменяемую ссылку на `self`, потому что мы изменяем экземпляр структуры `Post`, для которой вызываем метод `add_text`. Затем мы вызываем `push_str` для экземпляра типа `String` в поле `content` и передаем аргумент `text` для добавления в сохраненное содержимое `content`. Это поведение не зависит от состояния, в котором находится пост, поэтому оно не является частью паттерна переходов между состояниями. Метод `add_text` вообще не взаимодействует с полем `content`, но представляет собой часть нужного нам поведения.

Делаем пустой черновик

Даже после того, как мы вызвали метод `add_text` и добавили некое содержимое в статью, нужно, чтобы метод `content` по-прежнему возвращал пустой строковый срез, так как статья пока находится в черновом состоянии, как показано в ❸ в лис-

тинге 17.11. Пока что давайте реализуем метод `content` с простейшей вещью, которая будет выполнять это требование: всегда возвращать пустой строковый срез. Мы изменим это поведение позже, когда займемся реализацией возможности изменять состояние статьи, чтобы ее можно было публиковать. Пока что статьи могут быть только в черновом состоянии, поэтому содержимое статьи всегда должно быть пустым. В листинге 17.14 показана такая заполнительная реализация.

Листинг 17.14. Добавление заполнителя для метода content в структуру Post, которая всегда возвращает пустой строковый срез &str

src/lib.rs

```
impl Post {
    // --пропуск--
    pub fn content(&self) -> &str {
        ""
    }
}
```

После того как был добавлен метод `content`, в листинге 17.11 все работает, как и задумывалось, вплоть до строки в ❸.

Запрос на проверку статьи изменяет ее состояние

Далее нам нужно добавить функциональность запроса на проверку статьи, который должен изменить ее состояние с `Draft` на `PendingReview`. Этот код показан в листинге 17.15.

Листинг 17.15. Реализация методов request_review в структуре Post и типаже State

src/lib.rs

```
impl Post {
    // --пропуск--
  ❶ pub fn request_review(&mut self) {
      ❷ if let Some(s) = self.state.take() {
          ❸ self.state = Some(s.request_review())
        }
    }
}

trait State {
  ❹ fn request_review(self: Box<Self>) -> Box<dyn State>;
}

struct Draft {}

impl State for Draft {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
      ❺ Box::new(PendingReview {})
    }
}
```

```
struct PendingReview {}

impl State for PendingReview {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
      ❻ self
    }
}
```

Мы даем экземпляру структуры `Post` публичный метод `request_review`, который будет брать изменяемую ссылку на `self` ❶. Затем мы вызываем внутренний метод `request_review` для текущего состояния структуры `Post` ❸, и этот второй метод `request_review` поглощает текущее состояние и возвращает новое состояние.

Мы добавили метод `request_review` в типаж `State` ❹. Все типы, которые реализуют указанный типаж, теперь должны будут реализовывать метод `request_review`. Обратите внимание, что вместо `self`, `&self` или `&mut self` в качестве первого параметра метода у нас `self: Box<Self>`. Этот синтаксис означает, что метод допустим только при вызове для умного указателя `Box`, содержащего тип. Этот синтаксис берет `Box<Self>` во владение, делая старое состояние недействительным, благодаря чему значение состояния структуры `Post` способно трансформироваться в новое состояние.

Для того чтобы поглотить старое состояние, метод `request_review` должен стать владельцем значения состояния. Именно здесь появляется тип `Option` в поле `state` структуры `Post`: мы вызываем метод `take`, чтобы взять значение `Some` из поля `state` и не трогать `None`, потому что в языке Rust нельзя иметь незаполненные поля в структурах ❷. Это позволяет нам переместить значение `state` из структуры `Post`, а не заимствовать его. Затем мы установим значение `state` статьи равным результату этой операции.

Чтобы завладеть значением `state`, нужно временно установить `state` равным `None`, а не устанавливать его непосредственно с помощью кода, такого как `self.state = self.state.request_review();`. Благодаря этому структура `Post` не сможет использовать старое значение `state` после того, как мы трансформировали его в новое состояние.

Метод `request_review` в структуре `Draft` должен возвращать новый, обернутый в умный указатель `Box` экземпляр новой структуры `PendingReview` ❺, которая представляет состояние, когда статья ожидает проверки. Структура `PendingReview` тоже реализует метод `request_review`, но он не выполняет никаких трансформаций. Напротив, он возвращает себя ❻, потому что, когда мы запрашиваем проверку статьи, уже находящейся в состоянии `PendingReview`, она должна оставаться в состоянии `PendingReview`.

Теперь мы видим преимущества паттерна переходов между состояниями: метод `request_review` в структуре `Post` остается тем же, независимо от значения его поля `state`. Каждое состояние отвечает за собственные правила.

Мы оставим метод `content` в структуре `Post` как есть, возвращая пустой строковый срез. Теперь статья может быть в состоянии `PendingReview`, а также в состоянии `Draft`, но мы хотим, чтобы то же самое поведение было в состоянии `PendingReview`. Листинг 17.11 теперь работает вплоть до строки ❺!

Добавление метода approve, который изменяет поведение метода content

Метод `approve` будет похож на метод `request_review`: он установит поле `state` равным значению, которое согласно текущему состоянию оно должно иметь, когда это состояние одобрено, как показано в листинге 17.16.

Листинг 17.16. Реализация методов approve в структуре Post и типаже Sate

src/lib.rs

```
impl Post {
    // --пропуск--
    pub fn approve(&mut self) {
        if let Some(s) = self.state.take() {
            self.state = Some(s.approve())
        }
    }
}

trait State {
    fn request_review(self: Box<Self>) -> Box<dyn State>;
    fn approve(self: Box<Self>) -> Box<dyn State>;
}

struct Draft {}

impl State for Draft {
    // --пропуск--
    fn approve(self: Box<Self>) -> Box<dyn State> {
     ❶ self
    }
}

struct PendingReview {}

impl State for PendingReview {
    // --пропуск--
    fn approve(self: Box<Self>) -> Box<dyn State> {
     ❷ Box::new(Published {})
    }
}

struct Published {}

impl State for Published {
```

```
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        self
    }

    fn approve(self: Box<Self>) -> Box<dyn State> {
        self
    }
}
```

Мы добавляем метод `approve` в типаж `State` и новую структуру, реализующую типаж `State`, состояние `Published`.

Аналогично методу `request_review`, если мы вызовем метод `approve` для структуры `Draft`, то он не будет иметь никакого эффекта, потому что он возвращает `self` ❶. Когда мы вызываем метод `approve` для структуры `PendingReview`, он возвращает новый, обернутый в умный указатель `Box` экземпляр структуры `Published` ❷. Структура `Published` реализует типаж `State`, и как для метода `request_review`, так и для метода `approve` она возвращает себя, поскольку в этих случаях статья должна оставаться в состоянии `Published`.

Теперь нам нужно обновить метод `content` для структуры `Post`: если состояние равно `Published`, то мы хотим вернуть значение, находящееся в поле `content` статьи. В противном случае мы хотим вернуть пустой строковый срез, как показано в листинге 17.17.

Листинг 17.17. Обновление метода content в структуре Post для делегирования полномочий методу content в типаже State

src/lib.rs

```
impl Post {
    // --пропуск--
    pub fn content(&self) -> &str {
        self.state.as_ref().unwrap().content(&self)
    }
    // --пропуск--
}
```

Поскольку цель состоит в том, чтобы держать все эти правила внутри структур, которые реализуют типаж `State`, мы вызываем метод `content` для значения в поле `state` и передаем экземпляр статьи (то есть `self`) в качестве аргумента. Затем мы возвращаем значение, получаемое в результате использования метода `content`, для значения поля `state`.

Мы вызываем метод `as_ref` для экземпляра типа `Option`, потому что нам нужна ссылка на значение внутри `Option`, а не владение значением. Поскольку поле `state` равно `Option<Box<dyn State>>`, когда мы вызываем `as_ref`, возвращается `Option<&Box<dyn State>>`. Если бы мы не вызывали `as_ref`, то возникла бы ошибка, потому что мы не можем переместить `state` из заимствованной ссылки `&self` параметра функции.

Затем мы вызываем метод `unwrap`, который, как известно, никогда не будет паниковать. Мы знаем, что благодаря методам для структуры `Post` поле `state` всегда содержит значение `Some`, когда эти методы заканчивают работу. Это один из случаев, о которых мы говорили в разделе «Случаи, когда у вас больше информации, чем у компилятора» (с. 206), когда известно, что значение `None` невозможно, даже если компилятор не в состоянии понять это.

В момент, когда мы вызываем метод `content` для `&Box<Dyn State>`, вступит в силу принудительное приведение типа посредством `deref` для `&` и `Box`, и поэтому метод `content` в конечном счете будет вызван для типа, который реализует типаж `State`. Таким образом, нам нужно добавить `content` в определение типажа `State`. Именно там мы разместим алгоритм для возврата содержимого в зависимости от имеющегося состояния, как показано в листинге 17.18.

Листинг 17.18. Добавление метода content в типаж State

src/lib.rs

```
trait State {
    // --пропуск--
    fn content<'a>(&self, post: &'a Post) -> &'a str {
      ❶ ""
    }
}

// --пропуск--

struct Published {}

impl State for Published {
    // --пропуск--
    fn content<'a>(&self, post: &'a Post) -> &'a str {
      ❷ &post.content
    }
}
```

Мы добавляем реализацию по умолчанию для метода `content`, которая возвращает пустой строковый срез ❶. То есть нам не нужно реализовывать `content` в структурах `Draft` и `PendingReview`. Структура `Published` будет переопределять метод `content` и возвращать значение в `post.content` ❷.

Обратите внимание, что для этого метода нужны аннотации жизненных циклов, как обсуждалось в главе 10. Мы берем ссылку на `post` в качестве аргумента и возвращаем ссылку на часть этого аргумента `post`, поэтому жизненный цикл возвращаемой ссылки связан с жизненным циклом аргумента `post`.

И дело сделано — теперь весь листинг 17.11 работает! Мы применили паттерн переходов между состояниями к процессу создания статьи для блога. Алгоритм, связанный с указанными правилами, размещен в объектах состояния, а не разбросан по всей структуре `Post`.

Компромиссы паттерна переходов между состояниями

Мы показали, что язык Rust способен реализовывать объектно-ориентированный паттерн переходов между состояниями с инкапсуляцией разных видов поведения, которые статья должна иметь в каждом состоянии. Методы для структуры `Post` ничего не знают о различных поведениях. Принятый нами принцип организации кода позволяет обращаться только в одно место, чтобы узнать, каким может быть поведение публикуемой статьи: к реализации типажа `State` в структуре `Published`.

Если бы мы создали альтернативную реализацию без паттерна переходов между состояниями, то можно было бы использовать выражения `match` в методах для структур `Post` или даже в коде функции `main`, которые проверяют состояние статьи и изменяют поведение в этих местах. Таким образом нам пришлось бы обращаться к нескольким местам, чтобы понять, какие последствия возможны, когда статья находится в опубликованном состоянии! Это увеличило бы число добавленных состояний: каждому из выражений `match` потребовался бы еще один рукав.

С паттерном переходов между состояниями методы структуры `Post` и места, которые мы используем в `Post`, не нуждаются в выражениях `match`. Для добавления нового состояния нам нужно будет только добавить новую структуру и реализовать методы типажа для этой структуры.

Реализация с паттерном переходов между состояниями легко расширяется, что дает большую функциональность. Убедитесь в простоте технического сопровождения кода с паттерном переходов между состояниями, выполнив несколько рекомендаций:

- Добавьте метод `reject`, который изменяет состояние статьи из `PendingReview` в `Draft`.
- Внесите требование делать два вызова метода `approve`, прежде чем состояние может быть изменено на `Published`.
- Разрешите пользователям добавлять текстовое содержимое только в том случае, если статья находится в состоянии `Draft`. Подсказка: дайте объекту состояния возможность отвечать за то, что может меняться в содержимом, но не отвечать за модифицирование структуры `Post`.

Одним из недостатков модели состояний является то, что, поскольку состояния реализуют переходы между собой, некоторые из них связаны друг с другом. Если бы между состояниями `PendingReview` и `Published` мы добавили еще одно состояние, к примеру, `Scheduled`, то нам пришлось бы изменить код в `PendingReview` для перехода в `Scheduled`. Было бы проще, если бы структуре `PendingReview` не требовалось вносить изменения с добавлением нового состояния, но это означало бы переход к другому паттерну проектирования.

Еще один недостаток заключается в том, что мы дублировали алгоритм. Ради устранения некоторых повторов мы могли бы сделать реализации по умолча-

нию для методов `request_review` и `approve` для типажа `State`, которые возвращают `self`. Однако это нарушило бы объектную безопасность, потому что этот типаж не знает, каким конкретно будет `self`. Мы хотим использовать типаж `State` как типажный объект, поэтому его методы должны быть безопасными для объектов.

Другой повтор включает в себя схожие реализации методов `request_review` и `approve` в структуре `Post`. Оба метода делегируют полномочия одинакового метода, определенного для значения поля `state` экземпляра типа `Option`, и устанавливают новое значение поля `state` равным результату. Если бы для структуры `Post` было много методов, которые следовали бы этому паттерну, то мы бы могли подумать об определении макрокоманды для устранения повтора (см. раздел «Макрокоманды» в главе 19).

Применяя паттерн переходов между состояниями в точности как он определен для объектно-ориентированных языков, мы не используем сильные стороны Rust в полной мере. Давайте взглянем на некоторые изменения упаковки `blog`, из-за которых недопустимые состояния и переходы могут превратиться в ошибки времени компиляции.

Кодирование состояний и поведения как типов

Мы покажем вам, как переосмыслить паттерн переходов между состояниями, чтобы получить другую совокупность преимуществ. Вместо полной инкапсуляции состояний и переходов, чтобы внешний код о них не знал, мы будем кодировать состояния в разные типы. Следовательно, система проверки типов Rust будет предотвращать попытки использовать черновики там, где разрешены только опубликованные статьи, выдавая ошибку компилятора.

Давайте рассмотрим первую часть функции `main` в листинге 17.11:

src/main.rs

```
fn main() {
    let mut post = Post::new();

    post.add_text("Сегодня на обед я ел салат");
    assert_eq!("", post.content());
}
```

Мы по-прежнему можем создавать новые статьи в черновом состоянии с помощью функции `Post::new`, а также добавлять текст в пост. Но у черновика-поста не будет метода `content`, который возвращает пустую строку. Вместо этого мы сделаем так, чтобы черновые статьи вообще не имели метода `content`. Благодаря этому, если мы попытаемся получить содержимое черновой статьи, то получим ошибку компилятора, сообщающую о том, что данный метод не существует. В результате мы не сможем случайно показать содержимое черновой статьи, потому что этот код даже не будет компилироваться. В листинге 17.19 показано определение структур `Post` и `DraftPost`, а также методы для каждой из них.

Листинг 17.19. Структуры Post с методом content и DraftPost без метода content
src/lib.rs

```
pub struct Post {
    content: String,
}

pub struct DraftPost {
    content: String,
}

impl Post {
 ❶ pub fn new() -> DraftPost {
        DraftPost {
            content: String::new(),
        }
    }

 ❷ pub fn content(&self) -> &str {
        &self.content
    }
}

impl DraftPost {
 ❸ pub fn add_text(&mut self, text: &str) {
        self.content.push_str(text);
    }
}
```

Структуры `Post` и `DraftPost` имеют приватное поле `content`, в котором хранится текст поста. Указанные структуры больше не имеют поля `state`, потому что мы перемещаем кодировку состояния в типы структур. Структура `Post` будет представлять опубликованный пост. У нее есть метод `content`, который возвращает в ❷ содержимое поля `content`.

У нас по-прежнему есть функция `Post::new`, но вместо того, чтобы возвращать экземпляр структуры `Post`, она возвращает экземпляр структуры `DraftPost` ❶. Поскольку поле `content` приватное и нет функций, которые возвращают `Post`, невозможно создать экземпляр структуры `Post` прямо сейчас.

Структура `DraftPost` имеет метод `add_text`, поэтому мы можем добавлять текст в поле `content`, как раньше ❸. Но обратите внимание, что у `DraftPost` не определен метод `content`! Таким образом, теперь программа делает так, что все статьи начинаются как черновые, а черновые статьи не имеют содержимого, доступного для показа. Любая попытка обойти эти ограничения приведет к ошибке компилятора.

Реализация переходов как трансформаций в разные типы

Тогда как получить опубликованную статью? Мы хотим, чтобы соблюдалось правило, согласно которому черновая статья должна быть проверена и одобрена до

публикации. Пост в состоянии ожидания проверки по-прежнему не должен иметь никакого содержимого. Давайте выполним эти ограничения, добавив еще одну структуру `PendingReviewPost`, определив метод `request_review` для `DraftPost`, который возвращает `PendingReviewPost`, и метод `approve` для `PendingReviewPost`, возвращающий `Post`, как показано в листинге 17.20.

Листинг 17.20. Структура PendingReviewPost, созданная вызовом request_review для DraftPost, и метод approve, который превращает PendingReviewPost в опубликованную статью Post

src/lib.rs

```
impl DraftPost {
    // --пропуск--
    pub fn request_review(self) -> PendingReviewPost {
        PendingReviewPost {
            content: self.content,
        }
    }
}

pub struct PendingReviewPost {
    content: String,
}

impl PendingReviewPost {
    pub fn approve(self) -> Post {
        Post {
            content: self.content,
        }
    }
}
```

Методы `request_review` и `approve` берут `self` во владение, тем самым поглощая экземпляры `DraftPost` и `PendingReviewPost` и трансформируя их соответственно в `PendingReviewPost` и опубликованный пост `Post`. Благодаря этому у нас не будет затянувшихся экземпляров `DraftPost` после вызова для них `request_review` и так далее. Для структуры `PendingReviewPost` не определен метод `content`, поэтому попытка прочитать содержимое этой структуры приводит к ошибке компилятора, как и в случае с `DraftPost`. Поскольку единственный способ получить экземпляр опубликованной статьи `Post`, у которой все-таки определен метод `content`, состоит в вызове метода `approve` для `PendingReviewPost`, а единственный способ получить `PendingReviewPost` — вызвать метод `request_review` для `DraftPost`, теперь мы закодировали процесс создания поста в систему типов.

Но мы также должны внести небольшие изменения в функцию `main`. Методы `request_review` и `approve` возвращают новые экземпляры вместо того, чтобы модифицировать структуру, для которой они вызываются, поэтому нужно добавить больше затеняющих присваиваний `let post =`, чтобы сохранять возвращаемые экземпляры. Мы также не можем иметь проверочные утверждения, в которых со-

держимое черновика и поста, ожидающего проверки, является пустыми строками, вместе с тем они нам и не нужны. Больше нельзя компилировать код, который пытается использовать содержимое постов в этих состояниях. Обновленный код в функции `main` показан в листинге 17.21.

Листинг 17.21. Изменения в функции main для новой реализации процесса создания поста

src/main.rs

```
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("Сегодня на обед я ел салат");

    let post = post.request_review();

    let post = post.approve();

    assert_eq!("Сегодня на обед я ел салат", post.content());
}
```

Изменения, которые нужно было внести в функцию `main` для переназначения переменной `post`, означают, что эта реализация больше не соответствует объектно-ориентированному паттерну переходов между состояниями в полной мере. Трансформации между состояниями больше не инкапсулируются полностью в структуре `Post`. Однако преимущество заключается в том, что недопустимые состояния теперь невозможны из-за системы типов и их проверки, которая происходит во время компиляции! Благодаря этому некоторые дефекты, такие как показ содержимого неопубликованного поста, будут обнаружены до того, как попадут в продакшен.

Опробуйте предложенные задачи о дополнительных требованиях, которые мы упомянули в начале этого раздела об упаковке `blog` после листинга 17.20, и оцените дизайн этой версии кода. Обратите внимание, что некоторые задачи могли бы быть выполнены уже в таком дизайне.

Мы видели, что даже если в Rust применимы объектно-ориентированные паттерны проектирования, другие паттерны, такие как кодирование состояния в систему типов, также в нем доступны. У этих паттернов разные преимущества. Хотя вы, возможно, хорошо знакомы с объектно-ориентированными паттернами, все-таки посмотрите на задачи с другой стороны и воспользуйтесь преимуществами средств Rust, благодаря которым, к примеру, можно предотвратить некоторые дефекты во время компиляции. Объектно-ориентированные паттерны не всегда будут лучшим решением в Rust из-за таких средств, как владение, которых нет в объектно-ориентированных языках.

Итоги

Независимо от того, считаете вы Rust объектно-ориентированным или нет, прочитав эту главу, вы узнали, что можно использовать типажные объекты для получения некоторых объектно-ориентированных средств в Rust. Динамическая диспетчеризация дает коду некоторую гибкость в обмен на крохотное снижение производительности времени выполнения. Вы можете использовать эту гибкость для применения объектно-ориентированных паттернов, которые будут поддерживать ваш код. В Rust также есть другие средства, такие как владение, которых нет в объектно-ориентированных языках. Объектно-ориентированный паттерн не всегда лучший способ воспользоваться преимуществами Rust, но к этому варианту всегда можно прибегнуть.

Далее мы рассмотрим паттерны Rust, обеспечивающие очень большую гибкость. Мы обращались к ним на протяжении всей книги, но пока еще не видели всех их возможностей. Поехали!

18 Паттерны и сопоставление

Паттерны — это специальный синтаксис в языке Rust для сопоставления со структурой сложных и простых типов. Использование паттернов в сочетании с выражениями `match` и другими конструкциями дает больше контроля над управлением потоком в программе. Паттерн состоит из той или иной комбинации следующих элементов:

- Литералы.
- Деструктурированные массивы, перечисления, структуры или кортежи.
- Переменные.
- Метасимволы.
- Заполнители.

Эти компоненты описывают форму данных, с которыми мы работаем и которые затем сопоставляем со значениями, чтобы выяснить, правильные ли данные у программы для выполнения того или иного фрагмента кода.

Для того чтобы применить паттерн, мы сравниваем его со значением. Если паттерн совпадает с этим значением, то мы используем части значения в коде. Вспомните выражения `match` из главы 6, в которых использовались паттерны, такие как пример машины для сортировки монет. Если значение укладывается в форму паттерна, то мы используем названные части. В противном случае код, связанный с паттерном, выполняться не будет.

Это глава-справочник, посвященная паттернам. Мы поговорим о местах, где допустимы паттерны, обсудим разницу между опровержимыми и неопровержимыми паттернами и разные виды синтаксиса паттернов, которые вы, возможно, будете встречать. К концу главы вы узнаете, как использовать паттерны, чтобы ясно выразить многие понятия.

Где могут использоваться паттерны

В Rust паттерны много где появляются, и вы часто их использовали, даже не осознавая этого! В данном разделе рассматриваются ситуации, в которых паттерны допустимы.

Ветви выражения match

Как обсуждалось в главе 6, мы используем паттерны в ветвях выражений `match`. Формально выражения `match` определяются как ключевое слово `match`, затем значение для сопоставления и один или несколько ветвей совпадения, состоящих из паттерна и выполняемого выражения, если значение совпадает с паттерном этой ветви, например:

```
match ЗНАЧЕНИЕ {
    ПАТТЕРН => ВЫРАЖЕНИЕ,
    ПАТТЕРН => ВЫРАЖЕНИЕ,
    ПАТТЕРН => ВЫРАЖЕНИЕ,
}
```

Одно из требований к выражениям `match` состоит в том, что они должны быть исчерпывающими в том смысле, что в `match` должны быть учтены все возможные значения. Чтобы вы учитывали все возможные варианты, нужно иметь всеохватывающий паттерн в последней ветви: например, имя переменной, совпадающее с любым значением, всегда будет срабатывать и, таким образом, охватывать все оставшиеся случаи.

Особый паттерн `_` будет совпадать с чем угодно, но он не привязывается к переменной и поэтому часто используется в последнем рукаве совпадения. Паттерн `_` бывает полезен, например, если вы хотите проигнорировать любое неуказанное значение. Мы рассмотрим паттерн `_` подробнее в разделе «Игнорирование значений в паттерне».

Условные выражения if let

В главе 6 мы обсуждали выражения `if let` главным образом как более краткий способ написания эквивалента выражения `match`, который совпадает только с одним случаем. Как вариант `if let` может иметь соответствующий `else`, содержащий выполняемый код, если паттерн в выражении `if let` не совпадает.

Листинг 18.1 показывает, что также существует возможность смешивать и сочетать выражения `if let`, `else if` и `else if let`. Благодаря этому мы получаем больше гибкости, чем при использовании выражения `match`, в котором можно выразить только одно сравниваемое с паттернами значение. Кроме того, условия в серии выражений `if let`, `else if` и `else if let` не обязательно должны относиться друг к другу.

Код в листинге 18.1 показывает серию проверок на несколько условий, которые решают, каким должен быть фоновый цвет. В этом примере мы создали переменные с жестко заданными значениями, которые реальная программа могла бы получить из данных, вводимых пользователем.

Листинг 18.1. Смешивание выражений if let, else if, else if let и else

src/main.rs

```
fn main() {
    let favorite_color: Option<&str> = None;
    let is_tuesday = false;
    let age: Result<u8, _> = "34".parse();

 ❶ if let Some(color) = favorite_color {
    ❷ println!("Используя ваш любимый цвет, {}, в качестве фона", color);
 ❸ } else if is_tuesday {
    ❹ println!("Вторник - зеленый день!");
 ❺ } else if let Ok(age) = age {
    ❻ if age > 30 {
        ❼ println!("Использование фиолетового цвета в качестве фона");
       } else {
        ❽ println!("Использование оранжевого цвета в качестве фона");
       }
 ❾ } else {
    ❿ println!("Использование синего цвета в качестве фона");
    }
}
```

Если пользователь указывает любимый цвет ❶, то это фоновый цвет ❷. Если сегодня — вторник ❸, то фоновый цвет — зеленый ❹. Если пользователь указывает свой возраст в качестве строки и мы можем успешно разобрать ее как число ❺, то цвет будет либо фиолетовым ❼, либо оранжевым ❽ в зависимости от значения числа ❻. Если ни одно из этих условий не применимо ❾, то фоновый цвет — синий ❿.

Эта условная структура позволяет поддерживать сложные требования. С жестко закодированными значениями, которые здесь есть, этот пример будет выводить

```
Использование фиолетового цвета в качестве фона.
```

Вы видите, что выражение `if let` может также вводить затененные переменные таким же образом, как и рукава выражения `match`: строка кода `if let Ok(age) = age` ❺ вводит новую затененную переменную `age`, содержащую значение внутри варианта `Ok`. Это означает, что нужно поместить условие `if age > 30` в этот блок ❻: мы не можем совместить эти два условия в выражении `if let Ok (age) = age && age > 30`. Затененная переменная `age`, которую мы хотим сравнить с 30, будет недействительна до тех пор, пока новая область видимости не начнется с фигурной скобки.

Недостатком использования выражений `if let` является то, что компилятор не проверяет исчерпываемость, в то время как с выражениями `match` он это делает.

Если бы мы пропустили последний блок `else` ❾ и, следовательно, обработку некоторых случаев, то компилятор не предупредил бы нас о возможной логической ошибке.

Условные циклы while let

Похожий по конструкции с выражением `if let` условный цикл `while let` позволяет циклу `while` работать до тех пор, пока паттерн совпадает. Пример в листинге 18.2 показывает цикл `while let`, который использует вектор в качестве стека и выводит значения в векторе в порядке, обратном тому, в котором они были добавлены.

Листинг 18.2. Использование цикла while let для печати значений, пока метод stack.pop() возвращает Some

```
let mut stack = Vec::new();

stack.push(1);
stack.push(2);
stack.push(3);

while let Some(top) = stack.pop() {
    println!("{}", top);
}
```

В этом примере выводятся 3, 2 и затем 1. Метод `pop` берет последний элемент из вектора и возвращает `Some(value)`. Если вектор пуст, то `pop` возвращает `None`. Цикл `while` продолжает выполнение кода в своем блоке до тех пор, пока `pop` возвращает `Some`. Когда `pop` возвращает `None`, цикл останавливается. Мы можем использовать условный цикл `while let`, чтобы удалить каждый элемент из стека.

Циклы for

В главе 3 мы упоминали, что цикл `for` — это наиболее распространенная циклическая конструкция в коде на языке Rust, но мы еще не обсуждали паттерн, который берет `for`. В цикле `for` паттерном является значение, следующее непосредственно за ключевым словом `for`, поэтому в `for x in y` паттерном является `x`.

Листинг 18.3 показывает использование паттерна в цикле `for` для деструктурирования, или разложения, кортежа в рамках `for`.

Листинг 18.3. Использование паттерна в цикле `for` для деструктурирования кортежа

```
let v = vec!['a', 'b', 'c'];

for (index, value) in v.iter().enumerate() {
    println!("{} находится в индексе {}", value, index);
}
```

Код из листинга 18.3 выводит следующее:

```
a находится в индексе 0
b находится в индексе 1
c находится в индексе 2
```

Мы используем метод `enumerate`, чтобы переделать итератор для порождения значения и индекса этого значения в итераторе, помещенных в кортеж. Первый вызов метода `enumerate` порождает кортеж `(0, 'a')`. Когда это значение сочетается с паттерном `(index, value)`, то `index` равен 0, а `value` равно `'a'`, выводится первая строка данных.

Инструкции let

До этой главы мы прямо обсуждали использование паттернов только с выражениями `match` и `if let`, но на самом деле мы использовали паттерны и в других местах, в том числе в инструкциях `let`. Рассмотрим простую передачу значения переменной с помощью `let`:

```
let x = 5;
```

На протяжении всей книги мы сотни раз использовали инструкции `let` подобного рода, и хотя вы, возможно, не осознавали этого, вы использовали паттерны! В более формальном плане инструкция `let` выглядит так:

```
let ПАТТЕРН = ВЫРАЖЕНИЕ;
```

В таких инструкциях, как `let x = 5;`, с именем переменной в слоте `ПАТТЕРН`, имя переменной — это всего лишь простая форма паттерна. Rust сравнивает выражение с паттерном и назначает любые имена, которые он находит. Поэтому в примере `let x = 5;` паттерном является `x`, который означает «связать то, что совпадает здесь, с переменной `x`». Поскольку имя `x` представляет весь паттерн, то этот паттерн фактически означает «связать все с переменной `x`, каким бы ни было значение».

Чтобы четче увидеть сопоставление с паттерном инструкции `let`, рассмотрим листинг 18.4, который использует паттерн с `let` для деструктурирования кортежа.

Листинг 18.4. Использование паттерна для деструктурирования кортежа и создания сразу трех переменных

```
let (x, y, z) = (1, 2, 3);
```

Здесь мы сопоставляем кортеж с паттерном. Rust сравнивает `(1, 2, 3)` с `(x, y, z)` и видит, что это значение совпадает с паттерном, поэтому Rust связывает 1 с `x`, 2 с `y` и 3 с `z`. Можно думать об этом кортежном паттерне как о вложении в него трех отдельных паттернов переменных.

Если число элементов в паттерне не совпадает с числом элементов в кортеже, то совокупный тип не будет совпадать и мы получим ошибку компилятора. Напри-

мер, листинг 18.5 показывает попытку деструктурирования кортежа из трех элементов в две переменные, которая не будет работать.

Листинг 18.5. Неправильное построение паттерна, переменные которого не совпадают с числом элементов в кортеже

```
let (x, y) = (1, 2, 3);
```

Попытка компиляции этого кода приводит к ошибке типа:

```
error[E0308]: mismatched types
 --> src/main.rs:2:9
  |
2 |     let (x, y) = (1, 2, 3);
  |         ^^^^^^ expected a tuple with 3 elements, found one with 2 elements
  |
  = note: expected type `({integer}, {integer}, {integer})`
             found type `(_, _)`
```

Если бы мы хотели проигнорировать одно или несколько значений в кортеже, то могли бы использовать `_` или `..`, как вы увидите в разделе «Игнорирование значений в паттерне». Если проблема состоит в том, что в паттерне слишком много переменных, то нужно сделать так, чтобы типы совпадали, удалив переменные, чтобы число переменных равнялось числу элементов в кортеже.

Параметры функций

Параметры функций также могут быть паттернами. Код в листинге 18.6, объявляющий функцию `foo`, которая берет один параметр `x` типа `i32`, теперь вам знаком.

Листинг 18.6. Сигнатура функции использует паттерны в параметрах

```
fn foo(x: i32) {
    // здесь будет код
}
```

Часть `x` — это паттерн! Как и в случае с `let`, мы можем сопоставить кортеж в аргументах функции с паттерном. Листинг 18.7 разбивает значения в кортеже в момент, когда мы передаем его внутрь функции.

Листинг 18.7. Функция с параметрами, которые деструктурируют кортеж

src/main.rs

```
fn print_coordinates(&(x, y): &(i32, i32)) {
    println!("Текущее местоположение: ({}, {})", x, y);
}

fn main() {
    let point = (3, 5);
    print_coordinates(&point);
}
```

Этот код выводит

```
Текущее местоположение: (3, 5)
```

Значения `&(3, 5)` совпадают с паттерном `&(x, y)`, поэтому `x` равно 3, а `y` — 5.

Кроме того, мы можем использовать паттерны в списках параметров замыкания таким же образом, как и в списках параметров функций, поскольку замыкания похожи на функции, как описано в главе 13.

Вы уже увидели несколько способов использования паттернов, но они не работают одинаково всюду, где можно их применить. В некоторых ситуациях эти паттерны должны быть неопровержимы, в других — они могут быть опровержимы. Далее мы обсудим эти два понятия.

Опровержимость: возможность несовпадения паттерна

Паттерны бывают двух видов: опровержимые и неопровержимые. Паттерны, которые будут совпадать с любым возможным переданным значением, неопровержимые. Примером является `x` в инструкции `let x = 5;`, потому что `x` совпадает абсолютно со всем и, следовательно, не может не совпасть. Паттерны, которые не совпадают с некоторыми возможными значениями, являются опровержимыми. Примером служит `Some(x)` в выражении `if let Some(x) = a_value`, потому что, если значение в переменной `a_value` равно `None`, а не `Some`, то паттерн `Some(x)` не совпадет.

Параметры функций, инструкции `let` и циклы `for` могут принимать только неопровержимые паттерны, поскольку программа не сможет делать ничего значимого, когда значения не совпадают. Выражения `if let` и `while let` принимают только опровержимые паттерны, поскольку по определению они предназначены для обработки возможной ошибки: функциональность условного выражения заключается в его способности выполнять разные действия в зависимости от успеха или провала.

В целом вам не следует беспокоиться о различии между опровержимыми и неопровержимыми паттернами. Однако вам все-таки нужно знать о понятии опровержимости, чтобы реагировать при виде его в сообщении об ошибке. В этих случаях вам потребуется изменить либо паттерн, либо конструкцию, с которой вы используете паттерн, в зависимости от предполагаемого поведения кода.

Давайте рассмотрим, что происходит, когда мы пытаемся использовать опровержимый паттерн в месте, где Rust требует неопровержимый паттерн, и наоборот. Листинг 18.8 показывает инструкцию `let`, но для паттерна мы задали `Some(x)`, опровержимый паттерн. Как и следовало ожидать, этот код не компилируется.

Листинг 18.8. Попытка использовать опровержимый паттерн с let

```
let Some(x) = some_option_value;
```

Если бы значение `some_option_value` было равно `None`, то оно не совпало бы с паттерном `Some(x)`, то есть паттерн является опровержимым. Однако инструкция `let` может принимать только неопровержимый паттерн, поскольку код не может сделать ничего допустимого со значением `None`. Во время компиляции язык Rust будет жаловаться, что мы пытались использовать опровержимый паттерн там, где требуется неопровержимый паттерн[1]:

```
error[E0005]: refutable pattern in local binding: `None` not covered
 -->
  |
3 | let Some(x) = some_option_value;
  |     ^^^^^^^ pattern `None` not covered
```

Поскольку мы не охватили (и не могли охватить!) каждое допустимое значение с паттерном `Some(x)`, Rust по праву выдает ошибку компилятора.

Для устранения проблемы, когда у нас опровержимый паттерн вместо неопровержимого, мы можем изменить код, использующий паттерн: вместо `let` можно применить `if let`. Тогда, если паттерн не совпадает, то код в фигурных скобках будет пропущен и работа продолжится корректно. Листинг 18.9 показывает, как исправить код из листинга 18.8.

Листинг 18.9. Использование выражения if let и блока с опровержимыми паттернами вместо let

```
if let Some(x) = some_option_value {
    println!("{}", x);
}
```

Код готов! Это абсолютно правильный код, хотя он означает, что мы не можем использовать неопровержимый паттерн без ошибки. Если мы дадим выражению `if let` паттерн, который всегда будет совпадать, например `x`, как показано в листинге 18.10, то он компилироваться не будет.

Листинг 18.10. Попытка использовать неопровержимый паттерн с выражением `if let`

```
if let x = 5 {
    println!("{}", x);
};
```

Компилятор жалуется, что использовать выражение `if let` с неопровержимым паттерном не имеет смысла[2]:

---

[1] `ошибка[E0005]: не охвачен опровержимый паттерн в локальной привязке `None``

[2] `ошибка[E0162]: неопровержимый паттерн if-let`

```
error[E0162]: irrefutable if-let pattern
 --> <anon>:2:8
  |
2 | if let x = 5 {
  |        ^ irrefutable pattern
```

По этой причине рукава выражения `match` должны использовать опровержимые паттерны, за исключением последнего рукава, который должен сопоставлять любые оставшиеся значения с неопровержимым паттерном. Rust позволяет использовать неопровержимый паттерн в выражении `match` только с одним рукавом, но этот синтаксис не особо полезен, и его можно заменить более простой инструкцией `let`.

Теперь, когда вы знаете, где используются паттерны и чем отличаются опровержимые и неопровержимые паттерны, давайте познакомимся с синтаксисом, который мы можем использовать для создания паттернов.

Синтаксис паттернов

В процессе чтения книги вы видели примеры многих видов паттернов. В этом разделе мы соберем весь синтаксис, допустимый в паттернах, и обсудим, почему вы можете использовать каждый из них.

Сопоставление литералов

В главе 6 вы видели, что можно напрямую сопоставлять паттерны с литералами. В следующем коде приводится несколько примеров:

```
let x = 1;

match x {
    1 => println!("один"),
    2 => println!("два"),
    3 => println!("три"),
    _ => println!("что угодно"),
}
```

Этот код выводит `один`, потому что значение в `x` равно `1`. Данный синтаксис полезен, когда вы хотите, чтобы код выполнял действие, если он получает то или иное конкретное значение.

Сопоставление именованных переменных

Именованные переменные — это неопровержимые паттерны, которые совпадают с любым значением, мы применяли их в этой книге неоднократно. Однако при использовании именованных переменных в выражениях `match` возникают сложности. Поскольку ключевое слово `match` начинает новую область видимости,

переменные, объявляемые как часть паттерна внутри выражения `match`, будут затенять переменные с тем же именем вне конструкции `match`, как и в случае со всеми переменными. В листинге 18.11 мы объявляем переменную `x` со значением `Some(5)` и переменную `y` со значением 10. Затем мы создаем выражение `match` для значения `x`. Посмотрите на паттерны в рукавах выражения `match` и в макрокоманде `println!` в конце и попытайтесь выяснить, что этот код будет печатать, прежде чем его выполнить или читать дальше.

Листинг 18.11. Выражение match с рукавом, который вводит затененную переменную y

src/main.rs

```
fn main() {
  ❶ let x = Some(5);
  ❷ let y = 10;

    match x {
      ❸ Some(50) => println!("Получено 50"),
      ❹ Some(y) => println!("Совпадение, y = {:?}", y),
      ❺ _ => println!("Вариант по умолчанию, x = {:?}", x),
    }

  ❻ println!("в конце: x = {:?}, y = {:?}", x, y);
}
```

Давайте посмотрим, что происходит, когда выполняется выражение `match`. Паттерн в первом рукаве совпадения ❸ не совпадает с определенным значением `x` ❶, поэтому код продолжает работу.

Паттерн во втором рукаве совпадения ❹ вводит новую переменную с именем `y`, которая будет совпадать с любым значением внутри значения `Some`. Поскольку мы находимся в новой области видимости внутри выражения `match`, эта переменная `y` является новой, а не той `y`, которую мы объявили в начале со значением 10 ❷. Эта новая привязка `y` будет совпадать с любым значением внутри `Some`, которое мы имеем в `x`. Следовательно, эта новая `y` связывается с внутренним значением `Some` в `x`. Указанное значение равно 5, поэтому выражение для данного рукава исполняется и выводит `Совпадение, y = 5`.

Если бы значение переменной `x` было равно `None` вместо `Some(5)`, то паттерны в первых двух рукавах не совпали бы, поэтому значение совпало бы с подчеркиванием ❺. Мы не ввели переменную `x` в паттерн рукава подчеркивания, поэтому `x` в выражении по-прежнему является внешним `x`, который не был затенен. В этом гипотетическом случае выражение `match` выведет:

```
Вариант по умолчанию, x = None.
```

Когда выражение `match` завершает работу, его область видимости заканчивается, и так же заканчивается область внутренней переменной `y`. Последняя инструкция `println!` ❻ выводит `в конце: x = Some(5), y = 10`.

Для того чтобы создать выражение `match`, которое сравнивает значения внешних `x` и `y`, вместо ввода затененной переменной нам нужно было бы применить условия с ограничителями совпадений. Мы поговорим об ограничителях совпадений позже, в разделе «Дополнительные условия с ограничителями совпадений».

Несколько паттернов

В выражениях `match` можно сопоставлять несколько паттернов с помощью синтаксиса `|`, который означает *или*. Например, код ниже сопоставляет значение `x` с рукавами совпадения, первый из которых имеет вариант *или*, то есть если значение `x` совпадает с любым из значений в этом рукаве, то будет выполняться код этого рукава:

```
let x = 1;

match x {
    1 | 2 => println!("один или два"),
    3 => println!("три"),
    _ => println!("что угодно"),
}
```

Этот код выводит `один или два`.

Сопоставление интервалов значений с помощью синтаксиса ...

Синтаксис `...` позволяет сопоставлять с широким интервалом значений. В следующем коде, когда паттерн совпадает с любым из значений в пределах интервала, этот рукав будет исполнен:

```
let x = 5;

match x {
    1 ... 5 => println!("с одного до пяти"),
    _ => println!("что-то еще \"),
}
```

Если `x` равно 1, 2, 3, 4 или 5, то будет совпадать первый рукав. Этот синтаксис более удобен, чем использование оператора `|` для выражения той же идеи. Вместо `1 ... 5` нам пришлось бы задать `1 | 2 | 3 | 4 | 5`, если бы мы использовали `|`. Описание интервала будет намного короче, в особенности если мы хотим сопоставить, скажем, любое число между 1 и 1000!

Интервалы допускаются только с числовыми значениями или значениями `char`, поскольку компилятор проверяет, не пуст ли интервал во время компиляции. Единственные типы, для которых Rust может различить пустые и непустые интервалы, — это значения `char` и числа.

Вот пример использования интервалов значений `char`:

```
let x = 'c';

match x {
    'a' ... 'j' => println!("буква вверху ASCII"),
    'k' ... 'z' => println!("буква внизу ASCII"),
    _ => println!("что-то еще"),
}
```

Язык Rust может различить, что `c` находится внутри интервала первого паттерна, и выводит `буква вверху ASCII`.

Деструктурирование для выделения значений

Мы также можем использовать паттерны для деструктурирования структур, перечислений, кортежей и ссылок, чтобы использовать разные части этих значений. Давайте разберем каждое значение.

Деструктурирование

Листинг 18.12 показывает структуру `Point` с двумя полями `x` и `y`, которую можно разбить с помощью паттерна с инструкцией `let`.

Листинг 18.12. Деструктурирование полей структуры на отдельные переменные

src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x: a, y: b } = p;
    assert_eq!(0, a);
    assert_eq!(7, b);
}
```

Приведенный код создает переменные `a` и `b`, которые совпадают со значениями полей `x` и `y` структуры `p`. Этот пример показывает, что имена переменных в паттерне не обязательно должны совпадать с именами полей структуры. Но обычно требуется, чтобы имена переменных совпадали с именами полей. Это нужно для того, чтобы было легче запомнить, какие переменные пришли из каких полей.

Поскольку сопоставление имен переменных с полями встречается часто и написание `let Point { x: x, y: y } = p;` содержит много повторов, для паттернов, которые сочетаются с полями структур, существует сокращенная форма. Нужно

задать только имена полей структуры, а переменные, создаваемые из паттерна, будут иметь те же имена. Листинг 18.13 показывает код, который ведет себя точно так же, как и код в листинге 18.12, но переменными, создаваемыми в паттерне `let`, являются `x` и `y` вместо `a` и `b`.

Листинг 18.13. Деструктурирование полей структуры с помощью сокращенной формы записи полей структуры

src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x, y } = p;
    assert_eq!(0, x);
    assert_eq!(7, y);
}
```

Этот код создает переменные `x` и `y`, которые совпадают с полями `x` и `y` переменной `p`. В результате переменные `x` и `y` содержат значения из структуры `p`.

Мы также можем выполнять деструктурирование с литеральными значениями в рамках структурного паттерна, а не создавать переменные для всех полей. Это позволяет проверять некоторые поля на наличие определенных значений при создании переменных для деструктурирования других полей.

Листинг 18.14 показывает выражение `match`, которое делит значения структуры `Point` на три случая: точки, лежащие непосредственно на оси `x` (что верно при `y = 0`), на оси `y` (`x = 0`) или ни на одной из них.

Листинг 18.14. Деструктурирование и сопоставление литеральных значений в одном паттерне

src/main. rs

```
fn main() {
    let p = Point { x: 0, y: 7 };

    match p {
        Point { x, y: 0 } => println!("На оси x в точке {}", x),
        Point { x: 0, y } => println!("На оси y в точке {}", y),
        Point { x, y } => println!("Ни на одной оси: ({}, {})", x, y),
    }
}
```

Первый рукав будет совпадать с любой точкой, лежащей на оси `x`, с указанием, что поле `y` совпадает, если его значение совпадает с литералом 0. Паттерн по-

прежнему создает переменную `x`, которую мы можем использовать в коде для этого рукава.

Схожим образом второй рукав совпадает с любой точкой на оси `y` с указанием, что поле `x` совпадает, если его значение равно 0, и создает переменную `y` для значения поля `y`. Третий рукав не конкретизирует никаких литералов, поэтому он совпадает с любой точкой и создает переменные для полей `x` и `y`.

В этом примере значение `p` совпадает со вторым рукавом в силу того, что `x` содержит `0`, поэтому этот код выведет:

```
На оси y в точке 7
```

Деструктурирование перечислений

Ранее мы уже делали деструктурирование перечислений, например, когда деструктурировали `Option<i32>` в листинге 6.5. Мы не упомянули, что паттерн для деструктурирования перечисления должен соответствовать тому, как определены данные, хранящиеся внутри перечисления. Например, в листинге 18.15 мы используем перечисление `Message` из листинга 6.2 и записываем выражение `match` с паттернами, которые будут деструктурировать каждое внутреннее значение.

Листинг 18.15. Деструктурирование вариантов перечисления, содержащих разные виды значений

src/main. rs

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

fn main() {
  ❶ let msg = Message::ChangeColor(0, 160, 255);

    match msg {
      ❷ Message::Quit => {
            println!("Вариант Quit не имеет данных для деструктурирования.")
        },
      ❸ Message::Move { x, y } => {
            println!(
                "Переместить в направлении x {} и в направлении y {}",
                x,
                y
            );
        }
      ❹ Message::Write(text) => println!("Текстовое сообщение: {}", text),
      ❺ Message::ChangeColor(r, g, b) => {
            println!(
                "Поменять цвет на красный {}, зеленый {} и синий {}",
```

```
                r,
                g,
                b
            )
        }
    }
}
```

Приведенный код выведет:

```
Поменять цвет на красный 0, зеленый 160 и синий 255
```

Попробуйте изменить значение `msg` ❶ и посмотрите, как будет выполняться код из других рукавов.

В случае вариантов перечисления без каких-либо данных, таких как `Message::Quit` ❷, мы не можем деструктурировать значение дальше. Можно выполнить сопоставление только для литерального значения `Message::Quit`, в этом паттерне никаких переменных нет.

В случае с вариантами перечисления, подобными структуре, такими как `Message::Move` ❸, мы можем использовать паттерн, похожий на тот, который указывается для сопоставления структур. После имени варианта мы помещаем фигурные скобки, а затем перечисляем поля с переменными, чтобы разложить части, используемые в коде для этого рукава. Здесь мы используем сокращенную форму, как и в листинге 18.13.

В случае с вариантами перечисления, подобными кортежу, такими как `Message::Write`, содержащий кортеж с одним элементом ❹ и `Message::ChangeColor`, который содержит кортеж с тремя элементами ❺, этот паттерн похож на тот, который мы указываем для сопоставления кортежей. Число переменных в паттерне должно совпадать с числом элементов в варианте, который мы сопоставляем.

Деструктурирование вложенных структур и перечислений

До сих пор во всех примерах сопоставлялись структуры или перечисления, которые были на один уровень в глубину. Сопоставление может работать и с вложенными элементами!

Например, можно выполнить рефакторинг кода из листинга 18.15 для поддержки цветов RGB и HSV в сообщении `ChangeColor`, как показано в листинге 18.16.

Листинг 18.16. Сопоставление вложенных перечислений

```
enum Color {
   Rgb(i32, i32, i32),
   Hsv(i32, i32, i32)
}

enum Message {
```

```
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(Color),
}

fn main() {
    let msg = Message::ChangeColor(Color::Hsv(0, 160, 255));

    match msg {
        Message::ChangeColor(Color::Rgb(r, g, b)) => {
            println!(
                "Поменять цвет на красный {}, зеленый {} и синий {}",
                r,
                g,
                b
            )
        },
        Message::ChangeColor(Color::Hsv(h, s, v)) => {
            println!(
                "Поменять цвет на тон {}, насыщенность {} и яркость {}",
                h,
                s,
                v
            )
        }
        _ => ()
    }
}
```

Паттерн первого рукава в выражении `match` совпадает с вариантом `Message::ChangeColor` перечисления, содержащим вариант `Color::Rgb`. Затем паттерн привязывается к трем внутренним значениям типа `i32`. Паттерн второго рукава также совпадает с вариантом `Message::ChangeColor` перечисления, но внутреннее перечисление совпадает с вариантом `Color::Hsv`. Мы можем указать эти сложные условия в одном выражении `match`, хотя здесь задействованы только два перечисления.

Деструктурирование структур и кортежей

Мы можем смешивать, сопоставлять и вкладывать деструктурирующие паттерны еще более сложными способами. Следующий пример показывает сложное деструктурирование, в котором мы вкладываем структуры и кортежи внутрь кортежа и деструктурируем все примитивные значения:

```
let ((feet, inches), Point {x, y}) = ((3, 10), Point { x: 3, y: -10 });
```

Этот код позволяет разбивать сложные типы на составные части, в результате чего мы можем использовать интересующие нас значения по отдельности.

За счет деструктурирования с помощью паттернов удобно использовать отдельно друг от друга фрагменты значений, например значений из каждого поля в структуре.

Игнорирование значений в паттерне

Вы видели, что иногда полезно игнорировать значения в паттерне, например в последнем рукаве выражения `match`, в результате получая всеохватывающий случай, который на самом деле ничего не делает, но учитывает все остальные возможные значения. Существует несколько способов игнорирования целых значений или частей значений в паттерне: использование паттерна `_` (который вы видели), паттерна `_` в другом паттерне, имени, начинающегося с подчеркивания, либо использование `..` для игнорирования оставшихся частей значения. Давайте рассмотрим, как и почему используются эти паттерны.

Игнорирование всего значения с помощью _

Мы использовали символ подчеркивания (`_`) в качестве подстановочного знака, который будет совпадать с любым значением, но он не будет привязан к нему. Хотя паттерн `_` особенно полезен в качестве последнего рукава в выражении `match`, мы можем использовать его в любом паттерне, включая параметры функций, как показано в листинге 18.17.

Листинг 18.17. Использование _ в сигнатуре функции

src/main. rs

```
fn foo(_: i32, y: i32) {
    println!("Этот код использует только параметр: {}", y);
}

fn main() {
    foo(3, 4);
}
```

Этот код будет полностью игнорировать значение, переданное в качестве первого аргумента 3, и выведет

```
Этот код использует только параметр: 4
```

В большинстве случаев, когда вам больше не нужен конкретный параметр функции, можно изменить сигнатуру так, чтобы она не включала неиспользуемый параметр. Игнорирование параметра функции может быть особенно полезно в некоторых случаях, например во время реализации типажа, когда вам нужна сигнатура некоторого типа, но тело функции в реализации не нуждается в одном из параметров. В таком случае компилятор не будет предупреждать о неиспользуемых параметрах функции, как это было бы, если бы вы использовали имя.

Игнорирование частей значения с помощью вложенных _

Мы также можем использовать _ внутри еще одного паттерна для игнорирования только части значения. Например, когда мы хотим протестировать только часть значения, но не используем другие части в соответствующем коде, который требуется выполнить. Листинг 18.18 показывает код, ответственный за управление значением настройки. Требования предписывают, что пользователю не разрешается вносить записи поверх существующих настроек. Но он может сбросить параметр и задать ему значение, если оно в данный момент не задано.

Листинг 18.18. Использование подчеркивания в паттернах, которые совпадают с вариантами Some, когда не нужно использовать значение внутри Some

```
let mut setting_value = Some(5);
let new_setting_value = Some(10);

match (setting_value, new_setting_value) {
    (Some(_), Some(_)) => {
        println!("Не получается перезаписать существующее настраиваемое
                  значение");
    }
    _ => {
        setting_value = new_setting_value;
    }
}

println!("значение настройки равно {:?}", setting_value);
```

Этот код выведет:

```
Не получается перезаписать существующее настраиваемое значение
```

а затем

```
значение настройки равно Some(5)
```

В первом рукаве совпадения не нужно сопоставлять или использовать значения внутри варианта `Some`, но требуется выполнить проверку ситуации, когда `setting_value` и `new_setting_value` равны варианту `Some`. В этом случае мы печатаем, почему не изменяем значение `setting_value`, и оно не изменяется.

Во всех остальных случаях (если либо `setting_value`, либо `new_setting_value` равны `None`), выражаемых паттерном _ во втором рукаве, мы хотим разрешать `new_setting_value` становиться `setting_value`.

Мы также можем использовать подчеркивания в нескольких местах внутри одного паттерна, чтобы игнорировать то или иное значение. Листинг 18.19 показывает пример игнорирования второго и четвертого значений в кортеже из пяти элементов.

Листинг 18.19. Игнорирование нескольких частей кортежа

```
let numbers = (2, 4, 8, 16, 32);

match numbers {
    (first, _, third, _, fifth) => {
        println!("Несколько чисел: {}, {}, {}", first, third, fifth)
    },
}
```

Этот код выводит:

```
Несколько чисел: 2, 8, 32
```

Значения `4` и `16` будут проигнорированы.

Игнорирование неиспользуемой переменной, начиная ее имя с _

Если вы создаете переменную, но нигде ее не используете, то Rust обычно выдает предупреждение, потому что это может быть дефектом. Но иногда полезно создать переменную, которую вы пока не будете применять, например, когда вы моделируете прототип или только начинаете проект. В этом случае Rust не будет предупреждать о неиспользуемой переменной, если ее имя будет начинаться с подчеркивания. В листинге 18.20 мы создаем две неиспользуемые переменные, но при выполнении этого кода должно появиться предупреждение только об одной из них.

Листинг 18.20. Написание имени переменной, начинающегося с подчеркивания, во избежание предупреждений о неиспользуемых переменных

src/main. rs

```
fn main() {
    let _x = 5;
    let y = 10;
}
```

Здесь есть предупреждение о том, что мы не используем переменную `y`, но нет предупреждения, что мы не применяем переменную, перед которой стоит символ подчеркивания.

Обратите внимание на небольшое отличие между использованием только _ и имени, которое начинается с подчеркивания. Синтаксис `_x` по-прежнему привязывает значение к переменной, в то время как _ не привязывает вообще. В листинге 18.21 показана ошибка в качестве иллюстрации случая, когда это различие имеет значение.

Листинг 18.21. Неиспользуемая переменная, начинающаяся с подчеркивания, по-прежнему привязывает значение, которое может стать владельцем значения

```
let s = Some(String::from("Здравствуй!"));

if let Some(_s) = s {
```

```
    println!("найдена строка");
}

println!("{:?}", s);
```

Мы получим ошибку, потому что значение переменной `s` все равно будет перенесено в `_s`, что не позволит использовать `s` снова. Однако если применять только подчеркивание, то значение не будет привязываться. Листинг 18.22 будет компилироваться без каких-либо ошибок, потому что `s` не перемещается в `_`.

Листинг 18.22. Использование подчеркивания не привязывает значение

```
let s = Some(String::from("Здравствуй!"));

if let Some(_) = s {
    println!("найдена строка");
}

println!("{:?}", s);
```

Этот код работает без проблем, потому что мы ни к чему не привязываем переменную `s`. Она не перемещается.

Игнорирование оставшихся частей значения с помощью ..

Со значениями, которые имеют много частей, мы можем прибегнуть к синтаксису `..`, чтобы использовать только несколько частей и игнорировать остальные. Таким образом, не нужно перечислять подчеркивания для каждого игнорируемого значения. Паттерн `..` игнорирует любые части значения, которые мы явно не сопоставили в остальной части паттерна. В листинге 18.23 представлена структура `Point`, которая содержит трехмерную координату. В выражении `match` мы хотим работать только с координатами на `x` и игнорировать значения в полях `y` и `z`.

Листинг 18.23. Игнорирование всех полей структуры `Point`, кроме `x`, с использованием `..`

```
struct Point {
    x: i32,
    y: i32,
    z: i32,
}

let origin = Point { x: 0, y: 0, z: 0 };

match origin {
    Point { x, .. } => println!("точка x равна {}", x),
}
```

Мы задаем значение `x`, а затем просто встраиваем паттерн `...` Это быстрее, чем перечислять `y: _` и `z: _`, в особенности когда мы работаем со структурами, где много полей, в ситуациях, когда важно только одно или два поля.

Синтаксис `..` будет расширяться до нужного количества значений. Листинг 18.24 показывает, как использовать `..` с кортежем.

Листинг 18.24. Сопоставление только первого и последнего значений в кортеже и игнорирование всех остальных значений

src/main.rs

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (first, .., last) => {
            println!("Несколько чисел: {}, {}", first, last);
        },
    }
}
```

В этом коде первое и последнее значения сопоставляются с первым `first` и последним `last`. Синтаксис `..` будет сопоставлять и игнорировать значения в середине.

Однако нужно использовать `..` однозначно. Если неясно, какие значения предназначены для сопоставления, а какие следует игнорировать, то Rust выдаст ошибку. Листинг 18.25 показывает пример использования `..` неоднозначным образом, поэтому код не компилируется.

Листинг 18.25. Попытка использования `..` неоднозначным образом

src/main. rs

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (.., second, ..) => {
            println!("Несколько чисел: {}", second)
        },
    }
}
```

Когда мы компилируем этот пример, то получаем такую ошибку[1]:

```
error: `..` can only be used once per tuple or tuple struct pattern
 --> src/main.rs:5:22
  |
5 |         (.., second, ..) => {
  |                      ^^
```

[1] ошибка: `..` может использоваться только один раз в кортеже или кортежно-структурном паттерне

У языка Rust нет возможности выяснить, сколько значений в кортеже следует игнорировать, прежде чем сопоставить значение со вторым `second`, а затем узнать, сколько еще значений проигнорировать после этого. Этот код может означать, что мы хотим проигнорировать 2, привязать переменную `second` к 4, а затем проигнорировать 8, 16 и 32. Либо что мы хотим проигнорировать 2 и 4, привязать `second` к 8, а затем проигнорировать 16 и 32; и так далее. Имя переменной `second` не означает для Rust ничего особенного, поэтому компилятор выдает ошибку — такого рода использование `..` в двух местах является неоднозначным.

Дополнительные условия с ограничителями совпадений

Ограничитель совпадения — это дополнительное условие `if`, задаваемое после паттерна в рукаве выражения `match`, которое, наряду с совпадением с паттерном, также должно совпадать, чтобы рукав был выбран. Ограничители совпадений полезны тем, что они помогают выражать более сложные идеи, чем это делают паттерны.

Условие может использовать переменные, создаваемые в паттерне. Листинг 18.26 показывает выражение `match`, в котором первый рукав имеет паттерн `Some(x)` и ограничитель совпадения `if x < 5`.

Листинг 18.26. Добавление ограничителя совпадения в паттерн

```
let num = Some(4);

match num {
    Some(x) if x < 5 => println!("меньше пяти: {}", x),
    Some(x) => println!("{}", x),
    None => (),
}
```

Указанный пример будет печатать `меньше пяти: 4`. Когда переменная `num` сравнивается с паттерном в первом рукаве, она совпадает, потому что `Some(4)` совпадает с `Some(x)`. Затем ограничитель совпадения проверяет, меньше ли значение в `x`, чем 5, и поскольку это так, то выбирается первый рукав.

Если бы вместо этого переменная `num` была равна `Some(10)`, то ограничитель совпадения в первом рукаве был бы ложным, потому что 10 не меньше 5. Затем компилятор перейдет ко второму рукаву, который совпадет, потому что второй рукав не имеет ограничителя совпадения и поэтому совпадает с любым вариантом `Some`.

Выразить условие `if x < 5` внутри паттерна невозможно, поэтому ограничитель совпадения дает возможность выражать эту логику.

В обсуждении листинга 18.11 мы упомянули, что можно было бы использовать ограничители совпадения для решения проблем с затенением паттерна. Напомним, что новая переменная создавалась внутри паттерна в выражении `match` вместо переменной вне выражения `match`. Эта новая переменная означала, что мы не

могли протестировать значение внешней переменной. Листинг 18.27 показывает, как использовать ограничитель совпадения для устранения этой проблемы.

Листинг 18.27. Использование ограничителя совпадения для проверки равенства с внешней переменной

src/main.rs

```
fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        Some(50) => println!("Получено 50"),
        Some(n) if n == y => println!("Совпадение, n = {:?}", n),
        _ => println!("Вариант по умолчанию, x = {:?}", x),
    }

    println!("в конце: x = {:?}, y = {:?}", x, y);
}
```

Теперь этот код выведет

```
Вариант по умолчанию, x = Some(5)
```

Паттерн во втором рукаве выражения `match` не вводит новую переменную `y`, которая затеняла бы внешнюю `y`, то есть мы можем использовать внешнюю `y` в ограничителе совпадения. Вместо задания паттерна как `Some(y)`, который затенял бы внешнюю `y`, мы задаем `Some(n)`. Так создается новая переменная `n`, которая ничего не затеняет, потому что вне выражения `match` переменной `n` нет.

Ограничитель совпадения `if n == y` не является паттерном и, следовательно, не вводит новые переменные. Эта переменная `y` является внешней, а не новой затененной переменной `y`, и мы можем найти такое же значение, как и у внешней `y`, сравнив `n` с `y`.

В ограничителе совпадения также можно использовать оператор *или* (`|`) для задания более одного паттерна. Условие ограничителя совпадения будет применяться ко всем паттернам. В листинге 18.28 показан приоритет комбинирования ограничителя совпадения с паттерном, использующим `|`. Важная часть этого примера заключается в том, что ограничитель совпадения `if y` применяется к 4, 5 и 6, хотя он может выглядеть так, как будто `if y` применяется только к 6.

Листинг 18.28. Сочетание нескольких паттернов с помощью ограничителя совпадения

```
let x = 4;
let y = false;

match x {
    4 | 5 | 6 if y => println!("да"),
    _ => println!("нет"),
}
```

Условие совпадения констатирует, что рукав совпадает только в том случае, если значение `x` равно 4, 5 или 6 и если `y` является истинным. Когда этот код выполняется, паттерн первого рукава совпадает, потому что `x` равно 4, но ограничитель совпадения `if y` является ложным, поэтому первый рукав не выбирается. Код переходит ко второму рукаву, который совпадает, а программа выведет `нет`. Причина в том, что условие `if` применяется ко всему паттерну `4 | 5 | 6`, не только к последнему значению 6. Другими словами, приоритет ограничителя совпадения по отношению к паттерну ведет себя так:

```
(4 | 5 | 6) if y => ...
```

а не так:

```
4 | 5 | (6 if y) => ...
```

После выполнения этого кода приоритет становится очевидным: если бы ограничитель совпадения применялся только к конечному значению в списке значений, заданных с помощью оператора `|`, то рукав совпал бы и программа вывела бы `да`.

Привязки @

Оператор *at* (`@`) позволяет создавать переменную, которая содержит значение. В то же время мы проверяем это значение, чтобы убедиться, что оно совпадает с паттерном. В листинге 18.29 показан пример, в котором нужно проверить, не находится ли поле `id` в `Message::Hello` в пределах интервала `3...7`. Но мы также хотим привязать значение к переменной `id_variable`, чтобы можно было его использовать в коде, связанном с рукавом совпадения. Мы могли бы назвать эту переменную `id`, так же как и поле, но в данном примере будем использовать другое имя.

Листинг 18.29. Использование @ для привязки к значению в паттерне и проверка

```
enum Message {
    Hello { id: i32 },
}

let msg = Message::Hello { id: 5 };

match msg {
    Message::Hello { id: id_variable @ 3...7 } => {
        println!("Найден идентификатор в интервале: {}", id_variable)
    },
    Message::Hello { id: 10...12 } => {
        println!("Найден идентификатор в еще одном интервале")
    },
    Message::Hello { id } => {
        println!("Найден какой-то другой идентификатор: {}", id)
    },
}
```

Код выведет:

```
Найден идентификатор в интервале: 5
```

Указывая `id_variable @` перед интервалом `3...7`, мы захватываем любое значение, совпадающее с интервалом, а также проверяем, что это значение совпадает с паттерном интервала.

Во втором рукаве, где есть только указанный в паттерне интервал, код, связанный с рукавом, не имеет переменной, которая содержит фактическое значение поля `id`. Значение поля `id` могло быть 10, 11 или 12, но код с этим паттерном не знает, какое это значение. Код паттерна не может использовать значение из поля `id`, потому что мы не сохранили значение `id` в переменной.

В последнем рукаве, где мы указали переменную без интервала, есть значение, доступное для использования в коде рукава в переменной с именем `id`. Причина в том, что мы использовали укороченный синтаксис поля структуры. Но, в отличие от первых двух рукавов, мы не проверили значение в поле `id` в этом рукаве: с этим паттерном будет совпадать любое значение.

Использование `@` позволяет тестировать значение и сохранять его в переменной внутри одного паттерна.

Итоги

Паттерны Rust очень полезны тем, что помогают различать виды данных. Благодаря их использованию в выражениях `match` паттерны охватывают все возможные значения, иначе программа не будет компилироваться. Паттерны в инструкциях `let` и параметрах функций делают эти конструкции полезнее, позволяя деструктурировать значения на более мелкие части и одновременно передавать их переменным. Мы можем создавать простые или сложные паттерны в соответствии со своими потребностями.

Далее, в предпоследней главе, мы рассмотрим некоторые продвинутые аспекты различных средств языка Rust.

19 Продвинутые средства

Вы уже изучили наиболее часто используемые средства языка программирования Rust. Прежде чем приступить к очередному проекту в главе 20, мы рассмотрим несколько аспектов языка, с которыми вы, возможно, будете сталкиваться время от времени. Можно использовать эту главу как справочник на случай, если столкнетесь с чем-то неизвестным при работе с Rust. Языковые средства, которые вы научитесь использовать в этой главе, полезны в очень специфических ситуациях. Возможно, вы нечасто будете к ним обращаться, но мы хотим, чтобы вы знали и понимали все средства, которые есть в арсенале Rust.

В этой главе мы рассмотрим:

- **Небезопасный язык Rust.** Как отказаться от встроенной защиты гарантий Rust и взять на себя ответственность за ручное соблюдение этих гарантий.
- **Продвинутые типажи.** Связанные типы, параметры типов по умолчанию, полный синтаксис, супертипажи и паттерн newtype применительно к типажам.
- **Продвинутые типы.** Больше сведений о паттерне newtype, а также псевдонимы типов, тип never и динамически изменяемые типы.
- **Продвинутые функции и замыкания.** Указатели функций и возвращающие замыкания.
- **Макрокоманды.** Способы, которые определяют больше кода во время компиляции.

В этом арсенале каждый найдет что-то для себя! Давайте приступим!

Небезопасный Rust

Код, который мы обсуждали, гарантировал безопасность памяти во время компиляции. Однако у Rust есть второй язык, скрытый внутри него, который не обеспечивает безопасность памяти: он называется «небезопасный язык Rust» (unsafe Rust), работает так же, как и обычный Rust, но дает дополнительные сверхспособности.

Небезопасный Rust существует потому, что по своей природе статический анализ консервативен. Когда компилятор пытается выяснить, поддерживает ли код гарантии, то лучше отклонять некоторые допустимые программы, чем принимать некоторые недопустимые программы. Хотя с точки зрения языка Rust код может быть в полном порядке, но самом деле все не так! Используйте небезопасный код, чтобы сказать компилятору: «Доверься мне, я знаю, что делаю». Важно понимать, что вы действуете на свой страх и риск: если вы работаете с небезопасным кодом неправильно, могут возникнуть проблемы из-за небезопасности памяти, такие как переход по указателю null.

Вторая причина существования «небезопасного двойника» заключается в том, что базовое аппаратное обеспечение небезопасно. Если бы Rust не позволял выполнять небезопасные операции, вы не смогли бы выполнять некоторые задачи. Компилятору нужно, чтобы вы могли заниматься низкоуровневым системным программированием, например взаимодействовать с операционной системой непосредственно или даже писать собственную ОС. И действительно, одна из целей языка — это работа с низкоуровневым системным программированием. Давайте узнаем, что и как можно делать с небезопасным Rust.

Небезопасные сверхспособности

Чтобы переключиться на небезопасный Rust, используйте ключевое слово `unsafe`, а затем начните новый блок, содержащий небезопасный код. В небезопасном Rust вы можете предпринять четыре действия, именуемые небезопасными сверхспособностями, которые нельзя совершить на безопасном языке Rust. Эти сверхспособности включают в себя способности:

- применять оператор разыменования к сырому указателю;
- вызывать небезопасную функцию или метод;
- обращаться или модифицировать изменяемую статическую переменную;
- реализовывать небезопасный типаж.

Важно понимать, что ключевое слово `unsafe` не отключает контролер заимствования или другие проверки безопасности, принятые в Rust: если вы используете ссылку в небезопасном коде, она все равно будет проверена. Ключевое слово `unsafe` предоставляет доступ только к этим четырем операциям, которые затем не проверяются компилятором на безопасность памяти. Внутри опасного блока вы все равно до некоторой степени будете в безопасности.

В дополнение к этому, небезопасность не означает, что код внутри блока обязательно опасен или что в нем обязательно будут проблемы с безопасностью памяти. Цель состоит в том, что, как программист, вы будете делать так, чтобы код внутри небезопасного блока обращался к памяти правильным образом.

Людям свойственно ошибаться, и ошибки, безусловно, будут. Но требуя, чтобы эти четыре небезопасные операции были внутри блоков, связанных с `unsafe`, вы

будете знать, что любые ошибки в безопасности памяти должны быть внутри небезопасного блока. Пусть небезопасные блоки будут небольшими. Вы будете благодарны позже при расследовании дефектов памяти.

В целях максимальной изоляции небезопасного кода лучше всего помещать его внутрь безопасной абстракции и предоставлять безопасный API, который мы обсудим позже при изучении небезопасных функций и методов. Части стандартной библиотеки реализованы как безопасные абстракции над проверенным небезопасным кодом. Благодаря обертыванию небезопасного кода в безопасную абстракцию использование `unsafe` не приводит к утечке небезопасного кода там, где вы или ваши пользователи могли бы использовать функциональность, реализованную с небезопасным кодом, потому что безопасная абстракция безопасна.

Давайте посмотрим на каждую из четырех небезопасных сверхспособностей. Мы также поговорим о некоторых абстракциях, которые обеспечивают безопасный интерфейс для небезопасного кода.

Применение оператора разыменования к сырому указателю

В разделе «Висячие ссылки» (с. 107) мы упомянули, что компилятор обеспечивает, чтобы ссылки всегда были действительными. Небезопасный Rust имеет два новых типа, именуемых сырыми указателями, которые похожи на ссылки. Как и в случае со ссылками, сырые указатели могут быть неизменяемыми либо изменяемыми и пишутся соответственно как `*const T` и `*mut T`. Звездочка не является оператором разыменования, это часть имени типа. В контексте сырых указателей «неизменяемый» означает, что указатель не может быть присвоен непосредственно после применения оператора разыменования.

В отличие от ссылок и умных указателей:

- Сырым указателям разрешено игнорировать правила заимствования, имея как неизменяемые, так и изменяемые указатели или несколько изменяемых указателей на одно и то же место.
- Сырые указатели гарантированно не указывают на действительную память.
- Сырым указателям разрешено быть null.
- Сырые указатели не реализуют никакой автоматической очистки.

Отказавшись от этих гарантий, вы отказываетесь от безопасности в обмен на более высокую производительность или возможность взаимодействовать с другим языком или оборудованием, где гарантии Rust не применяются.

Листинг 19.1 показывает, как создавать неизменяемый и изменяемый сырой указатель из ссылок.

Листинг 19.1. Создание сырых указателей из ссылок

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;
```

Обратите внимание, мы не включаем в этот код ключевое слово `unsafe`. Можно создавать сырые указатели в безопасном коде. Просто нельзя применять оператор разыменования к сырым указателям за пределами небезопасного блока, как вы увидите немного позже.

Мы создали сырые указатели, используя `as` для приведения неизменяемой и изменяемой ссылок к соответствующим типам сырых указателей. Поскольку мы создали их непосредственно из ссылок, гарантированно являющихся действительными, мы знаем, что эти сырые указатели действительны, но нельзя утверждать, что это верно для любого сырого указателя.

Далее мы создадим сырой указатель, в действительности которого не можем быть уверены. Листинг 19.2 показывает, как создавать сырой указатель на произвольное место в памяти. Попытка использовать произвольную память не имеет определения: данных по этому адресу может и не быть, компилятор может оптимизировать код так, чтобы не было доступа к памяти, либо программа выдаст ошибку сегментации. В обычных условиях нет веской причины писать такой код, но это возможно.

Листинг 19.2. Создание сырого указателя на произвольный адрес памяти

```
let address = 0x012345usize;
let r = address as *const i32;
```

Вспомните, что можно создавать сырые указатели в безопасном коде, но нельзя применять оператор разыменования к сырым указателям и читать данные, на которые они указывают. В листинге 19.3 мы используем оператор разыменования * для сырого указателя, требующего блок `unsafe`.

Листинг 19.3. Применение оператора разыменования к сырым указателям в небезопасном блоке

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1 равно: {}", *r1);
    println!("r2 равно: {}", *r2);
}
```

Указатель не приносит вреда. Мы можем иметь дело с недействительным значением, только когда пытаемся обратиться к значению, на которое он указывает.

Заметим также, что в листингах 19.1 и 19.3 мы создали сырые указатели `*const i32` и `*mut i32`, указывающие на одну и ту же ячейку памяти, где хранится `num`. Если бы мы попытались создать неизменяемую и изменяемую ссылку на `num`, то этот код не компилировался бы, потому что правила владения языком Rust не допускают изменяемую ссылку одновременно с любыми неизменяемыми ссылками. С помощью сырых указателей мы можем создавать изменяемый и неизменяемый указатели на одно и то же место и изменять данные через изменяемый указатель, потенциально создавая гонку данных. Будьте осторожны!

С учетом всего этого зачем вообще применять сырые указатели? Один из основных вариантов использования — это взаимодействие с кодом на языке C, как вы увидите в следующем разделе. Еще один вариант — создание безопасных абстракций, которые контролер заимствований не понимает. Мы представим небезопасные функции, а затем рассмотрим пример безопасной абстракции, использующей небезопасный код.

Вызов небезопасной функции или метода

Второй тип операций, требующих небезопасный блок, — это вызовы небезопасных функций. Небезопасные функции и методы выглядят так же, как и регулярные функции и методы, но у них есть дополнительное ключевое слово `unsafe` перед остальной частью определения. Ключевое слово `unsafe` в этом контексте указывает, что данная функция имеет требования, которые нужно поддерживать при вызове этой функции, потому что язык Rust не может гарантировать, что эти требования удовлетворены. Вызывая небезопасную функцию в блоке `unsafe`, мы говорим о том, что прочитали документацию к этой функции и берем на себя ответственность за соблюдение обязательств функции.

Вот небезопасная функция с именем `dangerous`, которая ничего не делает в своем теле:

```
unsafe fn dangerous() {}

unsafe {
    dangerous();
}
```

Нужно вызвать функцию `dangerous` внутри отдельного блока `unsafe`. Если мы попытаемся вызвать `dangerous` без блока `unsafe`, то получим ошибку[1]:

```
error[E0133]: call to unsafe function requires unsafe function or block
 -->
  |
4 |     dangerous();
  |     ^^^^^^^^^^ call to unsafe function
```

[1] `ошибка[E0133]: вызов небезопасной функции требует функции или блока с префиксом unsafe`

Вставляя блок `unsafe` в вызов функции `dangerous`, мы уверяем Rust, что прочитали документацию к указанной функции и понимаем, как правильно использовать ее свойства.

Тела небезопасных функций фактически являются блоками `unsafe`, поэтому для выполнения других небезопасных операций внутри небезопасной функции нам не нужно добавлять еще один блок `unsafe`.

Создание безопасной абстракции над небезопасным кодом

То, что функция содержит небезопасный код, не означает, что мы должны пометить всю функцию как небезопасную. На самом деле обертывание небезопасного кода в безопасную функцию — это часто встречающаяся абстракция. В качестве примера давайте изучим функцию из стандартной библиотеки `split_at_mut`, которая требует некоторого небезопасного кода, и узнаем, как ее можно реализовать. Этот безопасный метод определяется на изменяемых срезах: он берет один срез и делает из него два, разделяя срез по индексу, заданному в качестве аргумента. Листинг 19.4 показывает, как использовать `split_at_mut`.

Листинг 19.4. Использование безопасной функции split_at_mut

```
let mut v = vec![1, 2, 3, 4, 5, 6];

let r = &mut v[..];

let (a, b) = r.split_at_mut(3);

assert_eq!(a, &mut [1, 2, 3]);
assert_eq!(b, &mut [4, 5, 6]);
```

Мы не можем реализовать эту функцию, используя только безопасный Rust. Попытка выглядит примерно так, как код в листинге 19.5, который не компилируется. Для простоты мы реализуем `split_at_mut` как функцию, а не метод и только для срезов значений типа `i32`, а не для обобщенного типа `T`.

Листинг 19.5. Попытка реализовать функцию `split_at_mut` с использованием только безопасного языка Rust

```
fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = slice.len();

    assert!(mid <= len);

    (&mut slice[..mid],
     &mut slice[mid..])
}
```

Эта функция сначала получает суммарную длину среза. Затем она выполняет проверочное утверждение того, что индекс, заданный в качестве параметра, находится внутри среза, проверяя, что он меньше или равен длине. Проверочное

утверждение означает, что, если мы передадим индекс, который больше индекса, в котором делится срез, то эта функция поднимет панику до попытки использовать этот индекс.

Затем мы возвращаем два изменяемых среза в кортеже: один от начала исходного среза до индекса `mid`, а другой от `mid` до конца среза.

Когда мы попытаемся скомпилировать код из листинга 19.5, возникнет ошибка[1]:

```
error[E0499]: cannot borrow `*slice` as mutable more than once at a time
 -->
  |
6 |     (&mut slice[..mid],
  |           ----- first mutable borrow occurs here
7 |      &mut slice[mid..])
  |           ^^^^^ second mutable borrow occurs here
8 | }
  | - first borrow ends here
```

Контролер заимствования Rust не может понять, что мы заимствуем разные части среза. Он только знает, что мы заимствуем у одного и того же среза дважды. Заимствовать разные части среза в принципе нормально, потому что два среза не перекрываются, но язык Rust недостаточно умен, чтобы это знать. Когда нам известно, что с кодом все в порядке, а языку Rust — нет, самое время обратиться к небезопасному коду.

Листинг 19.6 показывает, как использовать блок `unsafe`, сырой указатель и несколько вызовов небезопасных функций, чтобы привести реализацию `split_at_mut` в рабочее состояние.

Листинг 19.6. Использование небезопасного кода в реализации функции `split_at_mut`

```
use std::slice;

fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
  ❶ let len = slice.len();
  ❷ let ptr = slice.as_mut_ptr();

  ❸ assert!(mid <= len);

  ❹ unsafe {
      ❺ (slice::from_raw_parts_mut(ptr, mid),
      ❻ slice::from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
    }
}
```

[1] `ошибка[E0499]: не получается позаимствовать `*slice` как изменяемый более одного раза`

Вспомните из раздела «Срезовый тип» (с. 109), что срез — это указатель на некие данные и длина среза. Мы используем метод `len` для получения длины среза ❶, а метод `as_mut_ptr` — для обращения к сырому указателю среза ❷. В данном случае, поскольку у нас изменяемый срез значений типа `i32`, метод `as_mut_ptr` возвращает сырой указатель с типом `*mut i32`, который мы сохранили в переменной `ptr`.

У нас есть проверочное утверждение того, что индекс `mid` находится внутри среза ❸. Затем мы переходим к небезопасному коду ❹: функция `slice::from_raw_parts_mut` берет сырой указатель и длину и создает срез. Мы используем эту функцию для создания среза, который начинается с `ptr` и является `mid` элементов в длину ❺. Затем мы вызываем метод `offset` для `ptr` с `mid` в качестве аргумента и получаем сырой указатель, который начинается в `mid`. Далее мы создаем срез, используя этот указатель и число оставшихся элементов после `mid` в качестве длины ❻.

Функция `slice::from_raw_parts_mut` небезопасна, поскольку она берет сырой указатель и должна принять на веру, что этот указатель действителен. Метод `offset` для сырых указателей также небезопасен, поскольку он должен принять на веру, что расположение смещения тоже является действительным указателем. Следовательно, для вызова функций `slice::from_raw_parts_mut` и `offset` мы должны окружить их блоком `unsafe`. Посмотрев на код и добавив проверочное утверждение, что `mid` должен быть меньше или равен `len`, мы можем сказать, что все сырые указатели, используемые в блоке `unsafe`, будут действительными указателями на данные внутри этого среза. Приемлемо и уместно так использовать блок `unsafe`.

Обратите внимание, не нужно помечать результирующую функцию `split_at_mut` как `unsafe`, можно вызывать эту функцию из безопасного Rust. Мы создали безопасную абстракцию для небезопасного кода, реализовав функцию, которая использует `unsafe` безопасным образом, потому что она создает только действительные указатели из данных, к которым эта функция обращается.

Напротив, `slice::from_raw_parts_mut` в листинге 19.7, скорее всего, приведет к аварийному сбою при использовании среза. Этот код берет произвольное место в памяти и создает срез длиной 10 000 элементов.

Листинг 19.7. Создание среза из произвольного места в памяти

```
use std::slice;

let address = 0x012345usize;
let r = address as *mut i32;

let slice: &[i32] = unsafe {
    slice::from_raw_parts_mut(r, 10000)
};
```

Мы не владеем памятью в этом произвольном месте, и нет гарантии, что срез, созданный кодом, содержит допустимые значения типа `i32`. Попытка использовать срез как допустимый приводит к неопределенному поведению.

Использование функций extern для вызова внешнего кода

Иногда коду Rust нужно взаимодействовать с кодом, написанным на другом языке. Для этого в Rust есть ключевое слово `extern`, которое облегчает создание и использование интерфейса с внешними функциями (FFI от англ. *Foreign Function Interface*). Интерфейс с внешними функциями — это способ, которым в языке программирования определяются функции и обеспечивается возможность вызова этих функций другим («иностранным» — foreign) языком программирования.

ВЫЗОВ ФУНКЦИЙ RUST ИЗ ДРУГИХ ЯЗЫКОВ

Мы также можем использовать `extern` для создания интерфейса, который позволяет другим языкам вызывать функции Rust. Вместо блока `extern` мы добавляем ключевое слово `extern` и указываем используемый ABI непосредственно перед ключевым словом `fn`. Нам также нужно добавить аннотацию `#[no_mangle]`, чтобы компилятор Rust не искажал имя этой функции. При искажении (mangling) компилятор изменяет имя, данное функции, на другое, содержащее больше информации для других частей процесса компиляции, но неудобное для восприятия человеком. Каждый компилятор языка программирования искажает имена по-разному, поэтому для того, чтобы другие языки могли вызывать функцию Rust, мы должны отключить искажение имени со стороны компилятора Rust.

В следующем примере мы делаем функцию `call_from_c` доступной из кода C, после того как она скомпилирована в совместную библиотеку и связана из C:

```
#[no_mangle]
pub extern "C" fn call_from_c() {
    println!("Только что была вызвана функция Rust из C!");
}
```

Это использование extern не требует ключевого слова `unsafe`.

В листинге 19.8 показано, как настраивать интеграцию с функцией `abs` из стандартной библиотеки C. Функции, объявляемые в блоках `extern`, всегда небезопасны для вызова из кода на Rust. Причина в том, что другие языки не обеспечивают соблюдение правил и гарантий Rust, а Rust не может их проверить, поэтому ответственность за безопасность ложится на программиста.

Листинг 19.8. Объявление и вызов функции extern, определенной на другом языке

src/main.rs

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Абсолютное значение для -3 согласно языку C: {}", abs(-3));
    }
}
```

В блоке `extern "C"` мы перечисляем имена и сигнатуры внешних функций из другого языка, который мы хотим вызвать. Фрагмент `"C"` определяет, какой двоичный интерфейс приложения (ABI от англ. *application binary interface*) используется внешней функцией. Двоичный интерфейс приложения определяет, как вызвать функцию на ассемблерном уровне. Двоичный интерфейс приложения `"C"` распространен больше всего и подчиняется ABI языка программирования C.

Обращение к изменяемой статической переменной или ее модифицирование

Мы пока не говорили о глобальных переменных, поддерживаемых в Rust, с которыми бывают проблемы с точки зрения правил владения Rust. Если два потока обращаются к одной и той же изменяемой глобальной переменной, то может возникнуть гонка данных.

В Rust глобальные переменные называются статическими переменными. В листинге 19.9 показан пример объявления и использования статической переменной со строковым срезом в качестве значения.

Листинг 19.9. Определение и использование неизменяемой статической переменной

src/main.rs

```
static HELLO_WORLD: &str = "Hello, world!";

fn main() {
    println!("имя равно: {}", HELLO_WORLD);
}
```

Статические переменные подобны константам, которые мы обсуждали в разделе «Различия между переменными и константами» (с. 63). Имена статических переменных по общему соглашению находятся в `КРИЧАЩЕМ_ЗМЕИНОМ_РЕГИСТРЕ`, и мы должны аннотировать тип переменной, в данном примере это `&'static str`. Статические переменные хранят ссылки только с жизненным циклом `'static`, то есть компилятор может узнавать жизненный цикл, нам не нужно явно его аннотировать. Обращаться к неизменяемой статической переменной безопасно.

Константы и неизменяемые статические переменные бывают похожи, но различие состоит в том, что значения в статической переменной имеют фиксированный адрес в памяти. При использовании этого значения всегда будут доступны одни и те же данные. Константам, с другой стороны, разрешено дублировать свои данные всякий раз, когда они используются.

Еще одно различие между константами и статическими переменными состоит в том, что статические переменные могут быть изменяемыми. Доступ к изменяемым статическим переменным и их модифицирование является небезопасным. Листинг 19.10 показывает объявление изменяемой статической переменной с именем `COUNTER`, обращение к ней и ее модифицирование.

Листинг 19.10. Небезопасно читать изменяемую статическую переменную или записывать в нее

src/main.rs

```
static mut COUNTER: u32 = 0;

fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    add_to_count(3);

    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}
```

Как и в случае с регулярными переменными, мы задаем изменяемость с помощью ключевого слова `mut`. Любой код, читающий или пишущий данные из `COUNTER`, должен находиться в блоке `unsafe`. Этот код ожидаемо компилирует и выводит `COUNTER: 3`, потому что это однопоточной код. Ситуация, когда к `COUNTER` обращается несколько потоков исполнения, скорее всего, приведет к гонкам данных.

Когда данные изменяемые и глобально доступные, трудно обойтись без гонок данных, поэтому Rust считает изменяемые статические переменные небезопасными. Там, где это возможно, предпочтительнее использовать параллельные технические приемы и безопасные для потоков умные указатели из главы 16, благодаря которым компилятор проверяет, что обращение к данным из разных потоков исполнения выполняется безопасным образом.

Реализация небезопасного типажа

Последнее действие, которое работает только с `unsafe`, — это реализация небезопасного типажа. Типаж небезопасен, если хотя бы один из его методов имеет некий инвариант, который компилятор не может проверить. Мы можем объявить типаж небезопасным, добавив ключевое слово `unsafe` перед ним и отметив реализацию типажа как `unsafe`, как показано в листинге 19.11.

Листинг 19.11. Определение и реализация небезопасного типажа

```
unsafe trait Foo {
    // здесь будут методы
}

unsafe impl Foo for i32 {
    // здесь будут реализации методов
}
```

Используя `unsafe impl`, мы обещаем, что будем поддерживать инварианты, которые компилятор не может проверить.

В качестве примера вспомним маркерные типажи `Sync` и `Send`, которые обсуждались в разделе «Расширяемая конкурентность с типажами Sync и Send» (с. 428): компилятор реализует эти типажи автоматически, если типы полностью состоят из `Send` и `Sync`. Если мы реализуем тип, содержащий тип, который не является `Send` и `Sync`, такой как сырые указатели, и хотим промаркировать этот тип как `Send` и `Sync`, нужно использовать `unsafe`. Rust не может проверить, что тип безопасен и его можно отправлять между потоками или получить к нему доступ из нескольких потоков. Следовательно, нужно делать проверки вручную и помечать их как `unsafe`.

Когда использовать небезопасный код

Использование ключевого слова `unsafe` для выполнения одного из четырех только что рассмотренных действий (сверхспособностей) не считают неправильным и не порицают. Но труднее получить небезопасный код в правильном виде, потому что компилятор не может поддержать безопасность памяти. Если по какой-то причине вам нужно использовать небезопасный код, вы можете это сделать. Если у вас есть явная аннотация `unsafe`, вам будет легче найти источник проблем, если они возникнут.

Продвинутые типажи

Мы впервые затронули типажи в разделе «Типажи: определение совместного поведения» (с. 223), но, как и в случае с жизненным циклом, не обсуждали эффективные элементы. Теперь, когда вы знаете о Rust больше, можно перейти к мелочам.

Детализация заполнительных типов в определениях типажей с помощью связанных типов

Связанные типы соединяют заполнитель типа с типажом таким образом, что определения типажных методов могут использовать эти заполнительные типы в своих сигнатурах. Реализация типажа укажет конкретный тип, который будет использоваться на месте этого типа в определенном случае. Благодаря этому мы можем определить типаж, который использует некоторые типы, и нам не нужно точно знать, что это за типы, пока типаж не будет реализован.

Мы говорили, что большинство продвинутых языковых средств из этой главе применяются редко. Связанные типы находятся где-то посередине: они используются реже, чем средства, описанные в остальной части книги, но чаще, чем многие другие средства, обсуждаемые в этой главе.

Один из примеров типажа со связанным типом — `Iterator`, предусмотренный стандартной библиотекой. Связанный тип называется `Item` и исполняет роль типа значений, которые перебирает тип, реализующий типаж `Iterator`. В разделе «Типаж Iterator и метод next» (с. 327) мы упомянули, что определение `Iterator` выглядит так, как показано в листинге 19.12.

Листинг 19.12. Определение типажа Iterator со связанным типом Item

```
pub trait Iterator {
    type Item;

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

Тип `Item` заполнительный, определение метода `next` показывает, что он будет возвращать значения типа `Option<Self::Item>`. Средства типажа `Iterator` будут описывать конкретный тип для `Item`, а метод `next` будет возвращать тип `Option`, содержащий значение этого конкретного типа.

Может показаться, что связанные типы похожи на обобщения, поскольку последние позволяют определять функцию, не указывая, с какими типами она может работать. Тогда зачем использовать связанные типы?

Рассмотрим разницу между этими идеями на примере из главы 13, в котором реализован типаж `Iterator` в структуре `Counter`. В листинге 13.21 мы описали, что у `Item` был тип `u32`:

src/lib.rs

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        // --пропуск--
```

Кажется, что этот синтаксис сопоставим с синтаксисом обобщений. Тогда почему бы просто не определить типаж `Iterator` с помощью обобщений, как показано в листинге 19.13?

Листинг 19.13. Гипотетическое определение типажа `Iterator` с использованием обобщений

```
pub trait Iterator<T> {
    fn next(&mut self) -> Option<T>;
}
```

Разница заключается в том, что при использовании обобщенных типов, как в листинге 19.13, мы должны аннотировать типы в каждой реализации. Поскольку мы также можем реализовать `Iterator<String> for Counter` или сделать это с участием любого другого типа, у нас может быть несколько реализаций `Iterator` для `Counter`. Другими словами, когда у типажа обобщенный параметр, его можно

реализовать для типа многократно, каждый раз изменяя конкретные типы параметров обобщенного типа. Используя метод `next` для `Counter`, нам пришлось бы предоставлять аннотации типов, чтобы задавать необходимую реализацию типажа `Iterator`.

При работе со связанными типами не нужно аннотировать типы, потому что мы не можем реализовать типаж в типе многократно. В листинге 19.12 с определением, использующим связанные типы, мы можем выбрать только один тип `Item`, поскольку может быть только одна инструкция `impl Iterator for Counter`. Не нужно описывать, что итератор значений типа `u32` требуется везде, где мы вызываем `next` для `Counter`.

Параметры обобщенного типа по умолчанию и перегрузка операторов

При работе с параметрами обобщенного типа вы можете задавать используемый по умолчанию конкретный тип для обобщенного типа. Таким образом, средствам типажа не нужно задавать конкретный тип, если работает тип по умолчанию. Синтаксис описания типа по умолчанию для обобщенного типа — это `<ЗаполнительныйТип=КонкретныйТип>` во время объявления обобщенного типа.

Этот технический прием полезен в такой ситуации, как перегрузка операторов. Перегрузка операторов — это индивидуальная настройка поведения оператора (такого, как `+`) в определенных ситуациях.

Rust не позволяет создавать собственные операторы или перегружать произвольные операторы. Но вы можете перегружать операции и соответствующие типажи, перечисленные в `std::ops`, реализуя типажи, связанные с оператором. Например, в листинге 19.14 мы перегружаем оператор `+`, чтобы складывать вместе два экземпляра структуры `Point`. Мы делаем это, реализуя типаж `Add` в структуре `Point`.

Листинг 19.14. Реализация типажа Add с целью перегрузки оператора + для экземпляров структуры Point

src/main.rs

```
use std::ops::Add;

#[derive(Debug, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
    type Output = Point;

    fn add(self, other: Point) -> Point {
```

```
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

fn main() {
    assert_eq!(Point { x: 1, y: 0 } + Point { x: 2, y: 3 },
               Point { x: 3, y: 3 });
}
```

Метод `add` складывает значения `x` двух экземпляров структуры `Point` и значения `y` двух экземпляров структуры `Point`, создавая новую структуру `Point`. Типаж `Add` имеет связанный тип с именем `Output`, который выясняет тип, возвращаемый методом `add`.

Обобщенный тип по умолчанию в этом коде находится внутри типажа `Add`. Вот его определение:

```
trait Add<RHS=Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}
```

В целом этот код должен быть вам знаком: типаж с одним методом и связанным типом. Новым фрагментом является `RHS=Self`: этот синтаксис называется параметром типа по умолчанию. Параметр обобщенного типа `RHS` (сокращенно *om right-hand side*, то есть «правая сторона») определяет тип параметра `rhs` в методе `add`. Если мы не зададим конкретный тип для `RHS` при реализации типажа `Add`, то `RHS` по умолчанию примет `Self`, который будет типом, в котором реализуется `Add`.

Когда мы реализовывали `Add` для `Point`, мы использовали для `RHS` тип по умолчанию, потому что хотели складывать два экземпляра структуры `Point`. Давайте рассмотрим пример реализации типажа `Add`, где требуется настраивать тип `RHS` индивидуально, а не использовать тип по умолчанию.

У нас есть две структуры, `Millimeters` и `Meters`, которые содержат значения в разных единицах измерения. Мы хотим прибавлять значения в миллиметрах к значениям в метрах и реализовывать `Add`, которая делала бы конвертацию правильно. Мы можем реализовать `Add` для `Millimeters`, где `Meters` будут в качестве `RHS`, как показано в листинге 19.15.

Листинг 19.15. Реализация типажа Add в Millimeters для прибавления Millimeters к Meters

src/lib.rs

```
use std::ops::Add;

struct Millimeters(u32);
```

```
struct Meters(u32);

impl Add<Meters> for Millimeters {
    type Output = Millimeters;

    fn add(self, other: Meters) -> Millimeters {
        Millimeters(self.0 + (other.0 * 1000))
    }
}
```

Для сложения `Millimeters` и `Meters` мы указываем `impl Add<Meters>`, устанавливая значение параметра `RHS` типа вместо использования `Self` по умолчанию.

Мы будем использовать параметры типа по умолчанию двумя главными способами:

- Для расширения типа без нарушения существующего кода.
- Для разрешения индивидуальной настройки в специфических случаях, которая большинству пользователей не понадобится.

Типаж `Add` стандартной библиотеки — это пример второго предназначения: обычно вы добавляете два подобных типа, но типаж `Add` обеспечивает возможность индивидуальной настройки помимо этого. Использование параметра типа по умолчанию в определении типажа `Add` означает, что большую часть времени вам не нужно указывать дополнительный параметр. Другими словами, большая часть стереотипного кода не требуется в реализации, что облегчает использование типажа.

Первое предназначение похоже на второе, но в обратном порядке: если вы хотите добавить параметр типа в существующий типаж, то можно дать ему тип по умолчанию, чтобы разрешить расширение функциональности типажа, не нарушая код существующей реализации.

Полный синтаксис для устранения неоднозначности: вызов методов с одинаковым именем

В Rust у типажа может быть метод с тем же именем, что и у метода другого типажа, а компилятор не мешает реализовывать оба типажа в одном типе. Также можно реализовывать метод непосредственно в типе с тем же именем, что и у метода из типажа.

При вызове методов с одинаковым именем нужно указывать, какой из них вы хотите использовать. Рассмотрим код в листинге 19.16, где мы определили два типажа, `Pilot` и `Wizard`, которые имеют метод с именем `fly`. Затем мы реализуем оба типажа в типе `Human`, у которого уже есть метод с именем `fly`, реализованный в нем. Каждый метод `fly` делает что-то свое.

Листинг 19.16. Два типажа определяются как имеющие метод fly и реализуются в типе Human, а метод fly реализуется непосредственно в Human

src/main.rs

```
trait Pilot {
    fn fly(&self);
}

trait Wizard {
    fn fly(&self);
}

struct Human;

impl Pilot for Human {
    fn fly(&self) {
        println!("Говорит ваш капитан.");
    }
}

impl Wizard for Human {
    fn fly(&self) {
        println!("Подъем!");
    }
}

impl Human {
    fn fly(&self) {
        println!("*яростно размахивает руками*");
    }
}
```

Когда мы вызываем `fly` для экземпляра типа `Human`, компилятор по умолчанию вызывает метод, который реализован непосредственно в типе, как показано в листинге 19.17.

Листинг 19.17. Вызов метод fly для экземпляра типа Human

src/main.rs

```
fn main() {
    let person = Human;
    person.fly();
}
```

Выполнение этого кода выведет `*яростно размахивает руками*`, показывая, что компилятор вызвал метод `fly`, реализованный непосредственно в `Human`.

Для того чтобы вызвать метод `fly` либо из `Pilot`, либо из `Wizard`, нужно использовать более явный синтаксис, который указывает, какой метод `fly` мы имеем в виду. Этот синтаксис показан в листинге 19.18.

Листинг 19.18. Детализация того, метод `fly` какого типажа мы хотим вызвать

src/main.rs

```
fn main() {
    let person = Human;
    Pilot::fly(&person);
    Wizard::fly(&person);
    person.fly();
}
```

Указание имени типажа перед именем метода проясняет, какую реализацию метода `fly` мы хотим вызвать. Мы также могли бы написать `Human::fly(&person)`, что эквивалентно `person.fly()`, использованному в листинге 19.18. Так выходит длиннее, но пригодится, если нам нужно устранить неоднозначность.

Выполнение этого кода выводит следующее:

```
Говорит ваш капитан.
Подъем!
*яростно размахивает руками*
```

Поскольку метод `fly` берет параметр `self`, если бы у нас было два типа, которые реализуют один типаж, язык Rust мог бы узнавать, какую реализацию типажа использовать, основываясь на типе `self`.

Однако связанные функции, которые являются частью типажей, не имеют параметра `self`. Когда два типа в одинаковой области видимости реализуют этот типаж, компилятор не может выяснить, какой тип вы имеете в виду, если вы не используете полный синтаксис, наиболее специфичный для применения при вызове функции. Например, типаж `Animal` в листинге 19.19 имеет связанную функцию `baby_name`, реализацию `Animal` для структуры `Dog` и связанную функцию `baby_name`, определенную непосредственно для `Dog`.

Листинг 19.19. Типаж со связанной функцией и тип со связанной функцией с тем же именем, который также реализует этот типаж

src/main.rs

```
trait Animal {
    fn baby_name() -> String;
}

struct Dog;

impl Dog {
    fn baby_name() -> String {
        String::from("Спот")
    }
}

impl Animal for Dog {
    fn baby_name() -> String {
```

```
        String::from("щенок")
    }
}

fn main() {
    println!("Детеныш собаки - это {}", Dog::baby_name());
}
```

Указанный код описывает приют для животных, в котором хотят давать всем щенкам кличку «Спот», что реализовано в связанной функции `baby_name`, которая определена для `Dog`. Тип `Dog` также реализует типаж `Animal`, который описывает характеристики, имеющиеся у всех животных. Детеныши собаки называются щенками, и это выражается в реализации типажа `Animal` в `Dog` в функции `baby_name`, связанной с типажом `Animal`.

В `main` мы вызываем функцию `Dog::baby_name`, которая вызывает связанную функцию, определенную непосредственно для `Dog`. Этот код выводит следующее:

```
Детеныш собаки - это Спот
```

Этот результат — не тот, который мы хотели получить. Мы хотим вызвать функцию `baby_name`, являющуюся частью типажа `Animal`, который реализован в `Dog`, чтобы код выводил

```
Детеныш собаки — это щенок
```

Технический прием детализации имени типажа, который мы использовали в листинге 19.18, здесь не помогает. При замене функции `main` на код в листинге 19.20 произойдет ошибка компиляции.

Листинг 19.20. Попытка вызвать функцию baby_name из типажа Animal, но Rust не знает, какую реализацию использовать

src/main.rs

```
fn main() {
    println!("Детеныш собаки - это {}", Animal::baby_name());
}
```

Поскольку `Animal::baby_name` является связанной функцией, а не методом, и, следовательно, не имеет параметра `self`, язык Rust не может определить, какая реализация `Animal::baby_name` нам нужна. Мы получим такую ошибку компилятора[1]:

```
error[E0283]: type annotations required: cannot resolve `_: Animal`
  --> src/main.rs:20:43
   |
20 |     println!("Детеныш собаки - это {}", Animal::baby_name());
   |                                          ^^^^^^^^^^^^^^^^
   |
   = note: required by `Animal::baby_name`
```

[1] ошибка[E0283]: требуются аннотации типов: не удается урегулировать `_: Animal ‘

Для того чтобы устранить неоднозначность и сообщить языку Rust, что нужна реализация `Animal` в `Dog`, требуется полный синтаксис. В листинге 19.21 показано, как применять полный синтаксис.

Листинг 19.21. Использование полного синтаксиса для указания, что мы хотим вызвать функцию baby_name из типажа Animal, реализованного в Dog

src/main.rs

```
fn main() {
    println!("Детеныш собаки - это {}", <Dog as Animal>::baby_name());
}
```

Мы предоставляем языку Rust аннотацию типа внутри угловых скобок. Она конкретизирует, что мы хотим вызвать метод `baby_name` из типажа `Animal`, реализованного в `Dog`, говоря, что нужно трактовать тип `Dog` как `Animal` для данного вызова функции. Этот код теперь будет печатать то, что мы хотим:

```
Детеныш собаки - это щенок
```

В общем случае полный синтаксис определяется следующим образом:

```
<Тип as Типаж>::функция(приемник_в_случае_метода, следующий_арг, ...);
```

Для связанных функций не будет приемника, будет только список других аргументов. Вы можете использовать полный синтаксис везде, где вызываете функции или методы. Однако можно опустить любую часть этого синтаксиса, которую компилятор может выяснить в другом месте программы. Вы должны использовать такой подробный синтаксис только тогда, когда есть много реализаций, использующих одинаковое имя, и Rust нужно помочь выявить, какую реализацию вы хотите вызвать.

Использование супертипажей, требующих функциональности одного типажа внутри другого типажа

Иногда вам, возможно, понадобится, чтобы один типаж использовал функциональность другого типажа. В этом случае нужно полагаться на то, что зависимый типаж также реализуется. Типаж, на который вы полагаетесь, является супертипажом того типажа, который вы реализуете.

Предположим, мы хотим создать типаж `OutlinePrint` с помощью метода `outline_print`, который будет печатать значение, обрамленное звездочками. То есть если задана структура `Point`, реализующая типаж `Display` для результата в `(x, y)`, то при вызове функции `outline_print` для экземпляра структуры `Point` с 1 для `x` и 3 для `y` она должна печатать следующее:

```
**********
*        *
* (1, 3) *
*        *
**********
```

В реализации `outline_print` мы хотим использовать функциональность типажа `Display`. Следовательно, нужно указать, что `OutlinePrint` будет работать только для типов, которые также реализуют `Display` и обеспечивают функциональность, необходимую типажу `OutlinePrint`. Мы можем сделать это в определении типажа, указав `OutlinePrint: Display`. Этот технический прием подобен добавлению типажа, привязанного к этому типажу. В листинге 19.22 показана реализация `OutlinePrint`.

Листинг 19.22. Реализация типажа OutlinePrint, которая требует функциональности из типажа Display

src/main.rs

```
use std::fmt;

trait OutlinePrint: fmt::Display {
    fn outline_print(&self) {
        let output = self.to_string();
        let len = output.len();
        println!("{}", "*".repeat(len + 4));
        println!("*{}*", " ".repeat(len + 2));
        println!("* {} *", output);
        println!("*{}*", " ".repeat(len + 2));
        println!("{}", "*".repeat(len + 4));
    }
}
```

Поскольку мы указали, что `OutlinePrint` требует типаж `Display`, можно использовать функцию `to_string`, которая автоматически выполняется для любого типа, реализующего типаж `Display`. Если бы мы попытались использовать `to_string` без добавления двоеточия и указания `Display` после имени типажа, то получили бы сообщение об ошибке, что метод с именем `to_string` не найден для типа `&Self` в текущей области видимости.

Давайте посмотрим, что происходит, когда мы пытаемся реализовать `OutlinePrint` в типе, который не реализует типаж `Display`. Примером служит структура `Point`:

src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

impl OutlinePrint for Point {}
```

Мы получаем ошибку, сообщающую о том, что необходим типаж `Display`, но он не реализован:

```
error[E0277]: the trait bound `Point: std::fmt::Display` is not satisfied
--> src/main.rs:20:6
   |
20 | impl OutlinePrint for Point {}
   |      ^^^^^^^^^^^^ `Point` cannot be formatted with the default formatter;
try using `:?` instead if you are using a format string
   |
= help: the trait `std::fmt::Display` is not implemented for `Point`
```

Для того чтобы это исправить, мы реализуем типаж `Display` в структуре `Point` и удовлетворяем ограничение, которое требует `OutlinePrint`:

src/main.rs

```
use std::fmt;

impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}
```

Тогда реализация типажа `OutlinePrint` в структуре `Point` будет успешно скомпилирована, и мы сможем вызывать `outline_print` для экземпляра структуры `Point`, чтобы выводить ее на экран в контуре из звездочек.

Использование паттерна newtype для реализации внешних типажей во внешних типах

В разделе «Реализация типажа в типе» (с. 224) мы упомянули о сиротском правиле, которое гласит, что нам разрешено реализовывать типаж в типе до тех пор, пока либо типаж, либо тип являются локальными для упаковки. Обойти это ограничение можно с помощью паттерна newtype, который включает в себя создание нового типа в кортежной структуре. (Мы рассмотрели кортежные структуры в разделе «Использование кортежных структур без именованных полей для создания разных типов» (с. 119).) Кортежная структура будет иметь одно поле и станет тонкой оберткой вокруг типа, для которого мы хотим реализовать типаж. Тогда оберточный тип является локальным для упаковки, и мы можем реализовать типаж в обертке. Newtype («новый тип») — это термин, который пришел из языка Haskell. При использовании этого паттерна производительность времени выполнения не страдает, а оберточный тип элиминируется во время компиляции.

Допустим, мы хотим реализовать типаж `Display` для типа `Vec<T>`, что сиротское правило запрещает делать непосредственно, потому что типаж `Display` и тип `Vec<T>` определены вне упаковки. Мы можем создать структуру `Wrapper`, содержа-

щую экземпляр типа `Vec<T>`, затем реализовать типаж `Display` для типа `Wrapper` и использовать значение типа `Vec<T>`, как показано в листинге 19.23.

Листинг 19.23. Создание типа `Wrapper` вокруг типа `Vec<String>` для реализации типажа `Display`

src/main.rs

```
use std::fmt;

struct Wrapper(Vec<String>);

impl fmt::Display for Wrapper {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "[{}]", self.0.join(", "))
    }
}

fn main() {
    let w = Wrapper(vec![String::from("hello"), String::from("world")]);
    println!("w = {}", w);
}
```

Реализация типажа `Display` использует `self.0` для доступа к внутреннему значению типа `Vec<T>`, потому что `Wrapper` — это кортежная структура, а значение типа `Vec<T>` — элемент с индексом 0 в кортеже. Тогда мы можем использовать функциональность типажа `Display` в типе `Wrapper`.

Недостаток использования этого технического приема в том, что `Wrapper` — это новый тип, поэтому у него нет методов для значения, которое он содержит. Мы должны были бы реализовать все методы типа `Vec<T>` непосредственно в типе `Wrapper` таким образом, чтобы методы делегировали полномочия `self.0`, — это позволило бы нам рассматривать `Wrapper` точно так же, как `Vec<T>`. Если бы мы хотели, чтобы новый тип имел все методы, которые есть у внутреннего типа, то решение состояло бы в реализации типажа `Deref` (обсуждается в разделе «Трактовка умных указателей как обыкновенных ссылок с помощью типажа Deref» (с. 364)) в типе `Wrapper` для того, чтобы возвращать внутренний тип. Если мы не хотим, чтобы тип `Wrapper` имел все методы внутреннего типа, например, чтобы ограничить поведение `Wrapper`, то нам нужно реализовывать те методы, которые мы действительно хотим, вручную.

Теперь вы знаете, как паттерн newtype используется применительно к типажам. Этот паттерн полезен, даже когда типажи не задействуются. Давайте займемся некоторыми эффективными способами взаимодействия с системой типов Rust.

Продвинутые типы

В системе типов Rust есть некоторые средства, о которых мы упоминали, но еще не обсуждали. Мы начнем с типов newtype в целом, поскольку мы исследуем, по-

чему они полезны как типы. Затем мы перейдем к псевдонимам типов, языковому средству, похожему на newtype, но с несколько иной семантикой. Мы также обсудим тип ! и динамически изменяемые типы.

ПРИМЕЧАНИЕ

Чтобы освоить материал следующего раздела, вам необходимо ознакомиться с разделом «Использование паттерна newtype для реализации внешних типажей во внешних типах» (с. 504).

Использование паттерна newtype для безопасности типов и абстракции

Паттерн newtype полезен для решения задач, выходящих за рамки рассмотренных примеров, включая статическое обеспечение того, чтобы значения не перемешивались и указывались единицы измерения значения. Вы видели пример использования типов newtype для обозначения единиц измерения в листинге 19.15: вспомните, что структуры `Millimeters` и `Meters` обертывали значения типа `u32` в newtype. Если бы мы написали функцию с параметром типа `Millimeters`, то не смогли бы скомпилировать программу, которая ненароком попыталась бы вызвать эту функцию со значением типа `Meters` или простым типом `u32`.

Еще одно применение паттерна newtype заключается в абстрагировании некоторых деталей реализации типа: новый тип может предоставлять публичный API, отличный от API приватного внутреннего типа, если мы используем новый тип непосредственно, к примеру, для ограничения имеющейся функциональности.

Типы newtype также могут скрывать внутреннюю реализацию. Например, мы можем предоставить тип `People` для обертывания `HashMap<i32, String>`, который хранит идентификатор человека, связанный с его именем. Код, использующий `People`, будет взаимодействовать только с публичным API, который мы предоставляем, например, метод добавления строки имени в коллекцию `People`. Этому коду не нужно будет знать, что мы назначаем идентификатор `i32` именам внутри. Паттерн newtype — это легкий способ достижения инкапсуляции для сокрытия деталей реализации, который мы обсуждали в разделе «Инкапсуляция, которая скрывает детали реализации» (с. 432).

Создание синонимов типов с помощью псевдонимов типов

Наряду с паттерном newtype язык Rust предоставляет возможность объявлять псевдоним типа, чтобы давать существующему типу другое имя. Для этого мы используем ключевое слово `type`. Например, можно создать псевдоним `Kilometers` для типа `i32` примерно так:

```
type Kilometers = i32;
```

Теперь псевдоним `Kilometers` является синонимом типа `i32`. В отличие от типов `Millimeters` и `Meters`, которые мы создали в листинге 19.15, `Kilometers` не является отдельным новым типом. Значения, имеющие тип `Kilometers`, будут трактоваться так же, как и значения типа `i32`:

```
type Kilometers = i32;

let x: i32 = 5;
let y: Kilometers = 5;

println!("x + y = {}", x + y);
```

Поскольку `Kilometers` и `i32` имеют одинаковый тип, мы можем добавлять значения обоих типов и передавать значения `Kilometers` функциям, берущим параметры типа `i32`. Однако, используя этот метод, мы не получаем преимуществ от проверки типов, которые есть при применении паттерна newtype, рассмотренного ранее.

Синонимы чаще всего используются для уменьшения повторов. Например, у нас может быть такой длинный тип:

```
Box<dyn Fn() + Send + 'static>
```

Писать его в сигнатурах функций и аннотациях типов по всему коду бывает утомительно, и могут возникнуть ошибки. Представьте себе, что ваш проект наполнен кодом такого рода, как в листинге 19.24.

Листинг 19.24. Использование длинного типа во многих местах

```
let f: Box<dyn Fn() + Send + 'static> = Box::new(|| println!("привет"));

fn takes_long_type(f: Box<dyn Fn() + Send + 'static>) {
    // --пропуск--
}

fn returns_long_type() -> Box<dyn Fn() + Send + 'static> {
    // --пропуск--
}
```

Псевдоним типа делает этот код более управляемым, уменьшая повторы. В листинг 19.25 мы ввели псевдоним `Thunk` для многословного типа и можем заменить все варианты его использования более коротким псевдонимом `Thunk`.

Листинг 19.25. Введение псевдонима `Thunk` для уменьшения повторов

```
type Thunk = Box<dyn Fn() + Send + 'static>;

let f: Thunk = Box::new(|| println!("привет"));

fn takes_long_type(f: Thunk) {
    // --пропуск--
}

fn returns_long_type() -> Thunk {
    // --пропуск--
}
```

Этот код намного легче читать и писать! Выбор осмысленного имени для псевдонима типа также помогает сообщать ваше намерение (thunk («заставка») — это слово для кода, который будет вычислен позже, поэтому данное имя уместно для сохраняемого замыкания).

Псевдонимы типов также обычно используются с типом `Result<T, E>` для уменьшения повторов. Рассмотрим модуль `std::io` в стандартной библиотеке. Операции ввода-вывода часто возвращают тип `Result<T, E>` для урегулирования ситуаций, когда операции не срабатывают. Эта библиотека имеет структуру `std::io::Error`, которая представляет все возможные ошибки ввода-вывода. Многие функции в `std::io` будут возвращать `Result<T, E>`, где `E` — это `std::io::Error`, такие как эти функции в типаже `Write`:

```
use std::io::Error;
use std::fmt;

pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize, Error>;
    fn flush(&mut self) -> Result<(), Error>;

    fn write_all(&mut self, buf: &[u8]) -> Result<(), Error>;
    fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<(), Error>;
}
```

Фрагмент `Result<..., Error>` часто повторяется. По этой причине `std::io` имеет в объявлении псевдонима такой тип:

```
type Result<T> = Result<T, std::io::Error>;
```

Поскольку это объявление находится в модуле `std::io`, мы можем использовать полный псевдоним `std::io::Result<T>`, то есть `Result<T, E>`, в котором `E` заполнен как `std::io::Error`. Сигнатуры типажа `Write` в итоге выглядят так:

```
pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()>;
    fn write_fmt(&mut self, fmt: Arguments) -> Result<()>;
}
```

Псевдоним типа помогает в двух отношениях: он облегчает написание кода и обеспечивает согласованный интерфейс для всех `std::io`. Поскольку он представляет собой псевдоним, то это просто еще один `Result<T, E>`, а значит, мы можем использовать любые методы, которые работают с типом `Result<T, E>`, а также специальный синтаксис, такой как оператор `?`.

Тип never, который никогда не возвращается

В Rust есть особый тип под названием `!`, который известен как пустой тип, потому что не имеет значений. Мы предпочитаем называть его типом never, потому что

он выступает вместо возвращаемого типа, когда функция никогда ничего не возвращает. Вот его пример:

```
fn bar() -> ! {
    // --пропуск--
}
```

Этот код читается как «эта функция никогда ничего не возвращает». Функции, которые никогда ничего не возвращают, называются *отклоняющимися функциями*. Мы не можем создавать значения типа `!`, и поэтому `bar` никогда не сможет ничего вернуть.

Но какая польза от типа, для которого вы никогда не сможете создавать значения? Вспомните код из листинга 2.5, мы воспроизвели его часть ниже, в листинге 19.26.

Листинг 19.26. Выражение match с рукавом, который заканчивается в continue

```
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};
```

В то время мы пропустили некоторые детали в этом коде. В разделе «Выражение `match` как оператор управления потоком» (с. 139) мы отметили, что все рукава выражения `match` должны возвращать один и тот же тип. Поэтому, например, следующий код не работает:

```
let guess = match guess.trim().parse() {
    Ok(_) => 5,
    Err(_) => "здравствуй",
}
```

Тип переменной `guess` в этом коде должен быть целым числом и строкой, а компилятор требует, чтобы переменная `guess` имела только один тип. Тогда что возвращает `continue`? Как же нам разрешили вернуть тип `u32` из одного рукава и иметь другой рукав, который заканчивается на инструкции `continue` в листинге 19.26?

Как вы могли догадаться, инструкция `continue` имеет значение типа `!`. То есть когда Rust вычисляет тип переменной `guess`, он смотрит на оба рукава выражения `match`, первое со значением типа `u32`, а второе со значением типа `!`. Так как тип `!` никогда не может иметь значения, Rust решает, что типом переменной `guess` является `u32`.

Формальным способом описания такого поведения является то, что выражения типа `!` могут принудительно приводиться к любому другому типу. Нам разрешено завершить этот рукав выражения `match` инструкцией `continue`, потому что `continue` не возвращает значение; вместо этого он перемещает управление обрат-

но в верхнюю часть цикла, поэтому в случае варианта `Err` мы никогда не передаем значение переменной `guess`.

Тип never также полезен с макрокомандой `panic!`. Помните метод `unwrap`, который мы вызываем для значений типа `Option<T>`, чтобы создать значение или поднять панику? Вот его определение:

```
impl<T> Option<T> {
    pub fn unwrap(self) -> T {
        match self {
            Some(val) => val,
            None => panic!("вызвана `Option::unwrap()` для значения `None`"),
        }
    }
}
```

В этом коде происходит то же самое, что и в выражении `match` в листинге 19.26: Rust видит, что `val` имеет тип `T`, а макрокоманда `panic!` имеет свой тип `!`, поэтому результатом совокупного выражения `match` является `T`. Этот код работает, потому что макрокоманда `panic!` не создает значение, а завершает программу. В случае варианта `None` мы не будем возвращать значение из метода `unwrap`, поэтому этот код допустим.

И последним выражением с типом `!` является цикл `loop`:

```
print!("на веки ");

loop {
    print!("вечные ");
}
```

Здесь `loop` никогда не кончается, и поэтому тип `!` является значением выражения. Однако оно не было бы истинным, если бы мы включили инструкцию `break`, потому что цикл заканчивался бы, дойдя до инструкции `break`.

Динамически изменяемые типы и типаж Sized

Из-за потребности языка Rust знать некоторые детали, например, сколько места нужно выделить для значения того или иного типа, в его системе типов имеется уголок, который может сбить с толку: речь об идее динамически изменяемых типов. Иногда называемые DST от англ. *dynamically sized type*, или безразмерными типами, они позволяют писать код со значениями, размер которых известен только во время выполнения.

Давайте углубимся в детали динамически изменяемого типа под названием `str`, который мы использовали на протяжении всей книги. Все верно, не `&str`, а `str` в отдельности является динамически изменяемым типом. Мы не знаем длину

строки до наступления выполнения, то есть мы не можем ни создать переменную типа `str`, ни взять аргумент типа `str`. Рассмотрим следующий фрагмент кода, который не работает:

```
let s1: str = "Привет!";
let s2: str = "Как дела?";
```

Rust должен знать, сколько памяти нужно выделить для любого значения некоторого типа, и все значения этого типа должны использовать одинаковый объем памяти. Если бы Rust позволял нам писать этот код, то оба значения `str` должны были бы занимать одинаковый объем пространства. Но они имеют разную длину: `s1` требуется 12 байт памяти, а `s2` — 15. Вот почему невозможно создать переменную, содержащую динамически изменяемый тип.

Тогда что же нам делать? В этом случае вы уже знаете ответ: мы создаем переменные `s1` и `s2` с типом `&str` вместо `str`. Напомним, в разделе «Строковые срезы» (с. 111) мы отметили, что срезовая структура данных хранит начальную позицию и длину среза.

Поэтому, в отличие от ссылки `&T`, представляющей собой одинарное значение, которое хранит адрес памяти, где находится `T`, ссылка `&str` состоит из двух значений: адреса `str` и его длины. По этой причине мы можем знать размер значения типа `&str` во время компиляции: он в два раза больше длины типа `usize`. То есть мы всегда знаем размер `&str`, независимо от того, насколько длинное строковое значение, на которое она ссылается. В общем случае именно таким образом динамически изменяемые типы используются в Rust: они имеют дополнительный кусок метаданных, который хранит размер динамической информации. Золотое правило динамически изменяемых типов состоит в том, что мы всегда должны помещать значения динамически изменяемых типов за какой-либо указатель.

Мы можем комбинировать `str` со всеми видами указателей: например, умными указателями `Box<str>` или `Rc<str>`. На самом деле вы уже встречали это раньше, но с другим динамически изменяемым типом — типажом. Каждый типаж является динамически изменяемым типом, на который мы можем ссылаться, используя имя типажа. В разделе «Использование типажных объектов, допускающих значения разных типов» (с. 435), мы упоминали, что для использования типажей в качестве типажных объектов мы должны их ставить за указатель, например, `&dyn Trait` или `Box<dyn Trait>` (`Rc<dyn Trait>` тоже будет работать).

Для работы с динамически изменяемыми типами в языке Rust имеется отдельный типаж под названием `Sized`, чтобы выяснять, известен ли размер типа во время компиляции. Указанный типаж автоматически реализуется для всего того, чей размер известен во время компиляции. В дополнение к этому, язык Rust неявно добавляет к типажу `Sized` границу в каждую обобщенную функцию. То есть определение обобщенной функции, подобное этому.

```
fn generic<T>(t: T) {
    // --пропуск--
}
```

На самом деле это определение рассматривается так, как будто мы написали вот это:

```
fn generic<T: Sized>(t: T) {
    // --пропуск--
}
```

По умолчанию обобщенные функции будут работать только с типами, размер которых известен во время компиляции. Однако для ослабления этого ограничения можно использовать специальный синтаксис:

```
fn generic<T: ?Sized>(t: &T) {
    // --пропуск--
}
```

Граница типажа для `?Sized` противоположна границе типажа `Sized`: мы читаем это как «`T` может быть, а может и не быть `Sized`». Этот синтаксис доступен только для `Sized`, а не для других типажей.

Также обратите внимание, что мы переключили тип параметра `t` с `T` на `&T`. Поскольку тип может и не быть `Sized`, нужно использовать его за каким-то указателем. В данном случае мы выбрали ссылку.

Далее мы поговорим о функциях и замыканиях.

Продвинутые функции и замыкания

Наконец мы познакомимся с несколькими эффективными языковыми средствами, связанными с функциями и замыканиями, которые включают указатели функций и возвращение замыканий.

Указатели функций

Мы уже говорили о том, как передавать замыкания в функции. Вы также можете передавать регулярные функции в функции! Этот технический прием полезен, когда нужно передать функцию, которую вы уже определили, а не определять новое замыкание. Его выполнение с указателями функций позволит использовать функции в качестве аргументов для других функций. Функции принудительно приводятся к типу `fn` (со строчной буквой f), не путать с типажом `Fn` замыкания. Тип `fn` называется указателем функции. Синтаксис, который описывает, что параметр является указателем функции, аналогичен синтаксису замыканий, как показано в листинге 19.27.

Листинг 19.27. Использование типа fn для принятия указателя функции в качестве аргумента

src/main.rs

```
fn add_one(x: i32) -> i32 {
    x + 1
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("Ответ равен {}", answer);
}
```

Этот код выводит:

```
Ответ равен 12
```

Мы указываем, что параметр `f` в функции `do_twice` является типом `fn`, который берет один параметр типа `i32` и возвращает тип `i32`. Затем мы можем вызвать `f` в теле функции `do_twice`. В функции `main` мы можем передать имя функции `add_one` в качестве первого аргумента функции `do_twice`.

В отличие от замыканий, тип `fn` представляет собой тип, а не типаж, поэтому мы указываем `fn` как тип параметра напрямую, а не объявляем параметр обобщенного типа с одним из типажей `Fn` в качестве связанного типажа.

Указатели функций реализуют все три типажа замыкания (`Fn`, `FnMut` и `FnOnce`), поэтому вы всегда можете передавать указатель функции в качестве аргумента функции, которая ожидает замыкания. Лучше всего писать функции, используя обобщенный тип и один из типажей замыкания, в результате чего функции будут принимать либо функции, либо замыкания.

Пример, где нужно принимать только тип `fn`, а не замыкания — это взаимодействие с внешним кодом, который не имеет замыканий: функции C могут принимать функции в качестве аргументов, но C не имеет замыканий.

В качестве примера того, где вы можете использовать либо замыкание, определенное внутри строки, либо именованную функцию, давайте рассмотрим функцию `map`. Чтобы использовать `map` для преобразования вектора чисел в вектор строк, мы могли бы применить замыкание, подобное этому:

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> = list_of_numbers
    .iter()
    .map(|i| i.to_string())
    .collect();
```

Либо в качестве аргумента функции `map` можно назначить функцию вместо замыкания, как тут:

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> = list_of_numbers
    .iter()
    .map(ToString::to_string)
    .collect();
```

Обратите внимание, мы должны использовать полный синтаксис, о котором говорилось в разделе «Продвинутые типажи» (с. 494), поскольку существует много функций с именем `to_string`. Здесь мы используем функцию `to_string`, определенную в типаже `ToString`, выполняемом стандартной библиотекой для любого типа, который реализует типаж `Display`.

Некоторым нравится этот стиль, а другие предпочитают замыкание. В итоге оба стиля компилируются в один и тот же код, поэтому используйте тот стиль, который вам понятнее.

У нас есть еще один полезный паттерн, в котором задействованы детали реализации кортежных структур и вариантов перечисления кортежных структур. Эти типы используют `()` в качестве инициализирующего синтаксиса, который выглядит как вызов функции. Инициализаторы фактически реализованы как функции, возвращающие экземпляр, который выводится из их аргументов. Мы можем использовать эти инициализирующие функции в качестве указателей функций, реализующих типажи замыкания, а значит, можно указывать инициализирующие функции в качестве аргументов методов, принимающих замыкания, например:

```
enum Status {
    Value(u32),
    Stop,
}

let list_of_statuses: Vec<Status> =
    (0u32..20)
    .map(Status::Value)
    .collect();
```

Здесь мы создаем экземпляры `Status::Value`, используя каждое значение типа `u32` в интервале, для которого вызывается функция `map`, за счет инициализирующей функции `Status::Value`. Некоторым нравится этот стиль, другие предпочитают замыкание. Оба стиля компилируются в один и тот же код, поэтому используйте тот стиль, который вам понятнее.

Возвращающие замыкания

Замыкания представлены типажами, а значит, вы не можете возвращать замыкания напрямую. В большинстве случаев, когда требуется вернуть типаж, вместо него

можно использовать конкретный тип, реализующий типаж в качестве значения, возвращаемого из функции. Но вы не можете делать это с замыканиями, потому что у них нет конкретного типа, который можно вернуть. Например, вы не можете использовать указатель функции `fn` в качестве типа, возвращаемого из функции.

Код ниже пытается вернуть замыкание напрямую, но он не компилируется:

```
fn returns_closure() -> Fn(i32) -> i32 {
    |x| x + 1
}
```

Ошибка компилятора заключается в следующем:

```
error[E0277]: the trait bound `std::ops::Fn(i32) -> i32 + `static:
std::marker::Sized` is not satisfied
 -->
  |
1 | fn returns_closure() -> Fn(i32) -> i32 {
  |                         ^^^^^^^^^^^^^^ `std::ops::Fn(i32) -> i32 +
'static` does not have a constant size known at compile-time
  |
  = help: the trait `std::marker::Sized` is not implemented for
  `std::ops::Fn(i32) -> i32 + 'static`
  = note: the return type of a function must have a statically known size
```

Ошибка снова ссылается на типаж `Sized`! Язык Rust не знает, сколько пространства ему понадобится для хранения замыкания. Ранее мы уже видели решение этой проблемы. Мы можем использовать типажный объект:

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

Этот код будет компилироваться без проблем. Дополнительные сведения о типажных объектах смотрите в разделе «Использование типажных объектов, допускающих значения разных типов» (с. 435).

Далее давайте посмотрим на макрокоманды!

Макрокоманды

Мы использовали макрокоманды, такие как `println!`, на протяжении всей книги, но мы еще не полностью изучили сам термин «макрокоманда» и способ ее действия. Термин «макрокоманда» относится к семейству средств языка Rust: декларативным макрокомандам с помощью макрокоманды `macro_rules!` и трем видам процедурных макрокоманд:

- Настраиваемым макрокомандам `#[derive]`, которые задают код, добавляемый при помощи атрибута `derive`, используемого в структурах и перечислениях.

- Макрокоманды, подобные атрибутам, которые определяют настраиваемые атрибуты, используемые с любым элементом.
- Макрокоманды, подобные функциям, которые выглядят как вызовы функций, но работают с токенами, указываемыми в качестве их аргумента.

Мы поговорим о каждой из них, но сначала давайте посмотрим, зачем вообще нужны макрокоманды, когда есть функции.

Разница между макрокомандами и функциями

В своей основе макрокоманды — это способ написания кода, именуемый «метапрограммированием», который пишет другой код. В приложении В в конце книги мы обсудим атрибут `derive`, который генерирует реализацию различных типажей. Мы также использовали макрокоманды `println!` и `vec!`. Все эти макрокоманды расширяются, производя больше кода, чем при написании вручную.

Метапрограммирование полезно тем, что оно сокращает объем кода, который вы должны писать и обслуживать, что также является одной из ролей функций. Однако макрокоманды имеют некоторые дополнительные свойства, которых нет у функций.

Сигнатура функции должна объявлять число и тип параметров, которыми обладает функция. Макрокоманды, с другой стороны, могут брать переменное число параметров: мы можем вызвать макрокоманду `println!("Здравствуй")` с одним аргументом или `println!("Здравствуй{}", имя)` с двумя аргументами. Кроме того, макрокоманды расширяются перед тем, как компилятор примется за интерпретацию кода, поэтому макрокоманда может, например, реализовать типаж в заданном типе. Функция же не может, потому что она вызывается во время выполнения, а типаж должен быть реализован во время компиляции.

Недостаток реализации макрокоманды вместо функции состоит в том, что определять макрокоманды сложнее, чем функции, потому что вы пишете код Rust, который пишет код Rust. Из-за этой опосредованности определения макрокоманд, как правило, труднее читать, понимать и сопровождать, чем определения функций.

Еще одно важное различие между макрокомандами и функциями заключается в том, что перед вызовом макрокоманд в файле их необходимо определить или ввести в область видимости, в отличие от функций, которые можно определять и вызывать в любом месте.

Декларативные макрокоманды с помощью macro_rules! для общего метапрограммирования

Наиболее широко используемой формой макрокоманд в Rust являются декларативные макрокоманды. Их также иногда называют «макрокомандами на приме-

рах», «макрокомандами `macro_rules!`» или просто «макросами». По своей сути декларативные макрокоманды позволяют писать что-то похожее на выражение `match` языка Rust. Как отмечалось в главе 6, выражения `match` — это управляющие структуры, которые берут выражение, сравнивают результирующее значение выражения с паттернами, а затем исполняют код, связанный с совпадающим паттерном. Макрокоманды тоже сравнивают значение с паттернами, связанными с тем или иным кодом: в этом случае значение является литералом исходного кода Rust, переданного в макрокоманду. Шаблоны сравниваются со структурой этого исходного кода, а код, связанный с каждым паттерном, при совпадении заменяет код, переданный в макрокоманду. Все это происходит во время компиляции.

Для того чтобы определить макрокоманду, вы используете конструкцию `macro_rules!`. Давайте узнаем, как применять `macro_rules!`, глядя на определение макрокоманды `vec!`. Глава 8 посвящена тому, как использовать макрокоманду `vec!` для создания нового вектора с конкретными значениями. Например, следующая макрокоманда создает новый вектор, содержащий три целых числа:

```
let v: Vec<u32> = vec![1, 2, 3];
```

Мы могли бы также использовать макрокоманду `vec!` для создания вектора из двух целых чисел или вектора из пяти строковых срезов. Мы не сможем использовать функцию, чтобы сделать то же самое, потому что мы не будем знать заранее число или тип значений.

Листинг 19.28 показывает несколько упрощенное определение макрокоманды `vec!`.

Листинг 19.28. Упрощенная версия определения макрокоманды vec!

src/lib.rs

```
❶ #[macro_export]
❷ macro_rules! vec {
    ❸ ( $( $x:expr ),* ) => {
          {
              let mut temp_vec = Vec::new();
           ❹ $(
                ❺ temp_vec.push($x❻);
                  )*
           ❼ temp_vec
          }
      };
  }
```

ПРИМЕЧАНИЕ

Фактическое определение макрокоманды `vec!` в стандартной библиотеке содержит код для предварительного выделения нужного объема памяти. Это код-оптимизация, он сюда не включен, чтобы упростить пример.

Аннотация `#[macro_export]` ❶ указывает, что эта макрокоманда должна быть доступна всякий раз, когда упаковка, в которой макрокоманда определена, вводится в область видимости. Без этой аннотации макрокоманда не может быть введена в область видимости.

Затем мы начинаем определение макрокоманды с помощью макрокоманды `macro_rules!` и приводим имя макрокоманды, которую определяем, без восклицательного знака ❷. Имя, в данном случае `vec`, сопровождается фигурными скобками, обозначающими тело определения макрокоманды.

Структура кода в теле `vec!` похожа на структуру выражения `match`. Здесь мы имеем один рукав с паттерном `( $( $x:expr ),* )`, за которым следует `=>` и блок кода, связанный с этим паттерном ❸. Если паттерн совпадает, то будет создан связанный блок кода. Учитывая, что это единственный паттерн в этой макрокоманде, существует только один допустимый способ совпадения, любой другой паттерн приведет к ошибке. У более сложных макрокоманд будет более одного рукава.

Допустимый синтаксис паттернов в определениях макрокоманд отличается от синтаксиса паттернов, описанного в главе 18, поскольку паттерны макрокоманд сопоставляются со структурой, а не со значениями кода Rust. Давайте изучим, что означают фрагменты паттерна в листинге 19.28. Полный синтаксис паттернов макрокоманд смотрите по адресу https://doc.rust-lang.org/stable/reference/macros.html.

Сначала весь паттерн охватывается набором скобок. Далее следует знак доллара (`$`), за которым идет набор скобок, захватывающий значения, совпадающие с паттерном внутри скобок, для использования в заменяющем коде. Внутри `$()` находится выражение `$x:expr`, которое совпадает с любым выражением языка Rust и дает выражению имя `$x`.

Запятая, следующая за `$()`, указывает, что символ-разделитель литерала запятой, как вариант, мог бы появиться после кода, совпадающего с кодом в `$()`. Символ * указывает на то, что паттерн совпадает с нулем или более из того, что предшествует *.

Когда мы вызываем эту макрокоманду с помощью `vec![1, 2, 3];`, паттерн `$x` трижды совпадает с тремя выражениями — `1`, `2` и `3`.

Теперь давайте посмотрим на паттерн в теле кода, связанного с этим рукавом: `temp_vec.push()` ❺ внутри `$()*` ❹ ❼ генерируется для каждой части, которая совпадает с `$()` в паттерне ноль или более раз в зависимости от того, сколько раз совпадает паттерн. Имя `$x` ❻ заменяется при каждом совпавшем выражении. Когда мы вызовем эту макрокоманду с `vec![1, 2, 3];`, генерируемый код, который заменяет этот макровызов, будет следующим:

```
let mut temp_vec = Vec::new();
temp_vec.push(1);
temp_vec.push(2);
temp_vec.push(3);
temp_vec
```

Мы определили макрокоманду, которая берет любое число аргументов любого типа и может генерировать код для создания вектора, содержащего указанные элементы.

У декларативной макрокоманды `macro_rules!` есть несколько странных пограничных случаев. В будущем в Rust появится второй вид декларативной макрокоманды, который будет работать схожим образом, но при этом пограничные случаи будут исправлены. После этого обновления макрокоманду `macro_rules!` практически объявят устаревшей. Имея это в виду, а также тот факт, что большинство программистов Rust будут больше использовать макрокоманды, чем писать их, мы не будем дальше обсуждать макрокоманду `macro_rules!`. Для того чтобы узнать больше о том, как писать макрокоманды, обратитесь к документации или другим ресурсам, таким как «Маленькая книга макрокоманд языка Rust» (*The Little Book of Rust Macros*) по адресу https://danielkeep.github.io/tlborm/book/index.html.

Процедурные макрокоманды для генерирования кода из атрибутов

Вторая форма макрокоманд — это процедурные макрокоманды, которые действуют больше как функции (и являются типом процедуры). Процедурные макрокоманды принимают некий код на входе, работают с этим кодом и производят некий код на выходе вместо сопоставления с паттернами и замены одного кода другим, как это делают декларативные макрокоманды.

Все три вида процедурных макрокоманд (настраиваемые с атрибутом `derive`, подобные атрибутам и подобные функциям) работают похожим образом.

При создании процедурных макрокоманд их определения должны находиться в собственной упаковке со специальным типом. Это происходит по техническим причинам, которые мы надеемся устранить в будущем. Использование процедурных макрокоманд выглядит как код в листинге 19.29, где `some_attribute` является заполнителем для использования конкретной макрокоманды.

Листинг 19.29. Пример использования процедурной макрокоманды

src/lib.rs

```
use proc_macro;

#[some_attribute]
pub fn some_name(input: TokenStream) -> TokenStream {
}
```

Функция, которая определяет процедурную макрокоманду, берет `TokenStream` на входе и производит `TokenStream` на выходе. Тип `TokenStream` определяется упаковкой `proc_macro`. Он входит в состав языка Rust и представляет собой последовательность токенов. Это ядро макрокоманды: исходный код, с которым работает макрокоманда, образует входной `TokenStream`, а код, производящий макрокоманду, является выходным `TokenStream`. Функция также имеет атрибут, который

указывает, какой тип процедурной макрокоманды мы создаем. В одной упаковке можно иметь несколько видов процедурных макрокоманд.

Давайте рассмотрим разные виды процедурных макрокоманд. Мы начнем с настраиваемой макрокоманды `derive`, а затем объясним небольшие различия, которые есть у других форм.

Как написать настраиваемую макрокоманду derive

Давайте создадим упаковку с именем `hello_macro`, которая определяет типаж `HelloMacro` с одной связанной функцией `hello_macro`. Вместо того чтобы заставлять пользователей упаковки реализовывать типаж `HelloMacro` в каждом типе, мы предоставим процедурную макрокоманду, с помощью которой пользователи могут аннотировать свой тип с `#[derive(HelloMacro)]`, получая реализацию по умолчанию функции `hello_macro`. Реализация по умолчанию будет печатать `Здравствуй, Макро! Меня зовут TypeName!`, где `TypeName` — это имя типа, для которого был определен этот типаж. Другими словами, мы напишем упаковку, которая позволит другому программисту писать код, подобный представленному в листинге 19.30, используя нашу упаковку.

Листинг 19.30. Код, который пользователь нашей упаковки сможет писать во время использования процедурной макрокоманды

src/main.rs

```
use hello_macro::HelloMacro;
use hello_macro_derive::HelloMacro;

#[derive(HelloMacro)]
struct Pancakes;

fn main() {
    Pancakes::hello_macro();
}
```

Когда дело будет сделано, этот код выведет:

```
Здравствуй, Макро! Меня зовут Pancakes!
```

Первый шаг — создать новую библиотечную упаковку, как показано ниже:

```
$ cargo new hello_macro --lib
```

Далее мы определим типаж `HelloMacro` и связанную с ним функцию:

src/lib.rs

```
pub trait HelloMacro {
    fn hello_macro();
}
```

У нас есть типаж и его функция. Здесь пользователь упаковки может реализовать этот типаж для достижения желаемой функциональности, например:

```
use hello_macro::HelloMacro;

struct Pancakes;

impl HelloMacro for Pancakes {
    fn hello_macro() {
        println!("Здравствуй, Макро! Меня зовут Pancakes!");
    }
}

fn main() {
    Pancakes::hello_macro();
}
```

Однако ему нужно будет написать блок реализации для каждого типа, который он хотел бы использовать с `hello_macro`. Мы хотим избавить его от необходимости выполнять эту работу.

Кроме того, мы пока не можем предоставить функцию `hello_macro` с реализацией по умолчанию, которая будет печатать имя типа, в котором выполнен типаж. В языке Rust нет возможности рефлексии (интроспекции), поэтому он не может найти имя типа во время выполнения. Нам нужна макрокоманда, которая будет генерировать код во время компиляции.

Следующий шаг — определить процедурную макрокоманду. На момент написания книги процедурные макрокоманды должны находиться в собственной упаковке. В перспективе это ограничение может быть снято. Общепринято структурировать упаковки и макроупаковки следующим образом: для упаковки с именем `foo` упаковка для настраиваемых процедурных макрокоманд с атрибутом `derive` называется `foo_derive`. Давайте создадим новую упаковку под названием `hello_macro_derive` внутри проекта `hello_macro`:

```
$ cargo new hello_macro_derive --lib
```

Обе упаковки тесно связаны, поэтому мы создаем упаковку для процедурных макрокоманд в каталоге упаковки `hello_macro`. Если мы изменим определение типажа в `hello_macro`, то нам также придется изменить реализацию процедурной макрокоманды в `hello_macro_derive`. Эти две упаковки нужно будет опубликовать по отдельности, а программисты, использующие эти упаковки, должны будут добавить обе в качестве зависимостей и ввести их в область видимости. Вместо этого мы могли бы использовать упаковку `hello_macro` в качестве зависимости и реэкспортировать код процедурной макрокоманды. Однако то, как мы структурировали проект, позволяет программистам использовать `hello_macro`, даже если они не хотят функциональности `derive`.

Нам нужно объявить `hello_macro_derive` как упаковку для процедурных макрокоманд. Нам также понадобится функциональность из упаковок `syn` и `quote`, как вы вскоре увидите, поэтому нужно добавить их в качестве зависимостей. Добавьте в файл Cargo.toml для `hello_macro_derive` следующее:

hello_macro_derive/Cargo.toml

```
[lib]
proc-macro = true

[dependencies]
syn = "0.14.4"
quote = "0.6.3"
```

Для того чтобы начать определение процедурной макрокоманды, поместите код из листинга 19.31 в файл src/lib.rs для упаковки `hello_macro_derive`. Обратите внимание, этот код не будет компилироваться до тех пор, пока мы не добавим определение функции `impl_hello_macro`.

Листинг 19.31. Код, который потребуется для обработки кода Rust большинству упаковок для процедурных макрокоманд

hello_macro_derive/src/lib.rs

```
extern crate proc_macro;

use crate::proc_macro::TokenStream;
use quote::quote;
use syn;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // Сконструировать представление кода Rust в виде синтаксического дерева,
    // которым можно манипулировать
    let ast = syn::parse(input).unwrap();

    // Создать реализацию типажа
    impl_hello_macro(&ast)
}
```

Обратите внимание, мы разделили код на функцию `hello_macro_derive`, которая отвечает за разбор потока `TokenStream`, и функцию `impl_hello_macro`, которая отвечает за преобразование синтаксического дерева: это делает написание процедурной макрокоманды удобнее. Код во внешней функции (в данном случае `hello_macro_derive`) будет одинаковым почти для каждой процедурной макрокоманды, которую вы встретите или создадите. Код, который вы приводите в теле внутренней функции (в данном случае `impl_hello_macro`), будет отличаться в зависимости от назначения процедурной макрокоманды.

Мы ввели три новые упаковки: `proc_macro`, `syn` (доступна на https://crates.io/crates/syn) и `quote` (доступна на https://crates.io/crates/quote). Упаковка `proc_macro` постав-

ляется в комплекте, поэтому нам не нужно было добавлять ее к зависимостям в Cargo.toml. Упаковка `proc_macro` представляет собой API компилятора, который позволяет читать код Rust и манипулировать им из нашего кода.

Упаковка `syn` проводит разбор кода Rust из строкового значения в структуру данных, с которой мы можем выполнять операции. Упаковка `quote` превращает структуры данных `syn` обратно в код Rust. Эти упаковки значительно упрощают выполнение разбора любого вида кода Rust, который мы, возможно, захотели бы обработать: написание полного анализатора кода Rust — задача не из простых.

Функция `hello_macro_derive` вызывается, когда пользователь библиотеки указывает для типа `#[derive (HelloMacro)]`. Это возможно, потому что здесь мы аннотировали функцию `hello_macro_derive` с `proc_macro_derive` и указали имя `HelloMacro`, которое соответствует имени нашего типажа. Этому соглашению следует большинство процедурных макрокоманд.

Функция `hello_macro_derive` сначала конвертирует `input` из `TokenStream` в структуру данных, которую мы затем можем интерпретировать и выполнять с ней операции. Вот тут и вступает в игру `syn`. Функция `parse` в `syn` берет `TokenStream` и возвращает структуру `DeriveInput`, представляющую разобранный код Rust. Листинг 19.32 показывает соответствующие части структуры `DeriveInput`, которые мы получаем при разборе строки `struct Pancakes;`.

Листинг 19.32. Экземпляр структуры DeriveInput, который мы получаем при разборе кода, имеющего атрибут макрокоманды из листинга 19.30

```
DeriveInput {
    // --пропуск--

    ident: Ident {
        ident: "Pancakes",
        span: #0 bytes(95..103)
    },
    data: Struct(
        DataStruct {
            struct_token: Struct,
            fields: Unit,
            semi_token: Some(
                Semi
            )
        }
    )
}
```

Поля этой структуры показывают, что код Rust, который мы разобрали, является пустой (unit) структурой с `ident` (идентификатором, означающим имя) типа `Pancakes`. В этой структуре есть и другие поля для описания всех видов кода Rust. Для получения дополнительной информации обратитесь к документации о `syn` для `DeriveInput` на https://docs.rs/syn/0.14.4/syn/struct.DeriveInput.html.

Вскоре мы определим функцию `impl_hello_macro`, где построим новый код Rust, который мы хотим включить. Но прежде чем мы это сделаем, обратите внимание, выход из макрокоманды `derive` также является `TokenStream`. Возвращаемый `TokenStream` добавляется в код, который пишут пользователи упаковки. Поэтому при компиляции упаковки они получат добавленную функциональность, которую мы предоставляем в модифицированном `TokenStream`.

Вы, возможно, заметили, что мы вызываем метод `unwrap`, чтобы побудить функцию `hello_macro_derive` поднять панику, если вызов функции `syn::parse` здесь не срабатывает. Процедурной макрокоманде необходимо паниковать при ошибках, потому что функции `proc_macro_derive` должны возвращать `TokenStream`, а не `Result`, чтобы соответствовать API процедурных макрокоманд. Мы упростили этот пример за счет метода `unwrap`. В рабочем коде вы должны предоставить более конкретные сообщения об ошибках с помощью макрокоманды `panic!` или метода `expect`.

Теперь, когда у нас есть код для преобразования аннотированного кода Rust из `TokenStream` в экземпляр `DeriveInput`, давайте сгенерируем код, реализующий типаж `HelloMacro` в аннотированном типе, как показано в листинге 19.33.

Листинг 19.33. Реализация типажа HelloMacro с использованием разобранного кода Rust

hello_macro_derive/src/lib.rs

```
fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {
    let name = &ast.ident;
    let gen = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Здравствуйо, Макро! Меня зовут {}", stringify!(#name));
            }
        }
    };
    gen.into()
}
```

Мы получаем экземпляр структуры `Ident`, содержащий имя (идентификатор) аннотированного типа с `ast.ident`. Структура в листинге 19.32 показывает, что, когда мы выполняем функцию `impl_hello_macro` в коде листинга 19.30, `ident`, который мы получаем, будет иметь поле `ident` со значением `"Pancakes"`. Таким образом, переменная `name` в листинге 19.33 будет содержать экземпляр структуры `Ident`, который при печати будет строкой `"Pancakes"`, именем структуры в листинге 19.30.

Макрокоманда `quote!` позволяет определять Rust, который мы хотим возвращать. Компилятор ожидает чего-то отличного от прямого результата исполнения макрокоманды `quote!`, поэтому нужно преобразовать его в `TokenStream`. Мы делаем это, вызывая метод `into`, который использует промежуточное представление и возвращает значение требуемого типа `TokenStream`.

Макрокоманда `quote!` также предоставляет очень интересную механику паттернов: мы можем ввести `#name`, и `quote!` заменит его значением в имени переменной. Вы даже можете выполнять повторение, подобно тому, как работают обычные макрокоманды. Для получения обширных сведений обратитесь к документации упаковки `quote` по адресу https://docs.rs/quote.

Мы хотим, чтобы процедурная макрокоманда генерировала реализацию типажа `HelloMacro` для аннотированного пользователем типа, который можно получить, используя `#name`. У реализации типажа есть одна функция, `hello_macro`, тело которой содержит нужную функциональность: печать `Здравствуй, Марко! Меня зовут`, а затем имени аннотированного типа.

Используемая здесь макрокоманда `stringify!` встроена в Rust. Она берет выражение Rust, такое как `1 + 2`, и во время компиляции преобразовывает его в строковый литерал `"1 + 2"`. Это отличается от макрокоманд `format!` или `println!`, которые вычисляют выражение и затем преобразовывают результат в экземпляр типа `String`. Существует возможность, что `#name` на входе может быть выражением для печати в буквальной форме, поэтому мы используем `stringify!`. Использование `stringify!`, кроме того, сохраняет выделенное пространство путем конвертирования `#name` в строковый литерал во время компиляции.

В этой точке команда `cargo build` должна успешно завершиться как в `hello_macro`, так и в `hello_macro_derive`. Давайте подключим эти упаковки к коду в листинге 19.30, чтобы увидеть процедурную макрокоманду в действии! Создайте новый бинарный проект в каталоге projects, используя `cargo new pancakes`. Нужно добавить `hello_macro` и `hello_macro_derive` в качестве зависимостей в файл Cargo.toml упаковки `pancakes`. Если вы публикуете свои версии `hello_macro` и `hello_macro_derive` в https://crates.io/, то они будут регулярными зависимостями; если нет, то можете указать их в качестве зависимостей от пути следующим образом:

```
[dependencies]
hello_macro = { path = "../hello_macro" }
hello_macro_derive = { path = "../hello_macro/hello_macro_derive" }
```

Поместите код из листинга 19.30 в src/main.rs и выполните команду `cargo run`: она должна вывести `Здравствуй, Марко! Меня зовут Pancakes!`. Реализация типажа `HelloMacro` из процедурной макрокоманды была включена так, чтобы упаковке `pancakes` не требовалость его выполнять. Реализация указанного типажа была добавлена за счет аннотации `#[derive(HelloMacro)]`.

Далее мы узнаем, чем отличаются другие виды процедурных макрокоманд от настраиваемых макрокоманд с атрибутом `derive`.

Макрокоманды, подобные атрибутам

Макрокоманды, подобные атрибутам, похожи на настраиваемые макрокоманды с атрибутом `derive`, но вместо того, чтобы генерировать код для `derive`, они позво-

ляют создавать новые атрибуты. Они также более гибкие: `derive` работает только для структур и перечислений, атрибуты могут применяться и к другим элементам, таким как функции. Вот пример использования макрокоманды, подобной атрибутам: допустим, у вас есть атрибут с именем `route`, который аннотирует функции при использовании каркаса веб-приложения:

```
#[route(GET, "/")]
fn index() {
```

Этот атрибут `#[route]` будет определен программным каркасом как процедурная макрокоманда. Сигнатура функции определения макрокоманд будет выглядеть следующим образом:

```
#[proc_macro_attribute]
pub fn route(attr: TokenStream, item: TokenStream) -> TokenStream {
```

Здесь мы имеем два параметра типа `TokenStream`. Первый — для содержимого атрибута: `GET`, часть «`/`». Второй — это тело элемента, к которому атрибут прикреплен: в данном случае `FN index () {}` и остальная часть тела функции.

Помимо этого, макрокоманды, подобные атрибутам, работают так же, как настраиваемые макрокоманды с атрибутом `derived`: вы создаете упаковку с типом упаковки `proc-macro` и реализуете функцию, которая генерирует нужный код.

Макрокоманды, подобные функциям

Макрокоманды, подобные функциям, определяют макрокоманды, которые выглядят как вызовы функций. Аналогично макрокомандам `macro_rules!` они более гибкие, чем функции: например, они могут брать неизвестное число аргументов. Однако макрокоманды `macro_rules!` могут быть определены только с помощью синтаксиса, похожего на выражение `match`, который мы обсуждали в разделе «Декларативные макрокоманды с помощью `macro_rules!` для общего метапрограммирования» (с. 516). Макрокоманды, подобные функциям, берут параметр `TokenStream`, а их определение манипулирует `TokenStream`, используя код Rust, как и другие два типа процедурных макрокоманд. Пример макрокоманды, подобной функциям, — макрокоманда `sql!`, которую можно вызвать так:

```
let sql = sql!(SELECT * FROM posts WHERE id=1);
```

Эта макрокоманда выполнит разбор инструкции SQL внутри себя и проверит ее на синтаксическую правильность, что представляет собой намного более сложную обработку, чем способна сделать макрокоманда `macro_rules!`. Макрокоманда `sql!` будет определена следующим образом:

```
#[proc_macro]
pub fn sql(input: TokenStream) -> TokenStream {
```

Это определение похоже на сигнатуру настраиваемой макрокоманды с атрибутом `derive`: мы получаем токены, находящиеся внутри скобок, и возвращаем код, который мы хотели сгенерировать.

Итоги

Фух! Теперь в вашем арсенале есть редкие инструменты, о которых вы знаете, что они существуют и задействуются в конкретных обстоятельствах. Мы рассмотрели несколько сложных тем, так что, когда вы столкнетесь с ними в сообщениях об ошибках или в чужом коде, то сможете распознать эти понятия и синтаксис. Используйте эту главу в качестве справочного пособия для поиска решений.

Далее мы применим на практике все, что обсуждали на протяжении книги, и выполним еще один проект!

20 Финальный проект: сборка многопоточного сервера

Путешествие получилось довольно долгим, но мы дошли до конца книги. В этой главе мы вместе создадим еще один проект, чтобы проиллюстрировать идеи из заключительных глав, а также вспомним некоторые предыдущие уроки.

В итоговом проекте мы создадим веб-сервер, который говорит: «Привет!» и выглядит в браузере так, как показано на рис. 20.1.

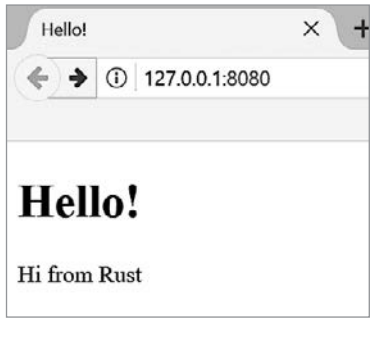

Рис. 20.1. Наш финальный совместный проект

Вот план сборки веб-сервера:

1. Узнать немного о TCP и HTTP.
2. Прослушать TCP-соединения на сокете.
3. Проанализировать небольшое число HTTP-запросов.
4. Создать соответствующий HTTP-ответ.
5. Улучшить пропускную способность сервера с помощью пула потоков исполнения.

Но прежде чем начать, мы должны уточнить одну деталь: метод, который мы будем использовать, — не лучший способ сборки веб-сервера с помощью Rust. Ряд готовых к производству упаковок доступен на https://crates.io/, и все они обеспечивают более полные реализации веб-сервера и пула потоков исполнения, чем та, что сделаем мы.

Однако наша цель — помочь вам освоить материал, а не идти по пути наименьшего сопротивления. Поскольку Rust — это язык системного программирования, мы можем выбрать уровень абстракции, с которым хотим работать, а можем перейти на более низкий уровень, чем это возможно в других языках. Мы напишем базовый HTTP-сервер и пул потоков исполнения вручную, чтобы вы могли усвоить общие идеи и технические приемы, лежащие в основе упаковок, которые вы, возможно, будете использовать в будущем.

Сборка однопоточного сервера

Начнем с того, что приведем в рабочее состояние однопоточный веб-сервер. Прежде чем начать, давайте кратко рассмотрим протоколы, участвующие в создании веб-серверов. Подробности выходят за рамки темы данной книги, поэтому мы кратко предоставим самую важную информацию.

Двумя главными протоколами, используемыми в веб-серверах, являются протокол передачи гипертекста (HTTP) и протокол управления передачей (TCP). Оба являются протоколами запросов-ответов, то есть клиент инициирует запросы, а сервер слушает их и дает клиенту ответ. Содержание этих запросов и ответов определяется протоколами.

Протокол TCP находится на более низком уровне и описывает детали того, как информация поступает из одного сервера на другой, но не указывает, что это за информация. HTTP строится поверх TCP, определяя содержимое запросов и ответов. Технически существует возможность использования HTTP с другими протоколами, но в подавляющем большинстве случаев HTTP отправляет данные по протоколу TCP. Мы будем работать с сырыми байтами запросов и ответов TCP и HTTP.

Прослушивание TCP-соединения

Веб-сервер должен прослушивать TCP-соединение, и поэтому данная часть будет первой, над которой мы будем работать. Стандартная библиотека предлагает модуль `std::net`, позволяющий это сделать. Давайте создадим новый проект обычным образом:

```
$ cargo new hello
     Created binary (application) `hello` project
$ cd hello
```

Теперь для начала введите код из листинга 20.1 в src/main.rs. Этот код будет слушать входящие TCP-потоки по адресу `127.0.0.1:7878`. Получая входящий поток, он будет выводить:

```
Соединение установлено!
```

Листинг 20.1. Прослушивание входящих потоков и печать сообщения при получении потока

src/main.rs

```
use std::net::TcpListener;

fn main() {
 ❶ let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

 ❷ for stream in listener.incoming() {
     ❸ let stream = stream.unwrap();

     ❹ println!("Соединение установлено!");
   }
}
```

Используя `TcpListener`, мы можем прослушивать TCP-соединения по адресу 127.0.0.1: 7878 ❶. В адресе секция перед двоеточием — это IP-адрес, представляющий ваш компьютер (этот адрес одинаков на каждом компьютере и не представляет компьютер определенного автора), а 7878 — это порт. Мы выбрали этот порт по двум причинам: на нем обычно принимается HTTP, а 7878 — это слово rust, набранное на телефоне.

Функция `bind` в этом сценарии работает как функция `new` в том, что она возвращает новый экземпляр типа `TcpListener`. Причина, по которой указанная функция называется `bind` (то есть «привязать»), заключается в том, что в сети подключение к порту для прослушивания называется привязкой к порту.

Функция `bind` возвращает экземпляр типа `Result<T, E>`, который говорит о том, что привязка может быть неуспешной. Например, для подключения к порту 80 требуются права администратора (неадминистраторы могут прослушивать только порты выше 1024), поэтому, если мы попытаемся подключиться к порту 80, не будучи администратором, то привязка не будет работать. Еще один пример, когда привязка не будет работать: если мы запускаем два экземпляра программы, и поэтому две программы прослушивают один и тот же порт. Поскольку мы пишем базовый сервер только для целей обучения, мы не будем беспокоиться об обработке таких ошибок. Мы используем метод `unwrap`, чтобы остановить программу в случае ошибок.

Метод `incoming` в типе `TcpListener` возвращает итератор, который дает последовательность потоков ❷ (более конкретно, потоков типа `TcpStream`). Один поток представляет собой открытое соединение между клиентом и сервером. Соединение — это имя для полного процесса запроса и ответа, в котором клиент подключается к серверу, сервер генерирует ответ и закрывает соединение. Таким образом, `TcpStream` будет читать из себя, чтобы увидеть, что отправил клиент, а затем будет

давать писать ответ в поток. В целом этот цикл `for` будет обрабатывать каждое соединение по очереди и производить серию потоков для обработки.

Пока что обработка потока состоит из вызова метода `unwrap` для завершения программы, если в потоке есть ошибки ❸. Если ошибок нет, то программа выводит сообщение ❹. Мы добавим больше функциональности для случая успеха в следующем листинге. Причина, по которой мы можем получать ошибки от метода `incoming`, когда клиент подключается к серверу, заключается в том, что фактически мы перебираем не соединения, а попытки соединения. Соединение может не быть успешным по ряду причин, многие из которых зависят от операционной системы. Например, многие операционные системы имеют лимит на число одновременно открытых соединений, которые они могут поддерживать. Новые попытки соединения, превышающие это число, будут выдавать ошибку до тех пор, пока некоторые из открытых соединений не будут закрыты.

Давайте попробуем выполнить этот код! Вызовите команду `cargo run` в терминале, а затем загрузите 127.0.0.1:7878 в веб-браузере. Браузер должен выдать сообщение об ошибке наподобие «Connection reset» («Сброс соединения»), так как сервер в данный момент не отправляет обратно никаких данных. Но посмотрев на терминал, вы должны увидеть несколько сообщений, которые были выведены, когда браузер соединился с сервером!

```
     Running `target/debug/hello`
Соединение установлено!
Соединение установлено!
Соединение установлено!
```

Иногда вы увидите, что со стороны браузера выводится несколько сообщений для одного запроса. Причина может быть в том, что браузер делает запрос страницы, а также других ресурсов, таких как иконка favicon.ico, которая появляется на вкладке браузера.

Возможно также, что браузер пытается соединиться с сервером несколько раз, потому что сервер не отвечает данными. Когда переменная `stream` выходит из области видимости и отбрасывается в конце цикла, соединение закрывается как часть реализации функции `drop`. Браузеры иногда регулируют закрытые соединения путем повторной попытки, потому что проблема может быть временной. Важно то, что мы успешно получили дескриптор для TCP-соединения!

Не забудьте остановить программу, нажав Ctrl-C, когда закончите выполнение отдельной версии кода. После внесения любых изменений в код заново выполните команду `cargo run`, чтобы работать с последней версией кода.

Чтение запроса

Давайте реализуем функциональность чтения запроса из браузера! Чтобы сначала получить соединение и затем выполнить некие действия с ним, мы начнем новую

функцию обработки соединений. В этой новой функции `handle_connection` мы будем читать данные из TCP-потока и печатать их, чтобы видеть данные, отправляемые из браузера. Измените код так, чтобы он выглядел как в листинге 20.2.

Листинг 20.2. Чтение из потока TcpStream и печать данных

src/main.rs

```
❶ use std::io::prelude::*;
  use std::net::TcpStream;
  use std::net::TcpListener;

  fn main() {
      let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

      for stream in listener.incoming() {
          let stream = stream.unwrap();

        ❷ handle_connection(stream);
      }
  }

  fn handle_connection(❸mut stream: TcpStream) {
    ❹ let mut buffer = [0; 512];

    ❺ stream.read(&mut buffer).unwrap();

    ❻ println!("Запрос: {}", String::from_utf8_lossy(&buffer[..]));
  }
```

Мы вводим `std::io::prelude` в область видимости, чтобы получить доступ к некоторым типажам, позволяющим читать и писать в поток ❶. В цикле `for` в функции `main`, вместо того, чтобы печатать сообщение о том, что у нас есть соединение, мы теперь вызываем новую функцию `handle_connection` и передаем ей переменную `stream` ❷.

В функции `handle_connection` мы сделали параметр `stream` изменяемым ❸. Причина в том, что внутренне экземпляр типа `TcpStream` отслеживает то, какие данные он нам возвращает. Он может прочитать данные в объеме, превышающем тот, который мы запрашивали, и сохранить их для следующего запроса. Следовательно, ему нужно ключевое слово `mut`, потому что его внутреннее состояние может измениться. Обычно мы думаем, что «чтение» не нуждается в изменении, но в данном случае указанное ключевое слово необходимо.

Далее нам фактически нужно читать из потока. Мы делаем это в два этапа: во-первых, объявляем переменную `buffer` в стеке для хранения считываемых данных ❹. Мы создали буфер размером 512 байт, которого будет достаточно для хранения данных базового запроса и для других целей этой главы. Если бы мы хотели обрабатывать запросы произвольного размера, то управление буфером было бы сложнее. Мы пока оставим его простым. Мы передаем буфер методу `stream.read`, который будет читать байты из `TcpStream` и помещать их в буфер ❺.

Затем мы конвертируем байты в буфере в экземпляр типа `String` и печатаем его ❻. Функция `String::from_utf8_lossy` берет `&[u8]` и производит из него экземпляр типа `String`. Часть имени «lossy» («с потерями») указывает на поведение этой функции при виде недопустимой последовательности UTF-8: она будет заменять недопустимую последовательность символом `U+FFFD`, то есть символом замены, `U+FFFD`. Вы можете увидеть символы замены для символов в буфере, которые не заполнены данными запроса.

Давайте испытаем этот код! Выполните программу и снова сделайте запрос в веб-браузере. Обратите внимание, мы все равно получим страницу ошибки в браузере, но данные программы в терминале теперь будут выглядеть примерно так:

```
$ cargo run
    Compiling hello v0.1.0 (file:///projects/hello)
     Finished dev [unoptimized + debuginfo] target(s) in 0.42 secs
      Running `target/debug/hello`
Request: GET / HTTP/1.1
Host: 127.0.0.1:7878
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:52.0) Gecko/20100101
Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
������������������������������������������
```

В зависимости от вашего браузера вы можете получить немного другие данные. Теперь, печатая данные запроса, мы можем понять, почему получается несколько соединений из одного запроса со стороны браузера, посмотрев на путь после `Request: GET`. Если все повторяющиеся соединения запрашивают ресурс `/`, то мы знаем, что браузер многократно пытается получить ресурс `/`, потому что он не получает ответа от программы.

Давайте разложим данные запроса, чтобы увидеть, что именно браузер запрашивает у программы.

HTTP-запрос

HTTP — это текстовый протокол, запрос принимает такой формат:

```
Method Request-URI HTTP-Version CRLF
headers CRLF
message-body
```

Первая строка формата — это строка запроса, содержащая информацию о том, что клиент запрашивает. Первая часть строки запроса указывает на используемый метод, например `GET` или `POST`, который описывает, как клиент делает запрос. Наш клиент использовал запрос `GET`.

Следующая часть строки запроса — это символ `/`, который указывает на единый идентификатор ресурса (URI), запрашиваемый клиентом: URI почти, но не совсем совпадает с единым локатором ресурсов (URL). Разница между URI- и URL-адресами в этой главе нам не важна, но спецификация HTTP использует термин URI, поэтому здесь можно просто мысленно подставить URL вместо URI.

Последняя часть — это версия HTTP, которую клиент использует, а затем строка запроса заканчивается последовательностью CRLF. (CRLF расшифровывается как *carriage return and line feed*, то есть «возврат каретки и подача строки», эти термины остались со времен пишущих машинок!) Последовательность CRLF также может быть записана как `\r\n`, где `\r` — это возврат каретки, а `\n` — подача строки. Последовательность CRLF отделяет строку запроса от остальных данных запроса. Обратите внимание, во время печати CRLF мы видим начало новой строки, а не `\r\n`.

Глядя на данные строки запроса, полученные из программы в ее настоящем состоянии, мы видим, что запрос имеет метод `GET`, URI запроса `/` и версию `HTTP/1.1`.

После строки запроса остальные строки, начиная с `Host:` и далее, являются заголовками. Запросы `GET` не имеют тела.

Попробуйте сделать запрос из другого браузера или запросить другой адрес, например 127.0.0.1:7878/test, чтобы увидеть, как изменяются данные запроса.

Теперь, зная, что именно запрашивает браузер, давайте отправим назад некоторые данные!

Написание ответа

Теперь мы реализуем отправку данных в ответ на запрос клиента. Ответы имеют следующий формат:

```
HTTP-Version Status-Code Reason-Phrase CRLF
headers CRLF
message-body
```

Первая строка отклика — это строка состояния, содержащая версию HTTP, используемую в ответе, числовой код состояния, который резюмирует результат запроса, и причину с текстовым описанием кода состояния. После последовательности CRLF идут любые заголовки, еще одна последовательность CRLF и тело ответа.

Вот пример ответа, который использует HTTP версии 1.1, имеет код состояния `200`, причину `OK`, без заголовков и без тела:

```
HTTP/1.1 200 OK\r\n\r\n
```

Код состояния 200 — это стандартный ответ об успешном выполнении. Текст представляет собой крошечный успешный HTTP-ответ. Давайте запишем это

в поток как наш ответ на успешный запрос! Из функции `handle_connection` удалите инструкцию `println!`, которая печатала данные запроса, и замените ее кодом из листинга 20.3.

Листинг 20.3. Запись крошечного успешного HTTP-ответа в поток

src/main.rs

```
fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];

    stream.read(&mut buffer).unwrap();

  ❶ let response = "HTTP/1.1 200 OK\r\n\r\n";

  ❷ stream.write(response.as_bytes()❸).unwrap();
  ❹ stream.flush().unwrap();
}
```

Первая новая строка определяет переменную `response`, которая содержит данные сообщения об успехе ❶. Затем мы вызываем метод `as_bytes` для переменной `response`, чтобы конвертировать строковые данные в байты ❸. Метод `write` для `stream` берет `&[u8]` и посылает эти байты непосредственно по соединению ❷.

Поскольку операция `write` может не сработать, мы используем метод `unwrap` для любого результата с ошибкой, как и раньше. Опять же, в реальном приложении вы бы добавили сюда обработку ошибок. Наконец, метод `flush` будет ждать и препятствовать продолжению программы до тех пор, пока все байты не будут записаны в соединение ❹. Тип `TcpStream` содержит внутренний буфер для минимизации вызовов опорной операционной системы.

Внеся эти изменения, давайте выполним код и сделаем запрос. Мы больше не печатаем данные в терминал, поэтому в нем будут только данные из Cargo. Когда вы загрузите 127.0.0.1:7878 в веб-браузере, вы должны получить пустую страницу вместо ошибки. Вы только что вручную закодировали HTTP-запрос и ответ!

Возвращение реального HTML

Давайте реализуем функциональность для возвращения не только пустой страницы. В корневом каталоге проекта, но не в каталоге src, создайте новый файл hello.html. Вы можете ввести любой HTML, который хотите. В листинге 20.4 показан один из вариантов.

Листинг 20.4. Пример HTML-файла для возвращения в ответе

hello.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
```

```
    <title>Привет!</title>
  </head>
  <body>
    <h1>Привет!</h1>
    <p>Привет от Rust</p>
  </body>
</html>
```

Это минимальный документ HTML5 с заголовком и некоторым текстом. Для того чтобы вернуть его с сервера при получении запроса, мы изменим функцию `handle_connection`, как показано в листинге 20.5, так, чтобы прочитать HTML-файл, добавить его в ответ в качестве тела и отправить.

Листинг 20.5. Отправка содержимого hello.html в качестве тела ответа

src/main.rs

```
❶ use std::fs;
  // --пропуск--

  fn handle_connection(mut stream: TcpStream) {
      let mut buffer = [0; 512];
      stream.read(&mut buffer).unwrap();

      let contents = fs::read_to_string("hello.html").unwrap();

   ❷ let response = format!("HTTP/1.1 200 OK\r\n\r\n{}", contents);

      stream.write(response.as_bytes()).unwrap();
      stream.flush().unwrap();
  }
```

Мы добавили строку вверху, чтобы ввести модуль файловой системы стандартной библиотеки в область видимости ❶. Код для чтения содержимого файла в строку должен выглядеть знакомо: мы использовали его в главе 12, когда читали содержимое файла для проекта ввода-вывода в листинге 12.4.

Далее мы используем макрокоманду `format!` для добавления содержимого файла в тело ответа об успешном выполнении ❷.

Выполните этот код с помощью команды `cargo run` и загрузите 127.0.0.1:7878 в свой браузер. Вы должны увидеть, что HTML отображен на экране!

В настоящее время мы игнорируем данные запроса в переменной `buffer` и просто отправляем обратно содержимое HTML-файла в безусловном порядке. Это означает, что если вы попытаетесь запросить в браузере *127.0.0.1:7878/что-то-еще*, то все равно получите тот же HTML-ответ. Наш сервер очень ограничен и не делает того, что делает большинство веб-серверов. Мы хотим настраивать ответы индивидуально, в зависимости от запроса, и отправлять обратно HTML-файл только для хорошо сформированного запроса ресурса /.

Проверка запроса и выборочный ответ

Прямо сейчас веб-сервер будет возвращать HTML-код файла независимо от того, что запросил клиент. Давайте добавим функциональность, которая проверяет, что браузер запрашивает ресурс / перед возвратом HTML-файла, и возвращает ошибку, если браузер запрашивает что-то еще. Для этого нужно изменить функцию `handle_connection`, как показано в листинге 20.6. Этот новый код проверяет содержимое полученного запроса на соответствие тому, что мы знаем о том, как выглядит запрос ресурса /, и добавляет блоки `if` и `else`, которые трактуют запросы по-разному.

Листинг 20.6. Сопоставление запроса и обработка запросов ресурса / иначе, чем другие запросы

src/main.rs

```
// --пропуск--

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];
    stream.read(&mut buffer).unwrap();

  ❶ let get = b"GET / HTTP/1.1\r\n";

  ❷ if buffer.starts_with(get) {
        let contents = fs::read_to_string("hello.html").unwrap();

        let response = format!("HTTP/1.1 200 OK\r\n\r\n{}", contents);

        stream.write(response.as_bytes()).unwrap();
        stream.flush().unwrap();
  ❸ } else {
        // какой-то другой запрос
    }
}
```

Во-первых, мы жестко закодируем данные, соответствующие запросу ресурса /, в переменную `get` ❶. Поскольку мы читаем сырые байты в буфер, мы трансформируем `get` в байтовую строку, добавляя синтаксис байтовой строки `b""` в начало данных содержимого. Затем мы проверяем, начинается ли значение переменной `buffer` с байтов в `get` ❷. Если это так, значит, мы получили хорошо сформированный запрос ресурса / — это случай успеха, который мы будем обрабатывать в блоке `if`, возвращающем содержимое HTML-файла.

Если буфер не начинается с байтов в `get`, значит, мы получили какой-то другой запрос. Вскоре мы добавим код в блок `else` ❸, который будет отвечать на все остальные запросы.

А сейчас выполните этот код и запросите 127.0.0.1:7878. Вы должны получить HTML в hello.html. Если вы сделаете какой-то другой запрос, например *127.0.0.1:7878/что-то-еще*, то получите ошибку соединения, подобную тем, которые вы видели при запуске кода в листингах 20.1 и 20.2.

Теперь давайте добавим код из листинга 20.7 в блок `else`, который будет возвращать ответ с кодом состояния 404, сигнализирующий о том, что содержимое запроса не найдено. Мы также вернем некоторый HTML, который будет отображаться на странице в браузере, сообщая ответ конечному пользователю.

Листинг 20.7. Ответ с кодом состояния 404 и страницей ошибок, если запрашивается любой ресурс, кроме /

src/main.rs

```
// --пропуск--

} else {
 ❶ let status_line = "HTTP/1.1 404 NOT FOUND\r\n\r\n";
 ❷ let contents = fs::read_to_string("404.html").unwrap();

    let response = format!("{}{}", status_line, contents);

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

Здесь ответ имеет строку состояния с кодом 404 и причиной НЕ НАЙДЕНО ❶. Мы по-прежнему не возвращаем заголовки, а телом отклика будет HTML в файле 404.html ❷. Вам нужно создать файл 404.html для страницы ошибок рядом с hello.html. Опять же, вы можете применять любой HTML, который хотите, либо используйте пример HTML в листинге 20.8.

Листинг 20.8. Пример содержимого страницы для отправки обратно с любым ответом 404

404.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Привет!</title>
  </head>
  <body>
    <h1>Ой!</h1>
    <p>Извините, я не знаю, что вы запрашиваете.</p>
  </body>
</html>
```

С этими изменениями запустите сервер снова. Запрос 127.0.0.1:7878 должен возвращать содержимое hello.html, а любой запрос, к примеру 127.0.0.1:7878/foo, должен возвращать HTML с сообщением об ошибке из 404.html.

Небольшой рефакторинг

На данный момент в блоках `if` и `else` много повторов: они одновременно читают файлы и пишут содержимое файлов в поток. Единственными различиями

являются строка состояния и имя файла. Давайте сделаем код более сжатым, выделив эти различия в отдельные блоки `if` и `else`, которые будут передавать значения строки состояния и имени файла переменным. Тогда мы сможем использовать эти переменные в безусловном порядке в коде для чтения файла и записи ответа. В листинге 20.9 показан итоговый код после замены крупных блоков `if` и `else`.

Листинг 20.9. Рефакторинг блоков if и else, в результате которого они содержат только тот код, который отличается в обоих случаях

src/main.rs

```
// --snip--

fn handle_connection(mut stream: TcpStream) {
    // --snip--

    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else {
        ("HTTP/1.1 404 НЕ НАЙДЕНО\r\n\r\n", "404.html")
    };

    let contents = fs::read_to_string(filename).unwrap();

    let response = format!("{}{}", status_line, contents);

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

Теперь блоки `if` и `else` возвращают только подходящие значения для строки состояния и имени файла в кортеже. Затем мы используем деструктурирование, передавая эти два значения переменным `status_line` и `filename`, применяя паттерн в инструкции `let`, как описано в главе 18.

Ранее повторенный код теперь находится вне блоков `if` и `else` и использует переменные `status_line` и `filename`. Так легче увидеть разницу между двумя случаями, и это означает, что есть только одно место для обновления кода, если мы хотим изменить способ чтения файла и записи ответа. Поведение кода в листинге 20.9 будет таким же, как и в листинге 20.8.

Потрясающе! Теперь у нас есть простой веб-сервер примерно в 40 строках кода Rust, который отвечает на один запрос страницей контента, а на все остальные — ответами 404.

Сейчас сервер работает в одном потоке, а значит, может обслуживать только один запрос за раз. Давайте узнаем, когда это может стать проблемой, разобрав несколько медленных запросов. Затем мы это исправим, в результате чего сервер сможет обрабатывать несколько запросов сразу.

Превращение однопоточного сервера в многопоточный

Прямо сейчас сервер обрабатывает каждый запрос по очереди, то есть он не обрабатывает второе соединение до тех пор, пока не завершится обработка первого. Если бы сервер получал все больше и больше запросов, то такое последовательное выполнение было бы все менее и менее оптимальным. Если сервер получит запрос, обработка которого занимает много времени, то последующие запросы должны будут ждать завершения продолжительного запроса, даже если новые могут обрабатываться быстро. Нам потребуется это исправить, но сначала рассмотрим указанную проблему на практике.

Моделирование медленного запроса в текущей реализации сервера

Мы рассмотрим, каким образом запрос с медленной обработкой влияет на другие запросы, выполняемые в текущей реализации сервера. В листинге 20.10 реализована обработка запроса ресурса /sleep со смоделированным медленным ответом, из-за которого сервер перед ответом будет засыпать на 5 секунд.

Листинг 20.10. Моделирование медленного запроса путем распознавания запроса /sleep и засыпания на 5 секунд

src/main.rs

```
use std::thread;
use std::time::Duration;
// --пропуск--

fn handle_connection(mut stream: TcpStream) {
    // --пропуск--

    let get = b"GET / HTTP/1.1\r\n";
 ❶ let sleep = b"GET /sleep HTTP/1.1\r\n";

    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
 ❷ } else if buffer.starts_with(sleep) {
     ❸ thread::sleep(Duration::from_secs(5));
     ❹ ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else {
        ("HTTP/1.1 404 НЕ НАЙДЕНО\r\n\r\n", "404.html")
    };

    // --пропуск--
}
```

Этот код немного запутан, но как модель он достаточно хорош. Мы создали второй запрос `sleep` ❶, данные которого сервер распознает. Мы добавили `else if`

после блока `if`, чтобы проверить запрос ресурса /sleep ❷. Когда этот запрос будет получен, сервер будет засыпать на 5 секунд ❸ перед отображением успешной HTML-страницы ❹.

Вы видите, насколько примитивен наш сервер: реальные библиотеки распознавали бы многочисленные запросы гораздо лаконичнее!

Запустите сервер с помощью команды `cargo run`. Затем откройте два окна браузера: одно для http://127.0.0.1:7878, а другое — для http://127.0.0.1:7878/sleep. Если вы введете URI-ресурс / несколько раз, как и раньше, то увидите, что он быстро отвечает. Но если вы введете /sleep, а затем загрузите /, то увидите, что ресурс / ждет до тех пор, пока `sleep` не проспит полные 5 секунд перед загрузкой.

Есть много способов изменить характер работы веб-сервера, чтобы не скапливалось большое количество запросов из-за медленного запроса. Способ, который мы применим, называется пулом потоков исполнения.

Повышение пропускной способности с помощью пула потоков исполнения

Пул потоков исполнения — это группа порожденных потоков исполнения, которые ожидают задачу и готовы ее выполнить. Когда программа получает новую задачу, она назначает ее одним из потоков исполнения в пуле — этот поток будет обрабатывать задачу. Остальные потоки в пуле могут обрабатывать другие задачи, поступающие, пока первый поток занимается обработкой. Когда первый поток завершает обработку своей задачи, он возвращается в пул незанятых потоков исполнения и готов к обработке новой задачи. Пул потоков исполнения позволяет обрабатывать соединения конкурентно, увеличивая пропускную способность сервера.

У нас в пуле будет несколько потоков исполнения для защиты от DoS-атак. Если бы программа создавала новый поток для каждого запроса по мере его поступления, то кто-то, сделав 10 миллионов запросов к серверу, создал бы хаос, израсходовав все ресурсы сервера и застопорив обработку запросов.

Мы не будем порождать неограниченное число потоков, а создадим фиксированное число ожидающих своей очереди в пуле. По мере поступления запросов они будут отправляться в пул для обработки. Пул будет поддерживать очередь входящих запросов. Каждый поток в пуле будет выталкивать запрос из этой очереди, обрабатывать его, а затем приступать к следующему запросу. С помощью этой конструкции мы можем конкурентно обрабатывать N запросов, где N — это число потоков исполнения. Если каждый поток отвечает на продолжительный запрос, то последующие запросы по-прежнему будут ожидать в очереди, но мы увеличили число продолжительных запросов, которые можно обрабатывать до момента достижения этой точки.

Это техническое решение — лишь одно из многих, которые повышают пропускную способность веб-сервера. Другие варианты — это модель ветвления/объеди-

нения (fork/join) и однопоточная модель асинхронного ввода-вывода. Если вас интересует эта тема, вы можете почитать о других решениях и реализовать их на языке Rust. Поскольку Rust является низкоуровневым языком, все эти варианты возможны.

Прежде чем приступить к реализации пула потоков исполнения, давайте поговорим о том, как должно выглядеть использование пула. При разработке кода, если сначала написать клиентский интерфейс, это поможет развить проект. Пишите API этого кода так, чтобы его структура соответствовала тому, как вы хотите его вызывать. Затем реализуйте свойства в рамках этой структуры вместо того, чтобы реализовывать свойства, а затем планировать публичный API.

Аналогично тому, как мы использовали разработку на основе тестов в проекте главы 12, здесь мы применим разработку на основе компилятора. Мы напишем код, который вызывает нужные функции, а затем посмотрим на ошибки компилятора и выясним, что требуется изменить, чтобы привести код в рабочее состояние.

Структура кода при возможности порождения потока исполнения для каждого запроса

Сначала давайте узнаем, как бы выглядел код, если бы он создавал новый поток для каждого соединения. Как упоминалось ранее, этот план не окончательный из-за проблем с потенциальным порождением неограниченного числа потоков исполнения, но это отправная точка. В листинге 20.11 показаны изменения, вносимые в функцию `main`, чтобы создавать новый поток для обработки каждого потока внутри цикла `for`.

Листинг 20.11. Создание нового потока исполнения для каждого потока

src/main.rs

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        thread::spawn(|| {
            handle_connection(stream);
        });
    }
}
```

Как вы узнали из главы 16, функция `thread::spawn` будет создавать новый поток, а затем выполнять код внутри замыкания в новом потоке. Если вы выполните этот код и загрузите ресурс /sleep в браузер, а затем ресурс / в двух других вкладках браузера, то увидите, что запросам ресурса / не придется ждать завершения запроса /sleep. Но, как мы уже упоминали, это в итоге приводит к переполнению системы, потому что вы будете создавать новые потоки без каких-либо ограничений.

Создание похожего интерфейса для конечного числа потоков исполнения

Мы хотим, чтобы пул потоков исполнения работал похожим, привычным образом, когда переключение с потоков исполнения на пул не требует крупных изменений в коде, использующем API. В листинге 20.12 показан гипотетический интерфейс для структуры `ThreadPool`, которую мы хотим использовать вместо функции `thread::spawn`.

Листинг 20.12. Идеальный интерфейс структуры ThreadPool

src/main.rs

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
 ❶ let pool = ThreadPool::new(4);

    for stream in listener.incoming() {
        let stream = stream.unwrap();

     ❷ pool.execute(|| {
            handle_connection(stream);
        });
    }
}
```

Мы используем функцию `ThreadPool::new` для создания нового пула потоков исполнения с конфигурируемым числом потоков, в данном случае четырьмя ❶. Затем, в цикле `for`, метод `pool.execute` имеет интерфейс, похожий на `othread::spawn` тем, что он берет замыкание, которое пул должен выполнять для каждого потока ❷. Мы должны реализовать метод `pool.execute` так, чтобы он брал замыкание и передавал его потоку из пула для выполнения. Этот код пока не компилируется, но мы постараемся, чтобы компилятор направил нас в нужную сторону.

Построение структуры ThreadPool с использованием разработки на основе компилятора

Внесите изменения в листинг 20.12 для src/main.rs, а затем давайте использовать ошибки компилятора из `cargo check`, которые будут направлять разработку. Вот первая полученная ошибка[1]:

```
$ cargo check
   Compiling hello v0.1.0 (file:///projects/hello)
error[E0433]: failed to resolve. Use of undeclared type or module `ThreadPool`
  --> src\main.rs:10:16
   |
10 |     let pool = ThreadPool::new(4);
   |                ^^^^^^^^^^^^^^^ Use of undeclared type or module
`ThreadPool`
error: aborting due to previous error
```

[1] ошибка[E0433]: не получилось урегулировать. Использование необъявленного типа или модуля `ThreadPool`

Отлично! Эта ошибка говорит о том, что нужен тип или модуль `ThreadPool`, поэтому создадим его прямо сейчас. Реализация типа `ThreadPool` не будет зависеть от вида работы, которую выполняет веб-сервер. Поэтому давайте переключим упаковку `hello` с двоичной на библиотечную, где будет храниться реализация типа `ThreadPool`. После перехода к библиотечной упаковке мы также можем использовать отдельную библиотеку пула потоков исполнения для любой работы, которую мы хотим выполнять с помощью указанного пула, а не только для обслуживания веб-запросов.

Создайте файл src/lib.rs, который содержит самое простое определение структуры `ThreadPool`, возможное на данный момент, а именно:

src/lib.rs

```
pub struct ThreadPool;
```

Затем создайте новый каталог, src/bin, и переместите двоичную упаковку из src/main.rs в src/bin/main.rs. В результате библиотечная упаковка станет первичной в каталоге hello. Мы по-прежнему можем выполнять двоичный код в src/bin/main.rs с помощью команды `cargo run`. После перемещения файла main.rs отредактируйте его так, чтобы внести библиотечную упаковку и ввести структуру `ThreadPool` в область видимости, добавив следующий код наверх src/bin/main.rs:

src/bin/main.rs

```
use hello::ThreadPool;
```

Этот код по-прежнему не будет работать, но давайте проверим его еще раз, чтобы получить следующую ошибку, которую нужно уладить[1]:

```
$ cargo check
    Compiling hello v0.1.0 (file:///projects/hello)
error[E0599]: no function or associated item named `new` found for type
`hello::ThreadPool` in the current scope
 --> src/bin/main.rs:13:16
   |
13 |      let pool = ThreadPool::new(4);
   |                 ^^^^^^^^^^^^^^^ function or associated item not found in
`hello::ThreadPool`
```

Эта ошибка указывает на то, что далее нужно создать связанную функцию с именем `new` для структуры `ThreadPool`. Мы также знаем, что функция `new` должна иметь один параметр, который может принимать 4 в качестве аргумента, и должна возвращать экземпляр структуры `ThreadPool`. Давайте реализуем простейшую функцию `new` с этими характеристиками:

[1] ошибка[E0599]: ни функция, ни связанный элемент с именем `new` не найдены для типа `hello::ThreadPool` в текущей области видимости

src/lib.rs

```
pub struct ThreadPool;

impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        ThreadPool
    }
}
```

Мы выбрали тип `usize` в качестве типа параметра `size`, поскольку знаем, что отрицательное число потоков исполнения не имеет смысла. Мы также знаем, что будем использовать 4 как число элементов в коллекции потоков исполнения, для чего и предназначен тип `usize`, как описано в разделе «Целочисленные типы» (с. 67).

Давайте проверим код еще раз[1]:

```
$ cargo check
    Compiling hello v0.1.0 (file:///projects/hello)
warning: unused variable: `size`
 --> src/lib.rs:4:16
  |
4 |     pub fn new(size: usize) -> ThreadPool {
  |                ^^^^
  |
  = note: #[warn(unused_variables)] on by default
  = note: to avoid this warning, consider using `_size` instead

error[E0599]: no method named `execute` found for type `hello::ThreadPool` in
the current scope
  --> src/bin/main.rs:18:14
   |
18 |         pool.execute(|| {
   |              ^^^^^^^
```

Теперь мы получаем предупреждение и ошибку. На мгновение проигнорируем предупреждение. Ошибка возникает, потому что у нас нет метода `execute` в структуре `ThreadPool`. Вспомните раздел «Создание похожего интерфейса для конечного числа потоков исполнения» (с. 543), там мы приняли решение, что пул потоков исполнения должен иметь интерфейс, подобный функции `thread::spawn`. В дополнение к этому мы реализуем функцию `execute`, которая принимает заданное замыкание и передает его для выполнения простаивающему потоку в пуле.

Мы определим метод `execute` для структуры `ThreadPool`, который будет принимать замыкание в качестве параметра. Вспомним раздел «Хранение замыканий с использованием обобщенных параметров и типажей `Fn`» (с. 319): мы можем принимать замыкания как параметры с тремя разными типажами — `Fn`, `FnMut` и `FnOnce`. Нужно решить, какой вид замыкания использовать здесь. Мы знаем, что в итоге

[1] ошибка[E0599]: не найден метод с именем `execute` для типа ‘ hello::ThreadPool` в текущей области видимости

сделаем что-то похожее на реализацию функции `thread::spawn` из стандартной библиотеки, поэтому можно посмотреть, какие границы у сигнатуры `thread::spawn` в ее параметре. Документация показывает следующее:

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
    where
        F: FnOnce() -> T + Send + 'static,
        T: Send + 'static
```

Здесь нас интересует именно параметр типа `F`. Параметр типа `T` связан с возвращаемым значением, он нам не интересен. Мы видим, что `spawn` использует `FnOnce` в качестве границы типажа в типе `F`. Вероятно, это тоже нужно, потому что мы в конечном итоге будем передавать аргумент, получаемый в `execute`, в `spawn`. Можно с уверенностью сказать, что `FnOnce` — нужный нам типаж, потому что поток будет выполнять замыкание запроса только один раз, что соответствует части `Once` в имени `FnOnce`.

Параметр типа `F` также имеет границу типажа `Send` и границу `'static` жизненного цикла, которые полезны в этой ситуации: граница `Send` нужна, чтобы передавать замыкание из одного потока в другой, а граница `'static` требуется потому, что мы не знаем, сколько времени займет исполнение потока. Давайте создадим метод `execute` для структуры `ThreadPool`, который будет принимать параметр обобщенного типа `F` с этими границами:

src/lib.rs

```
impl ThreadPool {
    // --пропуск--

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce()❶ + Send + 'static
    {

    }
}
```

Мы по-прежнему используем `()` после `FnOnce` ❶, потому что типаж `FnOnce` представляет собой замыкание, которое не берет параметров и не возвращает значение. Как и определения функций, возвращаемый тип может быть опущен из сигнатуры, но даже если у нас нет параметров, то все равно нужны круглые скобки.

Опять же, это самая простая реализация метода `execute`: она ничего не делает, мы лишь пытаемся скомпилировать код. Давайте проверим еще раз:

```
$ cargo check
   Compiling hello v0.1.0 (file:///projects/hello)
warning: unused variable: `size`
 --> src/lib.rs:4:16
  |
4 |     pub fn new(size: usize) -> ThreadPool {
```

```
  |                ^^^^
  |
  = note: #[warn(unused_variables)] on by default
  = note: to avoid this warning, consider using `_size` instead

warning: unused variable: `f`
 --> src/lib.rs:8:30
  |
8 |     pub fn execute<F>(&self, f: F)
  |                              ^
  |
  = note: to avoid this warning, consider using `_f` instead
```

Сейчас мы получаем только предупреждения, а это значит, он компилируется! Но обратите внимание: если вы попытаетесь выполнить команду `cargo run` и сделать запрос в браузере, то увидите ошибки, рассмотренные в начале главы. Библиотека на самом деле еще не вызывает замыкание, передаваемое в `execute`!

ПРИМЕЧАНИЕ

В случае языков со строгими компиляторами, таких как Haskell и Rust, можно услышать такое утверждение: «Если код компилируется, то он работает». Но это не истина в последней инстанции. Наш проект компилируется, но при этом абсолютно ничего не делает! Если бы мы создавали настоящий, законченный проект, то сейчас было бы самое время начать писать модульные тесты, чтобы проверить, что код компилируется *и* имеет желаемое поведение.

Проверка числа потоков исполнения в new

Мы продолжим получать предупреждения, потому что мы ничего не делаем с параметрами для функций `new` и `execute`. Давайте реализуем тела этих функций с желаемым поведением. Для начала подумаем о `new`. Ранее мы выбрали беззнаковый тип для параметра `size`, потому что пул с отрицательным числом потоков исполнения не имеет смысла. Однако пул с нулем потоков исполнения также не имеет смысла, хотя 0 является абсолютно допустимым значением типа `usize`. Мы добавим код для проверки, что величина `size` больше нуля, перед возвращением экземпляра структуры `ThreadPool`, и в случае если программа получит ноль, то она будет поднимать панику с помощью макрокоманды `assert!`, как показано в листинге 20.13.

Листинг 20.13. Реализация функции ThreadPool::new с паникой, в случае если size равен нулю

src/lib.rs

```
impl ThreadPool {
    /// Создать новый пул потоков исполнения.
    ///
    /// Размер — это число потоков исполнения в пуле.
    ///
```

```
❶ /// # Паники
   ///
   /// Функция `new` поднимет панику, если размер будет равен нулю.
   pub fn new(size: usize) -> ThreadPool {
    ❷ assert!(size > 0);

       ThreadPool
   }

   // --пропуск--
}
```

Мы добавили документацию для структуры `ThreadPool`, используя документационные комментарии. Обратите внимание, с точки зрения документирования мы поступили правильно, добавив секцию, озвучивающую ситуации, в которых функция может поднимать панику ❶, как описано в главе 14. Выполните команду `cargo doc --open` и щелкните на структуре `ThreadPool` — вы увидите, как выглядит сгенерированная документация для функции `new`!

Вместо добавления макрокоманды `assert!`, как мы сделали здесь ❷, можно было бы переделать функцию `new` так, чтобы она возвращала тип `Result`, как мы поступили с функцией `Config::new` в проекте ввода-вывода в листинге 12.9. Но в данном случае мы решили, что попытка создать пул потоков исполнения без потоков должна быть неустранимой ошибкой. Если вы амбициозны, то попробуйте написать версию функции `new` со следующей сигнатурой, чтобы сравнить обе версии:

```
pub fn new(size: usize) -> Result<ThreadPool, PoolCreationError> {
```

Создание пространства для хранения потоков исполнения

Теперь, имея способ узнать, что у нас есть допустимое число потоков исполнения для хранения в пуле, мы можем создать эти потоки и сохранить их в структуре `ThreadPool` перед ее возвращением. Но как мы будем «хранить» потоки? Давайте еще раз взглянем на сигнатуру функции `thread::spawn`:

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
    where
        F: FnOnce() -> T + Send + 'static,
        T: Send + 'static
```

Функция `spawn` возвращает `JoinHandle<T>`, где `T` — это тип, возвращаемый замыканием. Попробуем использовать `JoinHandle` и посмотрим, что получится. В нашем случае замыкания, которые мы передаем в пул потоков исполнения, будут обрабатывать соединение и ничего не возвращать, поэтому `T` будет пустым типом `()`.

Код в листинге 20.14 компилируется, но пока что не создает никаких потоков исполнения. Мы изменили определение структуры `ThreadPool` в части хранения вектора экземпляров `thread::JoinHandle<()>`, инициализировали вектор емкостью `size`, настроили цикл `for`, который будет выполнять некий код для создания потоков исполнения, и вернули содержащий их экземпляр структуры `ThreadPool`.

Листинг 20.14. Создание вектора для типа ThreadPool, который будет хранить потоки

src/lib.rs

```
❶ use std::thread;

  pub struct ThreadPool {
    ❷ threads: Vec<thread::JoinHandle<()>>,
  }

  impl ThreadPool {
      // --пропуск--
      pub fn new(size: usize) -> ThreadPool {
          assert!(size > 0);

        ❸ let mut threads = Vec::with_capacity(size);

          for _ in 0..size {
              // создать несколько потоков исполнения и сохранить их в векторе
          }

          ThreadPool {
              threads
          }
      }

      // --пропуск--
  }
```

Мы ввели `std::thread` в область видимости в библиотечной упаковке ❶, потому что используем `thread::JoinHandle` в качестве типа элементов вектора в структуре `ThreadPool` ❷.

После получения допустимого размера структура `ThreadPool` создает новый вектор, который может содержать `size` элементов ❸. В этой книге мы еще не использовали функцию `with_capacity`, которая выполняет ту же задачу, что и функция `Vec::new`, но с важным отличием: она предварительно выделяет пространство в векторе. Поскольку мы знаем, что нужно хранить `size` элементов в векторе, выполнение операции выделения заблаговременно несколько эффективнее, чем использование функции `Vec::new`, размер которой изменяется по мере вставки элементов.

Когда вы снова выполните команду `cargo check`, вы получите еще несколько предупреждений, но она должна быть успешной.

Структура Worker, ответственная за отправку кода из структуры ThreadPool в поток

Мы оставили комментарий в цикле `for` из листинга 20.14, касающийся создания потоков исполнения. Здесь мы рассмотрим, как на самом деле создаются потоки. Стандартная библиотека предоставляет функцию `thread::spawn` как способ соз-

дания потоков исполнения, а функция `thread::spawn` предусматривает получение некоторого программного кода, который поток должен исполнить после того, как она будет создана. Однако в нашем случае мы хотим создавать потоки — они будут ожидать программный код, который мы отправим позже. В реализации потоков исполнения стандартной библиотекой нет способов сделать это, мы должны реализовать такое поведение вручную.

Мы реализуем это поведение путем введения новой структуры данных между `ThreadPool` и потоками, которые будут управлять новым поведением. Мы обозначим эту структуру данных `Worker` («работник»), термином, часто встречающимся в реализациях операций с пулом. Представьте себе людей, работающих на кухне в ресторане: работники ждут заказов от клиентов, а затем отвечают за принятие этих заказов и их исполнение.

Вместо того чтобы хранить вектор экземпляров `JoinHandle<()>` в пуле потоков исполнения, мы будем хранить экземпляры структуры `Worker`. Каждый `Worker` будет хранить один экземпляр `JoinHandle<()>`. Затем мы реализуем метод для структуры `Worker`, который будет брать замыкание выполняемого кода и отправлять его в уже работающий поток для выполнения. Мы также дадим каждому работнику `id`, чтобы различать разных работников в пуле во время журналирования или отладки.

Давайте внесем следующие изменения в то, что происходит при создании пула потоков исполнения. Мы реализуем код, который отправляет замыкание потока после того, как `Worker` был настроен, следующим образом:

1. Определить структуру `Worker`, содержащую `id` и `JoinHandle<()>`.
2. Изменить `ThreadPool` так, чтобы он содержал вектор экземпляров структуры `Worker`.
3. Определить функцию `Worker::new`, которая берет `id` и возвращает экземпляр структуры `Worker`, содержащий `id` и поток, порожденный с пустым замыканием.
4. В функции `ThreadPool::new` использовать счетчик цикла `for` для генерирования `id`, создать новый экземпляр структуры `Worker` с этим `id` и сохранить его в векторе.

Если вы готовы принять вызов, то попробуйте реализовать эти изменения самостоятельно, прежде чем взглянуть на код в листинге 20.15.

Готовы? Вот листинг 20.15 с одним из способов внесения приведенных модификаций.

Листинг 20.15. Модифицирование структуры ThreadPool для хранения экземпляров структуры Worker вместо хранения потоков исполнения напрямую

src/lib.rs

```
use std::thread;

pub struct ThreadPool {
  ❶ workers: Vec<Worker>,
```

```
}

impl ThreadPool {
    // --пропуск--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut workers = Vec::with_capacity(size);

      ❷ for id in 0..size {
          ❸ workers.push(Worker::new(id));
        }

        ThreadPool {
            workers
        }
    }
    // --пропуск--
}

❹ struct Worker {
      id: usize,
      thread: thread::JoinHandle<()>,
}

impl Worker {
  ❺ fn new(id: usize) -> Worker {
      ❻ let thread = thread::spawn(|| {});

        Worker {
          ❼ id,
          ❽ thread,
        }
    }
}
```

Мы поменяли имя поля в типе `ThreadPool` с `threads` на `workers`, потому что теперь оно содержит экземпляры структуры `Worker` вместо экземпляров типа `JoinHandle<()>` ❶. Мы используем счетчик в цикле `for` ❷ в качестве аргумента для функции `Worker::new` и храним каждый новый `Worker` в векторе `workers` ❸.

Внешнему коду (как наш сервер в src/bin/main.rs) не нужно знать детали реализации, касающиеся структуры `Worker` в `ThreadPool`, поэтому мы делаем структуру `Worker` ❹ и ее функцию `new` приватными ❺. Функция `Worker::new` использует `id`, который мы ей даем ❼, и сохраняет экземпляр типа `JoinHandle<()>` ❽, созданный путем порождения нового потока исполнения с пустым замыканием ❻.

Этот код будет компилироваться и хранить число экземпляров структуры `Worker`, указанное в качестве аргумента для функции `ThreadPool::new`. Но мы пока что не обрабатываем замыкание, которое получаем в методе `execute`. Давайте посмотрим, как это сделать.

Отправка запросов нитям исполнения по каналам

Теперь мы займемся проблемой, касающейся того, что замыкания, передаваемые в функцию `thread::spawn`, абсолютно ничего не делают. В настоящее время мы получаем замыкание, которое хотим исполнить, в методе `execute`. Но нам нужно дать функции `thread::spawn` замыкание, подлежащее выполнению при возникновении каждого экземпляра структуры `Worker` во время создания `ThreadPool`.

Мы хотим, чтобы вновь создаваемые структуры `Worker` извлекали программный код, подлежащий выполнению, из очереди, хранящейся в пуле `ThreadPool`, и отправляли этот код в поток структуры `Worker` для выполнения.

В главе 16 вы узнали о каналах — простом способе связи между двумя потоками, который идеально подходит для данной ситуации. Мы будем использовать канал, функционирующий в качестве очереди заданий, а метод `execute` будет отправлять задание из `ThreadPool` в экземпляры `Worker`, которые будут отправлять это задание в свой поток. Вот план действий:

1. Пул `ThreadPool` будет создавать канал и стоять на отправляющей стороне канала.
2. Каждый работник `Worker` будет стоять на принимающей стороне канала.
3. Мы будем создавать новую структуру `Job`, содержащую замыкание для отправки по каналу.
4. Метод `execute` будет отправлять задание, которое он хочет исполнить, по передающей стороне канала.
5. В своем потоке `Worker` будет циклически перебирать принимающую сторону канала и исполнять замыкания любых заданий, которые он будет получать.

Давайте начнем с создания канала в функции `ThreadPool::new` и хранения отправляющей стороны в экземпляре типа `ThreadPool`, как показано в листинге 20.16. Структура `Job` пока ничего не содержит, но это тип элемента, который мы отправляем по каналу.

Листинг 20.16. Модифицирование структуры `ThreadPool` для хранения отправляющего конца канала, который отправляет экземпляры типа `Job`

src/lib.rs

```
// --пропуск--
use std::sync::mpsc;

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Job>,
}

struct Job;

impl ThreadPool {
```

```
    // --пропуск--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

     ❶ let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool {
            workers,
         ❷ sender,
        }
    }
    // --пропуск--
}
```

В функции `ThreadPool::new` мы создаем новый канал ❶, причем пул занимает отправляющий конец ❷. Это код будет успешно компилироваться, правда, попрежнему с предупреждениями.

Давайте попробуем передавать принимающий конец канала внутрь каждого работника тогда, когда пул потоков исполнения создает канал. Мы знаем, что хотим использовать принимающий конец в потоке, который работники порождают, поэтому мы будем ссылаться на параметр `receiver` в замыкании. Код в листинге 20.17 пока не компилируется полностью.

Листинг 20.17. Передача принимающего конца канала работникам

src/lib.rs

```
impl ThreadPool {
    // --пропуск--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
         ❶ workers.push(Worker::new(id, receiver));
        }

       ThreadPool {
           workers,
           sender,
        }
    }
    // --пропуск--
}
```

```
// --пропуск--

impl Worker {
    fn new(id: usize, receiver: mpsc::Receiver<Job>) -> Worker {
        let thread = thread::spawn(|| {
         ❷ receiver;
        });

        Worker {
            id,
            thread,
        }
    }
}
```

Мы внесли несколько небольших и простых изменений: передаем принимающий конец канала в функцию `Worker::new` ❶, а затем используем его внутри замыкания ❷.

Когда мы пытаемся проверить этот код, то получаем такую ошибку:

```
$ cargo check
   Compiling hello v0.1.0 (file:///projects/hello)
error[E0382]: use of moved value: `receiver`
  --> src/lib.rs:27:42
   |
27 |     workers.push(Worker::new(id, receiver));
   |                                  ^^^^^^^^ value moved here in
previous iteration of loop
   |
   = note: move occurs because `receiver` has type `std::sync::mpsc::Receiver<
Job>`, which does not implement the `Copy` trait
```

Код пытается передать `receiver` нескольким экземплярам структуры `Worker`. Как вы помните из главы 16, это работать не будет: реализация канала, которую предоставляет Rust, именуется как «несколько производителей, один потребитель». А значит, для исправления этого кода мы не можем просто клонировать потребляющий конец канала. Даже если бы мы могли, это нежелательное техническое решение. Вместо этого мы хотим распределять задания между потоками путем совместного использования одного приемника `receiver` всеми работниками.

Вдобавок, удаление задания из очереди каналов предусматривает изменение приемника `receiver`, поэтому нитям исполнения нужен безопасный способ совместного использования и модифицирования приемника. В противном случае мы могли бы получить гоночные состояния (как описано в главе 16).

Вспомните про умные указатели, безопасные для потоков, о которых мы говорили в главе 16: чтобы делиться владением между несколькими потоками и позволить потокам исполнения изменять значение, нужен умный указатель `Arc<Mutex<T>>`. Тип `Arc` будет позволять нескольким работникам владеть приемником, а тип

`Mutex` обеспечит, чтобы только один работник получал задание от приемника за один раз. В листинге 20.18 показаны изменения, которые необходимо внести.

Листинг 20.18. Совместное использование принимающего конца канала работниками с использованием типов Arc и Mutex

src/lib.rs

```
use std::sync::Arc;
use std::sync::Mutex;

// --пропуск--

impl ThreadPool {
    // --пропуск--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

      ❶ let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)❷));
        }

        ThreadPool {
            workers,
            sender,
        }
    }

    // --пропуск--
}

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        // --пропуск--
    }
}
```

В функции `ThreadPool::new` мы помещаем принимающий конец канала в `Arc` и `Mutex` ❶. Для каждого нового работника мы клонируем `Arc`, увеличивая число ссылок, чтобы работники могли совместно владеть принимающим концом ❷.

С этими изменениями код компилируется! Мы почти у цели!

Реализация метода execute

Давайте, наконец, реализуем метод `execute` в типе `ThreadPool`. Мы также изменим тип `Job` со структуры на псевдоним типажного объекта, содержащего тип замыка-

ния, который получает метод `execute`. Как описано в разделе «Создание синонимов типов с помощью псевдонимов типов» (с. 506), псевдонимы типов позволяют делать длинные типы короче. Взгляните на листинг 20.19.

Листинг 20.19. Создание псевдонима типа Job для умного указателя Box, содержащего каждое замыкание, а затем отправка задания по каналу

src/lib.rs

```
// --пропуск--

type Job = Box<FnOnce() + Send + 'static>;

impl ThreadPool {
    // --пропуск--

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {
     ❶ let job = Box::new(f);

     ❷ self.sender.send(job).unwrap();
    }
}

// --пропуск--
```

После создания нового экземпляра типа `Job` с использованием замыкания, которое мы получаем в методе `execute` ❶, мы отправляем это задание по отправляющему концу канала ❷. Мы вызываем метод `unwrap` для `send` на тот случай, если отправить сообщение не получится. Это может произойти, например, если мы остановим выполнение всех потоков, то есть принимающая сторона перестала получать новые сообщения. На данный момент мы не можем остановить исполнение потоков: они продолжают свою работу до тех пор, пока существует пул. Мы используем метод `unwrap`, потому что знаем, что сбоя не произойдет, но компилятору это неизвестно.

Но мы еще не закончили! В работнике замыкание, передаваемое в функцию `thread::spawn`, по-прежнему ссылается только на принимающий конец канала. Вместо этого нужно, чтобы замыкание работало в бесконечном цикле, запрашивая у принимающего конца канала задание и выполняя его при получении. Давайте внесем изменение, показанное в листинге 20.20, в функцию `Worker::new`.

Листинг 20.20. Получение и исполнение заданий в потоке работника `Worker`

src/lib.rs

```
// --пропуск--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
```

```
            let thread = thread::spawn(move || {
                loop {
                    let job = receiver.lock()❶.unwrap()❷.recv()❸.unwrap()❹;

                    println!("Работник {} получил задание; исполняется.", id);

                    (*job)();
                }
            });

            Worker {
                id,
                thread,
            }
        }
    }
```

Здесь мы сначала вызываем метод `lock` для `receiver`, чтобы получить мьютекс ❶, а затем — метод `unwrap`, чтобы поднимать панику при любых ошибках ❷. Получить замок не выйдет, если мьютекс находится в *отравленном* состоянии, что может произойти, если какой-то другой поток поднял панику, удерживая замок вместо того, чтобы его освободить. В этой ситуации правильно вызывать метод `unwrap`, чтобы этот поток мог поднимать панику. При необходимости поменяйте `unwrap` на метод `expect` с сообщением об ошибке, которое имеет для вас смысл.

Если мы получаем замок на мьютексе, то вызываем метод `recv`, чтобы получить задание `Job` из канала ❸. Заключительный вызов метода `unwrap` здесь проходит без ошибок ❹, которые тоже могут произойти, если поток, держащий передающую сторону канала, выключился, подобно тому, как метод `send` возвращает `Err`, если принимающая сторона выключилась.

Вызов метода `recv` блокируется, поэтому, если задания еще нет, то текущий поток будет ждать до тех пор, пока оно не появится. Умный указатель `Mutex<T>` обеспечивает, чтобы только один поток `Worker` запрашивал задание за один раз.

Теоретически этот код должен компилироваться. К сожалению, компилятор Rust еще не совершенен, и мы получаем такую ошибку[1]:

```
error[E0161]: cannot move a value of type std::ops::FnOnce() +
std::marker::Send: the size of std::ops::FnOnce() + std::marker::Send cannot
be statically determined
  --> src/lib.rs:63:17
   |
63 |                 (*job)();
   |                 ^^^^^^
```

[1] `ошибка[E0161]: не получается переместить значение типа std::ops::FnOnce() + std::marker::Send: размер std::ops::FnOnce() + std::marker::Send не получается определить статически`

Ошибка выглядит довольно загадочно, потому что и сама проблема довольно загадочная. Чтобы вызвать замыкание `FnOnce`, которое хранится в умном указателе `Box<T>` (являющемся псевдонимом типа `Job`), замыканию требуется переместиться из `Box<T>`, потому что оно берет `self` во владение, когда мы его вызываем. В общем, Rust не позволяет перемещать значение из `Box<T>`, потому что он не знает, насколько большим будет значение внутри `Box<T>`. Вспомните, что в главе 15 мы использовали умный указатель `Box<T>` именно потому, что у нас было нечто неизвестного размера, которое мы хотели сохранить в `Box<T>`, чтобы получить значение известного размера.

Как вы видели в листинге 17.15, мы можем писать методы, использующие синтаксис `self: Box<Self>`, который позволяет методу брать во владение значение `Self`, хранящееся в `Box<T>`. Здесь мы хотим сделать именно так, но, к сожалению, компилятор не позволит: та часть Rust, которая реализует поведение при вызове замыкания, не выполняется с `self: Box<Self>`. И поэтому Rust еще не понимает, что может использовать `self: Box<Self>` в этой ситуации, чтобы брать замыкание во владение и перемещать его из `Box<T>`.

Язык Rust по-прежнему находится в стадии разработки мест, где компилятор может быть усовершенствован, но в будущем код из листинга 20.20 должен работать без проблем. Мы будем рады, если вы присоединитесь к нам после прочтения этой книги и внесете свой вклад в развитие языка.

Но пока давайте обойдем эту проблему, используя хитрый трюк. Мы скажем компилятору, что в этом случае можно взять значение внутри `Box<T>` во владение, используя `self: Box<Self>`, и тогда, после того как мы овладеем замыканием, можно его вызвать. Это предусматривает определение нового типажа `FnBox` с методом `call_box`, который будет использовать `self: Box<Self>` в своей сигнатуре, определение `FnBox` для любого типа, реализующего `FnOnce()`, изменение псевдонима типа, чтобы он использовал новый типаж, и изменение структуры `Worker`, чтобы она использовала метод `call_box`. Эти изменения показаны в листинге 20.21.

Листинг 20.21. Добавление нового типажа FnBox для обхода текущих ограничений умного указателя Box<FnOnce()>

src/lib.rs

```
❶ trait FnBox {
    ❷ fn call_box(self: Box<Self>);
  }

❸ impl<F: FnOnce()> FnBox for F {
      fn call_box(self: Box<F>) {
        ❹ (*self)()
      }
  }

❺ type Job = Box<FnBox + Send + 'static>;

  // --пропуск--

  impl Worker {
```

```
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || {
            loop {
                let job = receiver.lock().unwrap().recv().unwrap();

                println!("Работник {} получил задание; исполняется.", id);

                ❻ job.call_box();
            }
        });

        Worker {
            id,
            thread,
        }
    }
}
```

Сначала мы создаем новый типаж под названием `FnBox` ❶. У него есть один метод `call_box` ❷, похожий на методы `call` для других типажей `Fn*`, за исключением того, что он берет `self: Box<Self>`, чтобы взять `self` во владение и переместить значение из `Box<T>`.

Далее мы реализуем типаж `FnBox` для любого типа `F`, который реализует типаж `FnOnce()` ❸. Это практически означает, что метод `call_box` может использоваться любым замыканием `FnOnce()`. Реализация метода `call_box` использует `(*self)()` для перемещения замыкания из `Box<T>` и вызова замыкания ❹.

Теперь нужно, чтобы псевдоним типа `Job` был типом `Box`, содержащим все, что реализует новый типаж `FnBox` ❺. Это позволит использовать метод `call_box` в `Worker`, когда мы получим значение `Job`, не вызывая замыкание непосредственно ❻. Реализация типажа `FnBox` для любого замыкания `FnOnce()` означает, что не нужно ничего менять в фактических значениях, которые мы посылаем по каналу. Теперь компилятору понятно, что наши действия правильные.

Этот трюк очень хитрый и сложный. Не волнуйтесь, если он выглядит совершенно нелогично, когда-нибудь он станет совершенно ненужным.

При реализации этого трюка пул потоков исполнения находится в рабочем состоянии! Запустите команду `cargo run` и сделайте несколько запросов:

```
$ cargo run
    Compiling hello v0.1.0 (file:///projects/hello)
warning: field is never used: `workers`
 --> src/lib.rs:7:5
  |
7 |     workers: Vec<Worker>,
  |     ^^^^^^^^^^^^^^^^^^^^
  |
  = note: #[warn(dead_code)] on by default

warning: field is never used: `id`
```

```
  --> src/lib.rs:61:5
   |
61 |     id: usize,
   |     ^^^^^^^^^
   |
   = note: #[warn(dead_code)] on by default

warning: field is never used: `thread`
  --> src/lib.rs:62:5
   |
62 |     thread: thread::JoinHandle<()>,
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: #[warn(dead_code)] on by default

    Finished dev [unoptimized + debuginfo] target(s) in 0.99 secs
     Running `target/debug/hello`
Работник 0 получил задание; исполняется.
Работник 2 получил задание; исполняется.
Работник 1 получил задание; исполняется.
Работник 3 получил задание; исполняется.
Работник 0 получил задание; исполняется.
Работник 2 получил задание; исполняется.
Работник 1 получил задание; исполняется.
Работник 3 получил задание; исполняется.
Работник 0 получил задание; исполняется.
Работник 2 получил задание; исполняется.
```

Получилось! Теперь у нас есть пул потоков, который выполняет соединения асинхронно. Никогда не создается более четырех потоков, поэтому система не будет перегружена, если сервер получит много запросов. Если мы сделаем запрос ресурса /sleep, то сервер будет способен обслуживать другие запросы, имея еще один поток, который будет их выполнять.

ПРИМЕЧАНИЕ

Если вы откроете ресурс /sleep в нескольких окнах браузера одновременно, то они могут загружаться по одному с интервалом в 5 секунд. Некоторые веб-браузеры исполняют несколько экземпляров одного и того же запроса последовательно из-за кэширования. Это ограничение не обусловлено нашим веб-сервером.

Узнав о цикле `while let` в главе 18, вы, возможно, задаетесь вопросом, почему мы не написали код потока `Worker`, как показано в листинге 20.22.

Листинг 20.22. Альтернативная реализация функции `Worker::new` с использованием цикла `while let`

src/lib.rs

```
// --пропуск--

impl Worker {
```

```
        fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
            let thread = thread::spawn(move || {
                while let Ok(job) = receiver.lock().unwrap().recv() {
                    println!("Работник {} получил задание; исполняется.", id);

                    job.call_box();
                }
            });

            Worker {
                id,
                thread,
            }
        }
    }
```

Этот код компилируется и выполняется, но не приводит к желаемому поведению потоков исполнения: из-за медленного запроса другие запросы все равно будут ждать своей обработки. Причина этого неочевидна: структура `Mutex` не имеет публичного метода `unlock`, поскольку владение замком основано на жизненном цикле `MutexGuard<T>` в `LockResult<MutexGuard<T>>`, который возвращается методом `lock`. Во время компиляции контролер заимствования может обеспечить соблюдение правила о том, что ресурс, охраняемый мьютексом, не может быть доступен, если мы не удерживаем замок. Но эта реализация также может привести к тому, что замок будет удерживаться дольше, чем предполагалось, если мы плохо продумаем жизненный цикл `MutexGuard<T>`. Поскольку значения в выражении `while` остаются в области видимости на протяжении всего блока, замок удерживается на протяжении всего вызова метода `job.call_box()`, то есть другие работники не могут получать задания.

Используя вместо этого цикл `loop` и приобретая замок и задание внутри блока, а не вне его, `MutexGuard`, возвращаемый из метода `lock`, отбрасывается, как только инструкция `let job` заканчивается. Таким образом, замок удерживается во время вызова метода `recv`, но освобождается перед вызовом `job.call_box()`, что позволяет обслуживать несколько запросов одновременно.

Корректное отключение и очистка

Код в листинге 20.21 откликается на запросы асинхронно с помощью пула потоков исполнения, как мы и предполагали. Мы получаем несколько предупреждений о полях `workers`, `id` и `thread`, которые не используются непосредственно, в качестве напоминания о том, что мы ничего не подчищаем. Когда мы применяем менее элегантный способ Ctrl-C для остановки главного потока исполнения, все остальные потоки также немедленно останавливаются, даже если находятся в процессе обслуживания запроса.

Теперь мы реализуем типаж `Drop`, который будет вызывать `join` для каждого потока в пуле, чтобы они могли завершить запросы, над которыми работают, перед

закрытием. Затем мы реализуем способ, который будет сообщать потокам, что они должны прекратить принимать новые запросы и выключиться. Для того чтобы увидеть этот код в действии, модифицируем сервер так, чтобы он принимал только два запроса перед тем, как корректно завершать работу своего пула потоков.

Реализация типажа Drop для ThreadPool

Давайте начнем с реализации типажа `Drop` в пуле потоков исполнения. Когда пул отбрасывается, все потоки должны быть собраны с тем, чтобы они завершили работу. В листинге 20.23 показана первая попытка реализации типажа `Drop`. Этот код пока не очень хорошо работает.

Листинг 20.23. Сбор всех потоков, когда пул потоков исполнения выходит из области видимости

src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
     ❶ for worker in &mut self.workers {
         ❷ println!("Отключение работника {}", worker.id);

         ❸ worker.thread.join().unwrap();
        }
    }
}
```

Сначала мы перебираем в цикле каждого работника пула потоков исполнения в `workers` ❶. Для этого используем `&mut`, потому что `self` является изменяемой ссылкой, и нам также нужно иметь возможность изменять `Worker`. Для каждого работника мы выводим сообщение о том, что данный работник отключается ❷, а затем вызываем `join` для потока этого работника ❸. Если вызов `join` оказывается неуспешным, то используем метод `unwrap`, который вынуждает компилятор поднять панику и перейти к некорректному отключению.

Вот ошибка, которую мы получаем при компиляции этого кода[1]:

```
error[E0507]: cannot move out of borrowed content
  --> src/lib.rs:65:13
   |
65 |             worker.thread.join().unwrap();
   |             ^^^^^^ cannot move out of borrowed content
```

Указанная ошибка говорит о том, что нельзя вызвать метод `join`, потому что у нас есть только неизменяемое заимствование каждого `Worker`, а `join` берет свой аргумент во владение. Для решения этой проблемы нужно переместить поток из экземпляра `Worker`, который владеет полем `thread`, чтобы метод `join` мог поглотить поток. Мы сделали это в листинге 17.15: если вместо этого `Worker` содержит

[1] `ошибка[E0507]: не получается переместить из заимствованного содержимого`

`Option<thread::JoinHandle<()>>`, то можно вызвать метод `take` для `Option`, чтобы переместить значение из варианта `Some` и не трогать вариант `None`. Другими словами, работающий `Worker` будет иметь вариант `Some` в поле `thread`, а когда мы захотим очистить `Worker`, мы поменяем `Some` на `None`, чтобы у `Worker` не было потока, подлежащего выполнению.

Итак, мы знаем, что хотим обновить определение `Worker` следующим образом:

src/lib.rs

```
struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>,
}
```

Теперь давайте доверимся компилятору — он найдет другие места, которые нужно изменить. Проверив этот код, мы получим две ошибки[1]:

```
error[E0599]: no method named `join` found for type `std::option::Option<std::
thread::JoinHandle<()>>` in the current scope
  --> src/lib.rs:65:27
   |
65 |             worker.thread.join().unwrap();
   |                           ^^^^

error[E0308]: mismatched types
  --> src/lib.rs:89:13
   |
89 |             thread,
   |             ^^^^^^
   |             |
   |             expected enum `std::option::Option`, found struct
`std::thread::JoinHandle`
   |             help: try using a variant of the expected type:
`Some(thread)`
   |
   = note: expected type `std::option::Option<std::thread::JoinHandle<()>>`
              found type `std::thread::JoinHandle<_>`
```

Давайте обратимся ко второй ошибке, которая указывает на код в конце функции `Worker::new`. Нам нужно завернуть значение `thread` в `Some` при создании нового экземпляра типа `Worker`. Внесите следующие изменения, чтобы устранить эту ошибку:

src/lib.rs

```
impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
```

[1] ошибка[E0599]: не найден метод с именем `join` для типа `std::option::Option<std::thread::JoinHandle<()>>` в текущей области видимости
ошибка[E0308]: несовпадающие типы

```
            // --пропуск--

            Worker {
                id,
                thread: Some(thread),
            }
        }
    }
```

Первая ошибка находится в реализации типажа `Drop`. Ранее мы упоминали, что намеревались вызывать `take` для значения `Option`, чтобы перемещать `thread` из `Worker`. Следующие изменения нужны как раз для этого:

src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            println!("Отключение работника {}", worker.id);

          ❶ if let Some(thread) = worker.thread.take() {
              ❷ thread.join().unwrap();
            }
        }
    }
}
```

Как обсуждалось в главе 17, метод `take` для `Option` вынимает вариант `Some` и не трогает `None`. Мы используем `if let` для деструктурирования `Some` и получения потока ❶; затем мы вызываем метод `join` для потока ❷. Если поток `Worker` равен `None`, то мы знаем, что поток `Worker` уже был очищен, поэтому в таком случае ничего не происходит.

Подача потокам сигнала об остановке прослушивания заданий

Благодаря внесенным изменениям код компилируется без предупреждений. Но есть и плохая новость: этот код пока что не функционирует так, как мы хотим. Ключом является алгоритм в замыканиях, выполняемых нитями экземпляров типа `Worker`: в данный момент мы вызываем метод `join`, но из-за этого потоки не отключатся, потому что находятся в бесконечном цикле в поисках заданий. Если мы попытаемся отбросить `ThreadPool` с текущей реализацией `drop`, то главный поток исполнения навсегда заблокируется, ожидая завершения первого потока.

Для устранения этой проблемы мы выполним модификацию потоков исполнения так, чтобы они прослушивали либо подлежащее выполнению задание `Job`, либо сигнал о том, что они должны прекратить прослушивание и выйти из бесконечного цикла. Вместо экземпляров типа `Job` канал будет отправлять один из приведенных ниже вариантов перечисления.

src/lib.rs

```
enum Message {
    NewJob(Job),
    Terminate,
}
```

Это перечисление `Message` будет либо вариантом `NewJob`, содержащим задание, которое поток должен выполнить, либо вариантом `Terminate`, побуждающим поток выйти из цикла и остановиться.

Нужно настроить канал для использования значений типа `Message` вместо `Job`, как показано в листинге 20.24.

Листинг 20.24. Отправка и получение значений типа Message и выход из цикла, если Worker получает сообщение Message::Terminate

src/lib.rs

```
pub struct ThreadPool {
    workers: Vec<Worker>,
  ❶ sender: mpsc::Sender<Message>,
}

// --пропуск--

impl ThreadPool {
    // --пропуск--

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {
        let job = Box::new(f);

      ❷ self.sender.send(Message::NewJob(job)).unwrap();
    }
}

// --пропуск--

impl Worker {
  ❸ fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Message>>>) -> Worker {

       let thread = thread::spawn(move ||{
           loop {
             ❹ let message = receiver.lock().unwrap().recv().unwrap();

               match message {
                 ❺ Message::NewJob(job) => {
                       println!("Работник {} получил задание; исполняется.", id);

                     ❻ job.call_box();
                   },
```

```
                ❼ Message::Terminate => {
                      println!("Работнику {} сказано завершить работу.", id);

                    ❽ break;
                    },
                }
            }
        });

        Worker {
            id,
            thread: Some(thread),
        }
    }
}
```

Для того чтобы встроить тип `Message`, нужно поменять `Job` на `Message` в двух местах: в определении типа `ThreadPool` ❶ и в сигнатуре функции `Worker::new` ❸. Метод `execute` структуры `ThreadPool` должен отправлять задания, завернутые в вариант `Message::NewJob` ❷. Тогда в функции `Worker::new`, там, где сообщение поступает из канала ❹, задание будет обрабатываться ❻, если получен вариант `NewJob` ❺, и поток будет выходить из цикла ❽, если получен вариант `Terminate` ❼.

Благодаря этим изменениям исходный код будет компилироваться и продолжать функционировать так же, как и после листинга 20.21. Но мы получим предупреждение, потому что мы не создаем никаких сообщений типа `Terminate`. Давайте устраним это предупреждение, изменив реализацию типажа `Drop` так, чтобы она выглядела как в листинге 20.25.

Листинг 20.25. Отправка сообщения Message::Terminate работникам перед вызовом метода join для каждого потока worker

src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        println!("Всем работникам отправляется сообщение о завершении.");

        for _ in &mut self.workers {
          ❶ self.sender.send(Message::Terminate).unwrap();
        }

        println!("Все работники выключаются.");

        for worker in &mut self.workers {
            println!("Выключается работник {}", worker.id);

            if let Some(thread) = worker.thread.take() {
              ❷ thread.join().unwrap();
            }
        }
    }
}
```

Теперь мы перебираем работников дважды: один раз для отправки одного сообщения `Terminate` для каждого работника ❶ и один раз для вызова `join` в поток каждого работника ❷. Если бы мы попытались отправить сообщение и сразу же вызвать `join` в том же цикле, то мы бы не смогли гарантировать, что работник в текущей итерации будет именно тем, кто получит сообщение из канала.

Чтобы лучше понять, зачем нужно два отдельных цикла, представьте себе сценарий с двумя работниками. Если бы мы использовали один цикл для итерации по каждому работнику, то во время первой итерации сообщение о завершении было бы отправлено по каналу, а метод `join` был бы вызван для потока первого работника. Если первый работник был бы в этот момент занят обработкой запроса, то второй работник подобрал бы указанное сообщение из канала и отключился бы. Нам пришлось бы ждать, пока отключится первый работник, но это так не произойдет, потому что сообщение о завершении приняли второй поток. Взаимоблокировка!

Для того чтобы предотвратить этот сценарий, мы сначала помещаем все сообщения `Terminate` в канале в один цикл, затем выполняем `join` для всех потоков исполнения в еще одном цикле. Каждый работник прекратит получать запросы на канале, как только получит сообщение о завершении. Таким образом, мы можем быть уверены в том, что, если отправляем такое же число сообщений о завершении, что и число работников, каждый работник получит такое сообщение до вызова метода `join` для своего потока.

Для того чтобы увидеть этот код в действии, давайте модифицируем функцию `main` так, чтобы принимать только два запроса, прежде чем корректно завершить работу сервера, как показано в листинге 20.26.

Листинг 20.26. Выключение сервера после выполнения двух запросов путем выхода из цикла

src/bin/main.rs

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take(2) {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    println!("Выключение.");
}
```

Не хотелось, чтобы реальный веб-сервер выключался после обслуживания только двух запросов. Этот код просто демонстрирует, что корректное отключение и очистка находятся в рабочем состоянии.

Метод `take` определен в типаже `Iterator` и ограничивает итерацию максимум двумя первыми элементами. `ThreadPool` выйдет из области видимости в конце функции `main`, а затем заработает реализация типажа `Drop`.

Запустите сервер с помощью команды `cargo run` и сделайте три запроса. Третий запрос должен выдавать ошибку, а в терминале вы должны увидеть данные, как показано ниже:

```
$ cargo run
    Compiling hello v0.1.0 (file:///projects/hello)
     Finished dev [unoptimized + debuginfo] target(s) in 1.0 secs
      Running `target/debug/hello`
Работник 0 получил задание; исполняется.
Работник 3 получил задание; исполняется.
Выключение.
Всем работникам отправляется сообщение о завершении.
Все работники выключаются.
Выключается работник 0
Работнику 1 сказано завершить работу.
Работнику 2 сказано завершить работу.
Работнику 0 сказано завершить работу.
Работнику 3 сказано завершить работу.
Выключается работник 1
Выключается работник 2
Выключается работник 3
```

Вы, возможно, увидите, что работники и сообщения выведены в другом порядке.

Мы видим работу этого кода из сообщений: работники 0 и 3 получили первые два запроса, а затем на третьем запросе сервер перестал принимать соединения. Когда `ThreadPool` выходит из области видимости в конце функции `main`, запускается его реализация типажа `Drop` и пул сообщает всем работникам о завершении работы. Каждый работник выводит сообщение, когда видит сообщение о завершении, а затем пул потоков исполнения вызывает метод `join`, чтобы выключить поток у каждого работника.

Обратите внимание на один интересный аспект этого исполнения: `ThreadPool` отправлял сообщения о завершении по каналу, а перед тем, как какой-либо работник получал сообщения, мы пытались применить метод `join` к работнику 0. Работник 0 еще не получил сообщение о завершении, а потому главный поток исполнения заблокировался в ожидании завершения работника 0. Тем временем каждый работник получал сообщение о завершении. Когда работник 0 закончил, главный поток исполнения ждал до тех пор, пока закончат остальные. В этот момент все они получили сообщение о завершении и смогли выключиться.

Поздравляем! Мы завершили проект, теперь у нас есть базовый веб-сервер, который использует пул потоков исполнения для асинхронного ответа. Мы можем выполнять корректное выключение сервера, которое очищает все потоки в пуле. Зайдите на https://www.nostarch.com/Rust2018/, откуда вы можете скачать полный код этой главы для справки.

Мы могли бы добиться здесь большего! Если вы хотите продолжить развитие этого проекта, то вот несколько идей:

- Добавьте документацию к `ThreadPool` и его публичным методам.
- Добавьте тесты функциональности библиотеки.
- Измените вызовы метода `unwrap` для более надежной обработки ошибок.
- Используйте `ThreadPool` для выполнения какой-нибудь задачи помимо обслуживания веб-запросов.
- Найдите упаковку пула потоков исполнения на https://crates.io/ и реализуйте похожий веб-сервер, используя найденную упаковку. Затем сравните его API и надежность с пулом потоков исполнения, который реализовали мы.

Итоги

Молодцы! Вы дошли до конца книги! Мы хотим поблагодарить вас за то, что вы путешествовали по языку Rust вместе с нами. Теперь вы готовы реализовывать собственные проекты на этом языке и участвовать в проектах других людей. Помните, что существует гостеприимное сообщество растиан, всегда готовых помочь в решении любых проблем, с которыми вы столкнетесь на планете Rust.

Приложение A
Ключевые слова

Список ниже содержит ключевые слова, зарезервированные в языке Rust для текущего или будущего использования. По этой причине их нельзя применять в качестве идентификаторов (за исключением сырых идентификаторов из раздела «Сырые идентификаторы» (с. 572)), включая имена функций, переменных, параметров, полей структур, модулей, упаковок, констант, макрокоманд, статических значений, атрибутов, типов, типажей или жизненных циклов.

Ключевые слова, употребляемые в настоящее время

Следующие ключевые слова в настоящее время имеют описанную функциональность.

`as`	выполнить примитивное приведение типов, устранить неоднозначность конкретного типажа, содержащего элемент, либо переименовать элементы в инструкциях `use` и `extern crate`
`break`	выйти из цикла немедленно
`const`	определить константные элементы или константные сырые указатели
`continue`	перейти к следующей итерации цикла
`crate`	связать внешнюю упаковку или макропеременную, представляющую упаковку, в которой определена макрокоманда
`dyn`	динамическая диспетчеризация типажного объекта

`else`	запасной вариант для управляющих конструкций `if` и `if let`
`enum`	определить перечисление
`extern`	связать внешнюю упаковку, функцию или переменную
`false`	булев литерал «ложь»
`fn`	определить тип функции или тип указателя функции
`for`	циклически перебрать элементы из итератора, реализовать типаж либо указать жизненный цикл более высокого ранга
`if`	выполнить ветвление, основываясь на результате условного выражения
`impl`	реализовать внутренне присущую либо типажную функциональность
`in`	часть синтаксиса цикла `for`
`let`	привязать переменную
`loop`	выполнить безусловный цикл
`match`	сопоставить значение с паттернами
`mod`	определить модуль
`move`	заставить замыкание владеть всеми элементами, захваченными в его среде
`mut`	обозначить изменяемость в ссылках, сырых указателях или привязках в паттернах
`pub`	обозначить публичную видимость в полях структур, блоках `impl` либо модулях
`ref`	привязать по ссылке
`return`	вернуться из функции
`Self`	псевдоним для типа, реализующего типаж
`self`	метод или текущий модуль
`static`	глобальная переменная или жизненный цикл, длящийся на протяжении исполнения всей программы
`struct`	определить структуру
`super`	родительский модуль текущего модуля
`trait`	определить типаж
`true`	булев литерал «истина»
`type`	определить псевдоним типа либо связанный тип
`unsafe`	обозначить небезопасный код, функции, типажи или реализации
`use`	ввести символы в область видимости
`where`	обозначить условия, которые ограничивают тип
`while`	выполнить условный цикл, основываясь на результате выражения

Ключевые слова, зарезервированные для использования в будущем

Следующие ключевые слова не имеют никакой функциональности, но зарезервированы в языке Rust для потенциального использования в будущем.

- `abstract`
- `async`
- `become`
- `box`
- `do`
- `final`
- `macro`
- `override`
- `priv`
- `try`
- `typeof`
- `unsized`
- `virtual`
- `yield`

Сырые идентификаторы

Сырые идентификаторы представляют собой синтаксис, который позволяет использовать ключевые слова там, где они обычно не разрешены. Сырой идентификатор применяется путем добавления к ключевому слову префикса `r#`.

Например, `match` — это ключевое слово. Если вы попытаетесь скомпилировать следующую функцию, которая использует `match` в качестве своего имени:

src/main.rs

```
fn match(needle: &str, haystack: &str) -> bool {
    haystack.contains(needle)
}
```

то получите такую ошибку:

```
error: expected identifier, found keyword `match`
 --> src/main.rs:4:4
  |
```

```
4 | fn match(needle: &str, haystack: &str) -> bool {
  |    ^^^^^ expected identifier, found keyword
```

Ошибка показывает, что вы не можете использовать ключевое слово `match` в качестве идентификатора функции. Для того чтобы использовать `match` в качестве имени функции, вы должны использовать синтаксис сырых идентификаторов, как тут:

src/main.rs

```
fn r#match(needle: &str, haystack: &str) -> bool {
    haystack.contains(needle)
}

fn main() {
    assert!(r#match("foo", "foobar"));
}
```

Этот код будет компилироваться без ошибок. Обратите внимание на префикс `r#` в имени функции в ее определении, а также на то, где функция вызывается в `main`.

Сырые идентификаторы позволяют использовать любое выбранное вами слово в качестве идентификатора, даже если это зарезервированное ключевое слово. Кроме того, благодаря сырым идентификаторам можно использовать библиотеки, написанные в редакции языка Rust, отличные от той, которую применяет ваша упаковка. Например, `try` не является ключевым словом в редакции 2015 года, но является таковым в редакции 2018-го. Если вы зависите от библиотеки, написанной в редакции 2015 года, в которой имеется функция `try`, то вам потребуется синтаксис сырых идентификаторов, в данном случае `r#try`, чтобы вызывать эту функцию из кода редакции 2018 года. Дополнительную информацию о редакциях смотрите в приложении Д ниже.

Приложение Б
Операторы и символы

Это приложение содержит глоссарий синтаксиса языка Rust, включая операторы и другие символы, которые появляются по отдельности или в контексте путей, обобщений, границ типажа, макрокоманд, атрибутов, комментариев, кортежей и скобок.

Операторы

Таблица Б.1 содержит операторы языка Rust, пример того, как оператор выглядит в контексте, краткое объяснение и информацию о перегрузке оператора. Если оператор перегружаемый, то в списке указывается соответствующий типаж, используемый для его перегрузки.

Таблица Б.1. Операторы

Оператор	Пример	Объяснение	Перегружаемый?
`!`	`ident!(...), ident!{...}, ident![...]`	Расширение макрокоманды	
`!`	`!expr`	Побитовое или логическое дополнение (отрицание)	Not
`!=`	`var != expr`	Сравнение неравенства	PartialEq
`%`	`expr % expr`	Арифметический остаток	Rem
`%=`	`var %= expr`	Арифметический остаток и присвоение	RemAssign
`&`	`&expr, &mut expr`	Заимствование	
`&`	`&type, &mut type, &'a type, &'a mut type`	Тип заимствованного указателя	
`&`	`expr & expr`	Побитовое И	BitAnd

Оператор	Пример	Объяснение	Перегружаемый?
&=	`var &= expr`	Побитовое И и присваивание	BitAndAssign
&&	`expr && expr`	Логическое И	
*	`expr * expr`	Арифметическое умножение	Mul
*=	`var *= expr`	Арифметическое умножение и присваивание	MulAssign
*	`*expr`	Разыменование (следование к значению по указателю с помощью оператора разыменования)	
*	`*const type, *mut type`	Сырой указатель	
+	`trait + trait, 'a + trait`	Ограничение составного типа	
+	`expr + expr`	Арифметическое сложение	Add
+=	`var += expr`	Арифметическое сложение и присваивание	AddAssign
,	`expr, expr`	Разделитель аргументов и элементов	
-	`- expr`	Арифметическое отрицание	Neg
-	`expr - expr`	Арифметическое вычитание	Sub
-=	`var -= expr`	Арифметическое вычитание и присваивание	SubAssign
->	`fn(...) -> type, |...| -> type`	Тип, возвращаемый из функции и замыкания	
	`expr.ident`	Доступ к члену	
	`.., expr.., ..expr, expr..expr`	Литерал интервала с правосторонним исключением	
..=	`..=expr, expr..=expr`	Литерал интервала с правосторонним включением	
	`..expr`	Синтаксис обновления литерала структуры	
	`variant(x, ..), struct_type {x, .. }`	Привязка к паттерну "И все остальное"	
	`expr...expr`	В паттерне: паттерн включающего интервала	
/	`expr / expr`	Арифметическое деление	Div
/=	`var /= expr`	Арифметическое деление и присваивание	DivAssign
:	`pat: type, ident: type`	Ограничения	

Оператор	Пример	Объяснение	Перегружаемый?
`:`	`ident: expr`	Инициализатор поля структуры	
`:`	`'a: loop {...}`	Метка цикла	
`;`	`expr;`	Инструкция либо терминатор элемента	
`;`	`[...; len]`	Часть синтаксиса массива с фиксированным размером	
`<<`	`expr << expr`	Сдвиг влево	Shl
`<<=`	`var <<= expr`	Сдвиг влево и присваивание	ShlAssign
`<`	`expr < expr`	Меньше, чем сравниваемое	PartialOrd
`<=`	`expr <= expr`	Меньше или равно сравниваемому	PartialOrd
`=`	`var = expr, ident = type`	Присваивание/эквивалентность	
`==`	`expr == expr`	Сравнение равенства	PartialEq
`=>`	`pat => expr`	Часть синтаксиса рукава выражения match	
`>`	`expr > expr`	Больше, чем сравниваемое	PartialOrd
`>=`	`expr >= expr`	Больше или равно сравниваемому	PartialOrd
`>>`	`expr >> expr`	Сдвиг вправо	Shr
`>>=`	`var >>= expr`	Сдвиг вправо и присваивание	ShrAssign
`@`	`ident @ pat`	Привязка к паттерну	
`^`	`expr ^ expr`	Побитовое исключающее ИЛИ	BitXor
`^=`	`var ^= expr`	Побитовое исключающее ИЛИ и присваивание	BitXorAssign
`|`	`pat | pat`	Альтернативы паттернов	
`|`	`expr | expr`	Побитовое ИЛИ	BitOr
`|=`	`var |= expr`	Побитовое ИЛИ и присваивание	BitOrAssign
`||`	`expr || expr`	Логическое ИЛИ	
`?`	`expr?`	Передача ошибок	

Неоператорные символы

В следующих таблицах содержатся все не-буквы, которые не функционируют как операторы, то есть они не ведут себя как вызов функции или метода.

Таблица Б.2 показывает символы, которые появляются по отдельности и являются действительными в различных местах.

Таблица Б.2. Автономный синтаксис

Символ	Объяснение
`'ident`	Именованный жизненный цикл или метка цикла
`...u8, ...i32, ...f64,...usize,etc.`	Числовой литерал определенного типа
`"..."`	Строковый литерал
`r"...", r#"..."#, r##"..."##` и т. д	Сырой строковый литерал; экранирующие символы не обрабатываются
`b"..."`	Байтовый строковый литерал; создает `[u8]` вместо строки
`br"...", br#"..."#, br##"..."##` и т. д	Сырой байтовый строковый литерал; комбинация сырого и байтового строкового литерала
`'...'`	Символьный (знаковый) литерал
`b'...'`	Байтовый литерал в кодировке ASCII
`|...| expr`	Замыкание
`!`	Всегда пустой нижний тип для отклоняющихся функций
`_`	Привязка к паттерну «проигнорировать»; также используется для придания целочисленным литералам удобочитаемого вида

Таблица Б.3 показывает символы, которые появляются в контексте пути через иерархию модулей к элементу.

Таблица Б.3. Синтаксис, имеющий отношение к путям

Символ	Объяснение
`ident::ident`	Путь пространства имен
`::path`	Путь относительно корня упаковки (то есть явно выраженный абсолютный путь)
`self::path`	Путь относительно текущего модуля (то есть явно выраженный относительный путь)
`super::path`	Путь относительно родителя текущего модуля
`type::ident, <type as trait>::ident`	Связанные константы, функции и типы
`<type>::`	Связанный элемент для типа, который не может быть назван напрямую (например, `<&T>::...`, `<[T]>::...` и т. д.)
`trait::method(...)`	Устранение неоднозначности вызова метода путем назначения имени типажу, который определяет этот метод
`type::method(...)`	Устранение неоднозначности вызова метода путем назначения имени типу, для которого он определен
`<type as trait>::method(...)`	Устранение неоднозначности вызова метода путем назначения имени типажу и типу

Таблица Б.4 показывает символы, которые появляются в контексте параметров обобщенного типа.

Таблица Б.4. Обобщения[1]

Символ	Объяснение
`path<...>`	Задает параметры для обобщенного типа в типе (например, `Vec<u8>`)
`path::<...>, method::<...>`	Задает параметры для обобщенного типа, функции или метода в выражении; часто упоминается как турборыба[1] (например, `"42".parse::<i32>()`)
`fn ident<...>`	Определить обобщенную функцию
`struct ident<...>`	Определить обобщенную структуру
`enum ident<...>`	Определить обобщенное перечисление
`impl<...>`	Определить обобщенную реализацию
`for<...> type`	Границы жизненного цикла с более высоким рангом
`type<ident=type>`	Обобщенный тип, в котором один или несколько связанных типов имеют конкретные присваивания (например, `Iterator<Item=T>`)

Таблица Б.5 показывает символы, которые появляются в контексте ограничения параметров обобщенного типа границами типажа.

Таблица Б.5. Ограничения, имеющие отношение к типажам

Символ	Объяснение
`T: U`	Обобщенный параметр `T`, ограниченный только теми типами, которые реализуют `U`
`T: 'a`	Обобщенный тип `T` должен исчерпывать жизненный цикл `'a` (то есть тип не может транзитивно содержать любые ссылки с жизненными циклам короче, чем у `'a`)
`T: 'static`	Обобщенный тип `T` не содержит заимствованных ссылок, кроме `'static`
`'b: 'a`	Обобщенный жизненный цикл `'b` должен быть больше жизненного цикла `'a`
`T:?Sized`	Позволить параметру обобщенного типа быть динамически изменяемым типом
`'a + trait, trait + trait`	Составное ограничение типа

[1] Турборыба (turbofish) — часть синтаксиса в языке Rust, которая принимает вид `::<НекийТип>`.

Таблица Б.6 показывает символы, которые появляются в контексте вызова или определения макрокоманд и указания атрибутов в элементе.

Таблица Б.6. Макрокоманды и атрибуты

Символ	Объяснение
`#[meta]`	Внешний атрибут
`#![meta]`	Внутренний атрибут
`$ident`	Макроподстановка
`$ident:kind`	Макрозахват
`$(...)`	Макроповторение

Таблица Б.7 показывает символы, создающие комментарии.

Таблица Б.7. Комментарии

Символ	Объяснение
`//`	Строчный комментарий
`//!`	Внутристрочный комментарий документации
`///`	Внешнестрочный комментарий документации
`/*...*/`	Блочный комментарий
`/*!...*/`	Внутренний блочный комментарий документации
`/**...*/`	Внешний блочный комментарий документации

Таблица Б.8 показывает символы, которые появляются в контексте кортежей.

Таблица Б.8. Кортежи

Символ	Объяснение
`()`	Пустой кортеж (так называемый unit), как литерал и как тип
`(expr)`	Выражение в скобках
`(expr,)`	Кортежное выражение с одним элементом
`(type,)`	Кортежный тип с одним элементом
`(expr, ...)`	Кортежное выражение
`(type, ...)`	Кортежный тип
`expr(expr, ...)`	Выражение вызова функции; также используется для инициализации кортежных структур и вариантов кортежных перечислений

Символ	Объяснение
`ident!(...)`, `ident!{...}`, `ident![...]`	Вызов макрокоманды
`expr.0`, `expr.1` и т. д.	Индексирование кортежа

Таблица Б.9 показывает контексты, в которых используются фигурные скобки.

Таблица Б.9. Фигурные скобки

Символ	Объяснение
`{...}`	Блочное выражение
`Type {...}`	Литерал `struct`

Таблица Б.10 показывает контексты, в которых используются квадратные скобки.

Таблица Б.10. Квадратные скобки

Контекст	Объяснение
`[...]`	Литерал массива
`[expr; len]`	Литерал массива, содержащий `len` копий выражения `expr`
`[type; len]`	Массивный тип, содержащий `len` экземпляров типа `type`
`expr[expr]`	Индексирование коллекции; перегружаемое (`Index`, `IndexMut`)
`expr[..]`, `expr[a..]`,	
`expr[..b]`, `expr[a..b]`	Индексирование коллекции, внешне похожее на нарезку коллекции, использующее `Range`, `RangeFrom`, `RangeTo` или `RangeFull` в качестве «индекса»

Приложение B
Генерируемые типажи

В разных частях книги мы обсуждали атрибут `derive`, который можно применять к определению структуры или перечисления. Атрибут `derive` генерирует код, который будет выполнять реализацию типажа с собственной реализацией по умолчанию в типе, аннотированном с помощью синтаксиса `derive`.

В этом приложении дается справочный материал обо всех типажах стандартной библиотеки, которые вы можете использовать с атрибутом `derive`. Каждый раздел охватывает:

- Операторы и методы, работа с которыми возможна благодаря генерированию этого типажа.
- Информацию о том, что делает реализация типажа, предусмотренная атрибутом `derive`.
- Информацию о том, что реализация типажа означает для типа.
- Условия, в которых разрешено или запрещено реализовывать типаж.
- Примеры операций, требующих типаж.

Если вы хотите иметь поведение, отличное от предусмотренного атрибутом `derive`, обратитесь к документации стандартной библиотеки о каждом типаже для получения подробных сведений о том, как реализовывать их вручную.

Остальные типажи, определенные в стандартной библиотеке, не могут быть реализованы в типах с помощью атрибута `derive`. Эти типажи не имеют разумного поведения по умолчанию, поэтому вы должны реализовать их таким образом, чтобы это имело смысл с точки зрения ваших целей.

Пример типажа, который нельзя сгенерировать с помощью указанного атрибута, — `Display`, который занимается форматированием для конечных пользователей. Вы всегда должны учитывать подходящий способ демонстрации типа конечному пользователю. Какие части этого типа конечный пользователь должен видеть? Какие части он счел бы уместными? Какой формат данных будет для него

наиболее релевантным? Компилятор Rust не имеет такого понимания, поэтому он не способен предоставить вам соответствующее поведение по умолчанию.

Список генерируемых типажей, приведенный в этом приложении, не является исчерпывающим: библиотеки могут реализовывать атрибут `derive` для собственных типажей, что в итоге делает типажи, с которыми вы можете использовать атрибут `derive`, поистине открытым. Реализация атрибута `derive` предусматривает использование процедурной макрокоманды, которая описана в разделе «Макрокоманды» (с. 515).

Debug для вывода рабочей информации

Типаж `Debug` обеспечивает возможность отладочного форматирования в форматных строках, которые вы указываете путем добавления `:?` внутри заполнителей `{}`.

Типаж `Debug` позволяет печатать экземпляры типа для отладочных целей, чтобы вы и другие программисты, использующие ваш тип, могли проверить экземпляр в определенный момент работы программы.

Типаж `Debug` необходим, например, при использовании макрокоманды `assert_eq!`. Указанная макрокоманда выводит значения экземпляров, переданных в качестве аргументов, если проверочное утверждение о равенстве не удовлетворяется, в результате чего программисты могут видеть, почему эти два экземпляра не равны.

PartialEq и Eq для сравнений равенств

Типаж `PartialEq` позволяет сравнивать экземпляры типа для проверки равенства и использовать операторы `==` и `!=`.

Генерирование типажа `PartialEq` реализует метод `eq`. Когда `PartialEq` генерируется в структурах, два экземпляра равны только в том случае, если все поля равны, и экземпляры не равны, если какие-то поля не равны. При генерировании в перечислениях каждый вариант равен самому себе и не равен другим вариантам.

Типаж `PartialEq` необходим, например, при использовании макрокоманды `assert_eq!`, которая должна сравнивать два экземпляра типа на равенство.

Типаж `Eq` не имеет методов. Его цель — подавать сигнал о том, что для каждого значения аннотированного типа данное значение равно самому себе. Типаж `Eq` может применяться только к типам, которые также реализуют `PartialEq`, хотя не все типы, выполняющие `PartialEq`, могут реализовывать `Eq`. Одним из примеров этого являются числовые типы с плавающей точкой: реализация таких чисел констатирует, что два экземпляра значения не-числа (`NaN`) не равны друг другу.

Примером ситуации, когда требуется типаж `Eq`, являются ключи в хеш-отображении `HashMap<K, V>`, благодаря которому `HashMap<K, V>` может отличать, одинаковы ли два ключа.

PartialOrd и Ord для сравнений порядка

Типаж `PartialOrd` позволяет сравнивать экземпляры типа для сортировки. Тип, реализующий `PartialOrd`, используется с операторами `<`, `>`, `<=` и `>=`. Типаж `PartialOrd` можно применять только к тем типам, которые также реализуют `PartialEq`.

Генерирование типажа `PartialOrd` реализует метод `partial_cmp`, возвращающий перечисление `Option<Ordering>`, которое будет равно `None`, если переданные значения не производят упорядоченности. Примером значения, которое не производит упорядоченности, хотя большинство значений этого типа могут сравниваться, является значение не-числа (`NaN`) с плавающей точкой. Вызов `partial_cmp` с любым числом с плавающей точкой и значением `NaN` с плавающей точкой вернет `None`.

При генерировании в структурах типаж `PartialOrd` сопоставляет два экземпляра путем сравнения значения в каждом поле в порядке, в котором поля появляются в определении структуры. При генерировании в перечислениях варианты перечисления, объявленные в определении перечисления ранее, считаются меньшими, чем варианты, следующие позже.

Типаж `PartialOrd` требуется, например, для метода `gen_range` из упаковки `rand`, который генерирует случайное значение в интервале, заданном низким и высоким значениями.

Типаж `Ord` позволяет узнавать, что для любых двух значений аннотированного типа существует допустимый порядок. Типаж `Ord` реализует метод `cmp`, который вместо `Option<Ordering>` возвращает `Ordering`, поскольку допустимое упорядочение `Ordering` возможно всегда. Вы можете применять типаж `Ord` только к типам, которые также реализуют `PartialOrd` и `Eq` (причем `Eq` требует `PartialEq`). При генерировании в структурах и перечислениях `cmp` ведет себя так же, как генерируемая (производная) реализация для `partial_cmp` ведет себя с `PartialOrd`.

Примером того, когда требуется `Ord`, является хранение значений в типе `BTreeSet<T>`, структуре, хранящей данные на основе порядка сортировки значений.

Clone и Copy для дублирования значений

Типаж `Clone` позволяет явным образом создавать глубокую копию значения, а процесс дублирования предусматривает выполнение произвольного кода и ко-

пирование данных кучи. Дополнительные сведения о клонировании смотрите в разделе «Пути взаимодействия переменных и данных: Clone» (с. 99).

Генерирование типажа `Clone` реализует метод `clone`, который при выполнении для всего типа целиком вызывает `clone` для каждой части типа. Это означает, что для генерирования типажа `Clone` все поля или значения в типе также должны реализовывать типаж `Clone`.

Пример того, когда требуется `Clone`, — вызов метода `to_vec` для среза. Срез не владеет экземплярами типа, которые он содержит, но вектор, возвращаемый из `to_vec`, должен будет владеть своими экземплярами, поэтому `to_vec` вызывает `clone` для каждого элемента. Таким образом, тип, хранящийся в срезе, должен реализовывать типаж `Clone`.

Типаж `Copy` позволяет дублировать значение путем копирования только тех частей, которые хранятся в стеке, никакого произвольного кода не требуется. Дополнительную информацию о типаже `Copy` смотрите в разделе «Данные только из стека: Copy» (с. 99).

Типаж `Copy` не определяет методов, которые не дают программистам перегружать эти методы и нарушать предположение о том, что никакой произвольный код не выполняется. Благодаря этому все программисты допускают, что значения будут копироваться очень быстро.

Вы можете сгенерировать типаж `Copy` для любого типа, все части которого реализуют `Copy`. Вы можете применять типаж `Copy` к типам, которые также реализуют `Clone`, поскольку тип, выполняющий `Copy`, имеет тривиальную реализацию `Clone`, выполняющую ту же задачу, что и `Copy`.

Типаж `Copy` требуется редко. Типы, реализующие `Copy`, имеют в своем распоряжении оптимизации, то есть не нужно вызывать `clone`, что делает код более сжатым.

Всего, что можно сделать с помощью `Copy`, можно также добиться с помощью `Clone`, но код, возможно, будет медленнее, либо местами придется использовать `clone`.

Хеш для отображения значения в значение фиксированного размера

Типаж `Hash` позволяет брать экземпляр типа произвольного размера и отображать этот экземпляр в значение фиксированного размера с помощью хеш-функции. Генерирование типажа `Hash` реализует метод `hash`. Сгенерированная (производная) реализация метода `hash` объединяет результат вызова `hash` для каждой части типа, то есть для генерирования `Hash` все поля или значения должны также реализовывать `Hash`.

Примером того, когда требуется `Hash`, является хранение ключей в `HashMap<K,V>` для эффективного хранения данных.

Default для значений по умолчанию

Типаж `Default` позволяет создавать значение по умолчанию для типа. Генерирование типажа `Default` реализует функцию `default`. Сгенерированная реализация функции `default` вызывает `default` для каждой части типа, то есть для генерирования типажа `Default` все поля или значения в типе должны также реализовывать `Default`.

Функция `Default::default` обычно используется в сочетании с синтаксисом обновления структуры, описанным в разделе «Создание экземпляров из других экземпляров с помощью синтаксиса обновления структуры» (с. 118). Вы можете настроить несколько полей структуры, а затем установить и использовать значение по умолчанию для остальных полей с помощью `..Default::default()`.

Типаж `Default` требуется, например, когда вы используете метод `unwrap_or_default` в экземплярах перечисления `Option<T>`. Если `Option<T>` равно `None`, то метод `unwrap_or_default` вернет результат `Default::default` для типа `T`, хранящегося в `Option<T>`.

Приложение Г
Полезные инструменты разработки

В этом приложении мы расскажем о некоторых полезных инструментах разработки проекта Rust. Мы рассмотрим автоматическое форматирование, быстрые способы применения предупреждающих исправлений, статический анализатор кода и интеграцию со средами разработки.

Автоматическое форматирование с помощью rustfmt

Инструмент rustfmt переформатирует код в соответствии со стилем кода, принятым в сообществе. Во многих совместных проектах используется rustfmt, чтобы предотвратить споры о том, какой стиль применять при написании кода на Rust: каждый форматирует свой код с помощью этого инструмента.

Для того чтобы установить rustfmt, введите следующее:

```
$ rustup component add rustfmt
```

Эта команда дает вам `rustfmt` и `cargo-fmt`, подобно тому, как Rust дает вам `rustc` и `cargo`. Для того чтобы отформатировать любой проект Cargo, введите следующее:

```
$ cargo fmt
```

Выполнение этой команды переформатирует весь код Rust в текущей упаковке. Она должна изменить только стиль кода, а не его семантику. Для получения дополнительной информации о rustfmt обратитесь к его документации по адресу https://github.com/rust-lang/rustfmt/.

Исправляйте код с помощью rustfix

Инструмент `rustfix` входит в комплект установки Rust и может автоматически устранять некоторые предупреждения компилятора. Если вы написали код на

Rust, то, вероятно, видели предупреждения компилятора. Например, рассмотрим код:

src/main.rs

```
fn do_something() {}

fn main() {
    for i in 0..100 {
        do_something();
    }
}
```

Здесь мы вызываем функцию `do_something` 100 раз, но никогда не используем переменную `i` в теле цикла `for`. Компилятор об этом предупреждает:

```
$ cargo build
    Compiling myprogram v0.1.0 (file:///projects/myprogram)
warning: unused variable: `i`
 --> src/main.rs:4:9
  |
4 |     for i in 1..100 {
  |         ^ help: consider using `_i` instead
  |
  = note: #[warn(unused_variables)] on by default

    Finished dev [unoptimized + debuginfo] target(s) in 0.50s
```

Предупреждение рекомендует, чтобы мы использовали `_i` в качестве имени: подчеркивание указывает на то, что мы не намерены использовать эту переменную. Можно автоматически применить эту рекомендацию с помощью инструмента rustfix, выполнив команду `cargo fix`:

```
$ cargo fix
    Checking myprogram v0.1.0 (file:///projects/myprogram)
      Fixing src/main.rs (1 fix)
    Finished dev [unoptimized + debuginfo] target(s) in 0.59s
```

Когда мы снова посмотрим на src/main.rs, мы увидим, что команда `cargo fix` изменила код:

src/main.rs

```
fn do_something() {}

fn main() {
    for _i in 0..100 {
        do_something();
    }
}
```

Переменная цикла `for` теперь называется `_i`, и предупреждение больше не появляется.

Вы также можете использовать команду `cargo fix` для транзита кода между разными редакциями Rust. Редакции рассматриваются в приложении Д далее.

Статический анализ кода с помощью Clippy

Инструмент Clippy представляет собой коллекцию проверок для анализа кода, благодаря которому вы можете фиксировать распространенные ошибки и улучшать код Rust.

Для того чтобы инсталлировать Clippy, введите следующее:

```
$ rustup component add clippy
```

Для запуска проверок Clippy в любом проекте Cargo введите:

```
$ cargo clippy
```

Например, вы пишете программу, которая использует аппроксимацию математической константы ϖ, как в этой программе:

src/main.rs

```
fn main() {
    let x = 3.1415;
    let r = 8.0;
    println!("площадь окружности равна {}", x * r * r);
}
```

Выполнение команды `cargo clippy` в этом проекте приведет к ошибке[1]:

```
error: approximate value of `f{32, 64}::consts::PI` found. Consider using it
directly
 --> src/main.rs:2:13
  |
2 |     let x = 3.1415;
  |             ^^^^^^
  |
  = note: #[deny(clippy::approx_constant)] on by default
  = help: for further information visit https://rust-lang-nursery.github.io/
rust-clippy/master/index.html#approx_constant
```

Благодаря ошибке понятно, что в языке Rust эта константа определена точнее, и программа могла бы быть правильнее, если бы вы использовали константу. Затем вы изменяете код так, чтобы в нем использовалась константа `PI`. Следующий код не приводит ни к ошибкам, ни к предупреждениям со стороны Clippy:

[1] `ошибка: найдено приближенное значение `f{32, 64}::conts::PI`. Подумайте о его прямом использовании`

src/main.rs

```
fn main() {
    let x = std::f64::consts::PI;
    let r = 8.0;
    println!("площадь окружности равна {}", x * r * r);
}
```

Для получения дополнительной информации о Clippy обратитесь к его документации по адресу https://github.com/rust-lang/rust-clippy/.

Интеграция с IDE с помощью языкового сервера Rust Language Server

В целях интеграции со средой разработки проект Rust распространяет языковой сервер *Rust Language Server* (`rls`). Этот инструмент общается через протокол языкового сервера *Language Server Protocol*, описанный на веб-сайте http://langserver.org/. Указанный протокол представляет собой спецификацию общения между интегрированной средой разработки и языками программирования. Разные клиенты могут использовать `rls`, такой как плагин Rust для кода Visual Studio, который можно найти по адресу https://marketplace.visualstudio.com/items?itemName=rust-lang.rust.

Для того чтобы установить rls, введите следующее:

```
$ rustup component add rls
```

Затем установите поддержку языкового сервера в вашей IDE. Вы получите такие возможности, как автозаполнение, переход к определению и внутристрочные ошибки.

Для получения дополнительной информации о `rls` обратитесь к документации по адресу https://github.com/rust-lang/rls/.

Приложение Д
Редакции

В главе 1 вы видели, что команда `cargo new` добавляет в файл Cargo.toml немного метаданных о редакции языка. В данном приложении речь идет именно об этом!

Язык Rust и компилятор имеют шестинедельный релизный цикл, то есть пользователи постоянно получают новые языковые средства. В других языках программирования бо́льшие изменения выпускаются реже. В Rust меньшие обновления выпускаются чаще. Через некоторое время все эти крошечные изменения складываются. Но от релиза к релизу бывает трудно оглянуться назад и сказать: «Ух ты, между Rust 1.10 и Rust 1.31 большая разница!»

Каждые два-три года команда разработчиков Rust выпускает новую редакцию. Каждая редакция объединяет языковые средства, включенные в чистый пакет, с полностью обновленной документацией и инструментами. Новые редакции выпускаются в рамках обычного шестинедельного релизного процесса.

Редакции служат разным целям для разных людей:

- Для активных пользователей Rust новая редакция объединяет изменения в простой для понимания пакет.
- Для тех, кто не пользуется Rust, новая редакция сигнализирует о том, что были достигнуты серьезные успехи, которые, возможно, побудят применять язык Rust.
- Тех, кто разрабатывает язык Rust, новая редакция сплачивает с точки зрения проекта в целом.

На момент написания этой книги имеются две редакции языка: Rust 2015 и Rust 2018. Эта книга написана под Rust редакции 2018 года.

Ключ `edition` в файле Cargo.toml говорит о том, какую редакцию компилятор должен использовать для вашего кода. Если ключа не существует, то в качестве значения используется 2015 по соображениям обратной совместимости.

В каждом проекте можно предпочесть редакцию, отличную от редакции по умолчанию 2015 года. Редакции могут содержать несовместимые изменения, например, включение нового ключевого слова, которое конфликтует с идентификаторами в коде. Однако если вы не согласны с этими изменениями, то ваш код по-прежнему будет компилироваться даже при обновлении версии используемого компилятора Rust.

Все версии компилятора поддерживают любую редакцию, существовавшую до релиза этого компилятора, они могут связать упаковки всех поддерживаемых редакций вместе. Изменения редакции влияют только на то, как компилятор первоначально анализирует код. Следовательно, если вы используете Rust 2015, а одна из зависимостей применяет Rust 2018, то проект будет компилироваться и сможет использовать эту зависимость. Противоположная ситуация, когда проект использует Rust 2018, а зависимость применяет Rust 2015, также возможна.

Таким образом, большинство языковых средств доступно во всех редакциях. Разработчики, использующие любую версию языка Rust, по-прежнему будут видеть улучшения по мере появления новых стабильных релизов. Однако в некоторых случаях, главным образом при добавлении новых ключевых слов, некоторые новые языковые средства могут быть доступны только в более поздних версиях. Вам придется переключить релизы для применения таких средств.

Дополнительные сведения можно найти в «Руководстве по редакциям языка» (*Edition Guide*) по адресу https://doc.rust-lang.org/stable/edition-guide/. Указанное руководство представляет собой полную книгу о редакциях, в которой перечислены различия между ними и объясняется способ автоматического обновления кода до новой редакции с помощью команды `cargo fix`.

Стив Клабник, Кэрол Николс

Программирование на Rust

Перевел с английского *А. Логунов*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*И. Кизилова*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Сидорова*
Верстка	*Л. Егорова*

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: 09.2020. 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 03.09.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 47,730. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87