

Язык программирования Rust

от авторов Steve Klabnik, Carol Nichols и участников сообщества Rust

Эта версия текста предполагает, что вы используете Rust версии 1.55 (или более позднюю) с настройкой `edition="2018"` внутри файлов конфигурации `Cargo.toml` всех проектов данной книги (для использования идиом Rust версии 2018 Edition). Смотрите раздел "[Установка](#)" главы 1, чтобы установить или обновить Rust, и посмотрите на новое [Приложение E](#) для получения информации об изданиях.

Язык Rust редакции 2018 года включает в себя ряд улучшений, которые делают Rust более эргономичным и лёгким в освоении. Эта версия книги содержит ряд изменений, отражающих эти улучшения:

- Глава 7, "Управление растущими проектами с помощью пакетов, крейтов и модулей", по большей части переписана. Система модулей и пути работы с ними стали более согласованными в 2018 редакции.
- Глава 10 обзавелась новыми разделами - "Типажи как параметры" и "Возврат типов, реализующих типаж", которые разъясняют новый синтаксис `impl Trait`.
- В главе 11 добавлен раздел "Использование `Result<T, E>` в тестах", который показывает как писать тесты, использующие оператор `?`.
- Раздел "Дополнительно о временах жизни" в главе 19 был удалён, так как улучшения в компиляторе сделали конструкции из этого раздела ещё более редкими.
- В приложение D, "Макросы", была добавлена информация о процедурных макросах. Само приложение стало разделом "Макросы" главы 19.
- Приложение А, "Ключевые слова", теперь также описывает возможности сырых идентификаторов, которые позволяют взаимодействовать коду 2015 редакции с кодом 2018 редакции.
- Приложение D теперь называется "Средства разработки" и описывает инструменты, которые помогут вам писать код на Rust.
- Мы исправили ряд небольших ошибок и неточностей формулировок. Спасибо читателям, которые сообщают нам об этом!

Обратите внимание, что любой код из более ранних версий книги, продолжит компилироваться без указания `edition="2018"` в `Cargo.toml` проекта, даже если вы обновите используемую версию компилятора Rust. Это гарантия обратной совместимости Rust!

HTML-версия книги доступна онлайн по адресам <https://doc.rust-lang.org/stable/book/>(англ.) и <https://doc.rust-lang.ru/book>(рус.) и офлайн, при установке Rust с помощью `rustup`: просто запустите `rustup docs --book` чтобы открыть её.

Английский вариант книги доступен [в печатном виде и в ebook формате от No Starch Press](#).

Предисловие

Не всегда было ясно, но язык программирования Rust в основном посвящён расширению возможностей: независимо от того, какой код вы пишете сейчас, Rust позволяет вам достичь большего, чтобы программировать уверенно в более широком диапазоне областей, чем вы делали раньше.

Возьмём, к примеру, работу «системного уровня» которая касается низкоуровневых деталей управления памятью, представления данных и многопоточности. Традиционно эта область программирования считается загадочной, доступной лишь немногим избранным, посвятившим долгие годы изучению всех её печально известных подводных камней. И даже те, кто практикует это, делают всё с осторожностью, чтобы их код не был уязвим для эксплойтов, сбоев или повреждений.

Rust разрушает эти барьеры, устранивая старые подводные камни и предоставляя дружелюбный, отполированный набор инструментов, которые помогут вам на этом пути. Программисты, которым необходимо «погрузиться» в низкоуровневое управление, могут сделать это с помощью Rust, не беря на себя привычный риск аварий или дыр в безопасности, и не изучая тонкости изменчивых наборов инструментов. Более того, язык предназначен для того, чтобы легко вести вас к надёжному коду, который эффективен с точки зрения скорости и использования памяти.

Программисты, которые уже работают с низкоуровневым кодом, могут использовать Rust для повышения своих амбиций. Например, внедрение параллелизма в Rust является операцией с относительно низким риском: компилятор поймает для вас классические ошибки. И вы можете заняться более агрессивной оптимизацией в своём коде с уверенностью, что не будете случайно добавлять в код сбои или уязвимости.

Но Rust не ограничивается низкоуровневым системным программированием. Он достаточно выразителен и эргономичен, чтобы приложения CLI (Command Line Interface - консольные программы), веб-серверы и многие другие виды кода были довольно приятными для написания — позже вы найдёте простые примеры того и другого в книге. Работа с Rust позволяет вырабатывать навыки, которые переносятся из одной предметной области в другую; Вы можете изучить Rust, написав веб-приложение, а затем применить те же навыки для Raspberry Pi.

Эта книга полностью раскрывает потенциал Rust для расширения возможностей

его пользователей. Это дружелюбный и доступный материал, призванный помочь вам повысить уровень не только ваших знаний о Rust, но и ваших возможностей и уверенности как программиста в целом. Так что погружайтесь, готовьтесь учиться - и добро пожаловать в сообщество Rust!

— Nicholas Matsakis и Aaron Turon

Введение

Примечание: это издание книги так же, как и [The Rust Programming Language](#) доступно в печатном и электронном виде от [No Starch Press](#).

Добро пожаловать в *The Rust Programming Language*, вводную книгу о Rust. Язык программирования Rust помогает создавать быстрые, более надёжные приложения. Хорошая эргономика и низкоуровневый контроль часто являются противоречивыми требованиями для дизайна языков программирования; Rust бросает вызов этому конфликту. Благодаря сбалансированности мощных технических возможностей с большим удобством разработки, Rust предоставляет возможности управления низкоуровневыми элементами (например, использование памяти) без трудностей, традиционно связанными с таким контролем.

Кому подходит Rust

Rust подходит для многих людей по разным причинам. Приведём несколько самых важных групп.

Команды разработчиков

Rust обеспечивает эффективные средства для совместной работы больших команд разработчиков с различным уровнем знаний системного программирования. Низкоуровневый код подвержен множеству незаметных ошибок, которые в большинстве других языков могут быть найдены только в результате обширного тестирования и тщательного анализа кода опытными разработчиками. В Rust компилятор играет роль привратника, отказываясь компилировать код с этими неуловимыми ошибками, включая ошибки параллелизма. Компилятор позволяет команде разработчиков больше сосредоточить своё внимание на логике, а не терять время на поиски ошибок.

Rust также предлагает современные инструменты разработки для системного программирования:

- Cargo, встроенный менеджер зависимостей и инструмент сборки, добавляет,

компилирует и управляет зависимостями безболезненно и согласованно, используя экосистему Rust.

- Rustfmt обеспечивает согласованный стиль кодирования для всех разработчиков.
- Rust Language Server поддерживает интегрированную среду разработки (IDE) с автодополнением кода и встроенными сообщениями об ошибках.

Эти и другие инструменты экосистемы Rust, обеспечивают разработчикам продуктивность при написании кода системного уровня.

Студенты

Rust полезен для студентов и тех, кто заинтересован в изучении системных концепций. Используя Rust, многие люди узнали о таких темах, как разработка операционных систем. Сообщество радушно и с удовольствием ответит на вопросы начинающих. Благодаря усилиям, таким как эта книга, команды Rust хотят сделать концепции систем более доступными для большего числа людей, особенно для новичков в программировании.

Компании

Сотни компаний, больших и малых, используют Rust для различных целей. Эти задачи включают в себя инструменты командной строки, веб-сервисы, инструменты DevOps, встраиваемые устройства, анализ и транскодирование аудио и видео, крипто-валюты, биоинформатика, поисковые системы, приложения интернета вещей, машинное обучение и даже основные части браузера Firefox.

Разработчики Open Source

Rust для людей, которые хотят построить язык программирования Rust, сообщество, инструменты разработчика и библиотеки. Мы хотели бы, чтобы вы внесли свой вклад в развитие языка Rust.

Люди, которые ценят скорость и стабильность

Rust для людей, которые жаждут скорости и стабильности в языке. Под скоростью

здесь мы подразумеваем и скорость программ, которые вы можете создать с помощью Rust, и скорость с которой Rust позволяет вам написать их. Проверки компилятора Rust обеспечивают стабильность через добавление функций и рефакторинг. Это в корне отличается от хрупкого устаревшего кода на языках без таких проверок, который разработчики часто боятся изменить. Стремясь к абстракциям с нулевой стоимостью, компилируя высокоуровневые функции в код более низкого уровня так же быстро, как код, написанный вручную, Rust стремится сделать безопасный код также и быстрым.

Язык Rust надеется также на поддержку многих других пользователей (здесь упомянуты только несколько крупнейших заинтересованных сторон). В целом, величайшая важность Rust заключается в устраниении компромиссов, которые программисты принимали десятилетиями, обеспечивая безопасность и производительность, скорость и эргономику. Попробуйте Rust и посмотрите, работает ли это для вас.

Для кого эта книга

В этой книге предполагается, что вы уже писали код на другом языке программирования, но не делается никаких предположений о том, на каком. Мы пытались сделать материал хорошо доступным для тех, кто имеет большой опыт в программировании. Мы не тратим много времени на разговоры о том, что такое программирование или как думать об этом. Если вы новичок в программировании, вам больше подойдёт чтение книг, в которых содержится введение в программирование.

Как использовать эту книгу

В целом, эта книга предполагает, что вы читаете её последовательно от начала до конца. Более поздние главы основываются на концепциях предыдущих. Иногда более ранние главы могут не углубляться в детали темы; мы обычно возвращаемся к теме в последующих главах.

В этой книге вы найдёте два вида глав: концептуальные главы и главы проекта. В концептуальных главах вы узнаете об аспектах Rust. В главах проекта мы будем вместе строить небольшие программы, применяя то, что вы узнали. Главы 2, 12 и

20 являются главами проекта; остальные являются концептуальными главами.

В главе 1 объясняется, как установить Rust, написать программу “Hello, world!” и использовать сборщик Cargo и менеджер пакетов в одном лице. Глава 2 является практическим введением в язык Rust. В ней объясняются концепции верхнего уровня и в более поздних главах предоставляются дополнительные детали о них. Если хотите сразу погрузиться в практику, то для этого предназначена глава 2. Вначале можно даже пропустить главу 3, которая рассказывает про возможности языка аналогичные тем, что есть в других языках и перейти к главе 4, для изучения системы владения в Rust. Однако, если вы дотошный ученик, предпочитающий изучить каждую особенность до перехода к следующей, то можно пропустить главу 2 и перейти сразу к главе 3, возвращаясь к главе 2, если захочется поработать над проектом и применить полученные знания.

Глава 5 описывает структуры и методы, а глава 6 охватывает перечисления, выражения `match` и конструкции управления потоком `if let`. Вы будете использовать структуры и перечисления для создания пользовательских типов в Rust.

В главе 7 вы узнаете о системе модулей Rust и о правилах конфиденциальности для организации вашего кода и его публичного программного интерфейса - Application Programming Interface (API). В главе 8 обсуждаются некоторые общие коллекции структур данных, которые предоставляет стандартная библиотека, например, векторы, строки и HashMap. Глава 9 исследует философию и методы обработки ошибок Rust.

В главе 10 рассматриваются шаблонные типы данных, типажи и времена жизни, которые дают вам силу разрабатывать код, который может использоваться разными типами. Глава 11 посвящена тестированию, которое даже с гарантиями безопасности в Rust, необходимо для обеспечения правильной логики вашей программы. В главе 12 мы создадим собственную реализацию подмножества функциональности инструмента командной строки `grep`, предназначенного для поиска текста в файлах. Для этого мы будем использовать многие концепции, которые обсуждались в предыдущих главах.

Глава 13 исследует замыкания и итераторы: особенности Rust, которые пришли из функциональных языков программирования. В главе 14 мы подробнее рассмотрим Cargo и расскажем о лучших способах предоставления пользования вашими библиотеками другим разработчикам. Глава 15 обсуждает умные указатели, которые предоставляет стандартная библиотека и свойства, которые обеспечивают их функциональность.

В главе 16 мы рассмотрим различные модели параллельного программирования и поговорим о том, как Rust помогает вам безбоязненно программировать в нескольких потоках. Глава 17 рассказывает о том, как идиомы Rust сравниваются с принципами объектно-ориентированного программирования, с которыми вы, возможно, знакомы.

Глава 18 является справочником по шаблонам и сопоставлению с образцом, которые являются мощным способом выражения идей в программах на Rust. Глава 19 содержит обзор продвинутых тем, представляющих интерес, включая небезопасный Rust, макросы, больше о временах жизни, шаблонах, типах, функциях и замыканиях.

В главе 20 мы завершим проект, в котором мы реализуем низкоуровневый многопоточный веб-сервер!

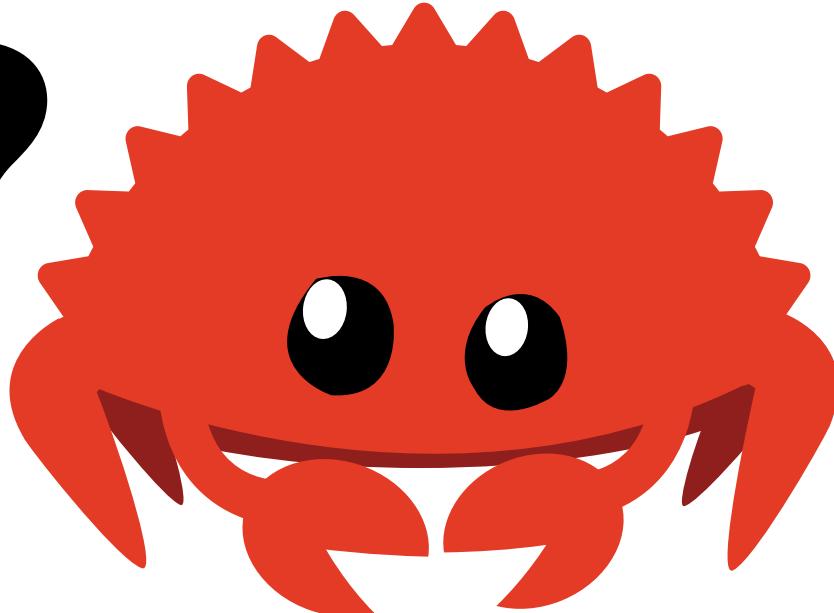
Наконец, некоторые приложения содержат полезную информацию о языке в формате, более похожем на справочник. В приложении A описаны ключевые слова Rust, в приложении B описаны операторы и символы Rust, в приложении C описаны производные свойства, предоставляемые стандартной библиотекой, в приложении D описаны некоторые полезные инструменты разработки, а в приложении E описаны редакции Rust.

Нет способа читать эту книгу неправильно: если вы хотите пропустить что-то и пройти вперёд, делайте это! Возможно, вам придётся вернуться к предыдущим главам, если у вас появятся какие-либо затруднения. Делайте так, как считаете удобным для себя.

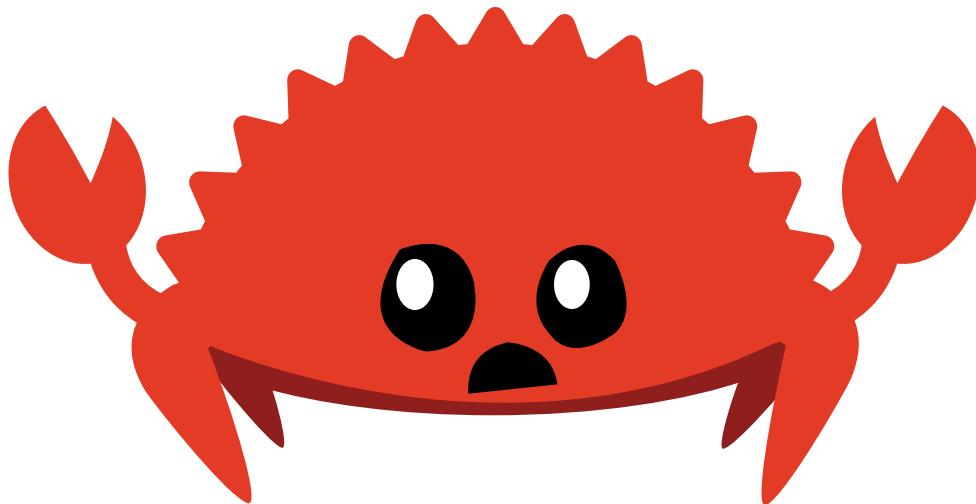
Важной частью процесса обучения Rust является изучение того, как читать сообщения об ошибках, которые отображает компилятор: они приведут вас к работающему коду. Мы изучим много примеров, которые не компилируются и отображают ошибки в сообщениях компилятора в разных ситуациях. Знайте, что если вы введёте и запустите случайный пример, он может не скомпилироваться! Убедитесь, что вы прочитали окружающий текст, чтобы понять, не предназначен ли пример, который вы пытаетесь запустить, для демонстрации ошибки. Ferris также поможет вам различить код, который не предназначен для работы:

Ferris	Пояснения

?

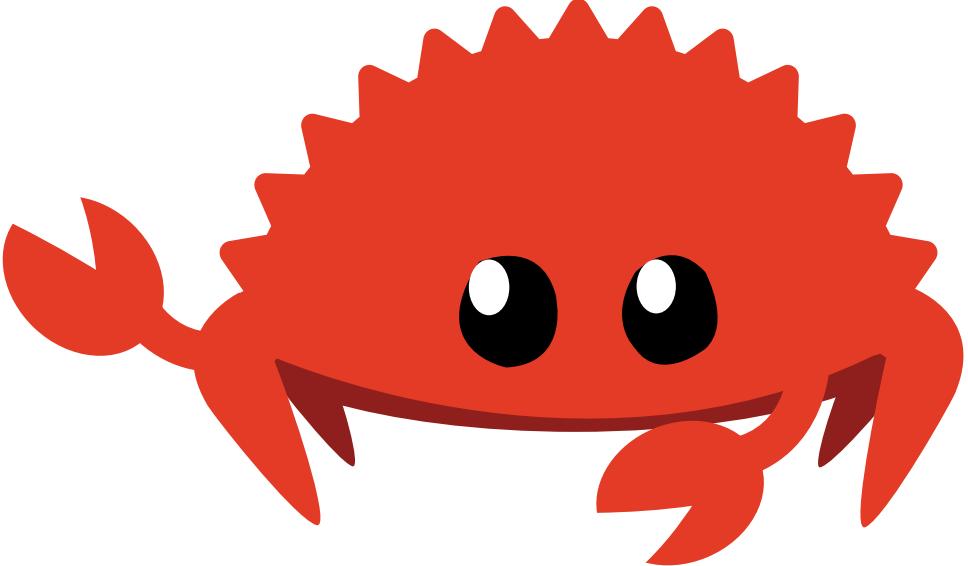


Этот код не
компилируется!



Этот код
вызывает
панику!

Этот код не
даёт

A large, stylized red crab logo with a spiky shell, wide eyes, and a smiling mouth, positioned on the left side of the page.

желаемого
поведения.

В большинстве случаев мы приведём вас к правильной версии любого кода, который не компилируется.

Исходные коды

Файлы с исходным кодом, используемым в этой книге, можно найти на [GitHub](#).

Начало работы

Начнём наше путешествие в Rust! Нужно много всего изучить, но каждое путешествие должно где-то начаться. В этой главе мы обсудим:

- установку Rust на Linux, macOS и Windows,
- написание программы, печатающей `Hello, world!`,
- использование `cargo`, менеджера пакетов и системы сборки в Rust в одном лице.

Установка

Первым шагом является установка Rust. Мы загрузим Rust, используя инструмент командной строки `rustup`, предназначенный для управления версиями Rust и другими связанными с ним инструментами. Вам понадобится интернет соединение для его загрузки.

Примечание: Если вы по каким-то причинам предпочитаете не использовать `rustup`, пожалуйста, посетите [страницу «Другие методы установки Rust»](#) для получения дополнительных опций.

Следующие шаги устанавливают последнюю стабильную версию компилятора Rust. Стабильность Rust гарантирует, что все примеры в книге, которые компилируются, будут продолжать компилироваться с более новыми версиями Rust. Вывод (данных) может немного отличаться в разных версиях, поскольку Rust часто улучшает сообщения об ошибках и предупреждениях. Другими словами, любая более новая, стабильная версия Rust, устанавливаемая с помощью этих шагов, должна работать в соответствии с содержанием этой книги.

Условные обозначения командной строки

В этой главе и на протяжении всей книги мы покажем некоторые команды, используемые в терминале командной строки. Строки, которые необходимо ввести в терминал, начинаются с `$`. Он указывает на начало каждой команды, поэтому вам не нужно вводить сам символ `$`. Строки, которые НЕ начинаются с `$`, обычно показывают вывод предыдущей команды. В дополнение, специфичные для PowerShell примеры используют символ `>` вместо символа `$`.

Установка `rustup` на Linux или macOS

Если вы используете Linux или macOS, пожалуйста, выполните следующую команду:

```
$ curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

Команда скачивает скрипт и начинает установку инструмента **rustup**, одновременно с установкой последней стабильной версии Rust. Вас могут запросить ввести локальный пароль. При успешной установке вы увидите следующий вывод:

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

Вам также понадобится компоновщик - программа, которую Rust использует для объединения своих скомпилированных выходных данных в один файл. Скорее всего, он у вас уже есть. При возникновении ошибок компоновки, вам следует установить компилятор C, который обычно будет включать в себя и компоновщик. Компилятор C также полезен, потому что некоторые распространённые пакеты Rust зависят от кода C и нуждаются в компиляторе C.

На macOS вы можете получить компилятор C, выполнив команду:

```
$ xcode-select --install
```

Пользователи Linux, как правило, должны устанавливать GCC или Clang в соответствии с документацией их дистрибутива. Например, при использовании Ubuntu можно установить пакет **build-essential**.

Установка **rustup** на Windows

На Windows перейдите по адресу <https://www.rust-lang.org/tools/install> и следуйте инструкциям по установке Rust. На определённом этапе установки вы получите сообщение с пояснением о необходимости наличия инструментов сборки C++ из Visual Studio 2013 или более поздней версии. Самый простой способ заполучить инструменты сборки - установить [Build Tools for Visual Studio 2019](#). Когда спросят, какие рабочие нагрузки нужно установить, убедитесь, что выбрано "C++ build tools", а также включены компоненты Windows 10 SDK и английский языковой пакет.

В остальной части этой книги используются команды, которые работают как в *cmd.exe*, так и в PowerShell. При наличии специфических различий мы объясним, что необходимо сделать в таких случаях.

Обновление и удаление

После установки Rust с помощью `rustup`, обновление на последние версии выполняется с помощью следующего простого скрипта командой:

```
$ rustup update
```

Чтобы удалить Rust и `rustup`, выполните следующую команду:

```
$ rustup self uninstall
```

Устранение возможных ошибок

Чтобы проверить, правильно ли у вас установлен Rust, откройте оболочку и введите эту строку:

```
$ rustc --version
```

Вы должны увидеть номер версии, хэш коммита и дату выпуска последней стабильной версии в следующем формате:

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

Если вы видите данную информацию, то вы установили всё успешно! Если вы не видите этой информации и используете Windows, проверьте, что путь к Rust находится в системной переменной `%PATH%`. Если он корректный, но Rust все ещё не работает, то есть множество мест, где можно получить помощь. Самое простое это канал `#beginners` [официального Rust Discord](#) сервера. Там вы можете пообщаться с другими Rustaceans (это наше шуточное прозвище), которые смогут вам помочь. Другие замечательные ресурсы включают [Пользовательский форум](#) и [Stack Overflow](#).

Локальная документация

Установка Rust также включает локальную копию документации, поэтому вы можете читать её в offline режиме. Выполните команду `rustup doc`, чтобы открыть локальную документацию в вашем браузере.

Каждый раз, когда есть какой-либо тип или какая-либо функция, предоставляемые

стандартной библиотекой, а вы не знаете, что они делают и как их использовать, воспользуйтесь документацией по интерфейсу прикладного программирования (API) для поиска!

Привет, мир!

Итак, когда Rust уже установлен можно приступать к написанию вашей первой программы. Общая традиция при изучении нового языка программирования - писать маленькую программу которая печатает в строке вывода "Hello, world!". Давайте сделаем тоже самое.

Обратите внимание: данная книга подразумевает, что читатель должен быть знаком с командной строкой. Однако, Rust не выдвигает каких-либо специальных требований к тому, как вы пишете и по какому принципу храните код своих программ. По этому, если вы предпочитаете использовать интегрированную среду разработки (IDE) взамен командной строки терминала, чувствуйте себя спокойно и пользуйтесь тем, чем вам удобно. Разные IDE имеют разный уровень поддержки Rust, по этому не забывайте проверять документацию к выбранной IDE. Однако, не так давно команда Rust сфокусировалась на задаче большей поддержки языка в разных IDE и однозначно сделала определённый прогресс в данном направлении!

Создание папки проекта

Прежде всего начнём с создания директории, в которой будем сохранять наш код на языке Rust. На самом деле не важно, где сохранять наш код. Однако, для упражнений и проектов, обсуждаемых в данной книге, мы советуем создать директорию *projects* в вашем домашнем каталоге, там же и хранить в будущем код программ из книги.

Откройте терминал и введите следующие команды для того, чтобы создать директорию *projects* для хранения кода разных проектов, и, внутри неё, директорию *hello_world* для проекта "Hello, world!".

Для Linux, macOS и PowerShell на Windows, введите:

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

Для Windows в CMD, введите:

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

Написание и запуск первой Rust программы

Теперь создадим новый файл, в котором сохраним исходный код программы, назовём его *main.rs*. Заметьте, что файлы с кодом на языке программирования Rust всегда заканчиваются расширением *.rs*. В случае, если вы захотите использовать более одного слова в названии файла, используйте знак подчёркивания в качестве разделителя. Например, возможно использовать именование *hello_world.rs*, но не рекомендуется использовать вариант *helloworld.rs*.

Теперь откроем файл *main.rs* для редактирования и введём следующие строки кода:

Название файла: *main.rs*

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

Листинг 1-1: Программа которая печатает `Hello, world!`

Сохраните файл и вернитесь обратно в окно терминала, чтобы скомпилировать и запустить нашу первую программу. На Linux или macOS для этого введите следующие команды:

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

В Windows, введите команду `.\main.exe` вместо `./main`:

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

Независимо от операционной системы, строка `Hello, world!` должна напечататься в окне вашего терминала. Если вы не увидели вывода, вернитесь в часть "Решение проблем" "["Troubleshooting"](#)" раздела "Установка" для получения помощи.

Если напечаталось `Hello, world!`, то примите наши поздравления! Вы написали программу на Rust, что делает вас Rust программистом — добро пожаловать!

Анатомия программы на Rust

Давайте рассмотрим в деталях, что происходит в программе “Hello, world!”. Вот первый кусок пазла:

```
fn main() {  
}
```

Эти строки определяют функцию в Rust. Функция `main` имеет специальный смысл: она всегда является первым кодом, который запускается в исполняемой программе на Rust. Первая строка объявляет функцию с именем `main` у которой нет параметров и она ничего не возвращает. Если бы тут были параметры, они были бы записаны внутри круглых скобок, `()`.

Также заметим, что тело функции обёрнуто в фигурные скобки `{}` (curly brackets). В Rust тело любой функции обрамляется в фигурные скобки. Хорошим стилем является размещение открывающей скобки в строке объявления функции, оставляя пробел между ними.

Если вы хотите придерживаться единого стиля оформления кода во всех проектах Rust, вы можете использовать инструмент автоматического форматирования `rustfmt` для оформления вашего кода в определённом стиле. Команда Rust включила этот инструмент в стандартный дистрибутив Rust, как и `rustc`, поэтому он уже должен быть установлен на вашем компьютере! Более подробную информацию можно найти в онлайн-документации.

Содержимое функции `main`:

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

Эта строка делает всю работу в этой маленькой программе: печатает текст на экран. Можно заметить четыре важных детали.

Первая, не столь заметная, - в стиле Rust для отступа используются четыре пробела, а не знак табуляции.

Во-вторых, `println!` вызывает макрос Rust. Если бы вместо этого он вызывал функцию, она вводилась бы как `println` (без `!`). Мы обсудим макросы Rust более подробно в главе 19. На данный момент вам просто нужно знать, что использование `!` означает, что вы вызываете макрос вместо обычной функции, и что макросы не всегда следуют тем же правилам, что и функции.

Третья - вы видите строку `"Hello, world!"`. Мы передаём эту строку как аргумент в макрос `println!` и, благодаря этому, строка выводится макросом на экран.

Четвёртая - в конце строки стоит точка с запятой (`;`), которая означает, что выражение закончилось и следующее выражение можно начинать опять. Большая часть строк в Rust коде заканчивается точкой с запятой.

Компиляция и выполнение кода являются отдельными шагами

Только что вы запустили вновь созданную программу, теперь изучим каждый шаг этого процесса.

Перед запуском программы, её необходимо скомпилировать компилятором Rust с помощью ввода команды `rustc` и передачи имени вашего файла с исходным кодом, например так:

```
$ rustc main.rs
```

Если вы знакомы с C или C++, то заметили, что это весьма похоже на вызов компиляции при помощи `gcc` или `clang`. После успешной компиляции, Rust выдаст двоичный исполняемый файл.

Для того, чтобы посмотреть на исполняемый файл на Linux, macOS и в PowerShell на Windows достаточно выполнить команду `ls` в командной строке. На Linux или macOS будет отображено два файла. В PowerShell на Windows, как и в Windows CMD - три файла.

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

В CMD на Windows следует ввести следующие команды:

```
> dir /B %= the /B option says to only show the file names =%
main.exe
main.pdb
main.rs
```

В обоих случаях видно: файл исходного кода с расширением `.rs`, исполняемый двоичный файл (`main.exe` в Windows, но `main` на всех других платформах), а в случае использования Windows, ещё и файл включающий отладочную информацию с расширением `.pdb`.

Отсюда мы можем запустить нашу программу (исполнимый двоичный файл) `main` или `main.exe` соответствующей командой для Windows или иных ОС:

```
$ ./main # для Linux
> .\main.exe # для Windows
```

Если `main.rs` был с текстом “Hello, world!”, то строка `Hello, world` будет напечатана в терминале.

Если вам близки динамические языки, такие как Ruby, Python или JavaScript, вы, возможно, не привыкли к тому, что компиляция и выполнение программы - это отдельные шаги. Rust - это язык с предварительной компиляцией (*ahead-of-time compiled*), что означает, что, скомпилировав программу, вы можете передать исполняемый файл другому человеку и он сможет запустить её, даже не имея установленного Rust. А если вы дадите кому-то файл `.rb`, `.py` или `.js`, то для его выполнения получателю понадобится установленный интерпретатор Ruby, Python или JavaScript (соответственно). Но в этих языках вам нужна всего одна команда для компиляции и запуска вашей программы. Что ж, всё является компромиссом в дизайне языков.

Простая компиляция с помощью `rustc` подходит только для простых программ, но по мере того, как ваши проекты становятся сложнее, вы захотите управлять всеми опциями и упростить обмен своим кодом с другими. Далее мы представим инструмент Cargo, который поможет в написании настоящих, а значит и более сложных, программ на Rust.

Привет, Cargo!

Cargo - это система сборки и менеджер пакетов Rust. Большая часть разработчиков используют данный инструмент для управления проектами, потому что Cargo выполняет за вас множество задач, таких как сборка кода, загрузка библиотек, от которых зависит ваш код, и создание этих библиотек. (Мы называем библиотеки, которые нужны вашему коду, *зависимостями*.)

Простейшие Rust программы, вроде той, что мы уже написали, не имеют зависимостей. Если бы мы собрали "Hello, world!" проект с помощью Cargo, то сборка использовала бы часть возможностей Cargo: те её функции, которые выполняют только сборку кода. По мере того, как вы будете писать более сложные программы на Rust, вы будете добавлять в них разные зависимости. Если вы начнёте проект с использованием Cargo, то добавлять зависимости будет намного проще, чем без него.

Так как большая часть проектов использует Cargo, то остальная часть книги подразумевает, что вы также используете Cargo. Cargo устанавливается вместе с Rust при использовании официальных установщиков обсуждаемых в разделе "[Установка Rust](#)". Если вы установили Rust другим способом, то проверьте, работает ли он, введя команду проверки версии Cargo в терминале:

```
$ cargo --version
```

Если команда выдала номер версии, то значит Cargo установлен. Если вы видите ошибку, вроде `command not found` ("команда не найдена"), загляните в документацию для использованного вами способа установки, чтобы выполнить установку Cargo отдельно.

Создание проекта с помощью Cargo

Давайте создадим новый проект с помощью Cargo и посмотрим, как он отличается от нашего начального проекта "Hello, world!". Перейдите обратно в папку *projects* (или любую другую, где вы решили сохранять код). Затем, в любой операционной системе, запустите команду:

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

Первая команда создаёт новую папку с названием *hello_cargo*. Мы дали название проекту *hello_cargo*, поэтому Cargo создаёт свои файлы внутри папки с этим именем.

Перейдём в каталог *hello_cargo* и посмотрим файлы. Увидим, что Cargo сгенерировал два файла и одну директорию: файл *Cargo.toml* и каталог *src* с файлом *main.rs* внутри.

Кроме того, cargo инициализировал новый репозиторий Git вместе с файлом *.gitignore*. Файлы Git не будут сгенерированы, если вы запустите **cargo new** в существующем репозитории Git; вы можете изменить это поведение, используя **cargo new --vcs=git**.

Примечание: Git - это распространённая система контроля версий. Вы можете заставить **cargo new** использовать другую систему контроля версий или вообще отказаться от неё, используя флаг **--vcs**. Запустите **cargo new --help**, чтобы увидеть доступные параметры.

Откройте файл *Cargo.toml* в любом текстовом редакторе. Он должен выглядеть как код в листинге 1-2.

Cargo.toml:

```
[package]
name = "hello_cargo"
version = "0.1.0"
edition = "2018"

[dependencies]
```

Листинг 1-2: Содержимое файла *Cargo.toml* сгенерированное командой **cargo new**

Это файл в формате **TOML** (*Tom's Obvious, Minimal Language*), который является форматом конфигураций Cargo.

Первая строка, **[package]**, является заголовочной секцией, которая указывает что следующие инструкции настраивают пакет. По мере добавления больше информации в данный файл, будет добавляться больше секций и инструкций (строк).

Следующие три строки задают информацию о конфигурации, необходимую Cargo для компиляции вашей программы: имя, версию и редакцию Rust, который будет использоваться. Мы поговорим о ключе **edition** в [Приложении E](#).

Последняя строка, **[dependencies]** является началом секции для списка любых зависимостей вашего проекта. В Rust, это внешние пакеты кода, на которые ссылаются ключевым словом *crate*. Нам не нужны никакие зависимости в данном проекте, но мы будем использовать их в первом проекте главы 2, так что нам пригодится данная секция зависимостей потом.

Откройте файл *src/main.rs* и загляните в него:

Название файла: *src/main.rs*

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

Cargo сгенерировал программу "Hello, world!", такую же как мы писали в листинге 1-1! Различиями между нашим предыдущим проектом и проектом, который генерирует Cargo является то, что Cargo помещает исходный код в каталог *src* и снабжает нас конфигурационным файлом *Cargo.toml* в корневой директории нашего проекта.

Cargo ожидает, что ваши исходные файлы находятся внутри каталога *src*. Каталог верхнего уровня проекта предназначен только для файлов README, информации о лицензии, файлы конфигурации и чего то ещё не относящего к вашему коду. Использование Cargo помогает организовывать проект. Есть место для всего и все находится на своём месте.

Если вы начали проект без использования Cargo, как мы делали для "Hello, world!" проекта, то можно конвертировать его в проект с использованием Cargo. Переместите код в подкаталог *src* и создайте соответствующий файл *Cargo.toml* в папке.

Сборка и запуск Cargo проекта

Посмотрим, в чем разница при сборке и запуске программы "Hello, world!" с помощью Cargo. В каталоге *hello_cargo* соберите проекта следующей командой:

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

Данная команда создаёт выполнимый файл в папке *target/debug/hello_cargo* (или

`target\debug\hello_cargo.exe` на Windows), а не в текущей директории проекта. Можно запустить исполняемый файл командой:

```
$ ./target/debug/hello_cargo # or .\target\debug\hello_cargo.exe on Windows  
Hello, world!
```

Если все хорошо, то `Hello, world!` печатается в терминале. Запуск команды `cargo build` в первый раз также приводит к созданию нового файла `Cargo.lock` в папке верхнего уровня. Данный файл хранит точные версии зависимостей вашего проекта. Так как у нас нет зависимостей, то файл пустой. Вы никогда не должны менять этот файл вручную: Cargo сам управляет его содержимым для вас.

Мы только что собрали проект командой `cargo build` и запустили его из `./target/debug/hello_cargo`. Но мы также можем использовать команду `cargo run` для компиляции кода и затем его запуска одной командой:

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

Заметьте, что в этот раз вы не увидели вывода о том, что Cargo компилировал `hello_cargo`. Cargo понял, что файлы не менялись, поэтому он просто запустил уже имеющийся бинарный файл. Если бы вы модифицировали исходный код, то Cargo собрал бы проект заново перед его запуском как вы уже видели в выводе:

```
$ 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!
```

Также Cargo предоставляет команду `cargo check`. Данная команда быстро проверяет ваш код, чтобы убедиться, что он компилируется, но не создаёт исполняемого файла:

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

В каком случае не нужно создавать исполняемый файл? Часто команда `cargo check` является более быстрой по сравнению с `cargo build`, потому что она пропускает

шаг создания исполняемого файла. В случае, если вы постоянно проверяете работу во время написания кода с помощью `cargo check`, то это ускоряет процесс! Таким образом многие разработчики запускают `cargo check` периодически, по мере того, как пишут программу, чтобы убедится, что она компилируется. А запускают команду `cargo build`, когда готовы создать исполняемый файл.

Повторим полученные знания про Cargo:

- можно собирать проект, используя команду `cargo build`,
- можно одновременно собирать и запускать проект одной командой `cargo run`,
- можно собрать проект для проверки ошибок с помощью `cargo check`, не тратя время на кодогенерацию исполняемого файла,
- `cargo` сохраняет результаты сборки не в директорию с исходным кодом, а в отдельный каталог `target/debug`.

Дополнительным преимуществом использования Cargo является то, что его команды одинаковы для разных операционных систем. С этой точки зрения, мы больше не будем предоставлять отдельные инструкции для Linux, macOS или Windows.

Сборка финальной версии (Release)

Когда проект, наконец, готов к релизу, можно использовать команду `cargo build --release` для его компиляции с оптимизацией. Данная команда создаёт исполняемый файл в папке `target/release` в отличии от папки `target/debug`. Оптимизации делают так, что Rust код работает быстрее, но их включение увеличивает время компиляции. По этой причине есть два отдельных профиля: один для разработки, когда нужно осуществлять сборку быстро и часто, и другой, для сборки финальной программы, которую будете отдавать пользователям, которая готова к работе и будет выполняться максимально быстро. Если вы замеряете время выполнения вашего кода, убедитесь, что собрали проект с оптимизацией `cargo build --release` и тестируете исполняемый файл из папки `target/release`.

Cargo как конвенция

Для простых проектов Cargo не даёт большой пользы по сравнению с

использованием `rustc`, но он докажет свою пользу как только ваши программы станут более запутанными. С помощью Cargo гораздо проще координировать сборку на сложных проектах, скомбинированных из множества внешних библиотек (crates).

Не смотря на то, что проект `hello_cargo` простой, теперь он использует большую часть реального инструментария, который вы будете повседневно использовать в вашей карьере, связанной с Rust. Когда потребуется работать над проектами размещёнными в сети, вы сможете просто использовать следующую последовательность команд для получения кода с помощью Git, перехода в каталог проекта, сборку проекта:

```
$ git clone example.org/someproject  
$ cd someproject  
$ cargo build
```

Для более детальной информации про Cargo, загляните в [его документацию](#).

Итоги

Теперь вы готовы начать своё Rust путешествие! В данной главе вы изучили как:

- установить последнюю стабильную версию Rust, используя `rustup`,
- обновить Rust до последней версии,
- открыть локально установленную документацию,
- написать и запустить программу типа "Hello, world!", используя напрямую компилятор `rustc`,
- создать и запустить новый проект, используя соглашения и команды Cargo.

Пришло время для создания более содержательной программы, чтобы привыкнуть к чтению и написанию кода на Rust. В главе 2 мы создадим программу для угадывания числа. Если вы хотите начать с изучения общих концепций программирования в Rust, загляните в главу 3, а затем вернитесь к главе 2.

Программируем игру Угадайка

Давайте погрузимся в Rust, вместе выполнив практический проект! Эта глава познакомит с несколькими распространёнными концепциями Rust, показав, как использовать их в реальной программе. Вы узнаете о `let`, `match`, методах, ассоциированных функциях, использовании внешних пакетов и многом другом! В следующих главах рассмотрим эти идеи более подробно. В этой главе вы на практике познакомитесь с основами.

Мы реализуем классическую для начинающих программистов задачу: игру в угадывание. Вот как это работает: программа генерирует случайное целое число в диапазоне от 1 до 100. Затем она предлагает игроку ввести отгадку. После ввода отгадки программа укажет, является ли отгадка слишком заниженной или слишком завышенной. Если отгадка верна, игра напечатает поздравительное сообщение и завершится.

Настройка нового проекта

Для настройки нового проекта перейдите в каталог *projects*, который вы создали в главе 1, и создайте новый проект с использованием Cargo, как показано ниже:

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

Первая команда, `cargo new`, принимает в качестве первого аргумента имя проекта (`guessing_game`). Вторая команда изменяет каталог на новый каталог проекта.

Загляните в созданный файл *Cargo.toml*:

Имя файла: *Cargo.toml*

```
[package]  
name = "guessing_game"  
version = "0.1.0"  
edition = "2021"  
  
# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html  
  
[dependencies]
```

Как вы уже видели в Главе 1, `cargo new` создаёт программу "Hello, world!". Посмотрите файл `src/main.rs`:

Файл: `src/main.rs`

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

Теперь давайте скомпилируем программу "Hello, world!" и сразу на этом же этапе запустим её с помощью команды `cargo run`:

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

Команда `run` пригодится, когда необходимо ускоренно выполнить итерацию проекта, мы так собираемся сделать в этом проекте, быстро тестируя каждую итерацию, прежде чем перейти к следующей.

Снова откройте файл `src/main.rs`. Весь код вы будете писать в этом файле.

Обработка отгадки

Первая часть программы игры угадывания запрашивает ввод данных пользователем, обрабатывает их и проверяет, что вводимые данные имеют ожидаемую форму. Для начала мы позволим игроку ввести отгадку. Введите код из Листинга 2-1 в `src/main.rs`.

Имя файла: `src/main.rs`

```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

Листинг 2-1: Код, который получает отгадку от пользователя и печатает её

Этот код содержит много информации, поэтому давайте рассмотрим его построчно. Чтобы получить пользовательский ввод и затем вывести результат в качестве вывода, нам нужно включить в область видимости библиотеку ввода/вывода `io`. Библиотека `io` является частью стандартной библиотеки, известной как `std`:

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

По умолчанию Rust имеет несколько элементов, заданных в стандартной библиотеке, которые он включает в область видимости каждой программы. Этот набор называется *прелюдией (prelude)*, и [в документации по стандартной библиотеке](#) можно увидеть все входящее в её состав элементы.

Если тип, который требуется использовать, отсутствует в прелюдии, его нужно явно ввести в область видимости с помощью оператора `use`. Использование библиотеки `std::io` предоставляет ряд полезных функциональных возможностей, включая способность принимать пользовательский ввод.

Как уже отмечалось 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 `{` обозначает начало тела функции.

Также в главе 1 упоминалось, что `println!` - это макрос, который печатает строку на экран:

```
println!("Guess the number!");  
println!("Please input your guess.");
```

Этот код печатает подсказку об игре и запрашивает пользовательский ввод.

Хранение значений с помощью переменных

Далее мы создаём *переменную* для хранения пользовательского ввода, как показано ниже:

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

Вот теперь программа становится интересней! Очень многое происходит в этой маленькой строке. Для создания переменной мы используем оператор `let`. Вот ещё один пример:

```
let apples = 5;
```

Эта строка создаёт новую переменную с именем `apples` и связывает её со значением 5. В Rust переменные неизменяемые по умолчанию. Мы подробно обсудим эту концепцию в разделе "[Переменные и изменяемость](#)" в главе 3. Чтобы сделать переменную изменяемой, мы добавляем `mut` перед именем переменной:

```
let apples = 5; // неизменяемая  
let mut bananas = 5; // изменяемая
```

Примечание: Синтаксис `//` означает начало комментария, который продолжается до конца строки. Rust игнорирует все содержимое комментариев. Подробнее о комментариях мы поговорим в [главе 3](#).

Возвращаясь к программе игры Угадайка, теперь вы знаете, что `let mut guess` предоставит изменяемую переменную с именем `guess`. Знак равенства (`=`) сообщает Rust, что сейчас нужно связать что-то с этой переменной. Справа от знака равенства находится значение, связанное с `guess`, которое является результатом вызова функции `String::new`, возвращающей новый экземпляр `String`. `String` - это тип строки, предоставляемый стандартной библиотекой, который является расширяемым фрагментом текста в кодировке UTF-8.

Синтаксис `::` в строке `::new` указывает, что `new` является ассоциированной функцией `String` типа. Ассоциированная функция - это функция, реализованная для типа, в данном случае `String`. Функция `new` создаёт новую, пустую строку. Функцию `new` можно встретить во многих типах, это типичное название для функции, которая создаёт новое значение какого-либо типа.

В целом, строка `let mut guess = String::new();` создала изменяемую переменную, которая связывается с новым, пустым экземпляром `String`. Фух!

Получение пользовательского ввода

Напомним, мы подключили функциональность ввода/вывода из стандартной библиотеки с помощью `use std::io;` в первой строке программы. Теперь мы вызовем функцию `stdin` из модуля `io`, которая позволит нам обрабатывать пользовательский ввод:

```
io::stdin()
    .read_line(&mut guess)
```

Если бы мы не импортировали библиотеку `io` с помощью `use std::io` в начале программы, мы все равно могли бы использовать эту функцию, записав вызов этой функции как `std::io::stdin`. Функция `stdin` возвращает экземпляр `std::io::Stdin`, который является типом, представляющим дескриптор стандартного ввода для вашего терминала.

Далее строка `.read_line(&mut guess)` вызывает метод `read_line` на дескрипторе стандартного ввода для получения ввода от пользователя. Мы также передаём `&mut guess` в качестве аргумента `read_line`, сообщая ему, в какой строке хранить пользовательский ввод. Главная задача `read_line` - принять все, что пользователь вводит в стандартный ввод, и сложить это в строку (не переписывая её содержимое), поэтому мы передаём эту строку в качестве аргумента. Строковый аргумент должен быть изменяемым, чтобы метод мог изменить содержимое строки.

Символ `&` указывает, что этот аргумент является *ссылкой*, который предоставляет возможность нескольким частям вашего кода получить доступ к одному фрагменту данных без необходимости копировать эти данные в память несколько раз. Ссылки - это сложная функциональная возможность, а одним из главных преимуществ Rust является безопасность и простота использования ссылок. Чтобы дописать эту программу, вам не понадобится знать много таких подробностей. Пока вам

достаточно знать, что ссылки, как и переменные, по умолчанию неизменяемы. Соответственно, чтобы сделать её изменяемой, нужно написать `&mut guess`, а не `&guess`. (В главе 4 ссылки будут описаны более подробно).

Обработка потенциального сбоя с помощью типа `Result`

Мы все ещё работаем над этой строкой кода. Хотя сейчас мы обсуждаем третью строку текста, это все ещё часть одной логической строки кода. Следующей частью является этот метод:

```
.expect("Failed to read line");
```

Мы могли бы написать этот код так:

```
io::stdin().read_line(&mut guess).expect("Failed to read line");
```

Однако одну длинную строку трудно читать, поэтому лучше разделить её. При вызове метода с помощью синтаксиса `.method_name()` часто целесообразно вводить новую строку и другие пробельные символы, чтобы разбить длинные строки. Теперь давайте обсудим, что делает эта строка.

Как упоминалось ранее, `read_line` помещает все введённые пользователем данные в переданную ему строку, а также возвращает значение - в данном случае `io::Result`. В стандартной библиотеке Rust есть несколько типов с именем `Result`: общий `Result`, а также специальные версии для подмодулей, например `io::Result`. Типы `Result` - это *перечисления*, часто называемые *enum*s, имеющие фиксированный набор возможностей, известных как *варианты*. Перечисления часто используются с `match`, условием, позволяющим выполнять различный код в зависимости от того, какой вариант значения перечисления используется при оценке условия.

В главе 6 перечисления будут рассмотрены более подробно. Назначение всех типов `Result` заключается в передаче информации для обработки ошибок.

Вариантами `Result` являются `Ok` и `Err`. Вариант `Ok` указывает, что операция завершилась успешно, а внутри `Ok` находится успешно сгенерированное значение. Вариант `Err` означает, что операция не удалась, а `Err` содержит информацию о причинах неудачи.

Значения типа `Result`, как и значения других типов, имеют заданные для них методы. Экземпляр `io::Result` имеет метод `expect`, который можно вызвать. Если данный экземпляр `io::Result` является значением `Err`, `expect` приведёт к аварийному завершению программы и отображению сообщения, которое вы передали в качестве аргумента в `expect`. Если метод `read_line` возвращает `Err`, то скорее всего это результат ошибки, исходящей от основной операционной системы. Если данный экземпляр `io::Result` является значением `Ok`, `expect` возьмёт возвращаемое значение, содержащееся в `Ok`, и вернёт только это значение, чтобы его можно было использовать. В данном случае это значение - количество байтов в пользовательском вводе.

Если не вызвать `expect`, программа скомпилируется, но будет получено предупреждение:

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

warning: `guessing_game` (bin "guessing_game") generated 1 warning
Finished dev [unoptimized + debuginfo] target(s) in 0.59s
```

Rust предупреждает о не использовании значения `Result`, возвращаемого из `read_line`, показывая, что программа не учла возможность возникновения ошибки.

Правильный способ убрать предупреждение - это написать обработку ошибок, но в нашем случае мы просто хотим аварийно завершить программу при возникновении проблемы, поэтому используем `expect`. О способах восстановления после ошибок вы узнаете в [главе 9](#).

Напечатать значений с помощью заполнителей `println!`

Кроме закрывающей фигурной скобки, в коде на данный момент есть ещё только одна строка для обсуждения:

```
println!("You guessed: {guess}");
```

Эта строка печатает строку, которая теперь содержит ввод пользователя. Набор фигурных скобок `{}` является заполнителем: думайте о `{}` как о маленьких крабовых клешнях,держивающих значение на месте. С помощью фигурных скобок можно вывести более одного значения: первый набор фигурных скобок содержит первое значение, указанное после форматирующей строки, второй набор - второе значение и так далее. Печать нескольких значений за один вызов `println!` будет выглядеть следующим образом:

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

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

Этот код напечатает `x = 5 and y = 10`.

Тестирование первой части

Давайте протестируем первую часть игры. Запустите её используя `cargo run`:

```
$ cargo run
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 6.44s
    Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
6
You guessed: 6
```

На данном этапе первая часть игры завершена: мы получаем ввод с клавиатуры и затем печатаем его.

Генерация секретного числа

Далее нам нужно сгенерировать секретное число, которое пользователь попытается угадать. Секретное число должно быть каждый разенным, чтобы в игру можно было играть несколько раз. Мы будем использовать случайное число в

диапазоне от 1 до 100, чтобы игра не была слишком сложной. Rust пока не включает функциональность случайных чисел в свою стандартную библиотеку. Однако команда Rust предоставляет `rand` crate с подобной функциональностью.

Использование пакета для получения дополнительной функциональности

Помните, что пакет (crate) - это коллекция файлов исходного кода Rust. Проект, создаваемый нами, представляет собой *бинарный пакет* (*binary crate*), который является исполняемым файлом. Пакет `rand` - это *библиотечный пакет* (*library crate*), содержащий код, который предназначен для использования в других программах и поэтому не может исполняться сам по себе.

Координация работы внешних пакетов является тем местом, где Cargo действительно блистает. Чтобы начать писать код, использующий `rand`, необходимо изменить файл *Cargo.toml*, включив в него в качестве зависимости пакет `rand`. Итак, откройте этот файл и добавьте следующую строку внизу под заголовком секции `[dependencies]`, созданным для вас Cargo. Обязательно укажите `rand` в точности как здесь, с таким же номером версии, иначе примеры кода из этого урока могут не заработать.

Имя файла: *Cargo.toml*

```
rand = "0.8.3"
```

В файле *Cargo.toml* все, что следует за заголовком, является частью этой секции, которая продолжается до тех пор, пока не начнётся другая секция. В `[dependencies]` вы сообщаете Cargo, от каких внешних пакетов зависит ваш проект и какие версии этих пакетов вам нужны. В нашем случае мы указываем пакет `rand` с семантическим спецификатором версии `0.8.3`. Cargo понимает [Семантическое Версионирование \(Semantic Versioning\)](#) (иногда называемое *SemVer*), которое является стандартом для написания номеров версий. Число `0.8.3` на самом деле является сокращением для `^0.8.3`, что означает любую версию, которая не ниже `0.8.3`, но не выше `0.9.0`. Cargo полагает, что эти версии имеют публичные API, совместимые с версией `0.8.3`, и эта спецификация гарантирует, что вы получите последний выпуск патча, который все ещё будет компилироваться с кодом из этой главы. Любая версия `0.9.0` или выше не гарантирует наличие того же API, что используется в следующих примерах.

Теперь, не меняя ничего в коде, давайте соберём проект, как показано в листинге 2-2.

```
$ cargo build
  Updating crates.io index
Downloaded rand v0.8.3
Downloaded libc v0.2.86
Downloaded getrandom v0.2.2
Downloaded cfg-if v1.0.0
Downloaded ppv-lite86 v0.2.10
Downloaded rand_chacha v0.3.0
Downloaded rand_core v0.6.2
  Compiling rand_core v0.6.2
  Compiling libc v0.2.86
  Compiling getrandom v0.2.2
  Compiling cfg-if v1.0.0
  Compiling ppv-lite86 v0.2.10
  Compiling rand_chacha v0.3.0
  Compiling rand v0.8.3
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53s
```

Листинг 2-2: Результат выполнения `cargo build` после добавления пакета `rand` в качестве зависимости

Вы можете увидеть другие номера версий (но все они будут совместимы с кодом, благодаря SemVer!), другие строки (в зависимости от операционной системы), а также строки могут быть расположены в другом порядке.

Когда мы включаем внешнюю зависимость, Cargo берет последние версии всего, что нужно этой зависимости, из *реестра* (*registry*), который является копией данных с [Crates.io](#). Crates.io - это место, где участники экосистемы Rust размещают свои проекты Rust с открытым исходным кодом для использования другими.

После обновления реестра Cargo проверяет раздел `[dependencies]` и загружает все указанные в списке пакеты, которые ещё не были загружены. В нашем случае, хотя мы указали только `rand` в качестве зависимости, Cargo также захватил другие пакеты, от которых зависит работа `rand`. После загрузки пакетов Rust компилирует их, а затем компилирует проект с имеющимися зависимостями.

Если сразу же запустить `cargo build` снова, не внося никаких изменений, то кроме строки `Finished` вы не получите никакого вывода. Cargo знает, что он уже загрузил и скомпилировал зависимости, и вы не вносили никаких изменений в файл `Cargo.toml`. Cargo также знает, что вы ничего не изменили в своём коде, поэтому он

не перекомпилирует и его. Если делать нечего, он просто завершает работу.

Если открыть файл `src/main.rs`, внести незначительные изменения, а затем сохранить его и снова произвести сборку, то вы увидите только две строки вывода:

```
$ cargo build
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
```

Эти строки показывают, что Cargo обновляет сборку только на основании вашего крошечного изменения в файле `src/main.rs`. Поскольку зависимости не изменились, Cargo знает, что может повторно использовать ранее загруженные и скомпилированные зависимости.

Обеспечение воспроизводимых сборок с помощью файла `Cargo.lock`

В Cargo есть механизм, обеспечивающий возможность пересобрать все тот же артефакт каждый раз, когда вы или кто-либо другой собирает ваш код. Пока вы не укажете обратное, Cargo будет использовать только те версии зависимостей, которые были заданы ранее. Например, допустим, что на следующей неделе выходит версия 0.8.4 пакета `rand`, и эта версия содержит важное исправление ошибки, но также содержит регрессию, которая может сломать ваш код. Чтобы справиться с этим, Rust создаёт файл `Cargo.lock` при первом запуске `cargo build`, поэтому теперь он есть в каталоге `guessing_game`.

Когда вы собираете проект в первый раз, Cargo определяет все версии зависимостей, удовлетворяющие критериям, а затем записывает их в файл `Cargo.lock`. Когда вы будете собирать свой проект в будущем, Cargo увидит, что файл `Cargo.lock` уже существует, и будет использовать указанные в нем версии вместо повторения работы по определению версий. Это даёт возможность автоматически получить воспроизводимую сборку. Другими словами, благодаря файлу `Cargo.lock` ваш проект останется на версии `0.8.3`, пока вы явно не обновитесь.

Обновление пакета для получения новой версии

Если вы захотите обновить пакет, Cargo предоставляет команду `update`, которая игнорирует файл `Cargo.lock` и определяет последние версии, соответствующие вашим спецификациям из файла `Cargo.toml`. После этого Cargo запишет эти версии в файл `Cargo.lock`. Иначе, по умолчанию, Cargo будет искать только версии больше `0.8.3`, но при этом меньше `0.9.0`. Если пакет `rand` имеет две новые версии

0.8.4 и 0.9.0, то при запуске `cargo update` вы увидите следующее:

```
$ cargo update
Updating crates.io index
Updating rand v0.8.3 -> v0.8.4
```

Cargo игнорирует релиз 0.9.0. В этот момент также появится изменение в файле `Cargo.lock`, указывающее на то, что версия `rand`, которая теперь используется, равна 0.8.4. Чтобы использовать `rand` версии 0.9.0 или любой другой версии из серии 0.9.x, необходимо обновить файл `Cargo.toml` следующим образом:

```
[dependencies]
rand = "0.9.0"
```

В следующий раз, при запуске `cargo build`, Cargo обновит реестр доступных пакетов и пересмотрит ваши требования к `rand` в соответствии с новой версией, которую вы указали.

Много ещё можно рассказать о [Cargo и его экосистеме](#), о которой мы поговорим в главе 14, но сейчас это все, что следует знать. Cargo упрощает повторное использование библиотек, поэтому Rustaceans могут писать небольшие проекты, собранные из нескольких пакетов (packages).

Генерация случайного числа

Давайте начнём использовать `rand` чтобы сгенерировать число для угадывания. Следующим шагом будет обновление `src/main.rs`, как показано в Листинге 2-3.

Имя файла: `src/main.rs`

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

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1..=100);

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

Листинг 2-3: Добавление кода для генерации случайного числа

Сначала мы добавляем строку `use rand::Rng`. Типаж `Rng` определяет методы, реализующие генераторы случайных чисел, и этот типаж должен быть в области видимости, чтобы можно было использовать эти методы. В главе 10 мы подробно рассмотрим типажи.

Далее мы добавляем две строки посередине. В первой строке вызов функции `rand::thread_rng`, предоставляющей нам специальный генератор случайных чисел, который мы собираемся использовать: локальный для текущего потока выполнения и заполняемый операционной системой. Затем вызываем метод `gen_range` на генераторе случайных чисел. Этот метод определяется типажом `Rng`, который мы ввели в область видимости с помощью оператора `use rand::Rng`. Метод `gen_range` принимает выражение диапазона в качестве аргумента и генерирует случайное число в пределах диапазона. Выражение диапазона, которое здесь используется, имеет форму `start...end` и является инклузивным по нижней границе, но эксклюзивным по верхней, поэтому нужно указать `1...101`, чтобы запросить число от 1 до 100. Как вариант, можно передать диапазон `1..=100`, что будет эквивалентно.

Примечание: Не просто сразу разобраться, какие типажи использовать, какие методы и функции вызывать из пакета, поэтому каждый пакет имеет

документацию с инструкциями по его использованию. Ещё одной замечательной особенностью Cargo является выполнение команды `cargo doc --open`, которая локально собирает документацию, предоставляемую всеми вашими зависимостями, и открывает её в браузере. К примеру, если интересна другая функциональность из пакета `rand`, запустите `cargo doc --open` и нажмите `rand` в боковой панели слева.

Во второй новой строке печатается секретный номер. Полезно, пока разрабатывается программа, иметь возможность тестировать её, но в финальной версии мы это удалим. Конечно это не похоже на игру, если программа печатает ответ сразу после запуска!

Попробуйте запустить программу несколько раз:

```
$ cargo run
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 2.53s
    Running `target/debug/guessing_game`
Guess the number!
The secret number is: 7
Please input your guess.
4
You guessed: 4

$ cargo run
  Finished dev [unoptimized + debuginfo] target(s) in 0.02s
    Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
```

Вы должны получить разные случайные числа, и все они должны быть числами в диапазоне от 1 до 100. Отличная работа!

Сравнение догадки с секретным числом

Теперь, когда у нас есть пользовательский ввод и случайное число, мы можем сравнить их. Этот шаг показан в листинге 2-4. Учтите, что этот код ещё не скомпилируется, подробнее мы объясним дальше.

Имя файла: src/main.rs

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

fn main() {
    // --snip--

    println!("You guessed: {guess}");

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => println!("You win!"),
    }
}
```



Листинг 2-4: Обработка возможных возвращаемых значений при сравнении двух чисел

Сначала добавим ещё один оператор `use`, который вводит тип с именем `std::cmp::Ordering` в область видимости из стандартной библиотеки. Тип `Ordering` является ещё одним перечислением и имеет варианты `Less`, `Greater` и `Equal`. Это три возможных исхода, при сравнении двух величин.

После чего ниже добавляем пять новых строк, использующих тип `Ordering`. Метод `cmp` сравнивает два значения и может вызываться для всего, что можно сравнить. Он принимает ссылку на все, что требуется сравнить: здесь сравнивается `guess` с `secret_number`. В результате возвращается вариант перечисления `Ordering`, которое мы ввели в область видимости с помощью оператора `use`. Для принятия решения о том, что делать дальше, мы используем выражение `match`, определяющее, какой вариант `Ordering` был возвращён из вызова `cmp` со значениями `guess` и `secret_number`.

Выражение `match` состоит из *веток* (*arms*). Ветка состоит из шаблона для сопоставления и кода, который будет запущен, если значение, переданное в `match`, соответствует шаблону этой ветки. Rust принимает значение, заданное `match`, и по очереди просматривает шаблон каждой ветки. Шаблоны и конструкция `match` - это мощные возможности Rust, позволяющие выразить множество ситуаций, с которыми может столкнуться ваш код, и гарантировать их обработку. Эти возможности будут подробно раскрыты в Главе 6 и Главе 18 соответственно.

Давайте рассмотрим пример с выражением `match`, который используется здесь.

Допустим, пользователь предположил число 50, а случайно сгенерированное секретное число в этот раз равно 38. Когда код сравнил 50 с 38, метод `cmp` вернёт `Ordering::Greater`, потому что 50 больше 38. Выражение `match` получит значение `Ordering::Greater` и начнёт проверку шаблона каждой ветки. Сначала он посмотрит на шаблон первой ветки, `Ordering::Less`, но увидит, что значение `Ordering::Greater` не сопоставляется с `Ordering::Less`, поэтому код в этой ветке будет проигнорирован, и перейдёт к следующей ветке. Шаблон следующей ветки - `Ordering::Greater`, который успешно *сопоставляется* с `Ordering::Greater`! Связанный код из этой ветки будет выполнен и распечатает `Too big!` на экране. Так как теперь не требуется рассматривать последнюю ветку в этом сценарии, выражение `match` прекратит выполнение.

Однако, код в листинге 2-4 все ещё не скомпилируется. Давайте попробуем:

```
$ cargo build
Compiling libc v0.2.86
Compiling getrandom v0.2.2
Compiling cfg-if v1.0.0
Compiling ppv-lite86 v0.2.10
Compiling rand_core v0.6.2
Compiling rand_chacha v0.3.0
Compiling rand v0.8.3
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
error[E0308]: mismatched types
--> src/main.rs:22:21
22 |     match guess.cmp(&secret_number) {
|             ^^^^^^^^^^^^^^ expected struct `String`, found
integer
| = note: expected reference `&String`
|           found reference `&{integer}`

error[E0283]: type annotations needed for `'{integer}`
--> src/main.rs:8:44
8  |     let secret_number = rand::thread_rng().gen_range(1..=100);
|             -----^ expected type
for type `'{integer}`
|             |
|             consider giving `secret_number` a type
|
| = note: multiple `impl`s satisfying `'{integer}: SampleUniform` found in
the `rand` crate:
|   - impl SampleUniform for i128;
|   - impl SampleUniform for i16;
|   - impl SampleUniform for i32;
|   - impl SampleUniform for i64;
|   and 8 more
note: required by a bound in `gen_range`
--> /Users/carolnichols/.cargo/registry/src/github.com-
1ecc6299db9ec823/rand-0.8.3/src/rng.rs:129:12
129 |     T: SampleUniform,
|             ^^^^^^^^^^ required by this bound in `gen_range`
help: consider specifying the type arguments in the function call
|
8  |     let secret_number = rand::thread_rng().gen_range::<T, R>(1..=100);
|             ++++++++
|
Some errors have detailed explanations: E0283, E0308.
For more information about an error, try `rustc --explain E0283`.
error: could not compile `guessing_game` due to 2 previous errors
```

Суть ошибки заключается в наличии *несовпадающих типов*. У Rust строгая, статическая система типов. Однако он также имеет вывод типов. Когда мы написали `let mut guess = String::new()`, Rust смог сделать вывод, что `guess` должна быть `String` и не заставил указывать тип. С другой стороны, `secret_number` - это числовой тип. Несколько типов чисел в Rust могут иметь значение от 1 до 100: `i32`, 32-битное число; `u32`, беззнаковое 32-битное число; `i64`, 64-битное число, а также другие. Если не указано иное, Rust по умолчанию использует `i32`, который будет типом `secret_number`, если не добавлять информацию о типе в другом месте, которая заставит Rust вывести другой числовой тип. Причина ошибки заключается в том, что Rust не может сравнить строку и числовой тип.

В конечном итоге, необходимо преобразовать `String`, считываемую программой в качестве входных данных, в реальный числовой тип, чтобы иметь возможность числового сравнения с секретным числом. Для этого добавьте эту строку в тело функции `main`:

Имя файла: src/main.rs

```
// --snip--  
  
let mut guess = String::new();  
  
io::stdin()  
    .read_line(&mut guess)  
    .expect("Failed to read line");  
  
let guess: u32 = guess.trim().parse().expect("Please type a number!");  
  
println!("You guessed: {}", guess);  
  
match guess.cmp(&secret_number) {  
    Ordering::Less => println!("Too small!"),  
    Ordering::Greater => println!("Too big!"),  
    Ordering::Equal => println!("You win!"),  
}
```

Бот эта строка:

```
let guess: u32 = guess.trim().parse().expect("Please type a number!");
```

Мы создаём переменную с именем `guess`. Но подождите, разве в программе уже нет переменной с этим именем `guess`? Так и есть, но Rust позволяет нам *затенять* предыдущее значение `guess` новым. Затенение позволяет нам повторно

использовать имя переменной `guess`, чтобы избежать создания двух уникальных переменных, таких как `guess_str` и `guess`, например. Мы рассмотрим это более подробно в главе 3, а пока знайте, что эта функция часто используется, когда необходимо преобразовать значение из одного типа в другой.

Мы связываем эту новую переменную с выражением `guess.trim().parse()`. Переменная `guess` в этом выражении относится к исходной переменной `guess`, которая содержала входные данные в виде строки. Метод `trim` на экземпляре `String` удалит любые пробельные символы в начале и конце строки для того, чтобы мы могли сопоставить строку с `u32`, которая содержит только числовые данные. Пользователь должен нажать `enter`, чтобы выполнить `read_line` и ввести свою догадку, при этом в строку добавится символ новой строки. Например, если пользователь набирает 5 и нажимает `enter`, `guess` будет выглядеть так: `5\n`. Символ `\n` означает "новая строка". (В Windows нажатие `enter` сопровождается возвратом каретки и новой строкой, `\r\n`). Метод `trim` убирает `\n` или `\r\n`, оставляя только `5`.

Метод строк `parse` преобразует строку в некоторое число. Поскольку этот метод может преобразовывать различные числовые типы, мы должны сообщить Rust конкретный числовой тип, который нам нужен, используя `let guess: u32`. Двоеточие (`:`) после `guess` сообщает Rust, что мы будем аннотировать тип переменной. В Rust есть несколько встроенных числовых типов. Представленный здесь тип `u32` - это беззнаковое 32-битное целое число. Это хороший выбор по умолчанию для небольшого положительного числа. О других числовых типах вы узнаете в Главе 3. Кроме того, аннотация `u32` в этом примере программы и сравнение с `secret_number` позволяет Rust сделать вывод, что `secret_number` также должен быть `u32`. Таким образом, теперь сравнение будет проводиться между двумя значениями одного типа!

Метод `parse` будет работать только с символами, которые логически могут быть преобразованы в числа, и поэтому легко может вызвать ошибки. Если, например, строка содержит `A %`, преобразовать её в число невозможно. Так как метод `parse` может потерпеть неудачу, возвращается тип `Result`, так же как и метод `read_line` (обсуждалось ранее в разделе "Обработка потенциальной неудачи с помощью `Result` Type"). Мы будем точно так же обрабатывать данный `Result`, вновь используя метод `expect`. Если `parse` вернёт вариант `Result Err`, так как не смог создать число из строки, вызов `expect` аварийно завершит игру и распечатает переданное ему сообщение. Если `parse` сможет успешно преобразовать строку в число, он вернёт вариант `Result Ok`, а `expect` вернёт число, полученное из

значения **Ok**.

Давайте запустим программу теперь!

```
$ cargo run
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 0.43s
    Running `target/debug/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
76
You guessed: 76
Too big!
```

Хорошо! Несмотря на то, что были добавлены пробелы перед догадкой 76, программа все равно вывела пользовательскую догадку 76. Запустите программу несколько раз, чтобы проверить разное поведение при различных типах ввода: задайте число правильно, задайте слишком большое число и задайте слишком маленькое число.

Сейчас у нас работает большая часть игры, но пользователь может сделать только одну догадку. Давайте изменим это, добавив цикл!

Возможность нескольких догадок с помощью циклов

Ключевое слово **loop** создаёт бесконечный цикл. Мы добавляем цикл, чтобы дать пользователям больше шансов угадать число:

Имя файла: src/main.rs

```
// --snip--  
  
println!("The secret number is: {secret_number}");  
  
loop {  
    println!("Please input your guess.");  
  
    // --snip--  
  
    match guess.cmp(&secret_number) {  
        Ordering::Less => println!("Too small!"),  
        Ordering::Greater => println!("Too big!"),  
        Ordering::Equal => println!("You win!"),  
    }  
}  
}
```

Как видите, мы переместили все, начиная с подсказки ввода догадки, в цикл. Не забудьте добавить ещё по четыре пробела на отступы строк внутри цикла и запустите программу снова. Теперь программа будет бесконечно запрашивать ещё одну догадку, что фактически создаёт новую проблему. Похоже пользователь не сможет выйти из игры!

Пользователь может прервать выполнение программы с помощью сочетания клавиш `ctrl-c`. Но есть и другой способ спасти от этого ненасытного монстра, о котором говорилось при обсуждении [parse](#) в "Сравнение догадки с секретным числом": если пользователь введёт нечисловой ответ, программа завершится аварийно. Мы можем воспользоваться этим, чтобы позволить пользователю выйти из игры, как показано здесь:

```
$ cargo run
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 1.50s
    Running `target/debug/guessing_game`
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
You guessed: 59
You win!
Please input your guess.
quit
thread 'main' panicked at 'Please type a number!: ParseIntError { kind:
InvalidDigit }', src/main.rs:28:47
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Ввод **quit** приведёт к выходу из игры, но, как вы заметите, так же будет и при любом другом нечисловом вводе. Однако это, мягко говоря, не оптимально. Мы хотим, чтобы игра автоматически остановилась, когда будет угадано правильное число.

Выход после правильной догадки

Давайте запрограммируем игру на выход при выигрыше пользователя, добавив оператор **break**:

Имя файла: src/main.rs

```
// --snip--  
  
match guess.cmp(&secret_number) {  
    Ordering::Less => println!("Too small!"),  
    Ordering::Greater => println!("Too big!"),  
    Ordering::Equal => {  
        println!("You win!");  
        break;  
    }  
}  
}  
}
```

Добавление строки `break` после `You win!` заставляет программу выйти из цикла, когда пользователь правильно угадает секретное число. Выход из цикла также означает выход из программы, так как цикл является последней частью `main`.

Обработка недопустимого ввода

Чтобы ещё улучшить поведение игры, вместо аварийного завершения программы, когда пользователь вводит не число, давайте заставим игру проигрывать этот случай, позволяя пользователю продолжить угадывание. Для этого необходимо изменить строку, в которой `guess` преобразуется из `String` в `u32`, как показано в Листинге 2-5.

Имя файла: `src/main.rs`

```
// --snip--  
  
io::stdin()  
    .read_line(&mut guess)  
    .expect("Failed to read line");  
  
let guess: u32 = match guess.trim().parse() {  
    Ok(num) => num,  
    Err(_) => continue,  
};  
  
println!("You guessed: {}", guess);  
  
// --snip--
```

Листинг 2-5. Игнорирование нечисловой догадки и запрос другой догадки вместо завершения программы

Мы переключаем вызов `expect` на выражение `match`, чтобы перейти от аварийного завершения при ошибке к обработке ошибки. Помните, что `parse` возвращает тип `Result`, а `Result` - это перечисление, которое имеет варианты `Ok` и `Err`. Здесь мы используем выражение `match`, как и в случае с результатом `Ordering` метода `cmp`.

Если `parse` успешно преобразует строку в число, он вернёт значение `Ok`, содержащее полученное число. Это значение `Ok` будет соответствовать шаблону первой ветки, а выражение `match` просто вернёт значение `num`, которое `parse` произвёл и поместил внутрь значения `Ok`. Это число окажется в нужной нам переменной `guess`, которую мы создали.

Если метод `parse` не способен превратить строку в число, он вернёт значение `Err`, которое содержит более подробную информацию об ошибке. Значение `Err` не совпадает с шаблоном `Ok(num)` в первой ветке `match`, но совпадает с шаблоном `Err(_)` второй ветки. Подчёркивание `_` является всеохватывающим выражением. В этой ветке мы говорим, что хотим обработать совпадение всех значений `Err`, независимо от того, какая информация находится внутри `Err`. Поэтому программа выполнит код второй ветки, `continue`, который сообщает программе перейти к следующей итерации `loop` и запросить ещё одну догадку. В этом случае программа эффективно игнорирует все ошибки, с которыми может столкнуться `parse`!

Все в программе теперь должно работать как положено. Давайте попробуем:

```
$ cargo run
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Finished dev [unoptimized + debuginfo] target(s) in 4.45s
Running `target/debug/guessing_game`
Guess the number!
The secret number is: 61
Please input your guess.
10
You guessed: 10
Too small!
Please input your guess.
99
You guessed: 99
Too big!
Please input your guess.
foo
Please input your guess.
61
You guessed: 61
You win!
```

Потрясающе! С помощью одной маленькой последней правки мы закончим игру в угадывание. Напомним, что программа все ещё печатает секретное число. Это хорошо подходило для тестирования, но это портит игру. Давайте удалим `println!`, который выводит секретное число. В Листинге 2-6 показан окончательный вариант кода.

Имя файла: src/main.rs

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

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1..=100);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin()
            .read_line(&mut guess)
            .expect("Failed to read line");

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

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

Листинг 2-6: Полный код игры в угадывание

Итоги

На данный момент вы успешно создали игру в угадывание. Поздравляем!

Этот проект - практический способ познакомить вас со многими новыми концепциями Rust: `let`, `match`, функции, использование внешних пакетов и многое другое. В следующих нескольких главах вы изучите эти концепции более подробно. Глава 3 охватывает понятия, которые есть в большинстве языков программирования, такие как переменные, типы данных и функции, и показывает, как использовать их в Rust. В главе 4 рассматривается владение, особенность, которая отличает Rust от других языков. В главе 5 обсуждаются структуры и синтаксис методов, а в главе 6 объясняется, как работают перечисления.

Общие концепции программирования

Эта глава описывает концепции, которые реализованы во многих языках программирования, и как эти концепции работают в Rust. Многие языки в своей сути имеют много общего. Ни одна из представленных в этой главе концепций, не является уникальной для Rust, но мы обсудим как эти концепции воплощены в Rust и обсудим соглашения об использовании этих концепций.

Вы познакомитесь с такими понятиями, как переменные, базовые типы, функции, комментарии и управляющие конструкции. Все эти основы встретятся в любой программе на Rust, и их раннее изучение даёт хорошую основу для старта.

Ключевые слова

Язык Rust имеет набор *ключевых слов* (keywords), которые зарезервированы только для использования в языке, подобно тому, как это есть в других языках. Помните, что эти слова нельзя использовать в качестве названий переменных или функций. Большая часть этих слов имеет специальные значения и вы будете использовать их для выполнения различных задач в ваших программах. Некоторые из ключевых слов не имеют в данный момент связанной с ними функциональности, но были зарезервированы для функциональности, которая возможно будет добавлена в будущем. Список этих слов можно найти в [приложении A](#).

Переменные и понятие изменяемости

Как упоминалось в разделе “Сохранение значений в переменных”, по умолчанию переменные являются неизменяемыми. Это одна из многих подсказок, которые Rust даёт вам для написания кода таким образом, чтобы использовать преимущества безопасности и простого параллелизма, которые предлагает Rust. Однако у вас есть возможность сделать ваши переменные изменяемыми. Давайте рассмотрим, как и почему Rust поощряет неизменность и почему иногда вы можете отказаться от этого.

Когда переменная неизменяемая, то её значение нельзя менять, как только значение привязано к её имени. Приведём пример использования этого типа переменной. Для этого создадим новый проект *variables* в каталоге *projects* при помощи команды: `cargo new variables`.

Потом в созданной папке проекта *variables* откройте исходный файл *src/main.rs* и замените содержимое следующим кодом, который пока не будет компилироваться:

Файл: *src/main.rs*

```
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```



Сохраните код программы и выполните команду `cargo run`. В командной строке вы увидите сообщение об ошибке:

```
$ cargo run
Compiling variables v0.1.0 (file:///projects/variables)
error[E0384]: cannot assign twice to immutable variable `x`
--> src/main.rs:4:5
2 |     let x = 5;
|     |
|     |
|     first assignment to `x`
|     help: consider making this binding mutable: `mut x`
3 |     println!("The value of x is: {}", x);
4 |     x = 6;
|     ^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
error: could not compile `variables` due to previous error
```

Данный пример показывает как компилятор помогает найти ошибки в программах. Ошибки компилятора могут вызывать разочарование, они означают, что ваша программа ещё не выполняет то, что вы от неё хотите. Ошибки *не означают*, что вы пока не являетесь хорошим программистом! Опытные разработчики Rust также получают ошибки компиляции.

Сообщение об ошибке указывает, что причиной ошибки является то, что вы **cannot assign twice to immutable variable `x`** (не можете присвоить неизменяемой переменной новое значение), потому что вы пытались присвоить второе значение неизменяемой переменной **x**.

Важно, что мы получаем ошибку времени компиляции, при попытке изменить значение, обозначенное как неизменяемое, потому что такая ситуация может привести к ошибкам. Если одна часть нашего кода исходит из предположения, что значение никогда не изменится, а другая часть кода изменяет это значение, вполне возможно, что первая часть кода не будет делать то, для чего она предназначена. Причину такого рода ошибок может быть трудно отследить постфактум, особенно когда второй фрагмент кода изменяет значение только *иногда*. Компилятор Rust гарантирует, что если вы заявите, что значение не изменится, оно действительно не изменится, поэтому вам не нужно следить за ним самостоятельно. Таким образом, ваш код легче понять.

Но изменяемость может быть очень полезной. Переменные являются неизменяемыми только по умолчанию. Аналогично как вы делали в Главе 2, можно сделать переменные изменяемыми добавлением ключевого слова **mut** перед названием переменной. В дополнение к возможности изменить значение, указание

mut передаёт намерение будущим читателям кода, что другие части кода будут изменять значение этой переменной.

Например, изменим *src/main.rs* на следующий код:

Файл: *src/main.rs*

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

Запустив программу, мы получим результат:

```
$ cargo run
Compiling variables v0.1.0 (file:///projects/variables)
  Finished dev [unoptimized + debuginfo] target(s) in 0.30s
    Running `target/debug/variables`
The value of x is: 5
The value of x is: 6
```

Разрешено изменять значение, где **x** связывается с **5** и потом на **6**, когда используется **mut**. Есть несколько компромиссов, которые следует дополнительно учитывать для предотвращения ошибок. Например, в случаях, когда вы используете большие структуры данных, изменение экземпляра на месте может быть быстрее, чем копирование и возврат вновь созданных экземпляров. С меньшими структурами данных может быть легче продумать создание новых экземпляров и написание кода в более функциональном стиле, поэтому более низкая производительность может быть достойным штрафом за достижение ясности кода.

Константы

Подобно неизменяемым переменным, константы — это значения, которые связаны с именем и не могут изменяться, но между константами и переменными есть несколько различий.

Во-первых, не разрешается использовать **mut** с константами. Константы не просто неизменны по умолчанию — они неизменны всегда. Вы объявляете константы, используя ключевое слово **const** вместо ключевого слова **let** и тип должен быть

явно указан. Мы собираемся рассмотреть типы и аннотации типов в следующем разделе “[Типы данных](#)” так что не беспокойтесь о деталях сейчас. Просто знайте, что вы всегда должны указывать тип.

Константы можно объявить в любой области видимости, включая глобальную. Это делает их удобными для значений, про которые должны знать многие другие части кода.

Последней разницей является то, что константы можно установить только в константное выражение, а не в результат значения, которое можно посчитать только во время выполнения.

Вот пример объявления константы:

```
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

Имя константы `THREE_HOURS_IN_SECONDS` и её значение устанавливается в результате умножения числа 60 (количество секунд в минуте) на 60 (количество минут в часе) на 3 (количество часов, которое мы хотим подсчитать в этой программе). Соглашение об именах констант в Rust состоит в том, чтобы использовать все символы в верхнем регистре с символами подчёркивания между словами. Компилятор способен вычислить ограниченный набор операций во время компиляции, что позволяет нам записать это значение так, чтобы его было легче понять и проверить, вместо того, чтобы устанавливать для этой константы значение 10800. См [раздел справочника Rust, посвящённый вычислению констант](#) для получения дополнительной информации о том, какие операции можно использовать при объявлении констант.

Константы являются корректными для всего времени выполнения программы, внутри области видимости где они были объявлены. Это делает константы удобным выбором для значений в приложении, которые могут быть доступны во многих частях приложения. Например, максимально разрешённое количество очков игрока в игре или скорость света в вакууме.

Наименование не изменяемых значений во всей программе, таких как константа, является удобным способом выразить смысл значения для будущих пользователей кода. Этот помогает иметь только одно место в коде, которое придётся обновить, если будет необходимо поменять его значение в будущем.

Затенение (переменных)

Как вы видели в учебнике по игре по угадыванию числа в [Глава 2](#), можно объявить новую переменную с тем же именем, что и предыдущая переменная. Rust разработчики говорят, что первая переменная *затенена* второй, а это означает, что значение второй переменной — это то, что программа видит при её использовании. Мы можем затенить переменную, используя то же имя переменной и повторив использование ключевого слова `let` следующим образом:

Файл: `src/main.rs`

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

    let x = x + 1;

    {
        let x = x * 2;
        println!("The value of x in the inner scope is: {x}");
    }

    println!("The value of x is: {x}");
}
```

Программа сначала связывает значение `5` с переменной `x`. Затем `x` затеняется повторением кода `let x =` с помощью начального значения и прибавления к нему `1`, так что значение `x` становится равным `6`. Третье выражение `let` также затеняет переменную `x`, умножением предыдущее значение на `2`. Это даёт переменной `x` значение равное `12`. При запуске программы мы получим вывод:

```
$ cargo run
Compiling variables v0.1.0 (file:///projects/variables)
Finished dev [unoptimized + debuginfo] target(s) in 0.31s
Running `target/debug/variables`
The value of x in the inner scope is: 12
The value of x is: 6
```

Затенение отличается от объявления переменной с помощью `mut`, так как мы получим ошибку компиляции, если случайно попробуем переназначить значение без использования ключевого слова `let`. Используя `let`, можно выполнить несколько превращений над значением, при этом оставляя переменную неизменяемой, после того как все эти превращения завершены.

Другой разницей между `mut` и затенением является то, что мы создаём совершенно

новую переменную, когда снова используем слово `let` (ещё одну). Мы можем даже изменить тип значения, но снова использовать предыдущее имя. К примеру, наша программа спрашивает пользователя сколько пробелов он хочет разместить между некоторым текстом, запрашивая символы пробела, но мы на самом деле хотим сохранить данный ввод как число:

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

Первая переменная `spaces` — является строковым типом, а вторая переменная `spaces` — числовым типом. Таким образом, затенение избавляет нас от необходимости придумывать разные имена, такие как `spaces_str` и `spaces_num`, вместо этого мы можем повторно использовать более простое имя `spaces`. Однако, если мы попытаемся использовать для этого `mut`, как здесь показано, то мы получим ошибку времени компиляции:

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



Ошибка говорит, что не разрешается менять тип переменной:

```
$ cargo run
Compiling variables v0.1.0 (file:///projects/variables)
error[E0308]: mismatched types
--> src/main.rs:3:14
|
2 |     let mut spaces = "    ";
|           ----- expected due to this value
3 |     spaces = spaces.len();
|           ^^^^^^^^^^ expected `&str`, found `usize`
```

```
For more information about this error, try `rustc --explain E0308`.
error: could not compile `variables` due to previous error
```

Теперь, когда вы имеете представление о работе с переменными, посмотрим на большее количество типов данных, которые они могут иметь.

Типы данных

Любое значение в Rust является определённым *тиром данных* (data type), которое говорит о том, какие данные указаны, так что Rust знает как с ними работать. Рассмотрим два подмножества тип данных: скалярные (простые) и составные (сложные).

Не забывайте, что Rust является *статически типизированным* (statically typed) языком. Это означает, что он должен знать типы всех переменных во время компиляции. Обычно компилятор может вывести (*infer*) какой тип мы хотим использовать, основываясь на значении и на том, как мы с ним работаем. В случаях, когда может быть выведено несколько типов, необходимо вручную добавлять аннотацию типа. Например, когда мы конвертировали `String` в число с помощью вызова `parse` в разделе "[Сравнение предположения с загаданным номером](#)" Главы 2, мы должны добавить такую аннотацию:

```
let guess: u32 = "42".parse().expect("Not a number!");
```

Если её не добавить, то Rust покажет следующую ошибку, означающую, что компилятору необходимо больше информации, чтобы знать какой тип мы хотим использовать:

```
$ cargo build
Compiling no_type_annotations v0.1.0
(file:///projects/no_type_annotations)
error[E0282]: type annotations needed
--> src/main.rs:2:9
 |
2 |     let guess = "42".parse().expect("Not a number!");
|           ^^^^^ consider giving `guess` a type
For more information about this error, try `rustc --explain E0282`.
error: could not compile `no_type_annotations` due to previous error
```

В будущем вы увидите различные аннотации для разных типов данных.

Скалярные типы данных

Скалярный тип представляет единственное значение. В Rust есть четыре скалярных типа: целые и вещественные числа, логический тип и символы. Вы можете узнать

эти типы по другим языкам программирования. Посмотрим на то, как они работают в Rust.

Целые числа

Целое число, *integer*, является числом без дробной составляющей. Мы использовали целочисленный тип в Главе 2, это был `u32`. Данное объявление типа указывает, что значение связанное с ним должно быть беззнаковым целым (*unsigned integer*). Типы знаковых целых (*signed integer*) начинаются с буквы `i`, вместо буквы `u` и занимают до 32 бит памяти. Таблица 3-1 показывает встроенные целые типы Rust. Каждый вариант в колонках Знаковый и Беззнаковый, к примеру `i16`, может использоваться для объявления значения целочисленного типа.

Таблица 3-1: целые типы Rust

Размер	Знаковый	Беззнаковый
8-bit	<code>i8</code>	<code>u8</code>
16 бит	<code>i16</code>	<code>u16</code>
32 бита	<code>i32</code>	<code>u32</code>
64 бита	<code>i64</code>	<code>u64</code>
128 бит	<code>i128</code>	<code>u128</code>
arch	<code>isize</code>	<code>usize</code>

Каждый вариант типа может быть со знаком или без знака, и имеет точный размер в битах. Определение характеристики типа как *знаковый* и *без знаковый* означает, что число данного типа может быть либо отрицательным, либо только положительным. Другими словами характеристика показывает, нужно ли числу иметь знак (знаковое) или оно будет только положительным и может быть представлено без знака (без знаковое). Это похоже на написание чисел на бумаге: когда знак важен, то число отображено со знаком плюс или минус. Однако, когда можно с уверенностью предположить, что число положительное, то оно отображается без знака. Знаковые числа сохраняются при помощи [дополнительного кода](#) знака перед числом.

Каждый знаковый вариант может хранить числа от $-(2^{n-1})$ до $2^{n-1} - 1$ включительно, где n является количеством используемых бит. Так `i8` может хранить числа от $-(2^7)$ до $2^7 - 1$, что равно от -128 до 127. Беззнаковые варианты могут хранить числа от 0

до $2^n - 1$, так **u8** может хранить числа от 0 до $2^8 - 1$, т.е. от 0 до 255.

Также есть типы **isize** и **usize**, размер которых зависит от компьютера, на котором работает ваша программа: они имеют размер 64 бит, если операционная система использует 64-битную архитектуру, и 32 бита, если 32-битную.

Вы можете записывать целочисленные литералы в любой форме из таблицы 3-2. Заметим, что все числовые литералы, кроме байтового, позволяют использовать суффиксы, такие как **57u8** и **_** в качестве визуального разделителя, например **1_000**.

Таблица 3-2: целочисленные литералы в Rust

Числовые литералы	Пример
Десятичный	98_222
Шестнадцатеричный	0xff
Восьмеричный	0o77
Бинарный	0b1111_0000
Байтовый (только u8)	b'A'

Как узнать, какой литерал необходимо использовать? Если вы не уверены, то вариант предоставляемый в Rust по умолчанию является хорошим выбором. Для целых чисел типом по умолчанию является **i32**: в общем случае, данный тип самый быстрый даже на 64-битных системах. Основной ситуацией, когда вам необходимо использование **isize** или **usize**, является индексирование некоторых коллекций.

Целочисленное переполнение

Предположим, у нас есть переменная типа **u8**, которая может сохранять значения между 0 и 255. Если вы попытаетесь задать переменной значение вне данного диапазона, например в 256, то произойдёт *целочисленное переполнение*. В Rust есть несколько интересных правил, связанных с этим поведением. При компиляции кода в режиме отладки, компилятор Rust включает проверки, которые приводят к *панике* во время выполнения, если случится целочисленное переполнение. В Rust термин "паниковать" означает, что программа сразу завершается с ошибкой. Мы обсудим "панику" более детально разделе "[Необрабатываемые ошибки с помощью макроса panic!](#)"

главы 9.

При компиляции кода в финальную версию при помощи флага `--release`, Rust не включает проверки на целочисленное переполнение, приводящие к панике. Вместо этого, в случае переполнения Rust выполняет *оборачивание дополнительного кода*. Если кратко, то значения больше, чем максимальное значение, которое может хранить тип, превращаются в минимальное значение данного типа. Для типа `u8`, число 256 превращается в 0, 257 станет 1 и так далее. Программа не будет "паниковать", но переменная получит значение, которое вы возможно не ожидали. Полагаться на такое поведение считается ошибкой.

Чтобы явно обработать возможность переполнения, вы можете использовать следующие семейства методов, которые стандартная библиотека предоставляет для примитивных числовых типов:

- Обернуть все режимы с помощью `wrapping_*` методов, например `wrapping_add`
 - Вернуть значение `None` в случае переполнения при помощи методов `checked_*`
 - Вернуть значение и логическое значение, указывающее, было ли переполнение с помощью методов `overflowing_*`
 - Подавить минимальные или максимальные значения с помощью методов `saturating_*`
-

Числа с плавающей запятой

В Rust есть два примитивных типа для чисел с плавающей точкой (floating-point numbers), которые являются числами с десятичными точками. Числа с плавающей точкой в Rust представлены типами `f32` и `f64`, имеющими размер 32 и 64 бита соответственно. Типом по умолчанию является `f64`, потому что все современные CPU работают с ним приблизительно с такой же скоростью, как и с `f32`, но с большей точностью.

Пример для чисел с плавающей точкой в действии:

Файл: src/main.rs

```
fn main() {  
    let x = 2.0; // f64  
  
    let y: f32 = 3.0; // f32  
}
```

Числа с плавающей точкой представлены согласно стандарту IEEE-754. Тип **f32** является числом с плавающей точкой одинарной точности, а **f64** имеет двойную точность.

Числовые операции

Rust поддерживает базовые математические операции для всех числовых типов: сложение, вычитание, умножение, деление и получение остатка. Следующий код показывает использование каждой операции с помощью выражения **let**:

Файл: src/main.rs

```
fn main() {  
    // addition  
    let sum = 5 + 10;  
  
    // subtraction  
    let difference = 95.5 - 4.3;  
  
    // multiplication  
    let product = 4 * 30;  
  
    // division  
    let quotient = 56.7 / 32.2;  
    let floored = 2 / 3; // Results in 0  
  
    // remainder  
    let remainder = 43 % 5;  
}
```

Каждое из этих выражений использует математические операции и вычисляет значение, которое затем присваивается переменной. "Приложение Б" содержит список всех операторов, имеющихся в Rust.

Логический тип данных

Как и в большинстве языков программирования, логический тип в Rust может

иметь два значения: `true` и `false`, и занимает в памяти один байт. Логический тип в Rust аннотируется при помощи `bool`. Например:

Файл: src/main.rs

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

    let f: bool = false; // with explicit type annotation
}
```

Основной способ использования значений логического типа - это условные конструкции, такие как выражение `if`. Мы расскажем про работу выражения `if` в разделе "Условные конструкции".

Символьный тип данных

До этого момента мы работали только с числами, но Rust поддерживает также и буквы. Тип `char` является самым примитивным буквенным типом и следующий код показывает как им пользоваться. (Заметим, что литералы `char` определяются с помощью одинарных кавычек, в отличии от строк где используются двойные кавычки.)

Файл: src/main.rs

```
fn main() {
    let c = 'z';
    let z: char = 'ℤ'; // with explicit type annotation
    let heart_eyed_cat = '😻';
}
```

Тип `char` имеет размер в четыре байта и представляет собой скалярное юникод значение (Unicode Scalar Value), а значит, он может представить больше символов, чем есть в ASCII. Акцентированные буквы, китайские, японские и корейские символы, эмодзи и пробелы нулевой ширины - всё является корректными значениями `char` в Rust. Скалярное юникод значение имеет диапазон от `U+0000` до `U+D7FF` и от `U+E000` до `U+10FFFF` включительно. Тем не менее, "символ" на самом деле не является концептом в Юникод, так что человеческая интуиция о том, что такое "символ" может не совпадать с тем, чем является тип `char` в Rust. Более детально мы обсудим эту тему в разделе "Сохранение UTF-8 текста в строки" Главы 8.

Сложные типы данных

Сложные типы могут группировать несколько значений в один тип. В Rust есть два примитивных сложных (комбинированных) типа: кортежи и массивы.

Кортежи

Кортеж является общим способом совместной группировки нескольких значений различного типа в единый комбинированный тип. Кортежи имеют фиксированную длину: после объявления они не могут расти или уменьшаться в размере.

Кортеж создаётся при помощи записи списка значений, перечисленных через запятую внутри круглых скобок. Каждая позиция в кортеже имеет тип. Типы различных значений в кортеже могут не быть одинаковыми. В примере мы добавили не обязательные аннотации типов:

Файл: src/main.rs

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

Переменной с именем `tup` привязывается весь кортеж, потому что кортеж является единственным комбинированным элементом. Чтобы получить отдельные значения из кортежа, можно использовать сопоставление с образцом для деструктурирования значений кортежа, как в примере:

Файл: src/main.rs

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

    let (x, y, z) = tup;

    println!("The value of y is: {}", y);
}
```

Программа создаёт кортеж, привязывает его к переменной `tup`. Затем в `let` используется шаблон для превращения `tup` в три отдельных переменные: `x`, `y` и `z`. Такого рода операция называется *деструктуризацией* (destructuring), потому что она разрушает один кортеж на три части. В конце программа печатает значение `y`, которое равно `6.4`.

В дополнение к деструктурированию с помощью сопоставления шаблонов, можно напрямую получить доступ к элементам кортежа с помощью символа `.` и указанием индекса значения, к которому мы хотим обратиться. Например:

Файл: 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;
}
```

Программа создаёт кортеж с именем `x`, а затем создаёт новые переменные для каждого элемента, используя соответствующие индексы. Как и в большинстве языков, первый индекс в кортеже - это ноль.

Массивы

Другим способом получения коллекции из множества значений является **массив**. В отличии от кортежа, каждый элемент массива имеет одинаковый тип. Массивы в Rust отличаются от массивов в некоторых других языках тем, что в Rust они, подобно кортежам, имеют фиксированную длину.

В Rust, значения, хранящиеся в массиве, записываются как список разделённых запятой значений внутри квадратных скобок:

Файл: src/main.rs

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

Массивы являются полезными, когда вы хотите разместить данные в стеке, вместо выделения памяти в куче (мы обсудим стек и кучу в Главе 4) или когда мы хотим быть уверенными, что у нас есть место под фиксированное количество элементов. Массив не такой гибкий, как вектор. Вектор является похожим типом для коллекций, предоставленным стандартной библиотекой, которому *позволено* увеличиваться или уменьшаться в размере. Если вы не уверены, использовать массив или вектор, то возможно следует воспользоваться вектором. В Главе 8 обсуждаются вектора

более детально.

Примером, когда возможно лучше воспользоваться массивом вместо вектора, является программа в которой нужно знать названия месяцев в году. Вряд ли в такой программе необходимо добавлять или удалять месяцы, поэтому можно воспользоваться массивом, так как вы знаете, что он всегда включает в себя 12 элементов:

```
let months = ["January", "February", "March", "April", "May", "June", "July",
    "August", "September", "October", "November", "December"];
```

Для записи типа массива используются квадратные скобки, внутри которых сначала указывается тип каждого из элементов, а затем, через точку с запятой, указывают количество элементов массива, как тут:

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

Здесь, **i32** является типом каждого элемента массива. После точки с запятой указано число **5** показывающее, что массив содержит 5 элементов.

Подобный синтаксис определения типа массива выглядит аналогично альтернативному синтаксису инициализации массива: когда вы хотите создать массив состоящий из одинаковых значений, вы можете указать значение всех элементов перед точкой запятой (как мы бы это делали с типом массива), а затем длину массива внутри квадратных скобок, как в примере:

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

Массив в переменной **a** будет включать **5** элементов, значение которых будет равно **3**. Данная запись аналогична коду `let a = [3, 3, 3, 3, 3];`, но является более краткой.

Доступ к элементам массива

Массив является единственным блоком памяти выделенным на стеке. Можно получать доступ к элементам используя индекс:

Файл: src/main.rs

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

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

В данном примере, переменная `first` получит значение равное `1`, потому что это значение доступно по индексу `[0]` из массива. Переменная с именем `second` получит значение равное `2` из массива по индексу `[1]`.

Некорректный доступ к элементу массива

Что произойдёт, если вы попытаетесь получить доступ к элементу массива, который находится за пределами массива? Допустим, вы изменили пример на следующий, в котором используется код, аналогичный игре в угадывание в главе 2, для получения индекса массива от пользователя:

Файл: src/main.rs

```
use std::io;

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

    println!("Please enter an array index.");

    let mut index = String::new();

    io::stdin()
        .read_line(&mut index)
        .expect("Failed to read line");

    let index: usize = index
        .trim()
        .parse()
        .expect("Index entered was not a number");

    let element = a[index];

    println!("The value of the element at index {} is: {}", index, element);
}
```



Этот код успешно компилируется. Если запустить этот код с помощью `cargo run` и ввести `0, 1, 2, 3` или `4`, то программа распечатает соответствующее значение по

указанному индексу из массива. Если вы введёте число за границей массива, например 10, то вы увидите следующий результат:

```
thread 'main' panicked at 'index out of bounds: the len is 5 but the index is 10', src/main.rs:19:19
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Программа привела к ошибке *времени выполнения* в случае использования недопустимого значения для операции индексации массива. Программа завершилась с сообщением об ошибке и не выполнила последний оператор `println!`. Когда вы пытаетесь получить доступ к элементу с помощью индексации, Rust проверяет, что указанный вами индекс меньше длины массива. Если индекс больше или равен длине, Rust программа паникует. Эта проверка должна происходить во время выполнения, особенно в данном случае, потому что компилятор не может знать, какое значение позже введёт пользователь при выполнении кода.

Это первый пример на деле демонстрирующий принципы безопасности в Rust. Во многих низкоуровневых языках, такие типы проверок не выполняются, и когда вы предоставляете некорректный индекс, можно получить доступ к не корректной памяти. Rust защищает вас от такого рода ошибок тем, что немедленно завершает программу, вместо того, чтобы позволить получить такой доступ и продолжить выполнение. Обсуждение обработки ошибок в Rust ведётся в Главе 9.

ФУНКЦИИ

Функции широко распространены в коде Rust. Вы уже познакомились с одной из самых важных функций в языке: функцией `main`, которая является точкой входа большинства программ. Вы также видели ключевое слово `fn`, позволяющее объявлять новые функции.

Код Rust использует змеиный регистр (*snake case*) как основной стиль для имён функций и переменных, в котором все буквы строчные, а символ подчёркивания разделяет слова. Вот программа, содержащая пример определения функции:

Имя файла: `src/main.rs`

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

    another_function();
}

fn another_function() {
    println!("Another function.");
}
```

Для определения функции в Rust необходимо указать `fn`, за которым следует имя функции и набор круглых скобок. Фигурные скобки указывают компилятору, где начинается и заканчивается тело функции.

Мы можем вызвать любую определённую нами функцию, указав её имя, сопровожданное набором круглых скобок. Так как `another_function` определена в программе, её можно вызвать из функции `main`. Обратите внимание, в исходном коде мы определили `another_function` *после* функции `main`, но можно было бы определить её и до. Rust неважно, где вы определяете свои функции, главное, чтобы они были где-то определены.

Создадим новый бинарный проект с названием *functions* для дальнейшего изучения функций. Поместите пример `another_function` в файл `src/main.rs` и запустите его. Вы должны увидеть следующий вывод:

```
$ cargo run
Compiling functions v0.1.0 (file:///projects/functions)
  Finished dev [unoptimized + debuginfo] target(s) in 0.28s
    Running `target/debug/functions`
Hello, world!
Another function.
```

Строки выполняются в том порядке, в котором они расположены в функции `main`. Сначала печатается сообщение "Hello, world!", а затем вызывается `another_function`, которая также печатает сообщение.

Параметры функции

Можно определить функции с *параметрами*, которые являются специальными переменными, входящими в сигнатуру функции. Когда функция имеет параметры, вы можете предоставить ей конкретные значения этих параметров. С технической точки зрения конкретные значения называются *аргументами*, но в неформальной беседе люди обычно используют слова *параметр* и *аргумент* как взаимозаменяемые применительно к переменным в определении функции или конкретным значениям, передаваемым при вызове функции.

В этой версии `another_function` мы добавляем параметр:

Имя файла: `src/main.rs`

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

fn another_function(x: i32) {
    println!("The value of x is: {}", x);
}
```

Попробуйте запустить эту программу. Должны получить следующий результат:

```
$ cargo run
Compiling functions v0.1.0 (file:///projects/functions)
  Finished dev [unoptimized + debuginfo] target(s) in 1.21s
    Running `target/debug/functions`
The value of x is: 5
```

Объявление `another_function` имеет один параметр с именем `x`. Тип переменной

x задан как `i32`. Когда мы передаём `5` в `another_function`, макрос `println!` помещает `5` на место пары фигурных скобок в строке форматирования.

В сигнатурах функций вы должны объявить тип каждого параметра. Требование аннотаций типов в определениях функций - это намеренное решение в дизайне Rust, позволяющее компилятору дальше в коде их почти никогда не запрашивать, чтобы понять, какой тип имеется в виду.

При определении нескольких параметров, разделяйте объявления параметров запятыми, как показано ниже:

Имя файла: `src/main.rs`

```
fn main() {
    print_labeled_measurement(5, 'h');
}

fn print_labeled_measurement(value: i32, unit_label: char) {
    println!("The measurement is: {value}{unit_label}");
}
```

Этот пример создаёт функцию под именем `print_labeled_measurement` с двумя параметрами. Первый параметр называется `value` с типом `i32`. Второй называется `unit_label` и имеет тип `char`. Затем функция печатает текст, содержащий `value` и `unit_label`.

Попробуем запустить этот код. Замените текущую программу проекта `functions` в файле `src/main.rs` на предыдущий пример и запустите его с помощью `cargo run`:

```
$ cargo run
Compiling functions v0.1.0 (file:///projects/functions)
  Finished dev [unoptimized + debuginfo] target(s) in 0.31s
    Running `target/debug/functions`
The measurement is: 5h
```

Поскольку мы вызвали функцию с `5` в качестве значения для `value` и `'h'` в качестве значения для `unit_label`, вывод программы содержит эти значения.

Операторы и выражения

Тела функций состоят из ряда операторов, необязательно заканчивающихся выражением. До сих пор функции, которые мы рассматривали, не включали

завершающее выражение, но вы видели выражение как часть оператора. Поскольку Rust является языком, основанным на выражениях, это важное различие необходимо понимать. В других языках таких различий нет, поэтому давайте рассмотрим, что такое операторы и выражения, и как их различия влияют на тела функций.

Операторы - это инструкции, которые выполняют какое-либо действие и не возвращают значение. *Выражения* вычисляют результирующее значение. Давайте посмотрим на несколько примеров.

Фактически мы уже использовали операторы и выражения. Создание переменной и присвоение ей значения с помощью `let` - это оператор. В листинге 3-1 `let y = 6;` это оператор.

Файл: src/main.rs

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

Листинг 3-1: Объявление функции `main` включающей один оператор

Определение функции также является оператором. Весь предыдущий пример тоже является оператором.

Операции не возвращают значений. Тем не менее, нельзя назначить оператор `let` другой переменной, как это пытается сделать следующий код. Вы получите ошибку:

Файл: src/main.rs

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

Если вы запустите эту программу, то ошибка будет выглядеть так:

```
$ 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

error[E0658]: `let` expressions in this position are experimental
--> src/main.rs:2:14
|
2 |     let x = (let y = 6);
|           ^^^^^^^^^^
|
= note: see issue #53667 <https://github.com/rust-lang/rust/issues/53667>
for more information
= help: you can write `matches!(<expr>, <pattern>)` instead of `let
<pattern> = <expr>`

warning: unnecessary parentheses around assigned value
--> src/main.rs:2:13
|
2 |     let x = (let y = 6);
|           ^           ^
|
= note: `#[warn(unused_parens)]` on by default
help: remove these parentheses
|
2 -     let x = (let y = 6);
2 +     let x = let y = 6;
|
```

For more information about this error, try `rustc --explain E0658`.
warning: `functions` (bin "functions") generated 1 warning
error: could not compile `functions` due to 2 previous errors; 1 warning emitted

Оператор `let y = 6` не возвращает значения, так что нет ничего что можно было бы назначить переменной `x`. Такое поведение отлично от некоторых других языков, типа C и Ruby, где выражение присваивания возвращает присваиваемое значение. В таких языках можно писать код `x = y = 6` и обе переменные `x` и `y` будут иметь одинаковое значение `6`; но это не так в Rust.

Выражения вычисляют значение и составляют большую часть остального кода, который вы напишете на Rust. Рассмотрим математическую операцию, например `5`

+ 6, которая является выражением, результатом которого является значение 11. Выражения могут быть частью операторов: в листинге 3-1 цифра 6 в операторе let y = 6; - выражение, которое принимает значение 6. Вызов функции - это выражение. Вызов макроса - это выражение. Новый блок области видимости, созданный с помощью фигурных скобок, представляет собой выражение, например:

Файл: src/main.rs

```
fn main() {
    let y = {
        let x = 3;
        x + 1
    };

    println!("The value of y is: {y}");
}
```

Это выражение:

```
{
    let x = 3;
    x + 1
}
```

Это блок, который в данном случае будет вычислен в 4. Это значение привязывается к y как часть оператора let. Обратите внимание, что x + 1 не имеет точки с запятой в конце, в отличие от большинства строк, которые вы видели до сих пор. Выражения не включают точку с запятой в конце. Если вы добавите точку с запятой в конец выражения, вы превратите его в оператор и тогда оно не вернёт значение. Помните об этом, когда будете изучать возвращаемые функцией значения и выражения.

Функции возвращающие значения

Функции могут возвращать значения в вызывающий их код. Мы не именуем возвращаемые значения, но мы объявляем их тип после символа (->). В Rust, возвращаемое значение функции является синонимом значения последнего выражения в блоке тела функции. Можно выполнить ранний возврат из функции используя ключевое слово return и указав значение, но большинство функций явно возвращает последнее выражение в теле. Вот пример функции

возвращающей значение:

Файл: src/main.rs

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

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

    println!("The value of x is: {x}");
}
```

В коде функции `five` нет вызовов функций, макросов или даже операторов `let` - есть только одно число `5`. Это является абсолютно корректной функцией в Rust. Заметьте, что возвращаемый тип у данной функции определён как `-> i32`.

Попробуйте запустить; вывод будет таким:

```
$ cargo run
Compiling functions v0.1.0 (file:///projects/functions)
  Finished dev [unoptimized + debuginfo] target(s) in 0.30s
    Running `target/debug/functions`
The value of x is: 5
```

Число `5` в функции `five` является возвращаемым значением функции (можно сказать что функция `five` вычисляется в `5`), вот почему возвращаемым типом является `i32`. Рассмотрим пример более детально. Есть два важных момента: первый строка `let x = five();` показывает, что мы используем значение возвращаемое функцией для инициализации переменной. Так как функция `five` возвращает `5`, то эта строка эквивалентна следующей:

```
let x = 5;
```

Второй момент, функция `five` не имеет входных параметров и определяет тип возвращаемого значения. Само тело функции - единственная `5` без точки с запятой. Т.к. мы хотим, чтобы функция возвращала значение, последняя строка функции должна быть выражением (не иметь после себя знак точки с запятой). В данной функции мы хотим вернуть `5` - по этому `5` должно быть выражением (не должно иметь после себя знак точки с запятой).

Рассмотрим другой пример:

Файл: src/main.rs

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

    println!("The value of x is: {}", x);
}

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

Запуск кода выведет `The value of x is: 6`. Но если поместить точку с запятой в конец строки, включающей `x + 1`, то это изменит её с выражения на оператор и мы получим ошибку.

Файл: src/main.rs

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

    println!("The value of x is: {}", x);
}

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



Компиляция данного кода вызывает следующую ошибку:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error[E0308]: mismatched types
--> src/main.rs:7:24
   |
7 | fn plus_one(x: i32) -> i32 {
   |         -----      ^^^ expected `i32`, found `()`  
   |         |
   |         implicitly returns `()` as its body has no tail or `return` expression
8 |     x + 1;
   |         - help: consider removing this semicolon

For more information about this error, try `rustc --explain E0308`.
error: could not compile `functions` due to previous error
```

Основное сообщение об ошибке «несовпадающие типы» раскрывает основную проблему с этим кодом. В определении функции `plus_one` говорится, что она

вернёт `i32`, но операторы не вычисляют значение, которое выражается `()`. Следовательно, ничего не возвращается, что противоречит определению функции и приводит к ошибке. В этом выводе Rust предоставляет сообщение, которое, возможно, поможет исправить эту проблему: он предлагает удалить точку с запятой, что исправит ошибку.

Комментарии

Все хорошие программисты, создавая программный код, стремятся сделать его простым для понимания. Бывают всё же случаи, когда дополнительное описание просто необходимо. В этих случаях программисты пишут заметки (или как их ещё называют, комментарии). Комментарии игнорируются компилятором, но для тех кто код читает - это очень важная часть документации.

Пример простого комментария:

```
// Hello, world.
```

В Rust комментарии должны начинаться двумя символами `//` и простираются до конца строки. Чтобы комментарии поместились на более чем одной строке, необходимо разместить их на каждой строке, как в примере:

```
// So we're doing something complicated here, long enough that we need  
// multiple lines of comments to do it! Whew! Hopefully, this comment will  
// explain what's going on.
```

Комментарии могут быть размещены в конце строки имеющей код:

Файл: src/main.rs

```
fn main() {  
    let lucky_number = 7; // I'm feeling lucky today  
}
```

Но чаще вы увидите их использованные в следующем формате, здесь комментарий размещён на отдельной строке над кодом, который комментируется:

Файл: src/main.rs

```
fn main() {  
    // I'm feeling lucky today  
    let lucky_number = 7;  
}
```

Также в Rust есть другой тип комментариев - документирующие комментарии, они используются в документации, и их обсуждаются разделе "Публикация пакета на Crates.io" Главы 14.

Поток управления

Способность запускать некоторый код в зависимости от истинности условия или выполнять некоторый код многократно, пока условие истинно, является базовым элементом большинства языков программирования. Наиболее распространёнными конструкциями, позволяющими управлять потоком выполнения кода в Rust, являются выражения `if` и циклы.

Выражения `if`

Выражение `if` позволяет разветвлять код в зависимости от условий. Вы задаёте условие, а затем объявляете: "Если это условие соблюдено, то выполнить этот блок кода. Если условие не соблюдается, не выполнять этот блок кода".

Для изучения выражения `if` создайте новый проект под названием *branches* в каталоге *projects*. В файл *src/main.rs* поместите следующий код:

Имя файла: *src/main.rs*

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

    if number < 5 {
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
```

Все выражения `if` начинаются с ключевого слова `if`, за которым следует условие. В данном случае условие проверяет, имеет ли переменная `number` значение меньше 5. Мы помещаем блок кода, который будет выполняться, если условие истинно, сразу после условия внутри фигурных скобок. Блоки кода, связанные с условиями в выражениях `if`, иногда называют *ветками*, так же как и ветки в выражениях `match`, которые мы обсуждали в разделе "[Сравнение догадки с секретным числом](#)" главы 2.

Опционально можно включить выражение `else`, которое мы используем в данном примере, чтобы предоставить программе альтернативный блок выполнения кода, выполняющийся при ложном условии. Если не указать выражение `else` и условие будет ложным, программа просто пропустит блок `if` и перейдёт к следующему

фрагменту кода.

Попробуйте запустить этот код. Появится следующий результат:

```
$ cargo run
Compiling branches v0.1.0 (file:///projects/branches)
  Finished dev [unoptimized + debuginfo] target(s) in 0.31s
    Running `target/debug/branches`
condition was true
```

Попробуйте изменить значение `number` на значение, которое делает условие `false` и посмотрите, что произойдёт:

```
let number = 7;
```

Запустите программу снова и посмотрите на вывод:

```
$ cargo run
Compiling branches v0.1.0 (file:///projects/branches)
  Finished dev [unoptimized + debuginfo] target(s) in 0.31s
    Running `target/debug/branches`
condition was false
```

Также стоит отметить, что условие в этом коде *должно* быть логического типа `bool`. Если условие не является `bool`, возникнет ошибка. Например, попробуйте запустить следующий код:

Имя файла: `src/main.rs`

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

    if number {
        println!("number was three");
    }
}
```



На этот раз условие `if` вычисляется в значение `3`, и Rust бросает ошибку:

```
$ cargo run
Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: mismatched types
--> src/main.rs:4:8
 |
4 |     if number {
|         ^^^^^^ expected `bool`, found integer
|
For more information about this error, try `rustc --explain E0308`.
error: could not compile `branches` due to previous error
```

Ошибка говорит, что Rust ожидал тип `bool`, но получил значение целочисленного типа. В отличии от других языков вроде Ruby и JavaScript, Rust не будет пытаться автоматически конвертировать *нелогические* типы в логические. Необходимо быть явным и всегда использовать `if` с логическим типом в качестве условия. Если нужно, чтобы блок кода `if` запускался только, когда число не равно `0`, то, например, мы можем изменить выражение `if` на следующее:

Имя файла: `src/main.rs`

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

    if number != 0 {
        println!("number was something other than zero");
    }
}
```

Будет напечатана следующая строка `number was something other than zero.`

Обработка нескольких условий с помощью `else if`

Можно использовать несколько условий, комбинируя `if` и `else` в выражении `else if`. Например:

Имя файла: `src/main.rs`

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

    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

У этой программы есть четыре возможных пути выполнения. После её запуска вы должны увидеть следующий результат:

```
$ cargo run
Compiling branches v0.1.0 (file:///projects/branches)
Finished dev [unoptimized + debuginfo] target(s) in 0.31s
Running `target/debug/branches`
number is divisible by 3
```

Во время выполнения этой программы по очереди проверяется каждое выражение `if` и выполняется первое тело, для которого условие истинно. Заметьте, что хотя 6 делится на 2, мы не видим ни вывода `number is divisible by 2`, ни текста `number is not divisible by 4, 3, or 2` из блока `else`. Так происходит потому, что Rust выполняет блок только для первого истинного условия, а обнаружив его, даже не проверяет остальные.

Использование множества выражений `else if` приводит к загромождению кода, поэтому при наличии более чем одного выражения, возможно, стоит провести рефакторинг кода. В главе 6 описана мощная конструкция ветвления Rust для таких случаев, называемая `match`.

Использование `if` в `let`-операторах

Поскольку `if` является выражением, его можно использовать в правой части оператора `let` для присвоения результата переменной, как в листинге 3-2.

Имя файла: src/main.rs

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

    println!("The value of number is: {number}");
}
```

Листинг 3-2: Присвоение результата выражения `if` переменной

Переменная `number` будет привязана к значению, которое является результатом выражения `if`. Запустим код и посмотрим, что происходит:

```
$ cargo run
Compiling branches v0.1.0 (file:///projects/branches)
Finished dev [unoptimized + debuginfo] target(s) in 0.30s
Running `target/debug/branches`
The value of number is: 5
```

Вспомните, что блоки кода вычисляются последним выражением в них, а числа сами по себе также являются выражениями. В данном случае, значение всего выражения `if` зависит от того, какой блок выполняется. При этом значения, которые могут быть результатами каждого из ветвей `if`, должны быть одного типа. В Листинге 3-2, результатами обеих ветвей `if` и `else` являются целочисленный тип `i32`. Если типы не совпадают, как в следующем примере, мы получим ошибку:

Имя файла: src/main.rs

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

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

    println!("The value of number is: {number}");
}
```



При попытке компиляции этого кода, мы получим ошибку. Ветви `if` и `else` представляют несовместимые типы значений, и Rust точно указывает, где искать проблему в программе:

```
$ cargo run
Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: `if` and `else` have incompatible types
--> src/main.rs:4:44
 |
4 |     let number = if condition { 5 } else { "six" };
|                         -           ^^^^^^ expected integer, found
`&str'
 |
|           |
|           expected because of this

For more information about this error, try `rustc --explain E0308`.
error: could not compile `branches` due to previous error
```

Выражение в блоке `if` вычисляется как целочисленное, а выражение в блоке `else` вычисляется как строка. Это не сработает, потому что переменные должны иметь один тип, а Rust должен знать во время компиляции, какого типа переменная `number`. Зная тип `number`, компилятор может убедиться, что тип действителен везде, где мы используем `number`. Rust не смог бы этого сделать, если бы тип `number` определялся только во время выполнения.. Компилятор усложнился бы и давал бы меньше гарантий в отношении кода, если бы ему приходилось отслеживать несколько гипотетических типов для любой переменной.

Повторение выполнения кода с помощью циклов

Часто бывает полезно выполнить блок кода более одного раза. Для этой задачи Rust предоставляет несколько циклов, которые позволяют выполнить код внутри тела цикла до конца, а затем сразу же вернуться в начало. Для экспериментов с циклами давайте создадим новый проект под названием *loops*.

В Rust есть три вида циклов: `loop`, `while` и `for`. Давайте попробуем каждый из них.

Повторение выполнения кода с помощью `loop`

Ключевое слово `loop` говорит Rust выполнять блок кода снова и снова до бесконечности или пока не будет явно приказано остановиться.

В качестве примера, измените код файла `src/main.rs` в каталоге проекта *loops* на код ниже:

Имя файла: `src/main.rs`

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

Когда запустим эту программу, увидим, как `again!` печатается снова и снова, пока не остановить программу вручную. Большинство терминалов поддерживают комбинацию клавиш `ctrl-c` для прерывания программы, которая застряла в непрерывном цикле. Попробуйте:

```
$ cargo run
Compiling loops v0.1.0 (file:///projects/loops)
Finished dev [unoptimized + debuginfo] target(s) in 0.29s
    Running `target/debug/loops`
again!
again!
again!
again!
again!
^Cagain!
```

Символ `^C` обозначает место, где было нажато `ctrl-c`. В зависимости от того, где находился код в цикле в момент получения сигнала прерывания, вы можете увидеть или не увидеть слово `again!`, напечатанное после `^C`.

К счастью, Rust также предоставляет способ выйти из цикла с помощью кода. Ключевое слово `break` нужно поместить в цикл, чтобы указать программе, когда следует прекратить выполнение цикла. Напоминаем, мы делали так в игре "Угадайка" в разделе ["Выход после правильной догадки"](#) главы 2, чтобы выйти из программы, когда пользователь выиграл игру, угадав правильное число.

Мы также использовали `continue` в игре "Угадайка", которая указывает программе в цикле пропустить весь оставшийся код в данной итерации цикла и перейти к следующей итерации.

Если у вас есть циклы внутри циклов, `break` и `continue` будут применяться к самому внутреннему циклу в данной точке. Дополнительно можно указать *метку цикла*, которую затем можно использовать с `break` или `continue`, чтобы обозначить, что эти ключевые слова применяются к помеченному циклу, а не к самому внутреннему циклу. Вот пример с двумя вложенными циклами:

```
fn main() {
    let mut count = 0;
    'counting_up: loop {
        println!("count = {count}");
        let mut remaining = 10;

        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break 'counting_up;
            }
            remaining -= 1;
        }

        count += 1;
    }
    println!("End count = {count}");
}
```

Внешний цикл имеет метку `'counting_up'`, и он будет считать от 0 до 2. Внутренняя петля без метки ведёт обратный отсчёт от 10 до 9. Первый `break`, который не содержит метку, выйдет только из внутреннего цикла. Оператор `break 'counting_up;` завершит внешний цикл. Этот код напечатает:

```
$ cargo run
Compiling loops v0.1.0 (file:///projects/loops)
Finished dev [unoptimized + debuginfo] target(s) in 0.58s
    Running `target/debug/loops`
count = 0
remaining = 10
remaining = 9
count = 1
remaining = 10
remaining = 9
count = 2
remaining = 10
End count = 2
```

Возвращение значений из циклов

Одно из применений `loop` - это повторение операции, которая может закончиться неудачей, например, проверка успешности выполнения потоком своего задания. Также может понадобиться передать из цикла результат этой операции в

остальную часть кода. Для этого можно добавить возвращаемое значение после выражения `break`, которое используется для остановки цикла. Это значение будет возвращено из цикла, и его можно будет использовать, как показано здесь:

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

    let result = loop {
        counter += 1;

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

    println!("The result is {result}");
}
```

Перед началом цикла объявляем переменную `counter` и инициализируем значением `0`. Затем объявляем переменную `result` для хранения значения, возвращаемого из цикла. На каждой итерации цикла прибавляем `1` к переменной `counter`, а затем проверяем, равен ли счётчик `10`. Если это так, то используем ключевое слово `break` со значением `counter * 2`. После цикла используем точку с запятой для завершения оператора для присвоения значения `result`. Наконец, распечатаем значение из `result`, которое в данном случае равно `20`.

Циклы с условием `while`

В программе часто требуется проверить состояние условия в цикле. Пока условие истинно, цикл выполняется. Когда условие перестаёт быть истинным, программа вызывает `break`, останавливая цикл. Такое поведение можно реализовать с помощью комбинации `loop`, `if`, `else` и `break`. При желании попробуйте сделать это в программе. Это настолько распространённый паттерн, что в Rust реализована встроенная языковая конструкция для него, называемая цикл `while`. В листинге 3-3 мы используем `while`, чтобы выполнить три цикла программы, производя каждый раз обратный отсчёт, а затем, после завершения цикла, печатаем сообщение и выводим.

Имя файла: src/main.rs

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

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

        number -= 1;
    }

    println!("LIFTOFF!!!");
}
```

Листинг 3-3: Использование цикла `while` для выполнения кода, пока условие истинно

Эта конструкция устраняет множество вложений, которые потребовались бы при использовании `loop`, `if`, `else` и `break`, и она более понятна. Пока условие истинно, код выполняется, в противном случае происходит выход из цикла.

Цикл по элементам коллекции с помощью `for`

Для перебора элементов коллекции, например, массива, можно использовать конструкцию `while`. Например, цикл в листинге 3-4 печатает каждый элемент массива `a`.

Имя файла: `src/main.rs`

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

    while index < 5 {
        println!("the value is: {}", a[index]);

        index += 1;
    }
}
```

Листинг 3-4: Перебор каждого элемента коллекции с помощью цикла `while`

Этот код выполняет перебор элементов массива. Он начинается с индекса `0`, а затем циклически выполняется, пока не достигнет последнего индекса в массиве (то есть, когда `index < 5` уже не является истиной). Выполнение этого кода напечатает каждый элемент массива:

```
$ cargo run
Compiling loops v0.1.0 (file:///projects/loops)
  Finished dev [unoptimized + debuginfo] target(s) in 0.32s
    Running `target/debug/loops`
the value is: 10
the value is: 20
the value is: 30
the value is: 40
the value is: 50
```

Все пять значений массива появляются в терминале, как и ожидалось. Поскольку **index** в какой-то момент достигнет значения **5**, цикл прекратит выполнение перед попыткой извлечь шестое значение из массива.

Однако такой подход чреват ошибками. Можно вызвать панику в программе, если значение индекса или условие теста неверны. Например, если изменить определение массива **a** на четыре элемента, но забыть обновить условие на **while index < 4**, код вызовет панику. Также это медленно, поскольку компилятор добавляет код времени выполнения для обеспечения проверки нахождения индекса в границах массива на каждой итерации цикла.

В качестве более краткой альтернативы можно использовать цикл **for** и выполнять некоторый код для каждого элемента коллекции. Цикл **for** может выглядеть как код в листинге 3-5.

Имя файла: src/main.rs

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

    for element in a {
        println!("the value is: {}", element);
    }
}
```

Листинг 3-5: Перебор каждого элемента коллекции с помощью цикла **for**

При выполнении этого кода мы увидим тот же результат, что и в листинге 3-4. Что важнее, теперь мы повысили безопасность кода и устранили вероятность ошибок, которые могут возникнуть в результате выхода за пределы массива или недостаточно далёкого перехода и пропуска некоторых элементов.

При использовании цикла **for** не нужно помнить о внесении изменений в другой код, в случае изменения количества значений в массиве, как это было с

методом, использованным в листинге 3-4.

Безопасность и компактность циклов `for` делают их наиболее часто используемой конструкцией цикла в Rust. Даже в ситуациях необходимости выполнения некоторого кода определённое количество раз, как в примере обратного отсчёта, в котором использовался цикл `while` из Листинга 3-3, большинство Rustaceans использовали бы цикл `for`. Для этого можно использовать `Range`, предоставляемый стандартной библиотекой, который генерирует числа по порядку, начиная с одного числа и заканчивая другим числом.

Вот как будет выглядеть обратный отсчёт с использованием цикла `for` и другого метода, о котором мы ещё не говорили, `rev`, для разворота диапазона:

Имя файла: `src/main.rs`

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

Данный код выглядит лучше, не так ли?

Итоги

Вы справились! Это была объёмная глава: вы узнали о переменных, скалярных и составных типах данных, функциях, комментариях, выражениях `if` и циклах! Для практики работы с концепциями, обсуждаемыми в этой главе, попробуйте создать программы для выполнения следующих действий:

- Конвертация температур между значениями по Фаренгейту к Цельсия.
- Генерирование n-го числа Фибоначчи.
- Распечатайте текст рождественской песни "Двенадцать дней Рождества", воспользовавшись повторами в песне.

Когда вы будете готовы двигаться дальше, мы поговорим о концепции в Rust, которая *не существует* обычно в других языках программирования: владение.

Понимание владения

Владение является наиболее уникальной особенностью языка Rust. Благодаря ей в Rust осуществляется безопасная работа с памятью без необходимости использования автоматической системы сборки мусора (garbage collector). Тем не менее, важно понимать как владение работает в Rust. В этой главе мы расскажем о владении, а также затронем несколько связанных с ней функциональностей, таких как: заимствование и срезы, а также разберёмся каким образом Rust распределяет данные в памяти.

Что же такое владение?

Владение является центральной особенностью языка Rust. Хотя эту особенность легко объяснить, она весьма сильно повлияла на остальную часть языка.

Все программы во время выполнения используют память компьютера и используют разные подходы для управления своей памятью. В одних языках программирования для этой цели используют систему сборки мусора (garbage collection, GC) постоянно следящую за памятью программы, которая больше не используется программой. В других языках программист должен сам явно запрашивать и освобождать память. Rust же использует третий подход: память управляется с помощью системы владения с набором правил, которые компилятор проверяет только во время компиляции программы. Ни одно из правил владения не замедляет выполнение программы.

Так как эта концепция ещё нова для многих программистов, её осознание и эффективное использование потребует определённого времени. Хорошая новость в том, что чем более опытным вы становитесь, тем сможете более естественно разрабатывать код, являющийся безопасным и эффективным. Думаю, что цель этого стоит.

Понимание концепции владения даст вам основу для понимания всех остальных особенностей, делающих Rust уникальным. В этой главе вы изучите владение на примерах, которые сфокусированы на наиболее часто используемой структуре данных: строках.

Стек и куча (heap)

Во многих языках программирования вам обычно не приходится часто думать о памяти в стеке или памяти в куче. Для системного языка программирование место хранения переменной (на стеке или в куче) имеет большее влияния на то, как язык ведёт себя и почему необходимо принимать определённые решения. Позже в данной главе будет описана основная часть правил владения относительно стека и кучи, здесь же в целях подготовки к основной статье представлено краткое объяснение.

Стек и куча являются частями памяти компьютера, которая доступна вашему коду во время выполнения, но они структурированы по разному. Стек

сохраняет значения в порядке получения данных и удаляет их в обратном порядке. Такого рода концепция известна как *последний зашёл, первый вышел* (last in, first out). Думайте о стеке как о стопке тарелок: при добавлении тарелок вы размещаете их сверху стопки, а когда тарелка нужна берете её сверху. Добавление и удаление тарелок из середины или снизу запрещено и не работает! Добавление данных называется *помещением в стек* (pushing onto), а удаление называется *извлечением из стека* (popping off).

Все данные сохраняемые в стеке должны быть известны и иметь фиксированный размер. Данные с неизвестным размером во время компиляции или размером, который может изменится в ходе выполнения программы должны сохраняться в куче. Куча является менее организованной: при размещении данных в куче запрашивается определённое количество памяти. Операционная система находит пустой участок кучи, являющийся достаточно большим, помечает его как используемый и возвращает указатель, который является адресом данного участка. Данный процесс называется *выделением в куче* и иногда сокращённо называется просто *выделение* (allocating). Размещение значений в стеке не считается выделением. По причине того, что указатель имеет известный, фиксированный размер, его можно сохранить в стеке, но когда вам нужны сами данные необходимо проследовать по указателю.

Представьте что вы находитесь в ресторане. Когда вы заходите, вы указываете количество людей в вашей компании, а обслуживающий персонал ищет пустой стол подходящий для вас и ведёт к нему. Если кто-то из компании придёт позже, то он сможет спросить где в зале вы находитесь, чтобы найти вас.

Размещение в стек происходит быстрее, чем выделение в куче, потому что операционная система никогда не делает поиска места для хранения новых данных. Местом размещения всегда является верхушка стека. Выделение памяти в куче требует больше работы, потому что операционная система должна сначала найти достаточно большой участок памяти для хранения данных и затем выполнить резервирование, чтобы подготовится к следующему выделению.

Доступ данных в куче является более медленным, чем в стеке, потому что необходимо сначала проследовать по указателю для получения данных. Современные процессоры работают быстрее, если они меньше "прыгают" по памяти. Продолжая аналогию, представьте официанта в ресторане, который принимает заказы с нескольких столов по мере их поступления. Для

официанта наиболее эффективным было бы получить сразу все заказы со стола и уже затем перейти к следующему столу, не возвращаясь к гостю с первого стола который вдруг вспомнил, что ему ещё надо заказать десерт. Принимать один заказ со стола А, затем со стола В, а затем снова со стола А и снова со стола В будет гораздо более медленным процессом. Кроме того, процессор может лучше выполнить работу, если оперирует данными которые расположены в памяти близко к другим данным (подобно тому как в стеке), а не далеко (как это может быть в куче). Выделение большого количества памяти в куче также занимает время.

При вызове функции значения передаваемые в неё (потенциально включая и указатели на данные в куче) и локальные переменные функции размещаются в стеке. Когда функция завершается, эти значения извлекаются из стека.

Отслеживание какие части кода используют данные в куче, минимизация количества дубликатов данных в ней и очистка неиспользуемых там данных, чтобы не закончилась вся память - это все проблемы, которые решает владение. Как только вы поймёте владение, вам больше не понадобится слишком часто думать про стек и кучу. Понимание того, что владение существует для управления данными в куче, помогает объяснить, почему это все работает и как.

Правила владения

Прежде всего, давайте познакомимся с самими правилами. Пожалуйста, помните о них во время работы с примерами сделанными для их иллюстрации:

- каждое значение имеет переменную, которая называется *владельцем* значения,
- у значения может быть только один владелец в один момент времени,
- когда владелец покидает область видимости, значение удаляется.

Область видимости переменной

Мы уже видели как область видимости работает на примере Rust программы в Главе 2. После прохождения базового синтаксиса, мы не будем включать в примеры код функции `fn main() { }`, так что если вы будете следовать примерам, вам нужно

будет поместить следующие примеры внутрь функции `main` самостоятельно. В результате наш пример будут немного короче и мы сможем фокусироваться на деталях, а не на шаблонном коде.

В качестве первого примера владения мы рассмотрим *область видимости* переменных. Область видимости является диапазоном внутри программы, в котором элемент программы является действительным. Например, есть переменная, которая выглядит так:

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

Переменная `s` ссылается на строковый литерал и значение данной переменной жёстко задано в коде программы. Переменная считается действительной с момента её объявления до конца текущей *области видимости*. В листинге 4-1 есть комментарии с аннотациями где переменная `s` является действительной.

```
{                                     // s is not valid here, it's not yet declared
    let s = "hello";      // s is valid from this point forward

    // do stuff with s
}
                                     // this scope is now over, and s is no longer
valid
```

Листинг 4-1: переменная и область видимости в которой она действительна

Другими словами, здесь есть два важных момента:

- когда переменная `s` появляется в области видимости, она считается действительной,
- она остаётся действительной до момента выхода за границы этой области.

На этом этапе объяснения, взаимосвязь между областями действия и допустимостью переменных аналогична той, что существует в других языках программирования. Теперь мы будем опираться на это понимание, введя тип `String`.

Тип данных `String`

Для иллюстрации правил владения нужен тип данных более сложный чем, те которые были в разделе "[Типы данных](#)" Главы 3. Типы описанные ранее, являются

типами сохраняемыми в стеке и извлекаемые из него, когда их область видимости заканчивается. Но мы хотим рассмотреть данные сохранённые в куче и разобраться в том как Rust определяет, в какой момент нужно очищать эти данные.

Воспользуемся типом `String` в качестве примера и сконцентрируемся на частях `String` относящихся ко владению. Данные аспекты применимы и для более сложных типов данных, не важно предоставлены ли они из стандартной библиотеки или созданы вами. Мы ещё обсудим более детально тип `String` в Главе 8.

Мы видели строковые литералы в которых значение строки жёстко закодировано в программе. Строковые литералы удобны, но не подходят для любой ситуации в которой мы бы хотели использовать текст. Одна из причин - неизменяемость данных литерала. Другая причина в том, что не любое строковое значение может быть известным при написании кода: например, что если хочется получить ввод пользователя и сохранить его? В данной ситуации Rust имеет второй строковый тип `String`. Память для значений этого типа выделяется в куче, так что можно сохранять количество текста неизвестное во время компиляции. Можно создать `String` из строкового литерала используя функцию `from`, вот так:

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

Два двоеточия (`::`) является оператором, который позволяет воспользоваться в текущем пространстве имён функцией `from` для типа `String` вместо использования некоторого имени функции `string_from`. Мы обсудим синтаксис детальнее в разделе "Синтаксис методов" Главы 5 и когда поговорим про пространства имён с модулями "Путь для обращения к элементу в дереве модулей" Главы 7.

Такие строки могут быть изменены:

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

s.push_str(", world!"); // push_str() appends a literal to a String

println!("{}", s); // This will print `hello, world!`
```

В чем здесь разница? Почему `String` можно менять, а литерал нельзя? Разница в том, как эти два типа работают с памятью.

Память и способы её выделения

В случае строкового литерала мы знаем его содержимое во время компиляции, так что текст жёстко закодирован в итоговом исполняемом файле. Это причина того, что строковые литералы являются быстрыми и эффективными. Но эти свойства приходят только из-за неизменяемости строковых литералов. К сожалению, нельзя поместить неопределённый кусок памяти в выполняемый файл для каждого кусочка текста, размер которого неизвестен при компиляции и который может менять свой размер во время выполнения программы.

Чтобы поддерживать изменяемый, увеличивающийся кусок текста типа `String`, ему необходимо выделять память в куче для всего содержимого (объем которого неизвестен во время компиляции). Это означает, что:

- память должна запрашиваться у операционной системы во время выполнения программы,
- необходим способ возврата этой памяти операционной системе, когда мы закончили в программе работу со `String`.

Первая часть сделана нами, когда вызывается `String::from`: эта реализация запрашивает необходимую память. Это является довольно универсальным подходом в языках программирования.

Тем не менее, вторая часть отличается. В языках со *сборщиком мусора*, сборщик отслеживает и очищает память, которая больше не используется и нам не нужно заботиться об этом процессе. Без сборщика мусора, мы отвечаем за определение момента, когда память больше не используется и вызываем код явно возвращающий память, также как когда бы запрашивали её. Корректное выполнение этих действий было исторически сложной проблемой программирования. Если забываем освободить, то теряем память. Если освободим слишком рано, то получим недействительную переменную. Если освободим дважды, то это тоже будет ошибкой. Нужно связать ровно одно **выделение** (allocate) с ровно одним **освобождением** (free).

Rust выбирает другой путь: память автоматически возвращается как только переменная владеющая памятью выходит из области видимости. Вот версия примера с областью видимости из листинга 4-1 использующего тип `String` вместо строкового литерала:

```
{  
    let s = String::from("hello"); // s is valid from this point forward  
  
    // do stuff with s  
}  
// this scope is now over, and s is no  
// longer valid
```

Здесь есть естественная точка в которой можно вернуть память занимаемую `String` обратно в операционную систему: когда переменная `s` уходит из области видимости. Когда переменная выходит из области видимости, Rust вызывает специальную функцию для нас. Данная функция называется `drop` и это место где автор `String` может поместить код для возвращения памяти. Rust вызывает `drop` автоматически на закрывающей фигурной скобке.

Заметьте: Данный шаблон освобождения ресурсов в конце цикла жизни переменной в C++ иногда называется *Resource Acquisition Is Initialization (RAII)*. Функция `drop` в Rust будет вам знакома, если вы уже использовали шаблон RAII.

Этот шаблон оказывает глубокое влияние на способ написания кода в Rust. Сейчас это может казаться простым, но в более сложных ситуациях поведение кода может быть неожиданным, например, когда хочется иметь несколько переменных использующих данные выделенные в куче. Изучим несколько таких ситуаций.

Способы взаимодействия переменных и данных: перемещение

Множество переменных могут взаимодействовать разными способами с одинаковыми данными в Rust. Давайте рассмотрим пример использующий целое число в листинге 4-2.

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

Листинг 4-2: Назначение значения целого числа из переменной `x` в `y`

Возможно мы догадаемся, что делает данный код: "привязать значение `5` к переменной `x`; затем сделать копию значения `x` и привязать его к переменной `y`". Теперь у нас две переменные, `x` и `y`, обе равны `5`. Действительно тут происходит именно это потому что целые числа являются простыми значениями с

известным, фиксированным размером и эти два значения **5** размещаются в стеке.

Теперь рассмотрим версию с типом **String**:

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

Выглядит очень похоже на предыдущий код, настолько что мы могли бы подумать, что этот код работает таким же образом: то есть вторая строка могла бы сделать копию значения **s1** и привязать его к **s2**. Но это не совсем то, что происходит.

Посмотрим на рисунок 4-1 и разберём, что происходит со **String** под капотом. Тип **String** состоит из трёх частей показанных слева: указатель на память занятую содержимым строки, длину и ёмкость. Данная группа данных сохраняется в стеке. Справа память в куче, которая хранит содержимое строки.

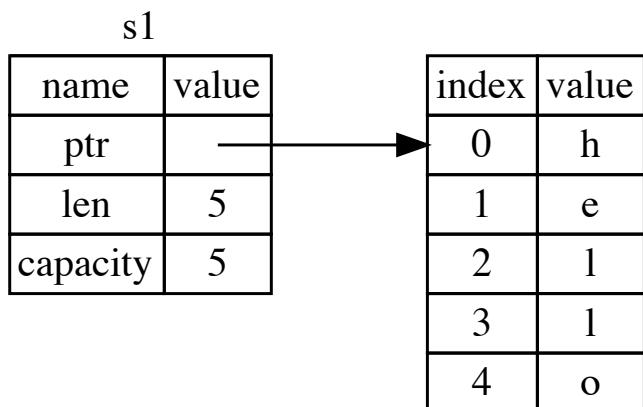
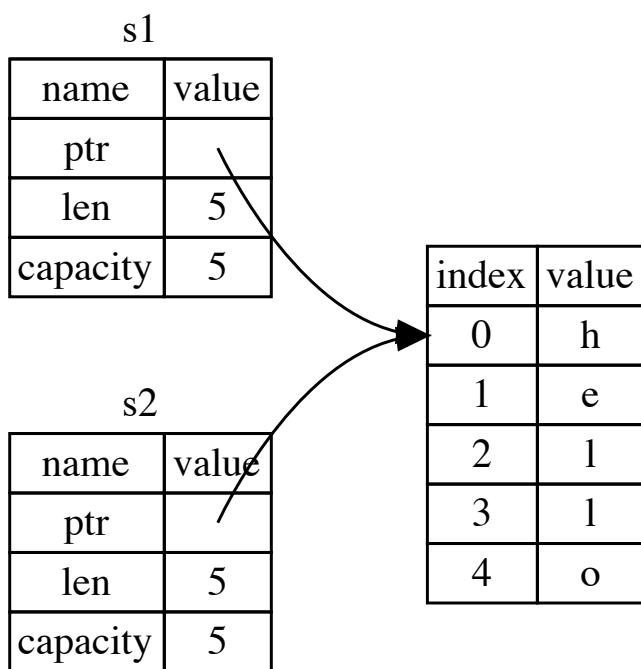


Рисунок 4-1: Представление в памяти строки **String** содержащей значение **"hello"** привязанное к **s1**

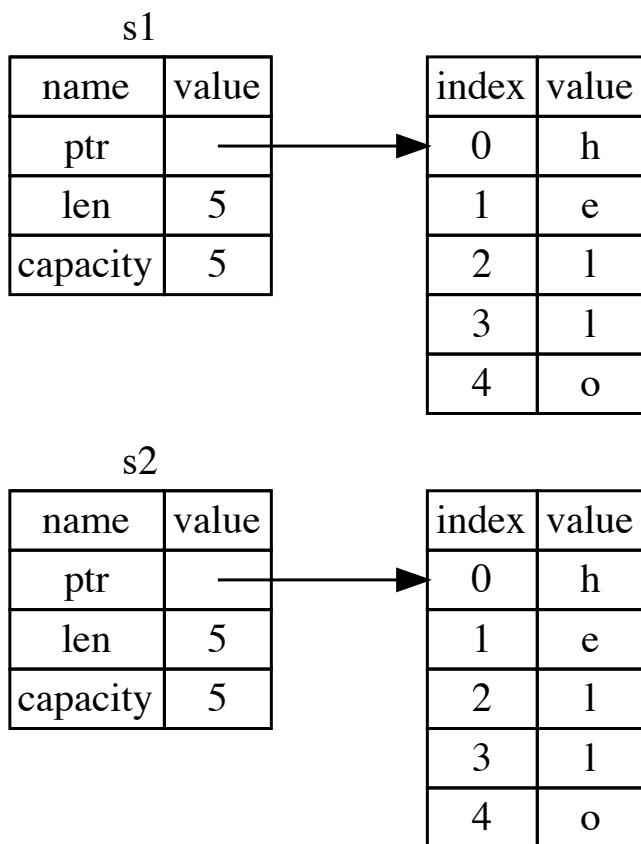
Длина - это сколько байт памяти использует содержимое **String** в данный момент. Ёмкость - это общее количество байт памяти, которые **String** получила от операционной системы. Разница между длиной и ёмкостью имеет значение, но не в данном контексте, сейчас можно игнорировать ёмкость.

Когда мы назначили **s1** переменной **s2**, то данные типа **String** были скопированы, что означает мы скопировали указатель, длину и ёмкость, которые находятся в стеке. Мы не копируем данные в куче на которые ссылается указатель. Другими словами данные представленные в памяти выглядят как на картинке 4-2.



Картина 4-2: Представление в памяти переменной `s2`, которая является копией указателя, длины и ёмкости переменной `s1`

Представление *НЕ* выглядит как на картинке 4-3, при котором память могла бы выглядеть так, как если бы Rust ешё скопировал и сами данные в куче. Если Rust сделал бы это, то операция `s2 = s1` могла бы быть очень дорогостоящей в смысле производительности: представьте если бы копируемые данные в куче были очень большими.



Картинка 4-3: Другая возможность того, как можно было бы сделать при `s2 = s1`, если бы Rust также копировал бы данные в куче

Ранее мы сказали, что когда переменная выходит из области видимости, Rust автоматически вызывает функцию `drop` и очищает память кучи для данной переменной. Но картинка 4-2 показывает, что теперь оба указателя указывают на одно и тоже место. Это проблема: когда переменная `s2` и переменная `s1` выходят из области видимости они обе будут пытаться освободить одну и ту же память в куче. Это известно как "ошибка двойного освобождения", *double free*, и является одной из ошибок безопасности памяти, упоминаемых ранее. Освобождение памяти дважды может привести к повреждению памяти, что потенциально может привести к уязвимостям безопасности.

Чтобы убедиться в безопасности использования памяти, расскажем детали того, что происходит в данной ситуации в Rust. Вместо попытки копировать выделенную память, Rust считает что переменная `s1` больше недействительна и таким образом в Rust ничего не нужно освобождать позже, когда `s1` покинет область видимости. Проверьте что происходит при попытке использования переменной `s1` после того как `s2` создана, это не сработает:

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

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

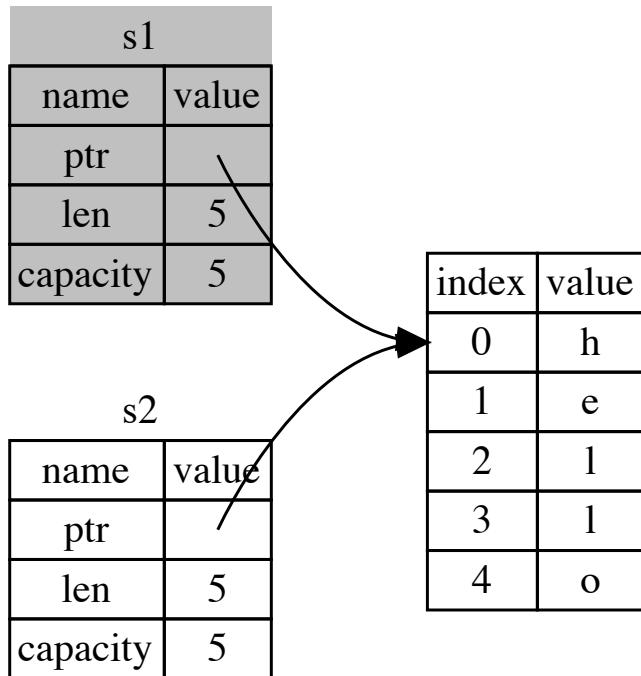


Вы получите ошибку ниже, потому что Rust не даст использовать недействительную ссылку `s1`:

```
$ cargo run
Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0382]: borrow of moved value: `s1`
--> src/main.rs:5:28
   |
2 |     let s1 = String::from("hello");
   |         -- move occurs because `s1` has type `String`, which does not
implement the `Copy` trait
3 |     let s2 = s1;
   |             -- value moved here
4 |
5 |     println!("{} , world!", s1);
   |             ^^^ value borrowed here after move
   |
= note: this error originates in the macro `$crate::format_args_nl` (in
Nightly builds, run with -Z macro-backtrace for more info)

For more information about this error, try `rustc --explain E0382`.
error: could not compile `ownership` due to previous error
```

Если вы слышали термины "поверхностное копирование" *shallow copy* и "глубокое копирование" *deep copy* в других языках, то концепция копирования указателя, длины и ёмкости без копирования самих данных в куче, возможно выглядит как создание "поверхностной копии". Но так как Rust делает первую переменную недействительной вместо создания поверхности копии, то такое действие известно как "перемещение" *move*. В данном примере, мы бы сказали, что `s1` была *перемещена* в переменную `s2`. То что происходит на самом деле показано на картинке 4-4.



Картина 4-4: представление памяти после того как `s1` была сделана не действительной

Это решает нашу проблему! Действительной остаётся только переменная `s2`, когда она выходит из области видимости, то она одна будет освобождать память в куче.

Дополнительно, присутствует выбор дизайна, который подразумевает следующее: Rust никогда не будет автоматически создавать "глубокие" копии ваших данных. Следовательно, любое такое *автоматическое* копирование, можно считать недорогим с точки зрения производительности во время выполнения.

Способы взаимодействия переменных и данных: клонирование

Если мы хотим сделать глубокое копирование данных в куче для типа `String`, а не только данных в стеке, то мы можем использовать общий метод называемый `clone`. Мы обсудим его синтаксис в Главе 5, но так как методы являются общими особенностями во многих языках программирования, то вы возможно уже видели их ранее.

Вот пример метода `clone` в действии:

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

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

Код работает отлично и явно выполняет поведение, показанное на картинке 4-3, где данные в куче *действительно скопированы*.

Когда вы видите вызов `clone`, то вы знаете о выполнении некоторого кода, который может быть дорогим. В то же время использование `clone` является визуальным индикатором того, что тут происходит что-то нестандартное (глубокое копирование вместо обыденного перемещения).

Стековые данные: Копирование

Это ещё одна особенность о которой мы ещё не говорили. Этот код, часть которого была показана ранее в листинге 4-2, использует целые числа. Этот код работает и не имеет ошибок:

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

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

Но данный код, кажется противоречит тому, что мы только что изучили: тут не нужно вызывать `clone`, но `x` является все ещё действительной переменной и не перемещена в `y`.

Причина такого поведения в том, что типы вроде целых чисел, размер которых известен во время компиляции, сохраняются полностью в стеке, поэтому такое копирование значений является быстрым. Это означает, что нет причин по которым мы бы хотели переменную `x` оставлять действительной после создания переменной `y`. Другими словами, здесь нет разницы между глубоким и поверхностным копированием, поэтому вызов `clone` не будет делать ничего отличного от обычного поверхностного копирования, и мы можем оставить это как есть.

В Rust есть специальная аннотация называемая типаж `Copy`, который можно применить на типы вроде целых чисел, размещенных в стеке (мы поговорим про типажи в Главе 10). Если тип имеет типаж `Copy`, то старая переменная этого типа все ещё может быть использована после её перемещения в новую. Rust не позволит аннотировать тип с типажом `Copy`, если тип или любая его часть имеет реализацию типажа `Drop`. Если типу нужно делать что-то особенное, когда значение уходит из области видимости и мы добавляем аннотацию `Copy` к данному типу, мы получим ошибку компиляции. Для изучения как добавлять аннотацию

Copy к вашему типу, смотрите раздел "Выводимые типажи" в приложении C.

Так какие типы имеют типаж **Copy**? Можно проверить документацию любого типа для уверенности, но как общее правило любая группа простых, скалярных значений может быть с типажом **Copy**, и ничего из типов, которые требуют выделения памяти в куче или являются некоторой формой ресурсов, не имеет типажа **Copy**. Вот некоторые типы, которые реализуют типаж **Copy**:

- все целочисленные типы, такие как **u32**,
 - логический тип данных **bool**, возможные значения которого **true** и **false**,
 - все числа с плавающей запятой такие как **f64**,
 - символьный тип **char**,
 - кортежи, но только если они содержат типы, которые также реализуют **Copy**.
- Например, **(i32, i32)** будет с **Copy**, но кортеж **(i32, String)** уже нет.

Владение и функции

Семантика передачи значений в функции является похожей на назначение значения переменной. Передача переменной в функцию в качестве входного параметра будет перемещать или копировать её значение, точно также как это делает операция присвоения. Пример в листинге 4-3 с некоторыми аннотациями, показывает на каких этапах переменные появляются и исчезают из области видимости.

Файл: src/main.rs

```

fn main() {
    let s = String::from("hello"); // s comes into scope

    takes_ownership(s);          // s's value moves into the function...
                                // ... and so is no longer valid here

    let x = 5;                   // x comes into scope

    makes_copy(x);              // x would move into the function,
                                // but i32 is Copy, so it's okay to still
                                // use x afterward

} // Here, x goes out of scope, then s. But because s's value was moved,
// nothing
// special happens.

fn takes_ownership(some_string: String) { // some_string comes into scope
    println!("{}", some_string);
} // Here, some_string goes out of scope and `drop` is called. The backing
// memory is freed.

fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{}", some_integer);
} // Here, some_integer goes out of scope. Nothing special happens.

```

Листинг 4-3: Функции с комментариями про владение и область видимости

Если попытаться использовать `s` после вызова `takes_ownership`, Rust выдаст ошибку времени компиляции. Такие статические проверки защищают от ошибок. Попробуйте добавить код в `main`, который использует переменную `s` и `x`, чтобы увидеть где их можно использовать и где правила владения предотвращают их использование.

Возвращение значений и область видимости

Возвращение значений также может перемещать владение. Листинг 4-4 является примером с похожими комментариями, что даны в листинге 4-3.

Файл: src/main.rs

```

fn main() {
    let s1 = gives_ownership();                      // gives_ownership moves its return
                                                       // value into s1

    let s2 = String::from("hello");                  // s2 comes into scope

    let s3 = takes_and_gives_back(s2);               // s2 is moved into
                                                       // takes_and_gives_back, which also
                                                       // moves its return value into s3
} // Here, s3 goes out of scope and is dropped. s2 was moved, so nothing
  // happens. s1 goes out of scope and is dropped.

fn gives_ownership() -> String {                  // gives_ownership will move its
                                                       // return value into the
function                                         // that calls it

    let some_string = String::from("yours"); // some_string comes into scope

    some_string                                // some_string is returned and
                                                       // moves out to the calling
                                                       // function
}

// This function takes a String and returns one
fn takes_and_gives_back(a_string: String) -> String { // a_string comes into
                                                       // scope

    a_string // a_string is returned and moves out to the calling function
}

```

Листинг 4-4: Перемещение владения и возврат значений

Владение переменной каждый раз следует похожему шаблону: присваивание значения другой переменной перемещает его. Когда переменная содержащая данные в куче выходит из области видимости, содержимое в куче будет очищено функцией `drop`, если только данные не были перемещены во владение другой переменной.

Приём во владение и затем возвращение владения из каждой функции немного утомительно. А что если мы позволим функции использовать значение, но не забирать его во владение? Весьма раздражает, если все, что мы передаём, также должно быть возвращено обратно. И это ещё надо будет делать в добавок к обслуживанию любых данных, которые мы также можем захотеть вернуть из тела функции.

Есть возможность возвращать несколько значений используя кортеж, как в листинге 4-5.

Файл: src/main.rs

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

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

    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String

    (s, length)
}
```

Листинг 4-5: возврат владения параметров

Но это слишком много церемоний и много работы для концепции, которая должна быть общей. К счастью для нас, в Rust есть функциональность для данной концепции, называемая *ссылка*.

Ссылочные переменные и заимствование

Основная проблематика в подходе с использованием кортежа в листинге 4-5 заключается в том, что мы должны вернуть `String` в вызывающую функцию, чтобы мы могли использовать `String` после вызова функции `calculate_length`, потому что `String` была перемещена в функцию `calculate_length`. Вместо этого мы можем предоставить ссылку на значение `String`. Ссылка похожа на указатель в том смысле, что это адрес, по которому мы можем получить доступ к данным, хранящимся по этому адресу, принадлежащему какой-либо другой переменной. В отличие от указателя, ссылка гарантированно указывает на допустимое значение определённого типа. Вот как вы могли бы определить и использовать функцию `calculate_length` которая имеет ссылку на объект в качестве параметра вместо того, чтобы владеть значением:

Файл: src/main.rs

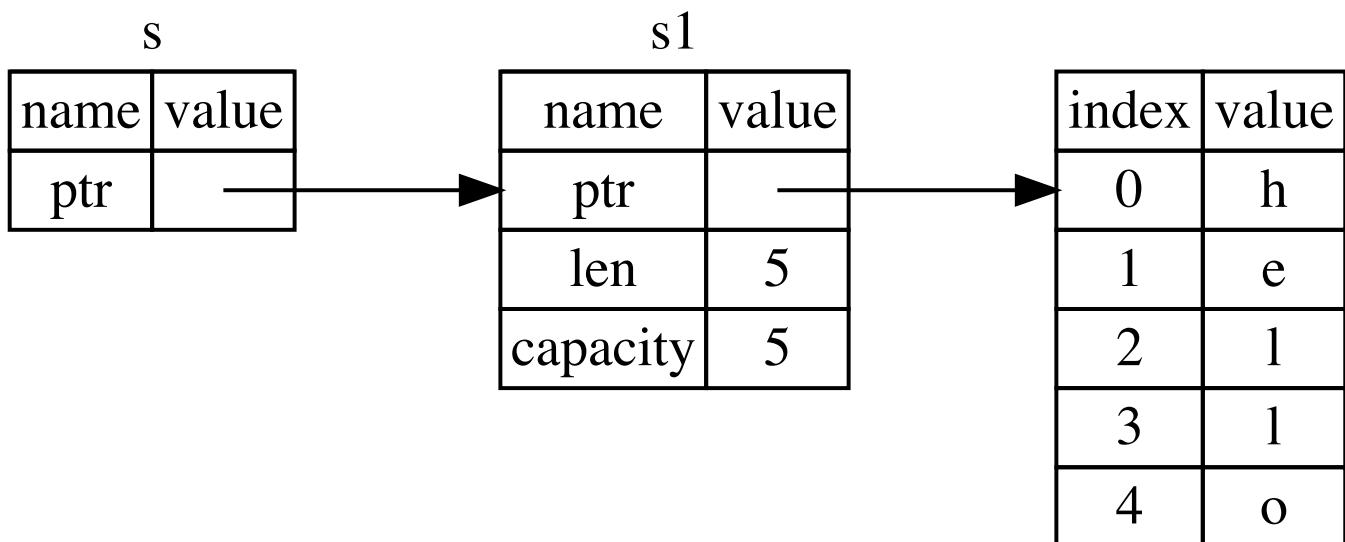
```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

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

Во-первых, обратите внимание, что весь код кортежа в объявлении переменной и возвращаемое значение функции исчезли. Во-вторых, обратите внимание, что мы передаём `&s1` в `calculate_length` и в его определении мы берём `&String` а не `String`. Эти амперсанды представляют собой **ссылки**, и они позволяют вам ссылаться на некоторое значение, не принимая владение им. Рисунок 4-5 изображает эту концепцию.



Картина 4-5: Диаграмма для `&String s` указывающей на `String s1`

Заметьте: Операцией обратной созданию ссылки используя `&` является операция *разыменования*, которая выполняется с помощью оператора разыменования `*`. Вы увидите использование этого оператора в главе 8 и мы обсудим детали ещё в главе 15.

Давайте подробнее рассмотрим механизм вызова функции:

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

`&s1` позволяет нам создать ссылку, которая *ссылается* на значение `s1`, но не владеет им. Поскольку он не владеет им, значение на которое он указывает, не будет удалено, когда ссылка перестанет использоваться.

Сигнатура функции использует `&` для индикации того, что тип параметра `s` является ссылкой. Добавим объясняющие комментарии:

```
fn calculate_length(s: &String) -> usize { // s is a reference to a String
    s.len()
} // Here, s goes out of scope. But because it does not have ownership of
// what
// it refers to, it is not dropped.
```

Область действия, в которой `s` такая же, как и область действия любого параметра функции, но значение, на которое указывает ссылка, не удаляется, когда `s` перестаёт использоваться, потому что `s` не имеет владельца. Когда функции имеют ссылки в качестве параметров вместо фактических значений, нам не нужно возвращать значения, чтобы вернуть право собственности, потому что мы никогда не владели ими.

Мы называем действие создания ссылки *заимствованием*. Как и в реальной жизни, если человек чем-то владеет, вы можете это у него позаимствовать. Когда вы закончите, вы должны вернуть его. Вы им не владеете.

А что произойдёт, если попытаться изменить то, что было позаимствовано? Попробуйте код листинга 4-6 Предупреждаем, этот код не сработает!

Файл: src/main.rs

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

    change(&s);
}

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



Listing 4-6: Попытка модификации заимствованной переменной

Вот ошибка:

```
$ cargo run
Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0596]: cannot borrow `*some_string` as mutable, as it is behind a `&` reference
--> src/main.rs:8:5
 |
7 | fn change(some_string: &String) {
|             ----- help: consider changing this to be a
mutable reference: `&mut String`
8 |     some_string.push_str(", world");
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ `some_string` is a `&` reference, so
the data it refers to cannot be borrowed as mutable

For more information about this error, try `rustc --explain E0596`.
error: could not compile `ownership` due to previous error
```

Как и переменные являются не изменяемыми по умолчанию, ссылочные переменные тоже являются неизменяемыми. Т.е. нельзя изменять данные по ссылке.

Изменяемые ссылочные переменные

Мы можем исправить код из листинга 4-6, чтобы позволить нам изменять заимствованное значение с помощью всего лишь нескольких небольших настроек, которые вместо этого используют *изменяемую ссылку*:

Файл: 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 мы меняем `s` на `mut`. Затем мы создаём изменяемую ссылку с помощью `&mut s` у которой вызываем `change` и обновляем сигнатуру функции, чтобы принять изменяемую ссылку с помощью `some_string: &mut String`. Это даёт понять, что `change` изменит значение, которое она заимствует.

У изменяемых ссылок есть одно большое ограничение: вы можете иметь только одну изменяемую ссылку на определённый фрагмент данных одновременно. Этот код, который пытается создать две изменяемые ссылки на `s`, потерпит неудачу:

Файл: src/main.rs

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

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

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



Описание ошибки:

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

For more information about this error, try `rustc --explain E0499`.
error: could not compile `ownership` due to previous error
```

Эта ошибка говорит о том, что этот код недействителен, потому что мы не можем заимствовать **s** как изменяемые более одного раза в один момент. Первое изменяемое заимствование находится в **r1** и должно длиться до тех пор, пока оно не будет использовано в **println!**, но между созданием этой изменяемой ссылки и её использованием мы попытались создать другую изменяемую ссылку в **r2**, которая заимствует те же данные, что и **r1**.

Ограничение, предотвращающее одновременное использование нескольких изменяемых ссылок на одни и те же данные, допускает изменение, но очень контролируемым образом. Это то, с чем борются новые Rustaceans, потому что большинство языков позволяют изменять значение, когда захотите. Преимущество этого ограничения заключается в том, что Rust может предотвратить гонки данных во время компиляции. Гонка данных похожа на состояние гонки и происходит, когда возникают следующие три сценария:

- два или больше указателей используют те же данные в одно и тоже время,
- минимум один указатель используется для записи данных,
- отсутствуют механизмы для синхронизации доступа к данным.

Гонки данных вызывают неопределенное поведение и их может быть сложно диагностировать и исправить, когда вы пытаетесь отследить их во время выполнения; Rust предотвращает эту проблему, отказываясь компилировать код с гонками данных!

Как всегда, мы можем использовать фигурные скобки для создания новой области видимости, позволяющей использовать несколько изменяемых ссылок, но не одновременно:

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

{
    let r1 = &mut s;
} // r1 goes out of scope here, so we can make a new reference with no
   problems.

let r2 = &mut s;
```

Rust применяет аналогичное правило для комбинирования изменяемых и неизменяемых ссылок. Этот код приводит к ошибке:

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

let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM

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



Ошибка:

```
$ cargo run
Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
immutable
--> src/main.rs:6:14
|
4 |     let r1 = &s; // no problem
|             -- immutable borrow occurs here
5 |     let r2 = &s; // no problem
6 |     let r3 = &mut s; // BIG PROBLEM
|             ^^^^^^ mutable borrow occurs here
7 |
8 |     println!("{}, {}, and {}", r1, r2, r3);
|                         -- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` due to previous error
```

Ух! У нас *также* не может быть изменяемой ссылки, пока у нас есть неизменяемая ссылка на то же значение. Пользователи неизменной ссылки не ожидают, что значение внезапно изменится! Однако разрешены множественные неизменяемые ссылки, потому что никто, кто просто читает данные, не может повлиять на чтение данных кем-либо другим.

Обратите внимание, что область действия ссылки начинается с того места, где она была введена, и продолжается до последнего использования этой ссылки. Например, этот код будет компилироваться, потому что последнее использование неизменяемых ссылок `println!`, происходит до того, как вводится изменяемая ссылка:

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

let r1 = &s; // no problem
let r2 = &s; // no problem
println!("{} and {}", r1, r2);
// variables r1 and r2 will not be used after this point

let r3 = &mut s; // no problem
println!("{}", r3);
```

Области неизменяемых ссылок `r1` и `r2` заканчиваются после `println!` где они использовались в последний раз, то есть до создания изменяемой ссылки `r3`. Эти области не пересекаются, поэтому этот код разрешён. Способность компилятора сообщить, что ссылка больше не используется в точке до конца области видимости, называется *нелексическим временем жизни* (сокращённо NLL), и вы можете прочитать об этом больше в [The Edition Guide](#).

Не смотря на то, что ошибки заимствования могут иногда вызывать разочарование, помните, что компилятор Rust указывает про потенциальную проблему на ранних этапах (во время компиляции, а не во время выполнения) и показывает точно, где находится проблема. Так что вам не нужно отслеживать, почему ваши данные не соответствуют вашим ожиданиям.

Недействительные ссылки

В языках с указателями весьма легко ошибочно создать недействительную, висячую (*dangling*) ссылку. Ссылку указывающую на участок памяти, который мог быть передан кому-то другому, путём освобождения некоторой памяти при сохранении указателя на эту память. Rust компилятор гарантирует, что ссылки никогда не станут недействительными: если у вас есть ссылка на какие-то данные, компилятор обеспечит что эти данные не выйдут из области видимости прежде, чем из области видимости исчезнет ссылка.

Попытаемся смоделировать подобную, висячую ссылку, появление которой компилятор предотвратит:

Файл: src/main.rs

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

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

    &s
}
```



Здесь ошибка:

```
$ cargo run
Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0106]: missing lifetime specifier
--> src/main.rs:5:16
   |
5 | fn dangle() -> &String {
   |           ^ expected named 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 using the ``static`` lifetime
   |
5 | fn dangle() -> &'static String {
   |           ~~~~~~
For more information about this error, try `rustc --explain E0106`.
error: could not compile `ownership` due to previous error
```

Эта ошибка сообщает об ещё не освещённой нами возможности языка Rust: *времени жизни переменной (lifetime)*. Мы расскажем подробнее об этой возможности в Главе 10. Но если вы проигнорируете раздел ошибки который говорит о времени жизни, то все ещё будете способны найти ключ к тому, почему этот код является проблемным:

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

Давайте пристальней рассмотрим, что же происходит на каждой стадии работы кода функции `dangle`:

Файл: src/main.rs

```
fn dangle() -> &String { // dangle returns a reference to a String  
    let s = String::from("hello"); // s is a new String  
  
    &s // we return a reference to the String, s  
} // Here, s goes out of scope, and is dropped. Its memory goes away.  
// Danger!
```



По причине того, что переменная `s` создана внутри функции `dangle`, то при завершении `dangle` содержимое памяти для `s` будет удалено из памяти. Но мы пытаемся вернуть ссылку на эту память. Это означает, что данная ссылка могла бы указывать на недействительную `String`. Это плохо! Rust не позволит нам этого сделать.

Решением является вернуть непосредственно `String`:

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

Это решение работает без проблем. Владение перемещено наружу и ничего не удаляется из памяти.

Правила работы с ссылками

Давайте повторим все, что мы обсудили про ссылки:

- в один момент времени, может существовать либо одна изменяемая ссылочная переменная, либо любое количество неизменяемых ссылочных переменных,
- все ссылки должны быть действительными.

В следующей главе мы рассмотрим другой тип ссылочных переменных - срезы.

Срезы

Другим типом данных, который не забирает во владение данные является *срез* (slice). Срез позволяет ссылаться на смежную последовательность элементов из коллекции, вместо полной коллекции.

Рассмотрим небольшую программную проблему: необходимо написать функцию, входным параметром которой является строка, а выходным значением функции является первое слово, которое будет найдено в этой строке. Если функция не находит пробелы, она возвращает полную строку.

Давайте подумаем над сигнатурой этой функции:

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

Функция `first_word` имеет входной параметр типа `&String`. Нам не нужно владение переменной, так что это нормально. Но что мы должны вернуть? На самом деле у нас нет способа выразить *часть строки*. Тем не менее, для решения задачи мы можем найти индекс конца слова в строке используя пробел. Попробуем сделать как на листинге 4-7:

Файл: src/main.rs

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

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

    s.len()
}
```

Листинг 4-7: Пример функции `first_word`, которая возвращает значение индекса пробела внутри строкового параметра `String`

Для того, чтобы найти пробел в строке, мы превратим `String` в массив байт, используя метод `as_bytes` и пройдём по `String` элемент за элементом, проверяя является ли значение пробелом.

```
let bytes = s.as_bytes();
```

Далее, мы создаём *итератор* по массиву байт используя метод `iter`:

```
for (i, &item) in bytes.iter().enumerate() {
```

Мы изучим итераторы более детально в Главе 13. Сейчас, достаточно понять, что метод `iter` при каждом вызове возвращает следующий элемент коллекции, а метод `enumerate` обворачивает результаты работы метода `iter` и возвращает каждый элемент упакованным в кортеж. Первый элемент этого кортежа возвращён из `enumerate` и является индексом, а второй элемент - ссылка на элемент коллекции которую предоставил метод `iter`. Такой способ перебора элементов массива является более удобным - не надо считать индекс самостоятельно.

Так как метод `enumerate` возвращает кортеж, мы можем использовать шаблон деструктуризации кортежа, как и везде в Rust. Так в цикле `for`, мы указываем шаблон `(i, &item)` который распакует значения кортежа в `i` для хранения индекса из кортежа и в `&item` который сразу же возьмёт по ссылке байт символа из кортежа. Мы используем `&` в шаблоне по причине того, что метод `.iter().enumerate()` возвращает нам ссылку на элемент.

Внутри цикла `for`, ищем байт представляющий пробел используя синтаксис байт литерала. Если пробел найден, возвращается его позиция. Иначе, возвращается длина строки `s.len()`:

```
if item == b' ' {
    return i;
}
s.len()
```

Теперь у нас есть способ узнать индекс байта указывающего на конец первого слова в строке, но есть проблема. Мы возвращаем сам `usize`, но это число имеет значение только в контексте `&String`. Другими словами, поскольку это значение отдельное от `String`, то нет гарантии, что оно все ещё будет действительным в будущем. Рассмотрим программу из листинга 4-8, которая использует функцию `first_word` листинга 4-7.

Файл: src/main.rs

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

    let word = first_word(&s); // word will get the value 5

    s.clear(); // this empties the String, making it equal to ""

    // word still has the value 5 here, but there's no more string that
    // we could meaningfully use the value 5 with. word is now totally
    // invalid!
}
```

Listing 4-8: Сохранение результата вызова функции `first_word`, а затем изменение содержимого `String`

Данная программа компилируется без ошибок и будет успешно работать, даже после того как мы воспользуемся переменной `word` после вызова `s.clear()`. Так как значение `word` совсем не связано с состоянием переменной `s`, то `word` сохраняет своё значение `5` без изменений. Мы могли бы использовать `5` вместе с переменной `s` и попытаться извлечь первое слово из строки, но это приведёт к ошибке, потому что содержимое `s` изменилось после того как мы сохранили `5` в переменной `word` (стало пустой строкой в вызове `s.clear()`).

Необходимость беспокоиться о том, что индекс в переменной `word` не синхронизируется с данными в переменной `s` является утомительной и подверженной ошибкам! Управление этими индексами становится ещё более хрупким, если мы напишем функцию `second_word`. Её сигнатура могла бы выглядеть так:

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

Теперь мы отслеживаем *начальный* и *конечный* индексы, у нас стало ещё больше значений, которые рассчитаны на основе данных о содержимом в определённом состоянии строки, но которые также совсем не привязаны к данному состоянию. Теперь есть уже три не связанные переменные, которые необходимо синхронизировать.

К счастью в Rust есть решение данной проблемы: строковые срезы.

Строковые срезы

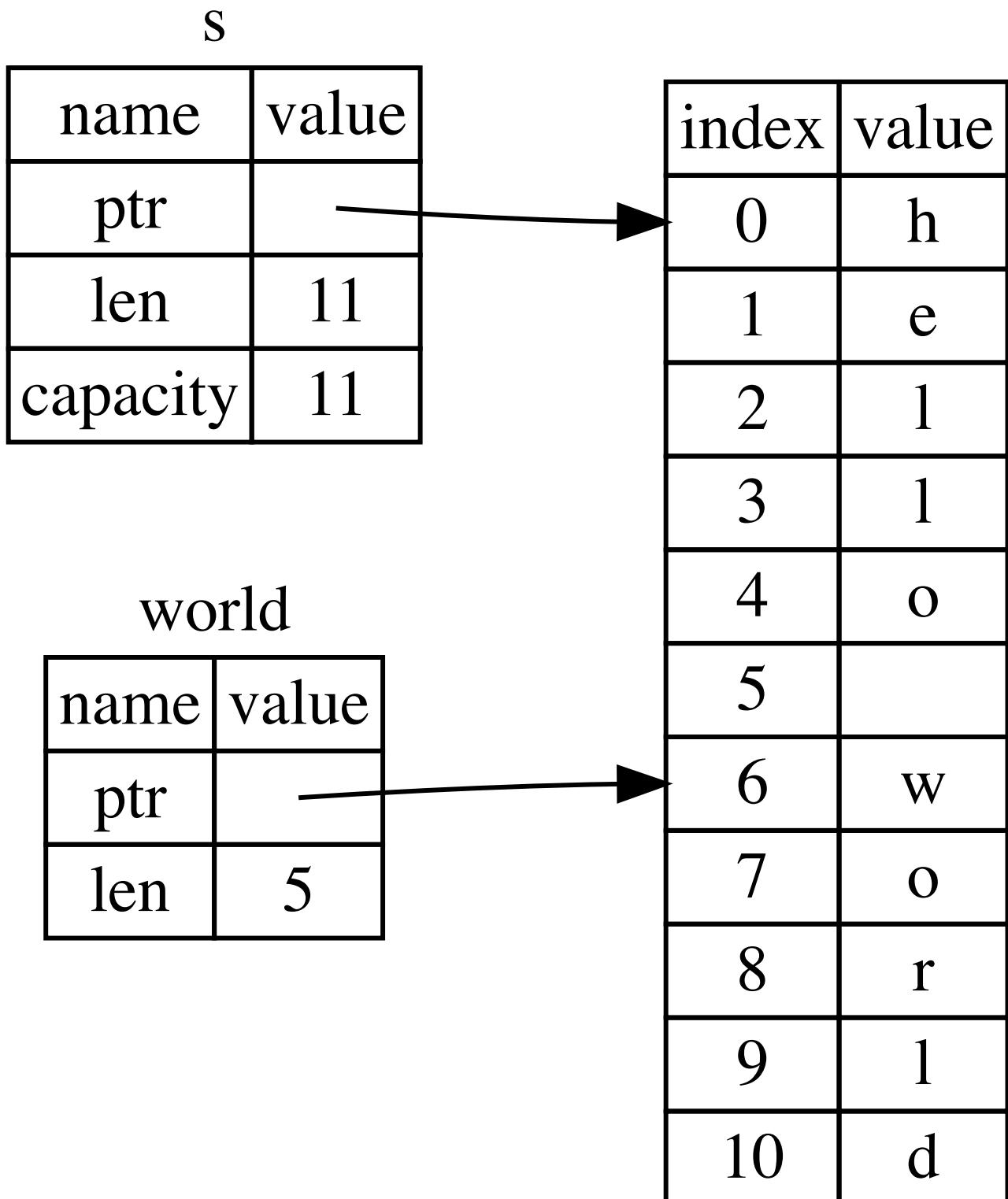
Строковый срез - это ссылка на часть строки `String` и он выглядит следующим образом:

```
let s = String::from("hello world");  
  
let hello = &s[0..5];  
let world = &s[6..11];
```

Эта инициализация похожа на создание ссылки на переменную `String`, но с дополнительным условием - указанием отрезка `[0..5]`. Вместо ссылки на всю `String`, срез ссылается на её часть.

Мы можем создавать срезы, используя диапазон в квадратных скобках указывая `[starting_index..ending_index]`, где `starting_index` означает первую позицию в срезе, а `ending_index` на единицу больше, чем последняя позиция. Во внутреннем представлении, срез хранит начальную позицию и длину среза, которая соответствует числу `ending_index` минус `starting_index`. Таким образом, в примере `let world = &s[6..11];`, переменная `world` будет срезом, который содержит ссылку на 7-ой байт в `s` со значением длины равным 5.

Рисунок 4-12 отображает это на диаграмме.

Рисунок 4-6: Строковый срез ссылается на часть **String**

Возможно использовать синтаксис диапазона `..` и другим способом. Если хочется начать с начального индекса (с нуля), то можно убрать число перед двоеточием. Другими словами, это эквивалентно:

```
let s = String::from("hello");  
  
let slice = &s[0..2];  
let slice = &s[..2];
```

Таким же образом, если срез включает последний байт строки `String`, можно убрать завершающее число. Это эквивалентно:

```
let s = String::from("hello");  
  
let len = s.len();  
  
let slice = &s[3..len];  
let slice = &s[3..];
```

Также можно не указывать оба значения, чтобы получить срез всей строки. Это эквивалентно:

```
let s = String::from("hello");  
  
let len = s.len();  
  
let slice = &s[0..len];  
let slice = &s[..];
```

Внимание: Индексы среза строк должны соответствовать границам UTF-8 символов. Если вы попытаетесь получить срез нарушая границы символа в котором больше одного байта, то вы получите ошибку времени исполнения. В рамках этой главы мы будем предполагать только ASCII кодировку. Более детальное обсуждение UTF-8 находится в секции "[Сохранение текста с кодировкой UTF-8 в строках](#)" Главы 8.

Давайте используем полученную информацию и перепишем метод `first_word` так, чтобы он возвращал срез. Для обозначения типа "срез строки" существует запись `&str`:

Файл: 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 способом аналогичным тому, как мы это делали в листинге 4-7: ищем индекс первого вхождения пробела, когда пробел найден, возвращается строковый срез, используя начало строки в качестве начального индекса и индекс пробела в качестве конечного индекса среза.

Теперь, вызвав метод `first_word`, мы получим одно единственное значение, которое привязано к нижележащим данным. Значение, которое составлено из ссылки на начальную точку среза и количества элементов в срезе.

Аналогичным образом можно переписать и второй метод `second_word`:

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

Теперь есть простое API, работу которого гораздо сложнее испортить, потому что компилятор обеспечивает нам то, что ссылки на `String` останутся действительными. Помните ошибку в программе листинга 4-8, когда мы получили индекс конца первого слова, но затем очистили строку, так что она стала недействительной? Тот код был логически некорректным, хотя не показывал никаких ошибок. Проблемы возникли бы позже, если бы мы попытались использовать индекс первого слова для пустой строки. Срезы делают невозможной данную ошибку и позволяют понять о наличии проблемы гораздо раньше. Так, использование версии метода `first_word` со срезом вернёт ошибку компиляции:

Файл: src/main.rs

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

    let word = first_word(&s);

    s.clear(); // error!

    println!("the first word is: {}", word);
}
```



Ошибка компиляции:

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

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` due to previous error
```

Напомним вам правила заимствования: если у нас есть неизменяемая ссылка на что-либо, то нельзя взять изменяемую ссылку для этого чего-то. Так как методу `clear` требуется обрезать `String`, ему нужно получить изменяемую ссылку. Rust не позволяет это сделать и компиляции не проходит. Rust не только упростил использование нашего API, но и исключил целый класс ошибок во время компиляции!

Строковые литералы это срезы

Напомним, что мы говорили о строковых литералах, хранящихся внутри бинарного файла. Теперь, когда мы знаем чем являются срезы, мы правильно понимаем что такое строковые литералы:

```
let s = "Hello, world!";
```

Тип `s` здесь является `&str` срезом, указывающим на конкретное место в бинарном файле программы. Это также объясняет, почему строковый литерал является неизменяемым, потому что тип `&str` это неизменяемая ссылка.

Строковые срезы как параметры

Знание о том, что можно брать срезы строковых литералов и `String` строк приводит к ещё одному улучшению метода `first_word`, улучшению его сигнатуры:

```
fn first_word(s: &String) -> &str {
```

Более опытные разработчики Rust написали бы сигнатуру из листинга 4-9, потому что она позволяет использовать одну функцию для значений обоих типов `&String` и `&str`.

```
fn first_word(s: &str) -> &str {
```

Листинг 4-9: Улучшение функции `first_word` используя тип строкового среза для параметра `s`

Если есть строковый срез, то можно его передавать напрямую. Если есть `String`, можно передавать срез полностью всей строки `String`. Определение функции принимающей строковый срез вместо ссылки на `String` делает API более общим и полезным без потери функциональности:

Файл: src/main.rs

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

    // `first_word` works on slices of `String`s, whether partial or whole
    let word = first_word(&my_string[0..6]);
    let word = first_word(&my_string[..]);
    // `first_word` also works on references to `String`s, which are
    // equivalent to whole slices of `String`s
    let word = first_word(&my_string);

    let my_string_literal = "hello world";

    // `first_word` works on slices of string literals, whether partial or
    // whole
    let word = first_word(&my_string_literal[0..6]);
    let word = first_word(&my_string_literal[..]);

    // Because string literals *are* string slices already,
    // this works too, without the slice syntax!
    let word = first_word(my_string_literal);
}
```

Другие срезы

Как вы могли бы представить, строковые срезы относятся к строкам. Но также есть более общий тип среза. Рассмотрим массив:

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

Подобно тому как мы хотели бы ссылаться на часть строки, мы можем захотеть ссылаться на часть массива. Мы можем делать это вот так:

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

let slice = &a[1..3];

assert_eq!(slice, &[2, 3]);
```

Данный срез имеет тип `&[i32]`. Он работает таким же образом, как и строковый срез, сохраняя ссылку на первый элемент и длину. Вы будете использовать данную разновидность среза для всех видов коллекций. Мы обсудим коллекции детально,

когда будем говорить про векторы в Главе 8.

Итоги

Концепции владения, заимствования и срезов обеспечивают защиту использования памяти в Rust. Rust даёт вам возможность контролировать использование памяти тем же способом, как другие языки системного программирования, но дополнительно предоставляет возможность автоматической очистки данных, когда их владелец покидает область видимости функции. Это означает, что не нужно писать и отлаживать дополнительный код, чтобы добиться такого контроля.

Владение влияет на множество других частей и концепций языка Rust. Мы будем говорить об этих концепциях на протяжении оставшихся частей книги. Давайте перейдём к Главе 5 и рассмотрим группировку частей данных в структуры `struct`.

Использование структур для объединения логически связанных данных

Структура, struct - от английского *structure* - это пользовательский тип данных, который позволяет назвать и упаковать вместе несколько связанных значений, которые составляют логическую группу. Если вы знакомы с объектно-ориентированными языками, то *struct* будет напоминать вам на атрибуты данных объекта. В этой главе мы сравним и сопоставим кортежи со структурами, продемонстрируем, как использовать структуры. Так же мы обсудим, как создавать ассоциированные с структурой функции и методы структуры определяющие поведение данных, связанных со структурой. Структуры и перечисления (обсуждаемые в Главе 6) - это строительные блоки для создания новых типов данных, диктуемых бизнес-областью вашей программы, которые позволяют в полной мере воспользоваться преимуществом проверки типов во время компиляции в Rust.

Определение и инициализация структур

Структуры похожи на кортежи, рассмотренные в разделе "Тип Кортеж", так как оба хранят несколько связанных значений. Как и кортежи, части структур могут быть разных типов. В отличие от кортежей, в структуре необходимо именовать каждую часть данных для понимания смысла значений. Добавление этих имён обеспечивает большую гибкость структур по сравнению с кортежами: не нужно полагаться на порядок данных для указания значений экземпляра или доступа к ним.

Для определения структуры указывается ключевое слово `struct` и её название. Название должно описывать значение частей данных, сгруппированных вместе. Далее, в фигурных скобках для каждой новой части данных поочерёдно определяются имя части данных и её тип. Каждая пара `имя: тип` называется *полем*. Листинг 5-1 описывает структуру для хранения информации об учётной записи пользователя:

```
struct User {  
    active: bool,  
    username: String,  
    email: String,  
    sign_in_count: u64,  
}
```

Листинг 5-1: определение структуры `User`

После определения структуры можно создавать её экземпляр, назначая определённое значение каждому полю с соответствующим типом данных. Чтобы создать экземпляр, мы указываем имя структуры, затем добавляем фигурные скобки и включаем в них пары `ключ: значение` (key: value), где ключами являются имена полей, а значениями являются данные, которые мы хотим сохранить в полях. Нет необходимости чётко следовать порядку объявления полей в описании структуры (но всё-таки желательно для удобства чтения). Другими словами, объявление структуры - это как шаблон нашего типа, в то время как экземпляр структуры использует этот шаблон, заполняя его определёнными данными, для создания значений нашего типа. Например, можно объявить пользователя как в листинге 5-2:

```
fn main() {
    let user1 = User {
        email: String::from("someone@example.com"),
        username: String::from("someusername123"),
        active: true,
        sign_in_count: 1,
    };
}
```

Листинг 5-2: Создание экземпляра структуры `User`

Чтобы получить конкретное значение из структуры, используется точечная нотация. Если нужен только адрес электронной почты этого пользователя, мы можем использовать `user1.email` везде, где хотим использовать это значение. Если экземпляр является изменяемым, мы можем изменить значение, используя точечную нотацию и присваивая его конкретному полю. В листинге 5-3 показано, как изменить значение в поле `email` изменяемого экземпляра `User`.

```
fn main() {
    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");
}
```

Листинг 5-3: Изменение значения в поле `email` экземпляра `User`

Заметим, что весь экземпляр структуры должен быть изменяемым; Rust не позволяет помечать изменяемыми отдельные поля. Как и для любого другого выражения, мы можем использовать выражение создания структуры в качестве последнего выражения тела функции для неявного возврата нового экземпляра.

На листинге 5-4 функция `build_user` возвращает экземпляр `User` с указанным адресом и именем. Поле `active` получает значение `true`, а поле `sign_in_count` получает значение `1`.

```
fn build_user(email: String, username: String) -> User {
    User {
        email: email,
        username: username,
        active: true,
        sign_in_count: 1,
    }
}
```

Листинг 5-4: Функция `build_user`, которая принимает `email` и имя пользователя и возвращает экземпляр `User`

Имеет смысл называть параметры функции теми же именами, что и поля структуры, но необходимость повторять `email` и `username` для названий полей и переменных несколько утомительна. Если структура имеет много полей, повторение каждого имени станет ещё более раздражающим. К счастью, есть удобное сокращение!

Использование сокращённой инициализации поля

Так как имена входных параметров функции и полей структуры являются полностью идентичными в листинге 5-4, возможно использовать синтаксис *сокращённой инициализации поля*, чтобы переписать `build_user` так, чтобы он работал точно также, но не содержал повторений для `email` и `username`, как в листинге 5-5.

```
fn build_user(email: String, username: String) -> User {
    User {
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
}
```

Листинг 5-5: Функция `build_user`, использующая сокращённую инициализацию поля, когда параметры `email` и `username` имеют те же имена, что и поля `struct`

Здесь происходит создание нового экземпляра структуры `User`, которая имеет поле с именем `email`. Мы хотим установить поле структуры `email` значением входного параметра `email` функции `build_user`. Так как поле `email` и входной параметр функции `email` имеют одинаковое название, можно писать просто `email` вместо

кода `email: email`.

Создание экземпляра структуры из экземпляра другой структуры с помощью синтаксиса обновления структуры

Часто бывает полезно создать новый экземпляр структуры, который включает большинство значений из другого экземпляра, но некоторые из них изменяет. Это можно сделать с помощью *синтаксиса обновления структуры*.

Сначала в листинге 5-6 показано, как обычно создаётся новый экземпляр `User` в `user2` без синтаксиса обновления. Мы задаём новое значение для `email`, но в остальном используем те же значения из `user1`, которые были заданы в листинге 5-2.

```
fn main() {
    // --snip--

    let user2 = User {
        active: user1.active,
        username: user1.username,
        email: String::from("another@example.com"),
        sign_in_count: user1.sign_in_count,
    };
}
```

Листинг 5-6: Создание нового экземпляра `User` с использованием одного из значений из экземпляра `user1`

Используя синтаксис обновления структуры, можно получить тот же эффект, используя меньше кода как показано в листинге 5-7. Синтаксис `..` указывает, что оставшиеся поля устанавливаются неявно и должны иметь значения из указанного экземпляра.

```
fn main() {
    // --snip--

    let user2 = User {
        email: String::from("another@example.com"),
        ..user1
    };
}
```

Листинг 5-7: Использование синтаксиса обновления структуры для установки нового значения

`email` для экземпляра `User`, но использование остальных значений из экземпляра `user1`

Код в листинге 5-7 также создаёт экземпляр в `user2`, который имеет другое значение для `email`, но с тем же значением для полей `username`, `active` и `sign_in_count` из `user1`. Оператор `..user1` должен стоять последним для указания на получение значений всех оставшихся полей из соответствующих полей в `user1`, но можно указать значения для любого количества полей в любом порядке, независимо от порядка полей в определении структуры.

Заметим, что синтаксис обновления структуры использует `=` как присваивание. Это связано с перемещением данных, как мы видели в разделе "Способы взаимодействия переменных и данных: перемещение". В этом примере мы больше не можем использовать `user1` после создания `user2`, потому что `String` в поле `username` из `user1` было перемещено в `user2`. Если бы мы задали `user2` новые значения `String` для `email` и `username`, и при этом использовать только значения `active` и `sign_in_count` из `user1`, то `user1` все ещё будет действительным после создания `user2`. Типы `active` и `sign_in_count` являются типами, реализующими типаж `Copy`, поэтому будет применяться поведение, о котором мы говорили в разделе "Stack-Only Data: Copy".

Кортежные структуры: структуры без именованных полей для создания разных типов

Rust также поддерживает структуры, похожие на кортежи, которые называются *кортежные структуры*. Кортежные структуры обладают дополнительным смыслом, который даёт имя структуры, но при этом не имеют имён, связанных с их полями. Скорее, они просто хранят типы полей. Кортежные структуры полезны, когда вы хотите дать имя всему кортежу и сделать кортеж отличным от других кортежей, и когда именование каждого поля, как в обычной структуре, было бы многословным или избыточным.

Чтобы определить кортежную структуру, начните с ключевого слова `struct` и имени структуры, за которым следуют типы в кортеже. Например, здесь мы определяем и используем две кортежные структуры с именами `Color` и `Point`:

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
}
```

Заметим, что значения `black` и `origin` являются разными типами, потому что это экземпляры разных кортежных структур. Каждая определённая вами структура имеет свой собственный тип, даже если поля внутри структуры имеют одинаковые типы. Например, функция, принимающая параметр типа `Color`, не сможет принять в качестве аргумента `Point`, даже если оба типа состоят из трёх значений `i32`. В остальном экземпляры кортежных структур ведут себя как кортежи: вы можете деструктурировать их на отдельные части, вы можете использовать `.` с последующим индексом для доступа к отдельному значению, и так далее.

Единично-подобные структуры: структуры без полей

Также можно определять структуры, не имеющие полей! Они называются *единично-подобными структурами*, поскольку ведут себя аналогично `()`, единичному типу, о котором мы говорили в разделе "Тип кортежа". Единично-подобные структуры могут быть полезны, когда требуется реализовать типаж для некоторого типа, но у вас нет данных, которые нужно хранить в самом типе. Мы обсудим типажи в главе 10. Вот пример объявления и создание экземпляра единичной структуры с именем `AlwaysEqual`:

```
struct AlwaysEqual;

fn main() {
    let subject = AlwaysEqual;
}
```

Чтобы определить `AlwaysEqual`, мы используем ключевое слово `struct`, желаемое имя, а затем точку с запятой. Нет необходимости в фигурных или круглых скобках! Затем мы можем получить экземпляр `AlwaysEqual` в переменной `subject` аналогичным образом: используя имя, которое мы определили, без фигурных и круглых скобок. Представим, что в дальнейшем мы реализуем поведение для этого типа таким образом, что каждый экземпляр `AlwaysEqual` всегда будет равен каждому экземпляру любого другого типа, возможно, с целью получения

ожидаемого результата для тестирования. Для реализации такого поведения нам не нужны никакие данные! В главе 10 вы увидите, как определять черты и реализовывать их для любого типа, включая единично-подобные структуры.

Владение данными структуры

В определении структуры `User` в листинге 5-1 мы использовали владеющий тип `String` вместо типа строковой срез `&str`. Это осознанный выбор, поскольку мы хотим, чтобы каждый экземпляр этой структуры владел всеми своими данными и чтобы эти данные были действительны до тех пор, пока действительна вся структура.

Структуры также могут хранить ссылки на данные, принадлежащие кому-то другому, но для этого необходимо использовать возможность Rust *время жизни*, которую мы обсудим в главе 10. Время жизни гарантирует, что данные, на которые ссылается структура, будут действительны до тех пор, пока существует структура. Допустим, если попытаться сохранить ссылку в структуре без указания времени жизни, как в следующем примере; это не сработает:

Имя файла: `src/main.rs`

```
struct User {
    active: bool,
    username: &str,
    email: &str,
    sign_in_count: u64,
}

fn main() {
    let user1 = User {
        email: "someone@example.com",
        username: "someusername123",
        active: true,
        sign_in_count: 1,
    };
}
```



Компилятор будет жаловаться на необходимость определения времени жизни ссылок:

```
$ cargo run
Compiling structs v0.1.0 (file:///projects/structs)
error[E0106]: missing lifetime specifier
--> src/main.rs:3:15
| 
3 |     username: &str,
|             ^ expected named lifetime parameter
| 
help: consider introducing a named lifetime parameter
| 
1 ~ struct User<'a> {
2 |     active: bool,
3 ~     username: &'a str,
| 

error[E0106]: missing lifetime specifier
--> src/main.rs:4:12
| 
4 |     email: &str,
|             ^ expected named lifetime parameter
| 
help: consider introducing a named lifetime parameter
| 
1 ~ struct User<'a> {
2 |     active: bool,
3 |     username: &str,
4 ~     email: &'a str,
| 

For more information about this error, try `rustc --explain E0106`.
error: could not compile `structs` due to 2 previous errors
```

В главе 10 мы обсудим, как исправить эти ошибки, чтобы иметь возможность хранить ссылки в структурах, а пока мы исправим подобные ошибки, используя владеющие типы вроде **String** вместо ссылок **&str**.

Пример использования структур

Чтобы понять, когда нам может понадобиться использование структур, давайте напишем программу, которая вычисляет площадь прямоугольника. Мы начнём с использования одиночных переменных, а затем будем улучшать программу до использования структур.

Давайте создадим новый проект программы при помощи Cargo и назовём его *rectangles*. Наша программа будет получать на вход длину и ширину прямоугольника в пикселях и затем рассчитывать площадь прямоугольника. Листинг 5-8 показывает один из коротких вариантов кода который позволит нам сделать именно то, что надо, код в файле проекта *src/main.rs*.

Имя файла: *src/main.rs*

```
fn main() {
    let width1 = 30;
    let height1 = 50;

    println!(
        "The area of the rectangle is {} square pixels.",
        area(width1, height1)
    );
}

fn area(width: u32, height: u32) -> u32 {
    width * height
}
```

Листинг 5-8: Вычисление площади прямоугольника, заданного отдельными переменными ширины и высоты

Теперь, проверим её работу `cargo run`:

```
$ cargo run
Compiling rectangles v0.1.0 (file:///projects/rectangles)
Finished dev [unoptimized + debuginfo] target(s) in 0.42s
Running `target/debug/rectangles`
The area of the rectangle is 1500 square pixels.
```

Этот код успешно вычисляет площадь прямоугольника, вызывая функцию `area` с каждым измерением, но мы можем сделать больше для повышения ясности и читабельности этого кода.

Проблема данного метода очевидна из сигнатуры `area`:

```
fn area(width: u32, height: u32) -> u32 {
```

Функция `area` должна вычислять площадь одного прямоугольника, но функция, которую мы написали, имеет два параметра, и нигде в нашей программе не ясно, что эти параметры взаимосвязаны. Было бы более читабельным и управляемым сгруппировать ширину и высоту вместе. Мы уже обсуждали один из способов сделать это в разделе "Тип кортеж" главы 3: использование кортежей.

Рефакторинг при помощи кортежей

Листинг 5-9 это другая версия программы, использующая кортежи.

Имя файла: src/main.rs

```
fn main() {
    let rect1 = (30, 50);

    println!(
        "The area of the rectangle is {} square pixels.",
        area(rect1)
    );
}

fn area(dimensions: (u32, u32)) -> u32 {
    dimensions.0 * dimensions.1
}
```

Листинг 5-9: Определение ширины и высоты прямоугольника с помощью кортежа

С одной стороны, эта программа лучше. Кортежи позволяют добавить немного структуры, и теперь мы передаём только один аргумент. Но с другой стороны, эта версия менее понятна: кортежи не называют свои элементы, поэтому нам приходится индексировать части кортежа, что делает наше вычисление менее очевидным.

Если мы перепутаем местами ширину с высотой при расчёте площади, то это не имеет значения. Но если мы хотим нарисовать нарисовать прямоугольник на экране, то это уже будет иметь значение! Мы должны помнить, что ширина `width` находится в кортеже с индексом `0`, а высота `height` с индексом `1`. Если кто-то другой поработал бы с кодом, ему бы пришлось разобраться в этом и также

помнить про порядок. Легко забыть и перепутать эти значения и это вызовет ошибки, потому что данный код не передаёт наши намерения.

Рефакторинг при помощи структур: добавим больше смысла

Мы используем структуры чтобы добавить смысл данным при помощи назначения им осмысленных имён . Мы можем переделать используемый кортеж в структуру с единственным именем для сущности и частными названиями её частей, как показано в листинге 5-10.

Файл: src/main.rs

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
    );
}

fn area(rectangle: &Rectangle) -> u32 {
    rectangle.width * rectangle.height
}
```

Листинг 5-10: Определение структуры `Rectangle`

Здесь мы определили структуру и дали ей имя `Rectangle`. Внутри фигурных скобок определили поля как `width` и `height`, оба - типа `u32`. Затем в `main` создали конкретный экземпляр `Rectangle` с шириной в 30 и высотой в 50 единиц.

Наша функция `area` теперь определена с одним параметром названным `rectangle`, чей тип является неизменяемым заимствованием структуры `Rectangle`. Как упоминалось в Главе 4, необходимо заимствовать структуру, а не передавать её во владение. Таким образом функция `main` сохраняет `rect1` в собственности и

может её использовать дальше, по этой причине мы и используем `&` в сигнатуре и в месте вызова функции.

Функция `area` имеет доступ к полям `width` и `height` экземпляра `Rectangle`.

Сигнатура нашей функции для `area` теперь точно говорит, что мы имели в виду: посчитать площадь `Rectangle` используя поля `width` и `height`. Такой подход сообщает, что ширина и высота связаны по смыслу друг с другом. А названия значений структуры теперь носят понятные описательные имена, вместо ранее используемых значений индексов кортежа `0` и `1`. Это плюс к ясности.

Добавление полезной функциональности при помощи Выводимых Типажей

Было бы полезно иметь возможность печатать экземпляр `Rectangle` во время отладки программы и видеть значения всех полей. Листинг 5-11 использует макрос `println!` `macro`, который мы уже использовали в предыдущих главах. Тем не менее, это не работает.

Файл: `src/main.rs`

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rect1 is {}", rect1);
}
```



Листинг 5-11: Попытка распечатать экземпляр `Rectangle`

При компиляции этого кода мы получаем ошибку с сообщением:

```
error[E0277]: `Rectangle` doesn't implement `std::fmt::Display`
```

Макрос `println!` умеет выполнять множество видов форматирования, по умолчанию фигурные скобки в `println!` говорят использовать форматирование

известное как типаж **Display**: вывод в таком варианте форматирования предназначен для непосредственного использования конечным пользователем. Примитивные типы изученные ранее, по умолчанию реализуют типаж **Display**, потому что есть только один способ отобразить число **1** или любой другой примитивный тип пользователю. Но для структур у которых **println!** должен форматировать способ вывода данных, это является менее очевидным, потому что есть гораздо больше возможностей для отображения: Вы хотите запятые или нет? Вы хотите печатать фигурные скобки? Должны ли отображаться все поля? Из-за этой неоднозначности Rust не пытается угадать, что нам нужно, а структуры не имеют встроенной реализации **Display** для использования в **println!** с заполнителем **{}**.

Продолжив чтение текста ошибки, мы найдём полезное замечание:

```
= help: the trait `std::fmt::Display` is not implemented for `Rectangle`
= note: in format strings you may be able to use `{:?}` (or `{:#?}` for
pretty-print) instead
```

Давайте попробуем! Вызов макроса **println!** теперь будет выглядеть так **println!**
("rect1 is {:?}", **rect1****);**. Ввод спецификатора **:?** внутри фигурных скобок говорит макросу **println!**, что мы хотим использовать другой формат вывода известный как **Debug**. Типаж **Debug** позволяет печатать структуру способом, удобным для разработчиков, чтобы видеть значение во время отладки кода.

Скомпилируем код с этими изменениями. Упс! Мы всё ещё получаем ошибку:

```
error[E0277]: `Rectangle` doesn't implement `Debug`
```

Снова компилятор даёт нам полезное замечание:

```
= help: the trait `Debug` is not implemented for `Rectangle`
= note: add `#[derive(Debug)]` to `Rectangle` or manually `impl Debug for
Rectangle`
```

Rust *реализует* функциональность для печати отладочной информации, но *не включает (не выводит) её по умолчанию*, мы должны явно включить эту функциональность для нашей структуры. Чтобы это сделать, добавляем внешний атрибут **#[derive(Debug)]** сразу перед определением структуры как показано в листинге 5-12.

Файл: src/main.rs

```

#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rect1 is {:?}", rect1);
}

```

Листинг 5-12: Добавление атрибута для вывода типажа `Debug` и печати экземпляра `Rectangle` с отладочным форматированием

Теперь при запуске программы мы не получим ошибок и увидим следующий вывод:

```

$ cargo run
Compiling rectangles v0.1.0 (file:///projects/rectangles)
  Finished dev [unoptimized + debuginfo] target(s) in 0.48s
    Running `target/debug/rectangles`
rect1 is Rectangle { width: 30, height: 50 }

```

Отлично! Это не самый красивый вывод, но он показывает значения всех полей экземпляра, которые определённо помогут при отладке. Когда у нас более крупные структуры, то полезно иметь более простой для чтения вывод; в таких случаях можно использовать код `{:#?}` вместо `{:?}` в строке макроса `println!`. В этом примере использование стиля `{:#?}` приведёт к такому выводу:

```

$ cargo run
Compiling rectangles v0.1.0 (file:///projects/rectangles)
  Finished dev [unoptimized + debuginfo] target(s) in 0.48s
    Running `target/debug/rectangles`
rect1 is Rectangle {
    width: 30,
    height: 50,
}

```

Другой способ вывести значение в формате `Debug` — это использовать макрос `dbg!` `macro`, который забирает во владение выражение, печатает номер файла и строки, где происходит `dbg!` вызов макроса в коде вместе с результирующим

значением этого выражения и возвращает владение значения.

Примечание: вызов `dbg!` макрос печатает в стандартный поток консоли для ошибок (`stderr`), а не в `println!` который печатает в стандартный поток консоли вывода (`stdout`). Подробнее о `stderr` и `stdout` мы поговорим в разделе “[Запись ошибок в поток вывода для ошибок вместо стандартного потока вывода](#)” Главы 12.

Вот пример, когда нас интересует значение, которое присваивается полю `width`, а также значение всей структуры в `rect1`:

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let scale = 2;
    let rect1 = Rectangle {
        width: dbg!(30 * scale),
        height: 50,
    };

    dbg!(&rect1);
}
```

Можем написать макрос `dbg!` вокруг выражения `30 * scale`, потому что `dbg!` возвращает владение значения выражения, поле `width` получит то же значение, как если бы у нас не было вызова `dbg!`. Мы не хотим чтобы макрос `dbg!` становился владельцем `rect1`, поэтому мы используем ссылку на `rect1` в следующем вызове. Вот как выглядит вывод этого примера:

```
$ cargo run
Compiling rectangles v0.1.0 (file:///projects/rectangles)
Finished dev [unoptimized + debuginfo] target(s) in 0.61s
Running `target/debug/rectangles`
[src/main.rs:10] 30 * scale = 60
[src/main.rs:14] &rect1 = Rectangle {
    width: 60,
    height: 50,
}
```

Мы можем увидеть, что первый бит вывода поступил из строки 10 `src/main.rs`, там где мы отлаживаем выражение `30 * scale` и его результирующее значение равно 60 (`Debug` форматирование, реализованное для целых чисел, заключается в печати только их значения). Вызов `dbg!` в строке 14 `src/main.rs` выводит значение `&rect1`, которое является структурой `Rectangle`. В этом выводе используется красивое форматирование `Debug` типа `Rectangle`. Макрос `dbg!` может быть очень полезен, когда вы пытаетесь понять, что делает ваш код!

В дополнение к `Debug`, Rust предоставил нам ряд типажей, которые мы можем использовать с атрибутом `derive` для добавления полезного поведения к нашим пользовательским типам. Эти типажи и их поведение перечислены в [Приложении C](#). Мы расскажем, как реализовать эти трейты с пользовательским поведением, а также как создать свои собственные трейты в главе 10. Кроме того, есть много других атрибутов помимо `derive`; для получения дополнительной информации смотри раздел “[Атрибуты](#)” [справочника Rust](#).

Функция `area` является довольно специфичной: она считает только площадь прямоугольников. Было бы полезно привязать данное поведение как можно ближе к структуре `Rectangle`, потому что наш специфичный код не будет работать с любым другим типом. Давайте рассмотрим, как можно улучшить наш код, превращая функцию `area` в метод `area`, определённый для типа `Rectangle`.

Синтаксис метода

Методы похожи на функции: они объявлены с помощью ключевого слова `fn` и его имени. Они могут иметь параметры и возвращаемое значение, могут содержать некоторый код, который выполняется при вызове из другого места. Тем не менее, методы отличаются от функций тем, что они определены внутри контекста структуры (также перечисления или объекта-типажа, которые мы рассмотрим в главе 6 и 17, соответственно), а их первым параметром всегда является `self`, который представляет экземпляр структуры для которого этот метод будет вызван.

Определение методов

Давайте изменим функцию `area` так, чтобы она имела экземпляр `Rectangle` в качестве входного параметра и сделаем её методом `area`, определённым для структуры `Rectangle`, как показано в листинге 5-13:

Файл: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

Листинг 5-13: Определение метода `area` у структуры `Rectangle`

Для определения функции в контексте типа `Rectangle`, мы начинаем блок `impl` (implementation - реализация). Затем переносим функцию `area` внутрь фигурных скобок `impl` и меняем первый (в данном случае единственный) параметр в сигнатуре на `self`, и далее везде в теле метода. В `main`, там где мы вызывали функцию `area` и передавали ей переменную `rect1` в качестве аргумента, теперь можно использовать *синтаксис метода* для вызова метода `area` на экземпляре типа `Rectangle`. Синтаксис метода идёт после экземпляра: мы добавляем точечную нотацию за которой следует название метода, круглые скобки и любые аргументы.

В сигнатуре `area`, используется `&self` вместо `rectangle: &Rectangle` потому что Rust знает, что тип `self` является типом `Rectangle`, так как данный метод находится внутри `impl Rectangle` контекста. Заметьте, всё ещё нужно использовать `&` перед `self`, как мы делали с `&Rectangle`. Методы могут принимать во владение `self`, заимствовать неизменяемый `self`, как мы сделали здесь, или заимствовать изменяемый `self`, а также любые другие параметры.

Мы выбрали `&self` здесь по той же причине, по которой использовали `&Rectangle` в версии кода с функцией: мы не хотим брать структуру во владение, мы просто хотим прочитать данные в структуре, а не писать в неё. Если бы мы хотели изменить экземпляр, на котором мы вызывали метод силами самого метода, то мы бы использовали `&mut self` в качестве первого параметра. Наличие метода, который берёт экземпляр во владение, используя только `self` в качестве первого параметра, является редким; эта техника обычно используется, когда метод превращает `self` во что-то ещё, и вы хотите запретитьзывающей стороне использовать исходный экземпляр после превращения.

Основным преимуществом использования методов вместо функций, в дополнение к использованию синтаксиса метода, где не нужно повторять тип `self` в каждой сигнатуре метода, является организация кода. Мы собрали все, что мы можем сделать с экземпляром типа в одном блоке `impl`, не заставляя будущих пользователей нашего кода искать дополнительный реализованный функционал для `Rectangle` в разных местах библиотеки.

Где используется оператор `->?`

В языках C и C++, используются два различных оператора для вызова методов: используется `.`, если вызывается метод непосредственно у экземпляра

структуре и используется `->`, если вызывается метод у ссылки на объект. Другими словами, если `object` является ссылкой, то вызовы метода `object->something()` и `(*object).something()` являются аналогичными.

Rust не имеет эквивалента оператора `->`, наоборот, в Rust есть функциональность называемая *автоматическое обращение по ссылке и разыменование* (automatic referencing and dereferencing). Вызов методов является одним из немногих мест в Rust, в котором есть такое поведение.

Вот как это работает: когда вы вызываете метод `object.something()`, Rust автоматически добавляет `&`, `&mut` или `*`, таким образом, чтобы `object` соответствовал сигнатуре метода. Другими словами, это то же самое:

```
p1.distance(&p2);  
(&p1).distance(&p2);
```

Первый пример выглядит намного понятнее. Автоматический вывод ссылки работает потому, что методы имеют понятного получателя - тип `self`. Учитывая получателя и имя метода, Rust может точно определить, что в данном случае делает код: читает ли метод (`&self`), делает ли изменение (`&mut self`) или поглощает (`self`). Тот факт, что Rust делает заимствование неявным для принимающего метода, в значительной степени способствует тому, чтобы сделать владение эргономичным на практике.

Методы с несколькими параметрами

Давайте попрактикуемся в использовании методов, реализовав второй метод у структуры `Rectangle`. На этот раз мы хотим, чтобы экземпляр `Rectangle` использовал другой экземпляр типа `Rectangle` и возвращал `true`, если второй `Rectangle` может полностью разместиться внутри площади экземпляра `self`; в противном случае он должен возвращать `false`. То есть мы хотим иметь возможность написать программу, показанную в листинге 5-14, в которой определили метод `can_hold`.

Файл: `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,
    };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

Листинг 5-14: Использование ещё не написанного метода `can_hold`

И ожидаемый результат будет выглядеть следующим образом, т.к. оба размера в экземпляре `rect2` меньше, чем размеры в экземпляре `rect1`, а `rect3` шире, чем `rect1`:

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

Мы знаем, что хотим определить метод, поэтому он будет находиться в `impl Rectangle` блоке. Имя метода будет `can_hold`, и оно будет принимать неизменяемое заимствование на другой `Rectangle` в качестве параметра. Мы можем сказать, какой это будет тип параметра, посмотрев на код вызывающего метода: метод `rect1.can_hold(&rect2)` передаёт в него `&rect2`, который является неизменяемым заимствованием экземпляра `rect2` типа `Rectangle`. В этом есть смысл, потому что нам нужно только читать `rect2` (а не писать, что означало бы, что нужно изменяемое заимствование), и мы хотим, чтобы `main` сохранил право собственности на экземпляр `rect2`, чтобы мы могли использовать его снова после вызова метода `can_hold`. Возвращаемое значение `can_hold` имеет булевый тип, а реализация проверяет, являются ли ширина и высота `self` больше, чем ширина и высота другого `Rectangle` соответственно. Давайте добавим новый метод `can_hold` в `impl` блок из листинга 5-13, как показано в листинге 5-15.

Файл: 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
    }
}
```

Листинг 5-15: реализация метода `can_hold` у структуры `Rectangle`, который принимает другой экземпляр `Rectangle` в качестве параметра

Когда мы запустим код с функцией `main` листинга 5-14, мы получим желаемый вывод. Методы могут принимать несколько параметров, которые мы добавляем в сигнатуру после первого параметра `self`, и эти параметры работают так же, как параметры в функциях.

Ассоциированные функции

Ещё одной полезной особенностью блоков `impl` является то, что мы можем определить функции внутри блоков `impl`, которые не принимают `self` в качестве параметра. Они называются *ассоциированными функциями*, потому что они всё ещё связаны со структурой в отличии от простых функций. Так же они всё ещё функции, а не методы, потому что у них нет экземпляра структуры над которой они могут работать. Вы уже использовали ассоциированную функцию `String::from`.

Ассоциированные функции часто используются в качестве конструкторов - функций, которые будут возвращать новый экземпляр структуры. Например, мы могли бы предоставить ассоциированную функцию, которая будет иметь один параметр измерения и использовать его как ширину и высоту, тем самым облегчая создание квадратного прямоугольника `Rectangle`, вместо указания одно и того же значения дважды:

Файл: src/main.rs

```
impl Rectangle {
    fn square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
        }
    }
}
```

Чтобы вызвать эту ассоциированную функцию, используется синтаксис `:::` с именем структуры; пример `let sq = Rectangle::square(3);`. Эта функция относится к структуре: синтаксис `:::` используется как для ассоциированных функций, так и для пространства имён, созданных модулями. Мы обсудим модули в Главе 7.

Несколько блоков `impl`

Для каждой структуры разрешено иметь множество `impl` блоков. Например, листинг 5-15 является эквивалентным коду из листинга 5-16, который описывает метод в своём отдельном блоке `impl`.

```
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
    }
}
```

Листинг 5-16: Переписанный листинг 5-15 с использованием нескольких блоков `impl`

В данном случае нет причин разделять эти методы на несколько блоков `impl`, но это тоже является правильным синтаксисом. Мы увидим случай, в котором полезно иметь несколько `impl` блоков в Главе 10, где мы будем обсуждать обобщённые типы и типажи.

Итоги

Структуры позволяют создавать собственные типы, которые имеют смысл в вашей предметной области. Используя структуры, вы храните ассоциированные друг с другом фрагменты данных и даёте название частям данных, чтобы ваш код был более понятным. Методы позволяют определить поведение, которое имеют экземпляры ваших структур, а ассоциированные функции позволяют привязать функциональность к вашей структуре, не обращаясь к её экземпляру.

Но структуры - это не единственный способ создания пользовательских типов: давайте обратимся к перечислениям в Rust, чтобы добавить ещё один инструмент в наш арсенал.

Перечисления и Сопоставление с образцом

В этой главе мы рассмотрим *перечисления*, также называемые *enum*s. Перечисления позволяют определять тип, перечисляя его возможные *варианты*. Сначала, мы определим и воспользуемся перечислением, чтобы показать, как перечисление может закодировать значение вместе с данными. Далее мы рассмотрим особенно полезный *enum*, называемый **Option**, который выражает факт того, что значение может быть *либо чем-то, либо ничем*. Потом мы посмотрим на сопоставление с образцом в **match** выражении, позволяющем легко выполнять разный код для различных значений перечисления. Наконец, мы рассмотрим конструкцию **if let** - ещё одну удобную и лаконичную идиому, которая позволяет вам управлять перечислениями в коде.

Перечисления являются особенностью многих языков, но в каждом языке их возможности различаются. Перечисления в Rust наиболее похожи на *алгебраические типы данных*, *Algebraic Data Types*, представленные в таких функциональных языках как F#, OCaml и Haskell.

Определение перечисления

Давайте посмотрим на ситуацию, которую мы могли бы выразить в коде, и рассмотрим почему перечисления полезны и более уместны чем структуры в данном случае. Представим, что нам нужно работать с IP-адресами. В настоящее время используются два основных стандарта IP-адресов: версия четыре и версия шесть. Это единственные варианты IP адресов, с которым столкнётся наша программа: мы можем *перечислить* (*enumerate*) все возможные варианты, отсюда и появляется понятие *перечисление* (*enumeration*, *enum*).

Любой IP-адрес может быть либо адресом версии четыре, либо версии шесть - но не может быть одновременно и шестой и четвёртой версии. Это свойство IP-адресов делает перечисление подходящей структурой данных для их хранения, т.к. значения *enum*, как и версия IP-адреса, могут быть только одним из возможных в данном перечислении вариантом. Адреса как версии четыре, так и версии шесть по-прежнему являются IP-адресами, поэтому они должны рассматриваться как один и тот же тип, когда код обрабатывает ситуации применимые к любому виду IP-адреса.

Можно выразить эту концепцию в коде, определив перечисление `IpAddrKind` и составив список возможных видов IP-адресов, `V4` и `V6`. Вот варианты перечислений:

```
enum IpAddrKind {
    V4,
    V6,
}
```

`IpAddrKind` теперь является пользовательским типом данных, который мы можем использовать в другом месте нашего кода.

Значения перечислений

Экземпляры каждого варианта перечисления `IpAddrKind` можно создать следующим образом:

```
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;
```

Обратите внимание, что варианты перечисления находятся в пространстве имён

его идентификатора, мы используем двойное двоеточие чтобы отделить вариант от пространства имён. Причина по которой это полезно в том, что сейчас оба значения `IpAddrKind::V4` и `IpAddrKind::V6` имеют одинаковый тип: `IpAddrKind`. Благодаря этому в дальнейшем мы имеем возможность определять функции, которые принимают любой вариант `IpAddrKind`:

```
fn route(ip_kind: IpAddrKind) {}
```

Можно вызвать эту функцию с любым из вариантов:

```
route(IpAddrKind::V4);
route(IpAddrKind::V6);
```

Использование перечислений имеет даже больше преимуществ. Размышляя о нашем типе IP-адреса в данный момент, мы понимаем, что у нас нет способа сохранить фактические *данные* IP-адреса; мы только знаем, каким *вариантом* он является. Учитывая то, что вы недавно узнали о структурах в Главе 5, можно решить эту проблему как показано в листинге 6-1.

```
enum IpAddrKind {
    V4,
    V6,
}

struct IpAddr {
    kind: IpAddrKind,
    address: String,
}

let home = IpAddr {
    kind: IpAddrKind::V4,
    address: String::from("127.0.0.1"),
};

let loopback = IpAddr {
    kind: IpAddrKind::V6,
    address: String::from("::1"),
};
```

Листинг 6-1. Сохранение данных и `IpAddrKind` IP-адреса с использованием `struct`

Здесь мы определили структуру `IpAddr`, которая имеет два поля: поле `kind` имеет тип `IpAddrKind` (перечисление, которое мы определили ранее) и поле `address` типа `String`. У нас есть два экземпляра этой структуры. Первый, `home`, имеет

значение `kind` равное `IpAddrKind::V4` и связан с адресом `127.0.0.1`. Второй экземпляр, `loopback`, имеет другой вариант `IpAddrKind` качестве значения `kind` - вариант `V6` и имеет связанный с ним адрес `::1`. Мы использовали структуру для объединения значений `kind` и `address`, теперь вариант связан со значением.

Мы можем представить ту же концепцию в более сжатой форме, используя только перечисление, вместо перечисления запакованного внутри структуры, и помещать данные непосредственно в каждый вариант перечисления. Это новое определение перечисления `IpAddr` говорит, что оба варианта `V4` и `V6` будут иметь связанные с ними значения типа `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 нет необходимости в дополнительной структуре. Здесь также легче увидеть ещё одну деталь того, как работают перечисления: имя каждого варианта перечисления, который мы определяем, также становится функцией, которая создаёт экземпляр перечисления. То есть `IpAddr::V4()` - это вызов функции, который принимает `String` и возвращает экземпляр типа `IpAddr`. Мы автоматически получаем эту функцию-конструктор, определяемую в результате определения перечисления.

Ещё одно преимущество использования перечисления вместо структуры заключается в том, что каждый вариант перечисления может иметь разное количество ассоциированных данных представленных в разных типах. Версия 4 для типа IP адресов всегда будет содержать четыре цифровых компонента, которые будут иметь значения между 0 и 255. При необходимости сохранить адреса типа `V4` как четыре значения типа `u8`, а также описать адреса типа `V6` как единственное значение типа `String`, мы не смогли бы с помощью структуры. Перечисления решают эту задачу легко:

```
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 способов определения структур данных для хранения IP-адресов стандарта версии четыре и версии шесть. Однако, как выясняется, желание хранить IP-адреса и кодировать какого они типа, настолько распространено среди разработчиков, что в стандартной библиотеке уже есть [готовое для нашей задачи определение](#), которое мы можем использовать! Давайте посмотрим, как стандартная библиотека определяет тип `IpAddr`: она так же как и у нас имеет аналогичное перечисление с аналогичными вариантами (подобными тем, которые мы определили и использовали ранее), но она представляет (а затем и встраивает в варианты) данные IP-адресов в форме двух разных структур, которые определяются по-разному для каждого варианта.

```
struct Ipv4Addr {
    // --snip--
}

struct Ipv6Addr {
    // --snip--
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

Этот код иллюстрирует что мы можем добавлять любой тип данных в значение перечисления: строку, число, структуру и пр. Вы даже можете включить в перечисление другие перечисления! Стандартные типы данных не очень сложны, хотя, потенциально, могут быть очень сложными (вложенность данных может быть очень глубокой).

Обратите внимание, что хотя определение перечисления `IpAddr` есть в стандартной библиотеке, мы смогли объявлять и использовать свою собственную реализацию с аналогичным названием без каких-либо конфликтов, потому что мы не добавили определение стандартной библиотеки в область видимости кода.

Подробнее об этом поговорим в Главе 7.

Рассмотрим другой пример перечисления в листинге 6-2: в этом примере каждый элемент перечисления имеет свой особый тип данных внутри:

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

Листинг 6-2. Перечисление `Message`, в каждом из вариантов которого хранятся разные количества и типы значений.

Это перечисление имеет 4 элемента:

- `Quit` - пустой элемент без ассоциированных данных,
- `Move` имеет именованные поля, как и структура.
- `Write` - элемент с единственной строкой типа `String`,
- `ChangeColor` - кортеж из трёх значений типа `i32`.

Определение перечисления с вариантами, такими как в листинге 6-2, похоже на определение значений различных типов внутри структур, за исключением того, что перечисление не использует ключевое слово `struct` и все варианты сгруппированы внутри типа `Message`. Следующие структуры могут содержать те же данные, что и предыдущие варианты перечислений:

```
struct QuitMessage; // unit struct
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // tuple struct
struct ChangeColorMessage(i32, i32, i32); // tuple struct
```

Но когда мы использовали различные структуры, которые имеют свои собственные типы, мы не могли легко определять функции, которые принимают любые типы сообщений, как это можно сделать с помощью перечисления типа `Message`, объявленного в листинге 6-2, который является единым типом.

Есть ещё одно сходство между перечислениями и структурами: так же, как мы можем определять методы для структур с помощью `impl` блока, мы можем

определять и методы для перечисления. Вот пример метода с именем `call`, который мы могли бы определить в нашем перечислении `Message`:

```
impl Message {
    fn call(&self) {
        // method body would be defined here
    }
}

let m = Message::Write(String::from("hello"));
m.call();
```

Тело метода будет использовать `self`, чтобы получить значение из объекта на котором мы вызвали этот метод. В этом примере мы создали переменную `m` которой назначено значение из выражения `Message::Write(String::from("hello"))` и это то чем будет `self` в теле метода `call` при вызове `m.call()`.

Теперь посмотрим на другое наиболее часто используемое перечисление из стандартной библиотеки, которое является очень распространённым и полезным: `Option`.

Перечисление `Option` и его преимущества перед Null-значениями

В предыдущем разделе мы рассмотрели, как перечисление `IpAddr` позволило нам использовать систему типов Rust для кодирования в программе большего количества информации, чем просто данные. В этом разделе рассматривается пример использования `Option`, ещё одного перечисления, определённого стандартной библиотекой. Тип `Option` используется во многих местах, потому что он кодирует очень распространённый сценарий, в котором значение может быть *чем-то* или может быть *ничем*. Выражение этой концепции в терминах системы типов означает, что компилятор может проверить, обработаны ли все случаи для данного типа, которые должны обрабатываться; такого рода проверка компилятора может предотвратить ошибки, которые чрезвычайно распространены в других языках программирования.

Дизайн языка программирования часто рассматривается с точки зрения того, какие функции вы включаете в него, но те функции, которые вы исключаете, также важны. Например в Rust нет такого функционала как `null` значения, однако он есть во

многих других языках. *Null значение* - это значение, которое означает, что значения нет. В языках с null значением переменные всегда могут находиться в одном из двух состояний: *нет значения (null)* или *есть значение (not-null)*.

В своей презентации 2009 года «Null ссылки: ошибка в миллиард долларов» Тони Хоар (Tony Hoare), изобретатель null, сказал следующее:

Я называю это своей ошибкой на миллиард долларов. В то время я разрабатывал первую комплексную систему типов для ссылок на объектно-ориентированном языке. Моя цель состояла в том, чтобы гарантировать, что любое использование ссылок должно быть абсолютно безопасным, с автоматической проверкой компилятором. Но я не мог устоять перед соблазном вставить пустую ссылку просто потому, что это было так легко реализовать. Это привело к бесчисленным ошибкам, уязвимостям и системным сбоям, которые, вероятно, причинили боль и ущерб на миллиард долларов за последние сорок лет.

Проблема с null значениям заключается в том, что если вы попытаетесь использовать его значение в качестве не-null значения, вы получите какую-то ошибку. Из-за того, что null или не-null свойство всеобъемлющее и может быть использовано повсеместно, очень легко использовать null и в дальнейшем получить такого рода ошибку.

Тем не менее, концепция, которую null пытается выразить, является полезной: null - это значение, которое в настоящее время по какой-то причине недействительно или отсутствует.

Проблема не в самой концепции, а в конкретной реализации. Таким образом, в Rust нет null-значений, но есть перечисление, которое может закодировать концепцию наличия или отсутствия значения. Это перечисление `Option<T>` и оно [объявляется в стандартной библиотеке](#) следующим образом:

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

Перечисление `Option<T>` настолько полезно, что даже подключено в авто-импорте; его не нужно явно подключать в область видимости. Дополнительно подключены

также и его варианты: можно использовать `Some` и `None` напрямую, без префикса `Option::`. Перечисление `Option<T>` все ещё является обычным перечислением, а `Some(T)` и `None` являются вариантами типа `Option<T>`.

`<T>` - это особенность Rust, о которой мы ещё не говорили. Это параметр обобщённого типа, и мы рассмотрим его более подробно в главе 10. На данный момент всё, что вам нужно знать, это то, что `<T>` означает, что вариант `Some` `Option` может содержать один фрагмент данных любого типа, и что каждый конкретный тип, который используется вместо `T` делает общий `Option<T>` другим типом. Вот несколько примеров использования `Option` для хранения числовых и строковых типов:

```
let some_number = Some(5);
let some_string = Some("a string");

let absent_number: Option<i32> = None;
```

Тип `some_number` - `Option<i32>`. Тип `some_string` - `Option<&str>`, который другого типа. Rust может вывести эти типы, потому что мы указали значение внутри варианта `Some`. Для `absent_number` Rust требует, чтобы мы аннотировали общий тип для `Option`: компилятор не может вывести тип, который в `Some`, глядя только на значение `None`. Здесь мы сообщаем Rust, что мы имеем в виду, что `absent_number` должен иметь тип `Option<i32>`.

Когда есть значение `Some`, мы знаем, что значение присутствует и содержится внутри `Some`. Когда есть значение `None`, это означает то же самое, что и null в некотором смысле: у нас нет действительного значения. Так почему наличие `Option<T>` лучше, чем null?

Вкратце, поскольку `Option<T>` и `T` (где `T` может быть любым типом) относятся к разным типам, компилятор не позволит нам использовать значение `Option<T>` даже если бы оно было определено допустимым вариантом `Some`. Например, этот код не будет компилироваться, потому что он пытается добавить `i8` к значению типа `Option<i8>`:

```
let x: i8 = 5;
let y: Option<i8> = Some(5);

let sum = x + y;
```



Запуск данного кода даст ошибку ниже:

```
$ cargo run
Compiling enums v0.1.0 (file:///projects/enums)
error[E0277]: cannot add `Option<i8>` to `i8`
--> src/main.rs:5:17
5 |     let sum = x + y;
   |             ^ no implementation for `i8 + Option<i8>`
|
= help: the trait `Add<Option<i8>>` is not implemented for `i8`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `enums` due to previous error
```

Сильно! Фактически, это сообщение об ошибке означает, что Rust не понимает, как сложить `i8` и `Option<i8>`, потому что это разные типы. Когда у нас есть значение типа на подобие `i8`, компилятор гарантирует, что у нас всегда есть допустимое значение типа. Мы можем уверенно продолжать работу, не проверяя его на null перед использованием. Однако, когда у нас есть значение типа `Option<T>` (где `T` - это любое значение любого типа `T`, упакованное в `Option`, например значение типа `i8` или `String`), мы должны беспокоиться о том, что значение типа `T` возможно не имеет значения (является вариантом `None`), и компилятор позаботится о том, чтобы мы обработали такой случай, прежде чем мы бы попытались использовать `None` значение.

Другими словами, вы должны преобразовать `Option<T>` в `T` прежде чем вы сможете выполнять операции с этим `T`. Как правило, это помогает выявить одну из наиболее распространённых проблем с null: когда мы предполагаем, что что-то не равно null, хотя *на самом деле* оно null.

С Rust не нужно беспокоиться о неправильном предположении касательно не-null значения, это помогает чувствовать себя более уверенно. Для того, чтобы иметь значение, которое может быть null, вы должны явно сказать об этом, указав тип `T` этого значения как `Option<T>` (*обернуть его в Option*). Затем, когда вы используете это значение, вы обязаны явно обрабатывать случай, когда значение равно null. Везде, где значение имеет тип, не являющийся `Option<T>`, вы можете смело рассчитывать на то, что значение не равно null. Такой подход - продуманное проектное решение в Rust, ограничивающее распространение null и увеличивающее безопасность Rust кода.

Итак, как мы можем получить желанное значение типа `T`, упакованное в варианте `Some` типа `Option`, когда у нас на руках есть только значение типа `Option<T>?` `Option<T>` имеет большое количество методов, которые полезны в различных

ситуациях; можно проверить их в [документации](#). Знакомство с методами в `Option<T>` будет чрезвычайно полезным в вашем путешествии по языку Rust.

В общем случае, чтобы использовать значение `Option<T>`, нужен код, который будет обрабатывать все варианты перечисления `Option<T>`. Вам понадобится некоторый код, который будет работать только тогда, когда у вас есть значение `Some(T)`, и этому коду разрешено использовать внутреннее `T`. Также вам понадобится другой код, который будет работать, если у вас есть значение `None`, и у этого кода не будет доступного значения `T`. Выражение `match` — это конструкция управления потоком выполнения программы, которая делает именно это при работе с перечислениями: она запускает разный код в зависимости от того, какой вариант перечисления имеется, и этот код может использовать данные, находящиеся внутри совпадшего варианта.

Оператор управления потоком выполнения `match`

В Rust есть невероятно мощная конструкция управления потоком выполнения программы под названием `match`, которая позволяет вам сравнивать значение с серией шаблонов и затем выполнять код, связанный с совпадшим шаблоном. Шаблонами могут выступать литералы, имена переменных, подстановочные значения и многое другое; в Главе 18 описаны все разновидности шаблонов и что они делают. Сила `match` проистекает из выразительности шаблонов и того факта, что компилятор подтверждает, что все возможные случаи обрабатываются.

Думайте о выражении `match` как о машине для сортировки монет: монеты скользят вниз по дорожке с отверстиями разного размера и каждая монета проваливается в первое отверстие, в которое она проходит. Таким же образом значения проходят через каждый шаблон в конструкции `match` и при первом же совпадении с шаблоном значение "проводится" в соответствующий блок кода для дальнейшего использования.

Поскольку мы только что упомянули монеты, давайте использовать их в качестве примера, используя `match`! Можно написать функцию, которая возьмёт неизвестную монету Соединённых Штатов и, подобно счётной машине, определит какая это монета и вернёт её значение в центах, как показано в листинге 6-3.

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Листинг 6-3: Перечисление и выражение `match`, которое использует варианты перечисления в качестве шаблонов

Давайте разберём `match` в функции `value_in_cents`. Сначала пишется ключевое слово `match`, затем следует выражение, которое в данном случае является

значением `coin`. Это выглядит очень похоже на выражение `if`, но есть большая разница: с `if` выражение должно возвращать булево значение, а здесь это может быть любой тип. Тип `coin` в этом примере - перечисление типа `Coin`, объявленное в строке 1.

Далее идут ветки `match`. Ветки состоят из двух частей: шаблон и некоторый код. Здесь первая ветка имеет шаблон, который является значением `Coin::Penny`, затем идёт оператор `=>`, который разделяет шаблон и код для выполнения. Код в этом случае - это просто значение `1`. Каждая ветка отделяется от последующей при помощи запятой.

Когда выполняется выражение `match`, оно сравнивает полученное значение с образцом каждой ветки по порядку. Если шаблон совпадает со значением, то выполняется код, связанный с этим шаблоном. Если этот шаблон не соответствует значению, то выполнение продолжается со следующей ветки, так же, как в автомате по сортировке монет. У нас может быть столько веток, сколько нужно: в листинге 6-3 наш `match` состоит из четырёх веток.

Код, связанный с каждой веткой, является выражением, а полученное значение выражения в соответствующей ветке — это значение, которое возвращается для всего выражения `match`.

Фигурные скобки обычно не используются, если код ветки короткий, как в листинге 6-3, где каждая ветка только возвращает значение. Если необходимо выполнить несколько строк кода в ветке, можно использовать фигурные скобки. Например, следующий код будет выводить «Lucky penny!» каждый раз, когда метод вызывается со значением `Coin::Penny`, но возвращаться при этом будет результат последнего выражения в блоке, то есть значение `1`:

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => {
            println!("Lucky penny!");
            1
        }
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Шаблоны, которые привязывают значения

Есть ещё одно полезное качество у веток в выражении `match`: они могут привязываться к частям тех значений, которые совпали с шаблоном. Благодаря этому можно извлекать значения из вариантов перечисления.

В качестве примера, давайте изменим один из вариантов перечисления так, чтобы он хранил в себе данные. С 1999 по 2008 год Соединённые Штаты чеканили 25 центов с различным дизайном на одной стороне для каждого из 50 штатов. Ни одна другая монета не получила дизайна штата, только четверть доллара имела эту дополнительную особенность. Мы можем добавить эту информацию в наш `enum` путём изменения варианта `Quarter` и включить в него значение `UsState`, как сделано в листинге 6-4.

```
#[derive(Debug)] // so we can inspect the state in a minute
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

Листинг 6-4: Перечисление `Coin`, где вариант `Quarter` содержит также значение `UsState`

Давайте представим, что наш друг пытается собрать четвертаки всех 50 штатов. Пока мы сортируем мелочь по типу монет, мы также будем печатать имя штата, связанное с каждым четвертаком. Таким образом, если у нашего друга ещё нет такой монеты, то её можно добавить в его коллекцию.

В выражении `match` для этого кода мы добавляем переменную с именем `state` в шаблон, который соответствует значениям варианта `Coin::Quarter`. Когда `Coin::Quarter` совпадёт с шаблоном, переменная `state` будет привязана к значению штата этого четвертака. Затем мы сможем использовать `state` в коде этой ветки, вот так:

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?}!", state);
            25
        }
    }
}
```

Если мы сделаем вызов функции `value_in_cents(Coin::Quarter(UsState::Alaska))`, то `coin` будет иметь значение `Coin::Quarter(UsState::Alaska)`. Когда мы будем сравнивать это значение с каждой из веток, ни одна из них не будет совпадать, пока мы не достигнем варианта `Coin::Quarter(state)`. В этот момент `state` привязывается к значению `UsState::Alaska`. Затем мы сможем использовать эту привязку в выражении `println!`, получив таким образом внутреннее значение варианта `Quarter` перечисления `Coin`.

Сопоставление шаблона для Option<T>

В предыдущем разделе мы хотели получить внутреннее значение `T` для случая `Some` при использовании `Option<T>`; мы можем обработать тип `Option<T>` используя `match`, как уже делали с перечислением `Coin`! Вместо сравнения монет мы будем сравнивать варианты `Option<T>`, независимо от этого изменения механизм работы выражения `match` останется прежним.

Допустим, мы хотим написать функцию, которая принимает `Option<i32>` и если есть значение внутри, то добавляет 1 к существующему значению. Если значения нет, то функция должна возвращать значение `None` и не пытаться выполнить какие-либо операции.

Такую функцию довольно легко написать благодаря выражению `match`, код будет выглядеть как в листинге 6-5.

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

Листинг 6-5: Функция, которая использует выражение `match` с типом `Option<i32>`

Давайте рассмотрим процесс выполнения функции `plus_one` более подробно. Когда мы вызываем `plus_one(five)`, то переменная `x` в теле `plus_one` будет иметь значение `Some(5)`. Затем мы сравниваем это значение с каждой веткой выражения `match`.

`None => None,`

Значение `Some(5)` не соответствует шаблону `None`, поэтому мы продолжаем со следующей ветки.

`Some(i) => Some(i + 1),`

Совпадает ли `Some(5)` с шаблоном `Some(i)`? Да, это так! У нас такой же вариант. Тогда переменная `i` привязывается к значению, содержащемуся внутри `Some`, поэтому `i` получает значение `5`. Затем выполняется код ассоциированный для данной ветки, поэтому мы добавляем 1 к значению `i` и создаём новое значение `Some` со значением `6` внутри.

Теперь давайте рассмотрим второй вызов `plus_one` в листинге 6-5, где `x` является `None`. Мы входим в выражение `match` и сравниваем значение с первой веткой.

`None => None,`

Оно совпадает! Для данной ветки шаблон (`None`) не подразумевает наличие какого-то значения к которому можно было бы что-то добавить, поэтому программа останавливается и возвращает значение которое находится справа от `=>` - т.е. `None`. Так как шаблон первой ветки совпал, то никакие другие шаблоны веток не сравниваются.

Комбинирование `match` и перечислений полезно во многих ситуациях. Вы много где сможете увидеть подобный шаблон в коде программ на Rust: сделать сопоставление значения используя один из шаблонов `match`, привязать данные входного значения к данным внутри ветки, выполнить код на основе привязанных данных. Сначала это может показаться немного сложным, но как только вы привыкнете, то захотите чтобы такая возможность была бы во всех языках. Это неизменно любимый пользователями приём.

Match объемлет все варианты значения

Есть ещё один аспект выражения `match`, который необходимо обсудить.

Рассмотрим версию нашей функции `plus_one`, которая имеет ошибку и не будет компилироваться:

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        Some(i) => Some(i + 1),
    }
}
```



Мы не обработали вариант `None`, поэтому этот код вызовет дефект в программе. К счастью, Rust знает и умеет ловить такой случай. Если мы попытаемся скомпилировать такой код, мы получим ошибку компиляции:

```
$ cargo run
Compiling enums v0.1.0 (file:///projects/enums)
error[E0004]: non-exhaustive patterns: `None` not covered
--> src/main.rs:3:15
   |
3 |     match x {
   |         ^ pattern `None` not covered
   |
   = help: ensure that all possible cases are being handled, possibly by
adding wildcards or more match arms
   = note: the matched value is of type `Option<i32>`

For more information about this error, try `rustc --explain E0004`.
error: could not compile `enums` due to previous error
```

Rust знает, что мы не обработали все возможные варианты входного значения, и даже знает какие ветки с какими шаблонами мы забыли добавить! Сравнение по шаблону в Rust является *полными и исчерпывающими* (exhaustive): мы должны

обработать все возможные варианты до конца, чтобы код был корректным в понимании компилятора. Особенно в случае `Option<T>`, когда Rust не позволит нам забыть обработать случай `None` и защитит нас от ошибочного предположения, о том, что у нас *всегда* есть значение, хотя *на самом деле* мы могли бы получить `null`. Таким образом не дают допустить ошибку на миллиард долларов, рассмотренную ранее.

Универсальные шаблоны и заполнитель

Используя перечисления, мы также можем выполнять специальные действия для нескольких определённых значений, а для всех остальных значений выполнять одно действие по умолчанию. Представьте, что мы реализуем игру, в которой при выпадении 3 игрок не двигается, а получает новую модную шляпу. Если выпадает 7, игрок теряет шляпу. При всех остальных значениях ваш игрок перемещается на столько-то мест на игровом поле. Вот `match`, реализующий эту логику, в котором результат броска костей жёстко закодирован, а не является случайным значением, а вся остальная логика представлена функциями без тел, поскольку их реализация не входит в рамки данного примера:

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    other => move_player(other),
}

fn add_fancy_hat() {}
fn remove_fancy_hat() {}
fn move_player(num_spaces: u8) {}
```

Для первых двух веток шаблонами являются литеральные значения 3 и 7. Для последней ветки, которая охватывает все остальные возможные значения, шаблоном является переменная, которую мы решили назвать `other`. Код, выполняемый для другой ветки, использует эту переменную, передавая её в функцию `move_player`.

Этот код компилируется, даже если мы не перечислили все возможные значения `u8`, потому что последний паттерн будет соответствовать всем значениям, не указанным в конкретном списке. Этот универсальный шаблон удовлетворяет требованию, что соответствие должно быть исчерпывающим. Обратите внимание, что мы должны поместить ветку `catch-all` последней, потому что шаблоны

оцениваются по порядку. Rust предупредит нас, если мы добавим ветки после **catch-all**, потому что эти последующие ветки никогда не будут совпадать!

В Rust также есть шаблон, который можно использовать, когда мы не хотим использовать значение в шаблоне catch-all: `_`, который является специальным шаблоном, который соответствует любому значению и не привязывается к этому значению. Это говорит Rust, что мы не собираемся использовать это значение, поэтому Rust не будет предупреждать нас о неиспользуемой переменной.

Давайте изменим правила игры так: если выпадает что-то, кроме 3 или 7, нужно бросить ещё раз. Нам не нужно использовать значение в этом случае, поэтому мы можем изменить наш код, чтобы использовать `_` вместо переменной с именем **other**:

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    _ => reroll(),
}

fn add_fancy_hat() {}
fn remove_fancy_hat() {}
fn reroll() {}
```

Этот пример также удовлетворяет требованию исчерпывающей полноты, поскольку мы явно игнорируем все остальные значения в последней ветке; мы ничего не забыли.

Если мы изменим правила игры ещё раз, чтобы в ваш ход не происходило ничего другого, если вы бросаете не 3 или 7, мы можем выразить это, используя единичное значение (пустой тип кортежа, о котором мы упоминали в разделе "Тип кортежа") в качестве кода, который идёт вместе с веткой `_`:

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    _ => (),
}

fn add_fancy_hat() {}
fn remove_fancy_hat() {}
```

Здесь мы явно говорим Rust, что не собираемся использовать никакое другое значение, которое не соответствует шаблону в предыдущем плече, и не хотим запускать никакой код в этом случае.

Подробнее о шаблонах и совпадениях мы поговорим в [Главе 18](#). Пока же мы перейдём к синтаксису `if let`, который может быть полезен в ситуациях, когда выражение соответствия слишком многословно.

Компактное управление потоком выполнения с `if let`

Синтаксис `if let` позволяет скомбинировать `if` и `let` в менее многословную конструкцию, и затем обработать значения соответствующие только одному шаблону, одновременно игнорируя все остальные. Рассмотрим программу, в которой мы делаем поиск по шаблону значения `Option<u8>`, чтобы выполнить код только когда значение равно 3:

```
let config_max = Some(3u8);
match config_max {
    Some(max) => println!("The maximum is configured to be {}", max),
    _ => (),
}
```

Листинг 6-6. Выражение `match` которое выполнит код только при значении равном `Some(3)`

Мы хотим выполнить что-нибудь при совпадении значения с `Some(3)` и не хотим ничего делать с любым другим `Some<u8>` или значением `None`. Для удовлетворения `match`, после первой и единственной ветки, нам пришлось добавить дополнительный шаблонный код: ветку `_ => ()`.

Вместо этого мы могли бы решить нашу задачу более коротким способом, используя `if let`. Следующий код ведёт себя так же, как выражение `match` в листинге 6-6:

```
let config_max = Some(3u8);
if let Some(max) = config_max {
    println!("The maximum is configured to be {}", max);
}
```

Синтаксис `if let` принимает шаблон и выражение, разделённые знаком равенства. `if let` сработает так же, как `match`, когда в него на вход передадут выражение и подходящим шаблоном для этого выражения окажется первая ветка.

Используя `if let` мы меньше печатаем, меньше делаем отступов и меньше получаем шаблонного кода. Тем не менее, мы теряем полную проверку всех вариантов, предоставляемую выражением `match`. Выбор между `match` и `if let` зависит от того, что вы делаете в вашем конкретном случае и является ли получение краткости при потере полноты проверки подходящим компромиссом.

Другими словами, вы можете думать о конструкции `if let` как о синтаксическом сахаре для `match`, который выполнит код `match` только тогда, когда входное значение будет соответствовать единственному шаблону конструкции, а затем проигнорирует все остальные значения.

Можно добавлять `else` к `if let`. Блок кода, который находится внутри `else` аналогичен по смыслу блоку кода ветки связанной с шаблоном `_` выражения `match` (которое эквивалентно сборной конструкции `if let` и `else`). Вспомним объявление перечисления `Coin` в листинге 6-4, где вариант `Quarter` также содержит внутри значение штата типа `UsState`. Если бы мы хотели посчитать все монеты не являющиеся четвертьми, а для четвертей печатать название штата, то мы могли бы сделать это с помощью выражения `match` таким образом:

```
let mut count = 0;
match coin {
    Coin::Quarter(state) => println!("State quarter from {:+}!", state),
    _ => count += 1,
}
```

Или мы могли бы использовать выражение `if let` и `else` так:

```
let mut count = 0;
if let Coin::Quarter(state) = coin {
    println!("State quarter from {:+}!", state);
} else {
    count += 1;
}
```

Если у вас есть ситуация в которой ваша программа имеет логику которая слишком многословна для того чтобы её выражать используя `match`, помните, о том, что также в вашем наборе инструментов Rust есть `if let`.

Итоги

Мы рассмотрели как использовать перечисления для создания пользовательских типов, которые могут быть одним из наборов перечисляемых значений. Мы показали, как тип `Option<T>` из стандартной библиотеки помогает использовать систему типов для предотвращения ошибок. А когда значения перечисления имеют данные внутри них, можно использовать `match` или `if let`, чтобы извлечь и пользоваться значением, в зависимости от того, сколько случаев нужно обработать.

Теперь ваши программы Rust могут выражать концепции вашей предметной области используя структуры и перечисления. Создание и использование пользовательских типов в API обеспечивает *типовую безопасность*, *type safety*, вашего API: компилятор позаботится о том, чтобы функции получали значения только того типа, который они ожидают.

Чтобы предоставить хорошо организованный API пользователям, необходимо использовать и показывать только то, что нужно пользователям, давайте теперь обратимся к модулям в Rust.

Управление растущими проектами с помощью пакетов, крейтов и модулей

По мере роста кодовой базы ваших программ, организация проекта будет иметь большое значение, ведь отслеживание всей программы в голове будет становиться всё более сложным. Группируя связанные функции и разделяя код по основным функциональностям, (*фичам, feature*), вы делаете более прозрачным понимание о том, где искать код реализующий определённую функцию и где стоит вносить изменения для того чтобы изменить её поведение.

Программы, которые мы писали до сих пор, были в одном файле одного модуля. По мере роста проекта, мы можем организовывать код иначе, разделив его на несколько модулей и несколько файлов. Пакет может содержать несколько бинарных крейтов и дополнительно один крейт библиотеки. По мере роста пакета мы также можем извлекать части нашей программы в отдельные крейты, которые затем станут внешними зависимостями для основного кода нашей программы. Эта глава охватывает все эти техники. В свою очередь для очень крупных проектов, состоящих из набора взаимосвязанных пакетов развивающихся вместе, Cargo предоставляет рабочие пространства, *workspaces*, их мы рассмотрим за пределами данной главы, в разделе "["Рабочие пространства Cargo"](#)" Главы 14.

Дополнительно к группированию функциональности, инкапсуляция деталей реализации позволяет повторно использовать код на более высоком уровне: после реализации операции, другой код может вызывать этот код через открытый интерфейс, не зная как работает реализация. То, как вы пишете код, определяет какие части общедоступны для использования другим кодом и какие части являются закрытыми деталями реализации для которых вы оставляете право на изменения только за собой. Это ещё один способ ограничить количество деталей, которые вы должны держать в голове.

Связанное понятие - это область видимости: вложенный контекст в котором написан код имеющий набор имён, которые определены «в текущей области видимости». При чтении, письме и компиляции кода, программистам и компиляторам необходимо знать, относится ли конкретное имя в определённом месте к переменной, к функции, к структуре, к перечислению, к модулю, к константе или другому элементу и что означает этот элемент. Можно создавать области видимости и изменять какие имена входят или выходят за их рамки. Нельзя иметь два элемента с тем же именем в одной области; есть доступные инструменты для разрешения конфликтов имён.

Rust имеет ряд функций, которые позволяют управлять организацией кода, в том числе управлять тем какие детали открыты, какие детали являются частными, какие имена есть в каждой области вашей программы. Эти функции иногда вместе именуемые *модульной системой* включают в себя:

- **Пакеты, Packages**: Функционал Cargo позволяющий собирать, тестировать и делиться крейтами
- **Крейты, Crates**: Дерево модулей, которое создаёт библиотечный или исполняемый файл
- **Модули, Modules** и **use**: Позволяют вместе контролировать организацию, область видимости и конфиденциальность путей
- **Пути, Paths**: способ именования элемента, такого как структура, функция или модуль

В этой главе мы рассмотрим все эти функции, обсудим как они взаимодействуют и объясним, как использовать их для управления областью видимости. К концу у вас должно появиться солидное понимание модульной системы и умение работать с областями видимости на уровне профессионала!

Пакеты и крейты

Первые части модульной системы, которые мы рассмотрим - это пакеты и крейты. Крейт - это исполняемый файл или библиотека. Выделяют два типа крейтов: библиотечный и исполняемый. Библиотечные крейты можно подключать в другие крейты, но нельзя исполнять. Исполняемые же крейты - полная противоположность библиотечным - могут исполняться, но их нельзя подключить в другие крейты. Корень крейта - это исходный файл, на котором запускается и, исходя из которого, составляет корневой модуль вашего крейта Rust компилятор (мы расскажем о корневых модулях в разделе "["Определение модулей для управления областью видимости и конфиденциальностью"](#)"). Пакет состоит из одного или нескольких крейтов, которые предоставляют набор функций. Пакет содержит файл *Cargo.toml* описывающий, как собрать крейты пакета.

Несколько правил определяют, что может содержать пакет. Пакет может содержать не более одной библиотеки. Он может содержать сколько угодно бинарных крейтов, но должен содержать хотя бы один крейт (библиотечный или бинарный).

Давайте пройдёмся по тому, что происходит, когда мы создаём пакет. Сначала введём 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
```

После ввода команды Cargo создал файл *Cargo.toml*, предоставив пакет. Если мы просмотрим содержимое *Cargo.toml*, то не увидим упоминания о *src/main.rs* потому что Cargo следует соглашению, что *src/main.rs* является корнем исполняемого крейта с тем же именем, что и пакет. Аналогично, Cargo знает, что если каталог пакета содержит *src/lib.rs*, то пакет содержит библиотечный крейт с тем же именем, что и пакет, а *src/lib.rs* является корнем библиотечного крейта. Cargo передаёт корневой файл крейта в **rustc** и тот уже создаст библиотеку или бинарный исполняемый файл в зависимости от типа крейта.

Здесь мы имеем пакет, который содержит только *src/main.rs*, то есть содержит только бинарный крейт с именем **my-project**. Если пакет содержит *src/main.rs* и *src/lib.rs*, то он имеет два крейта: библиотечный и исполняемый, оба с одинаковыми

именами в качестве пакета. Пакет может иметь несколько исполняемых крейтов, размещая их файлы в каталоге `src/bin`: каждый файл будет отдельным исполняемым крейтом.

Крейт группирует в области видимости связанные вместе функциональности, поэтому функциональности легко распространить между несколькими проектами. Например, крейт `rand`, который мы использовали в Главе 2 обеспечивает функциональность генерации случайных чисел. Можно использовать эту функциональность в наших собственных проектах, привнося крейт `rand` в область видимости проекта. Вся функциональность предоставляемая крейтом `rand` станет доступна через имя крейта `rand`.

Сохранение функциональности крейта в его собственной области видимости проясняет, является ли конкретная функциональность определённой в нашем крейте или в крейте `rand`, таким образом предотвращая потенциальные конфликты. Например, крейт `rand` предоставляет типаж с именем `Rng`. Мы также можем определить `struct` с именем `Rng` в нашем собственном крейте. Так как функциональность крейта находится в пространстве имён собственной области видимости, то когда мы добавляем `rand` как зависимость, компилятор не смущён ссылкой на имя `Rng`. В нашем крейте ссылка относится к объявленной у нас `struct Rng`. А доступ к типажу `Rng` из крейта `rand` мы бы получили как `rand::Rng`.

Давайте будем двигаться дальше и поговорим о модульной системе!

Определение модулей для контроля видимости и конфиденциальности

В этом разделе мы поговорим о модулях и других частях системы модулей, а именно: *путях(paths)*, которые позволяют именовать элементы; ключевом слове **use**, которое приносит путь в область видимости; ключевом слове **pub**, которое делает элементы общедоступными. Мы также обсудим ключевое слово **as**, внешние пакеты и оператор `glob`. А пока давайте сосредоточимся на модулях!

Модули позволяют организовывать код внутри крейта по группам для удобства чтения и простого повторного использования. Модули также контролируют **конфиденциальность (privacy)** элементов: определяют может элемент использоваться внешним кодом, быть **публичным (public)** или является деталями внутренней реализации и недоступен для внешнего использования, т.е. является **приватным (private)**.

В качестве примера, давайте напишем библиотечный крейт предоставляющий функциональность ресторана. Мы определим сигнатуры функций, но оставим их тела пустыми, чтобы сосредоточиться на организации кода, вместо реализации кода для ресторана.

В ресторанной индустрии некоторые части ресторана называются *фронтом дома*, а другие *задней частью дома*. Фронт дома это там где находятся клиенты; здесь размещаются места клиентов, официанты принимают заказы и оплаты, а бармены делают напитки. Задняя часть дома это где шеф-повара и повара работают на кухне, моют посудомоечные машины, а менеджеры занимаются административной работой.

Чтобы структурировать крейт аналогично тому, как работает настоящий ресторан, можно организовать размещение функций во вложенных модулях. Создадим новую библиотеку (библиотечный крейт) с именем **restaurant** выполнив команду **cargo new --lib restaurant**; затем вставим код из листинга 7-1 в файл `src/lib.rs` для определения некоторых модулей и сигнатур функций.

Файл: `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() {}
    }
}
```

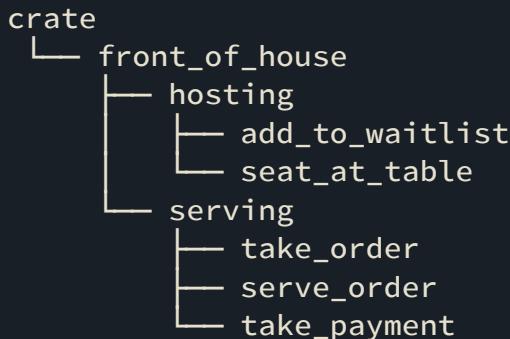
Листинг 7-1. Модуль `front_of_house` содержащий другие модули, которые затем содержат функции

Мы определяем модуль, начиная с ключевого слова `mod`, затем определяем название модуля (в данном случае `front_of_house`) и размещаем фигурные скобки вокруг тела модуля. Внутри модулей, можно иметь другие модули, как в случае с модулями `hosting` и `serving`. Модули также могут содержать определения для других элементов, таких как структуры, перечисления, константы, типажи или - как в листинге 7-1 - функции.

Используя модули, мы можем сгруппировать связанные определения вместе и сказать почему они являются связанными. Программистам, использующим такой код, будет легче найти определения, которые они хотели использовать, потому что они могли бы ориентироваться в сгруппированном коде, вместо того, чтобы прочитать все определения. Программисты, добавляющие новые функции в этот код, будут знать, где разместить код для поддержания порядка в программе.

Как мы упоминали ранее, файлы `src/main.rs` и `src/lib.rs` называются *корнями крейтов*. Причина такого именования в том, что любой из этих двух файлов определяет содержимое и размещает в корне структуры модуля, известной как *дерево модулей*, корневой модуль известный как `crate`.

В листинге 7-2 показано дерево модулей для структуры модулей приведённой в коде листинга 7-1.



Листинг 7-2: Дерево модулей для структуры модулей приведённой в коде в листинге 7-1

Это дерево показывает, как некоторые из модулей вкладываются друг в друга (например, `hosting` находится внутри `front_of_house`). Дерево также показывает, что некоторые модули являются *братьями* (*siblings*) друг для друга, то есть они определены в одном модуле (`hosting` и `serving`) - братья которые определены внутри `front_of_house`). Продолжая метафору с семьёй: если модуль А содержится внутри модуля В, мы говорим, что модуль А является *потомком* (*child*) модуля В, а модуль В является *родителем* (*parent*) модуля А. Обратите внимание, что корнем (отцом, главным предком) всего дерева модулей является неявный модуль с именем `crate`.

Дерево модулей может напомнить вам дерево каталогов файловой системы на компьютере; это очень удачное сравнение! По аналогии с каталогами в файловой системе, мы используемся модули для организации кода. И так же, как нам надо искать файлы в каталогах на компьютере, нам требуется способ поиска нужных модулей.

Ссылаемся на элементы дерева модулей при помощи путей

Чтобы показать Rust, где найти элемент в дереве модулей, мы используем путь так же, как мы используем путь при навигации по файловой системе. Например, если мы хотим вызвать функцию, то нам нужно знать её путь.

Пути бывают двух видов:

- *абсолютный путь* берет своё начало с корня крейта: названия крейта или ключевого слова `crate`,
- *относительный путь* начинается с текущего модуля и использует ключевые слова `self`, `super` или идентификатор в текущем модуле.

Как абсолютные, так и относительные, пути сопровождаются одним или несколькими идентификаторами разделёнными двойными двоеточиями (`:::`).

Давайте вернёмся к примеру в листинге 7-1. Как бы мы вызывали функцию `add_to_waitlist`? Наш вызов был бы похож на путь к функции `add_to_waitlist`? В листинге 7-3 мы немного упростили код листинга 7-1, удалив ненужные модули и функции. Мы покажем два способа вызова функции `add_to_waitlist` из новой функции `eat_at_restaurant` определённой в корне крейта. Функция `eat_at_restaurant` является частью нашей библиотеки публичного API, поэтому мы помечаем её ключевым словом `pub`. В разделе "Раскрытие путей с помощью ключевого слова `pub`", мы рассмотрим более подробно `pub`. Обратите внимание, что этот пример ещё не компилируется; мы скоро объясним почему.

Файл: src/lib.rs

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // Absolute path
    crate::front_of_house::hosting::add_to_waitlist();

    // Relative path
    front_of_house::hosting::add_to_waitlist();
}
```



Листинг 7-3. Вызов функции `add_to_waitlist` с использованием абсолютного и относительного пути

Первый раз, когда мы вызываем функцию `add_to_waitlist` в функции `eat_at_restaurant` мы используем абсолютный путь. Функция `add_to_waitlist` определена в том же крейте что и `eat_at_restaurant` и это означает, что мы можем использовать ключевое слово `crate` в начале абсолютного пути.

После ключевого слова `crate` мы включаем каждый из последующих дочерних модулей, пока не составим путь до `add_to_waitlist`. Вы можете представить файловую систему с такой же структурой, где мы указали бы путь `/front_of_house/hosting/add_to_waitlist` для запуска программы `add_to_waitlist`; мы используем слово `crate`, чтобы начать путь из корня крейта, подобно тому как используется `/` для указания корневой директории файловой системы.

Второй раз, когда мы вызываем `add_to_waitlist` внутри `eat_at_restaurant`, мы используем относительный путь. Путь начинается с имени модуля `front_of_house`, определённого на том же уровне дерева модулей, что и модуль `eat_at_restaurant`. Для относительного пути эквивалентный путь в вымышленной файловой системе выглядел бы так: `front_of_house/hosting/add_to_waitlist`. Начало пути совпадает с именем модуля, что указывает на то, что перед нами относительный путь.

Выбор, использовать относительный или абсолютный путь, является решением, которое вы примете на основании вашего проекта. Решение должно зависеть от того, с какой вероятностью вы переместите объявление элемента отдельно от или вместе с кодом использующим этот элемент. Например, в случае перемещения модуля `front_of_house` и его функции `eat_at_restaurant` в другой модуль с именем `customer_experience`, будет необходимо обновить абсолютный путь до `add_to_waitlist`, но относительный путь все равно будет действителен. Однако, если мы переместим отдельно функцию `eat_at_restaurant` в модуль с именем `dining`, то абсолютный путь вызова `add_to_waitlist` останется прежним, а относительный путь нужно будет обновить. Мы предпочитаем указывать абсолютные пути, потому что это позволяет проще перемещать определения кода и вызовы элементов независимо друг от друга.

Давайте попробуем скомпилировать листинг 7-3 и выяснить, почему он ещё не компилируется. Ошибка, которую мы получаем, показана в листинге 7-4.

```
$ 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();
  |     ^^^^^^^^^^ private module
  |
note: the module `hosting` is defined here
--> src/lib.rs:2:5
  |
2 |     mod hosting {
  |     ^^^^^^^^^^^^
  |
error[E0603]: module `hosting` is private
--> src/lib.rs:12:21
  |
12 |         front_of_house::hosting::add_to_waitlist();
  |         ^^^^^^^^^^ private module
  |
note: the module `hosting` is defined here
--> src/lib.rs:2:5
  |
2 |     mod hosting {
  |     ^^^^^^^^^^^^
  |
For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` due to 2 previous errors
```

Листинг 7-4. Ошибки компиляции при сборке кода из листинга 7-3

Сообщения об ошибках говорят о том, что модуль `hosting` является приватным. Таким образом, у нас есть правильные пути к модулю `hosting` и функции `add_to_waitlist`, но Rust не позволяет использовать их, потому что он не имеет доступа к разделам которые являются приватными, как в нашем случае.

Модули не только полезны для организации кода. Они также определяют *границы конфиденциальности* (privacy boundary) в Rust: граница, которая инкапсулирует детали реализации, которые внешний код не может знать, вызывать или полагаться на них. Итак, если вы хотите сделать элемент приватным, например функцию или структуру, то разместите его в модуль.

В Rust конфиденциальность (privacy) работает так, что все элементы (функции, методы, структуры, перечисления, модули и константы) являются по умолчанию приватными. Элементы в родительском модуле не могут использовать приватные элементы внутри дочерних модулей, но элементы в дочерних модулях могут

использовать элементы у своих родительских модулей. Причина в том, что дочерние модули обворачивают и скрывают детали своей реализации, но модули могут видеть контекст, в котором они определены. Продолжая метафору с рестораном, думайте о правилах конфиденциальности как о задней части ресторана: то что там происходит скрыто от клиентов ресторана, но открыто для менеджеров ресторана: они могут видеть и управлять рестораном в котором они все работают.

В Rust решили, что система модулей должна функционировать таким образом, чтобы по умолчанию скрывать детали реализации. Таким образом, вы знаете, какие части внутреннего кода вы можете изменять не нарушая работы внешнего кода. Но у нас всё же остаётся возможность раскрывать внутренние части кода дочерних модулей для внешних модулей - предков. Чтобы сделать элемент публичным мы можем использовать ключевое слово `pub`.

Раскрываем приватные пути с помощью ключевого слова `pub`

Давайте вернёмся к ошибке в листинге 7-4, которая говорит что модуль `hosting` является приватным. Мы хотим, чтобы функция `eat_at_restaurant` представленная в родительском модуле `eat_at_restaurant` имела доступ к функции `add_to_waitlist` в дочернем модуле, поэтому мы помечаем модуль `hosting` с ключевым словом `pub`, как показано в листинге 7-5.

Файл: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // Absolute path
    crate::front_of_house::hosting::add_to_waitlist();

    // Relative path
    front_of_house::hosting::add_to_waitlist();
}
```



Листинг 7-5. Объявление модуля `hosting` как `pub` для его использования из `eat_at_restaurant`

К сожалению, код в листинге 7-5 всё ещё приводит к ошибке, как показано в

листеинге 7-6.

```
$ 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();
|                                     ^^^^^^^^^^^^^^^^^^ private function
|
note: the function `add_to_waitlist` is defined here
--> src/lib.rs:3:9
3 |     fn add_to_waitlist() {}
|     ^^^^^^^^^^^^^^^^^^^^^^

error[E0603]: function `add_to_waitlist` is private
--> src/lib.rs:12:30
12 |     front_of_house::hosting::add_to_waitlist();
|                                     ^^^^^^^^^^^^^^ private function
|
note: the function `add_to_waitlist` is defined here
--> src/lib.rs:3:9
3 |     fn add_to_waitlist() {}
|     ^^^^^^^^^^^^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` due to 2 previous errors
```

Листинг 7-6: Ошибки компиляции при сборке кода в листинге 7-5

Что произошло? Добавление ключевого слова **pub** перед **mod hosting** сделало модуль публичным. После этого изменения, если мы можем получить доступ к модулю **front_of_house**, то мы можем доступ к модулю **hosting**. Но *содержимое* модуля **hosting** всё ещё является приватным: превращение модуля в публичный не делает его содержимое публичным. Ключевое слово **pub** позволяет внешнему коду в модулях предках обращаться только к модулю.

Ошибки в листинге 7-6 говорят, что функция **add_to_waitlist** является закрытой. Правила конфиденциальности применяются к структурам, перечислениям, функциям и методам, также как и к модулям.

Давайте также сделаем функцию **add_to_waitlist** общедоступной, добавив ключевое слово **pub** перед её определением, как показано в листинге 7-7.

Файл: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // Absolute path
    crate::front_of_house::hosting::add_to_waitlist();

    // Relative path
    front_of_house::hosting::add_to_waitlist();
}
```

Листинг 7-7. Добавление ключевого слова `pub` у модуля `mod hosting` и функции `fn add_to_waitlist` позволяет вызывать ранее скрытую функцию из функции `eat_at_restaurant`

Теперь код компилируется! Давайте посмотрим на абсолютный и относительный путь и перепроверим, почему добавление ключевого слова `pub` позволяет использовать эти пути в `add_to_waitlist` с учётом правила конфиденциальности.

В случае абсолютного пути мы начинаем с `crate`, корня дерева модуля нашего крейта. Затем в корне крейта определён модуль `front_of_house`. Модуль `front_of_house` приватный, потому что функция `eat_at_restaurant` определена в том же модуле, что и `front_of_house` (то есть `eat_at_restaurant` и `front_of_house` являются родственными), мы можем сослаться на `front_of_house` из `eat_at_restaurant`. Затем идёт модуль `hosting`, он также помечен с помощью `pub`. Мы можем получить доступ к родительскому модулю `hosting`, по этому `hosting` также доступен. Наконец функция `add_to_waitlist` тоже помечена как `pub`, и в то же время можно получить доступ к её родительскому модулю, значит вызов функции работает!

В случае относительного пути логика совпадает со случаем абсолютного пути, за исключением первого шага: вместо того, чтобы начинать с корня крейта, путь начинается с `front_of_house`. Модуль `front_of_house` определён в том же модуле, что и `eat_at_restaurant`, поэтому относительный путь, начинающийся с модуля в котором определён `eat_at_restaurant` тоже работает. Тогда, по причине того, что `hosting` и `add_to_waitlist` помечены как `pub`, остальная часть пути работает и вызов функции действителен!

Начинаем относительный путь с помощью `super`

Также можно построить относительные пути, которые начинаются в родительском модуле, используя ключевое слово `super` в начале пути. Это похоже на синтаксис начала пути файловой системы `..`. Зачем нам так делать?

Рассмотрим код в листинге 7-8, который моделирует ситуацию в которой повар исправляет неправильный заказ и лично выдаёт его клиенту. Функция `fix_incorrect_order` вызывает функцию `deliver_order`, указывая путь к `deliver_order` начинаящийся со `super`:

Файл: src/lib.rs

```
fn deliver_order() {}

mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        super::deliver_order();
    }

    fn cook_order() {}
}
```

Листинг 7-8: Вызов функции с использованием относительного пути, начинающегося со `super`

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 этом случае является корнем `crate`. Из корня мы пытаемся найти `deliver_order` и находим его. Успех! Мы считаем, что модуль `back_of_house` и функция `deliver_order` остаются в одинаковых отношениях друг с другом и должны быть перемещены вместе, если мы решим реорганизовать дерево модулей крейта. Поэтому мы использовали `super`. В итоге, в будущем нам не понадобится обновлять путь до модуля, при перемещении кода в другой модуль.

Делаем публичными структуры и перечисления

Мы также можем использовать `pub`, чтобы сделать структуры и перечисления публичными, но есть несколько дополнительных деталей. Если используется `pub` перед определением структуры, то структура становится публичной, но поля структуры все ещё остаются приватными. Делать ли каждое поле публичным или

нет решается в каждом конкретном случае. В листинге 7-9 мы определили публичную структуру `back_of_house::Breakfast` с открытым полем `toast`, но оставили приватным поле `seasonal_fruit`. Это моделирует случай в ресторане, когда клиент может выбрать тип хлеба к блюду, но повар решает, какие фрукты сопровождают блюдо на основании того, какой сейчас сезон и что есть на складе. Доступные фрукты быстро меняются, поэтому покупатели не могут выбирать фрукты или даже посмотреть, какие фрукты они получат.

Файл: 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("peaches"),
            }
        }
    }

    pub fn eat_at_restaurant() {
        // Order a breakfast in the summer with Rye toast
        let mut meal = back_of_house::Breakfast::summer("Rye");
        // Change our mind about what bread we'd like
        meal.toast = String::from("Wheat");
        println!("I'd like {} toast please", meal.toast);

        // The next line won't compile if we uncomment it; we're not allowed
        // to see or modify the seasonal fruit that comes with the meal
        // meal.seasonal_fruit = String::from("blueberries");
    }
}
```

Листинг 7-9: Структура с публичными и приватными полями

Поскольку поле `toast` в структуре `back_of_house::Breakfast` является открытым, то в функции `eat_at_restaurant` можно писать и читать поле `toast`, используя точечную нотацию. Обратите внимание, что мы не можем использовать поле `seasonal_fruit` в `eat_at_restaurant`, потому что `seasonal_fruit` является приватным. Попробуйте убрать комментирование с последней строки для значения поля `seasonal_fruit`, чтобы увидеть какую ошибку вы получите!

Также обратите внимание, что поскольку `back_of_house::Breakfast` имеет приватное поле, то структура должна предоставить публичную ассоциированную функцию, которая создаёт экземпляр `Breakfast` (мы назвали её `summer`). Если `Breakfast` не имел бы такой функции, мы бы не могли создать экземпляр `Breakfast` внутри `eat_at_restaurant`, потому что мы не смогли бы установить значение приватного поля `seasonal_fruit` в функции `eat_at_restaurant`.

В отличии от структуры, если мы сделаем перечисление публичным, то все его варианты будут публичными. Нужно только указать `pub` перед ключевым словом `enum`, как в листинге 7-10.

Файл: 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;
}
```

Листинг 7-10. Определяя перечисление публичным мы делаем все его варианты публичными

Поскольку мы сделали публичным список `Appetizer`, то можно использовать варианты `Soup` и `Salad` в функции `eat_at_restaurant`. Перечисления не очень полезны, если их варианты являются приватными: было бы досадно каждый раз аннотировать все перечисленные варианты как `pub`. По этой причине по умолчанию варианты перечислений являются публичными. Структуры часто полезны, если их поля не являются открытыми, поэтому поля структуры следуют общему правилу, согласно которому всё по умолчанию является приватными, если не указано `pub`.

Есть ещё одна ситуация с `pub` которую мы не освещали, и это последняя особенность модульной системы: ключевое слово `use`. Мы сначала опишем `use` само по себе, а затем покажем как сочетать `pub` и `use` вместе.

Подключение путей в область видимости с помощью ключевого слова `use`

Может показаться, что пути, которые мы писали для вызова функций неудобные, длинные и повторяющиеся. Например, в листинге 7-7, где мы выбирали абсолютный или относительный путь к функции `add_to_waitlist`, каждый раз для вызова `add_to_waitlist` мы должны были указать модули `front_of_house` и `hosting`. К счастью, есть способ упрощения этого процесса. Можно подключить путь в область видимости один раз, а затем вызывать элементы из этого пути будто это локальные элементы используя ключевое слово `use`.

В листинге 7-11 мы подключили модуль `crate::front_of_house::hosting` в область действия функции `eat_at_restaurant`, поэтому нам достаточно только указать `hosting::add_to_waitlist` для вызова функции `add_to_waitlist` внутри `eat_at_restaurant`.

Файл: 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();
}
```

Листинг 7-11. Подключение модуля в область видимости с помощью `use`

Добавление `use` и пути в область видимости аналогично созданию символьической ссылки в файловой системе. Добавляя `use crate::front_of_house::hosting` в корень крейта, `hosting` теперь является допустимым именем в этой области, как если бы `hosting` модуль был определён в корне крейта. Пути подключённые в область видимости с помощью `use` также проверяют конфиденциальность как и любые другие пути.

Также можно подключить элемент в область видимости с помощью `use` и относительного пути. Листинг 7-12 показывает как указать относительный путь, чтобы получить то же поведение, что и в листинге 7-11.

Файл: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

mod customer {
    pub fn eat_at_restaurant() {
        hosting::add_to_waitlist();
    }
}
```

Листинг 7-12. Подключение модуля в область видимости с помощью `use` и относительного пути

Создание идиоматических путей с `use`

В листинге 7-11 вы могли бы задаться вопросом, почему мы указали `use crate::front_of_house::hosting`, а затем вызвали `hosting::add_to_waitlist` внутри `eat_at_restaurant` вместо указания в `use` полного пути прямо до функции `add_to_waitlist` для получения того же результата, что в листинге 7-13.

Файл: 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();
}
```

Листинг 7-13. Подключение функции `add_to_waitlist` в область видимости с помощью `use` не идиоматическим способом

Хотя листинги 7-11 и 7-13 выполняют одну и ту же задачу, листинг 7-11 является идиоматическим способом подключения функции в область видимости с помощью

`use`. Подключение родительского модуля функции в область видимости при помощи `use`, и последующее указание родительского модуля в строке вызова его функций, даёт ясное понимание того, что эта функция определена не локально, и в то же время всё ещё минимизирует повторение полного пути. В коде листинга 7-13 не ясно, где именно определена `add_to_waitlist`.

С другой стороны, при подключении структур, перечислений и других элементов используя `use`, идиоматически правильным будет указывать полный путь. Листинг 7-14 показывает идиоматический способ подключения структуры стандартной библиотеки `HashMap` в область видимости исполняемого крейта.

Файл: src/main.rs

```
use std::collections::HashMap;

fn main() {
    let mut map = HashMap::new();
    map.insert(1, 2);
}
```

Листинг 7-14. Подключение `HashMap` в область видимости идиоматическим способом

За этой идиомой нет веской причины: это просто соглашение, которое появилось само собой. Люди привыкли читать и писать код Rust таким образом.

Исключением из этой идиомы является случай, когда мы подключаем два элемента с одинаковыми именами в область видимости используя оператор `use` - Rust просто не позволяет этого сделать. Листинг 7-15 показывает, как подключить в область действия два типа с одинаковыми именами `Result`, но из разных родительских модулей и как на них ссылаться.

Файл: src/lib.rs

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

fn function1() -> fmt::Result {
    // --snip--
}

fn function2() -> io::Result<()> {
    // --snip--
}
```

Листинг 7-15. Подключение двух типов с одинаковыми именами в одну область видимости требует использования их родительских модулей.

Как видите, использование имени родительских модулей позволяет различать два типа `Result`. Если бы вместо этого мы указали `use std::fmt::Result` и `use std::io::Result`, мы бы имели два типа `Result` в одной области видимости, и Rust не смог бы понять какой из двух `Result` мы имели в виду когда нашёл бы их употребление в коде.

Предоставление новых имен с помощью ключевого слова `as`

Есть ещё одно решение проблемы объединения двух типов с одинаковыми именами в одной области видимости используя `use`: после пути можно указать `as` и новое локальное имя (псевдоним) для типа. Листинг 7-16 показывает другой способ написать код в листинге 7-15 путём переименования одного из двух типов `Result` используя `as`.

Файл: `src/lib.rs`

```
use std::fmt::Result;
use std::io::Result as IoResult;

fn function1() -> Result {
    // --snip--
}

fn function2() -> IoResult<()> {
    // --snip--
}
```

Листинг 7-16. Переименование типа с помощью ключевого слова `as` при его подключении в область видимости

Во втором операторе `use` мы выбрали новое имя `IoResult` для типа `std::io::Result`, которое теперь не будет конфликтовать с типом `Result` из `std::fmt`, который также подключен в область видимости. Листинги 7-15 и 7-16 считаются идиоматичными, поэтому выбор за вами!

Реэкспорт имён с `pub use`

Когда мы подключаем имя в область видимости используя ключевое слово `use`, то имя доступное в новой области видимости является приватным. Чтобы позволить коду, который вызывает наш код, ссылаться на это имя, как если бы оно было определено в области видимости данного кода, можно объединить `pub` и `use`. Этот метод называется *рээкспортом* (*re-exporting*), потому что мы подключаем элемент в область видимости, но также делаем этот элемент доступным для подключения в других областях видимости.

Листинг 7-17 показывает код как в листинге 7-11 (где используется `use` в корневом модуле), но с изменениями: теперь применяется `pub use`.

Файл: `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();
}
```

Листинг 7-17. Делаем при помощи `pub use` имя доступным для любого кода из новой области видимости

Благодаря использованию `pub use`, внешний код теперь может вызывать функцию `add_to_waitlist` используя `hosting::add_to_waitlist`. Если бы мы не указали `pub use`, то только функция `eat_at_restaurant` могла бы вызывать `hosting::add_to_waitlist` в своей области видимости, но внешний код не смог бы так сделать.

Рээкспорт полезен, когда внутренняя структура вашего кода отличается от того, как другие программистызывающие ваш код, будут думать о предметной области. Например, в метафоре про ресторан, люди работающие в ресторане, думают о «фронтальной части дома» и «задней части дома». Но вероятно что клиенты посещающие ресторан, не буду думать о частях ресторана в таких терминах. С помощью `pub use`, можно структурировать код по одному принципу, но наружу публиковать другие варианты структуризации кода (подходящие под разные предметные области). Благодаря этому мы можем сделать нашу библиотеку удобно

организованной как для программистов, работающих над библиотекой так и для программистов вызывающих нашу библиотеку.

Использование внешних пакетов

В Главе 2 мы запрограммировали игру угадывания числа, где использовался внешний пакет для получения случайного числа, называемый `rand`. Чтобы использовать в нашем проекте пакет `rand`, мы добавили строку в *Cargo.toml*:

Файл: *Cargo.toml*

```
rand = "0.8.3"
```

Добавление `rand` в качестве зависимости в *Cargo.toml* указывает Cargo загрузить пакет `rand` и любые требующиеся для работы этого пакета зависимости из [crates.io](#) и сделать `rand` доступным для нашего проекта.

Затем, чтобы подключить определения `rand` в область видимости нашего пакета, мы добавили строку `use` начинающуюся с названия пакета `rand` и списка элементов, которые мы хотим подключить в область видимости. Напомним, что в разделе "[Генерация случайного числа](#)" Главы 2, мы подключили типаж `Rng` в область видимости и вызвали функцию `rand::thread_rng`:

```
use rand::Rng;

fn main() {
    let secret_number = rand::thread_rng().gen_range(1..=100);
}
```

Члены сообщества Rust сделали много пакетов доступными на ресурсе [crates.io](#), и добавление любого из них в свой пакет включает в себя одни и те же шаги: добавить пакет в файл *Cargo.toml* вашего пакета, использовать `use` для подключения элементов этого пакета в область видимости.

Обратите внимание, стандартная библиотека (`std`) также является крейтом, который является внешним по отношению к нашему пакету. Поскольку стандартная библиотека поставляется с языком Rust, то не нужно вносить изменения в *Cargo.toml* для подключения `std`. Но чтобы добавить её элементы в область видимости нашего пакета, нам нужно сослаться на неё используя `use`. Например, чтобы добавить `HashMap` в область видимости нам потребуется использовать следующую

строку:

```
use std::collections::HashMap;
```

Это абсолютный путь, начинающийся с `std`, имени крейта стандартной библиотеки.

Использование вложенных путей для уменьшения длинных списков `use`

Если мы используем несколько элементов определённых в одном пакете или в том же модуле, то перечисление каждого элемента в отдельной строке может занимать много вертикального пространства в файле. Например, эти два объявления `use` используются в программе угадывания числа (листинг 2-4) для подключения элементов из `std` в область видимости:

Файл: src/main.rs

```
// --snip--  
use std::cmp::Ordering;  
use std::io;  
// --snip--
```

Вместо этого, для того чтобы подключить в область видимости те же элементы одной строкой, можно использовать вложенные пути. Мы делаем это, как показано в листинге 7-18, указывая общую часть пути, за которой следуют два двоеточия, а затем фигурные скобки вокруг списка тех частей продолжения пути, которые отличаются.

Файл: src/main.rs

```
// --snip--  
use std::{cmp::Ordering, io};  
// --snip--
```

Листинг 7-18. Указание вложенных путей для подключения в область видимости нескольких элементов с одинаковым префиксом

В больших программах, подключение множества элементов из одного пакета или модуля с использованием вложенных путей может уменьшить количество

отдельных строк с `use`, в тех случаях когда подключаемых элементов много.

Можно использовать вложенный путь на любом уровне, что полезно при объединении двух операторов `use`, которые имеют общую часть пути. Например, в листинге 7-19 показаны два оператора `use`: один подключает `std::io`, другой подключает `std::io::Write` в область видимости.

Файл: src/lib.rs

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

Листинг 7-19. Два оператора `use` где один содержит часть пути другого

Общей частью этих двух путей является `std::io`, и это полный первый путь. Чтобы объединить эти два пути в одно выражение `use`, мы можем использовать ключевое слово `self` во вложенном пути, как показано в листинге 7-20.

Файл: src/lib.rs

```
use std::io::{self, Write};
```

Листинг 7-20. Объединение путей из листинга 7-19 в один оператор `use`

Эта строка подключает `std::io` и `std::io::Write` в область видимости.

Оператор * (Glob)

Если хотим подключить в область видимости все общие элементы, определённые в пути, можно указать путь за которым следует оператор `*` (звёздочка, *glob*):

```
use std::collections::*;


```

Этот оператор `use` подключает все открытые элементы из модуля `std::collections` в текущую область видимости. Будьте осторожны при использовании оператора `*`! Он может усложнить понимание, какие имена находятся в области видимости и где были определены имена, используемые в вашей программе.

Оператор `*` часто используется при тестировании для подключения всего что есть

в модуле `tests`; мы поговорим об этом в разделе "Как писать тесты" Главы 11. Оператор `*` также иногда используется как часть шаблона *автоматического импорта (prelude)*: смотрите [документацию по стандартной библиотеке](#) для получения дополнительной информации об этом шаблоне.

Разделение модулей на разные файлы

Пока что все примеры в этой главе определяли множество модулей в одном файле. Когда модули становятся большими, можно переместить их определения в отдельный файл, чтобы сделать код проще.

Например, давайте начнём с кода листинга 7-17 и первым шагом переместим модуль `front_of_house` в свой собственный файл `src/front_of_house.rs`, изменив корневой файл крейта так, чтобы он содержал код показанный в листинге 7-21. В этом случае, корневым файлом крейта является `src/lib.rs`, но эта процедура также работает с исполняемыми крейтами у которых корневой файл крейта `src/main.rs`.

Файл: `src/lib.rs`

```
mod front_of_house;

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

Листинг 7-21. Добавление в корневой файл крейта тела модуля `front_of_house` (которое далее будет вынесено в `src/front_of_house.rs`)

И на втором шаге в содержимом `src/front_of_house.rs` определим тело модуля `front_of_house` (которое мы изъяли из `src/lib.rs`), как показано в листинге 7-22.

Файл: `src/front_of_house.rs`

```
pub mod hosting {
    pub fn add_to_waitlist() {}
}
```

Листинг 7-22. Определения тела модуля `front_of_house` в файле `src/front_of_house.rs`

Использование точки с запятой после `mod front_of_house`, вместо объявления начала блока, говорит Rust загрузить содержимое модуля из другого файла имеющего такое же название как и имя модуля. Продолжим наш пример и выделим модуль `hosting` в отдельный файл, а затем поменяем содержимое файла `src/front_of_house.rs` так, чтобы он содержал только объявление модуля `hosting`:

Файл: `src/front_of_house.rs`

```
pub mod hosting;
```

Затем мы создаём каталог `src/front_of_house` и файл `src/front_of_house/hosting.rs` в данной директории. Чтобы вынести модуль мы, так же как и ранее, должны выделить содержимое модуля `hosting` из прежнего места и перенести его в свой файл модуля `hosting.rs`:

Файл: `src/front_of_house/hosting.rs`

```
pub fn add_to_waitlist() {}
```

Дерево модулей остаётся прежним, а вызовы функций в `eat_at_restaurant` будут работать без каких-либо изменений, даже если определения будут в разных файлах. Этот метод позволяет перемещать модули в новые файлы по мере их разрастания.

Обратите внимание на то, что выражение `pub use crate::front_of_house::hosting` в файле `src/lib.rs` не претерпело каких-либо изменений после переноса модулей в отдельные файлы. В то же время благодаря этому `use` не добавило какого-либо влияния на то какие файлы будут скомпилированы как часть крейта. Ключевое слово `mod` объявляет модули, а Rust просматривает файл с тем же именем, что и модуль: так он определяет код, который входит в этот модуль.

Итог

Rust позволяет разбить пакет на несколько крейтов и крейт на модули, так что вы можете ссылаться на элементы определённые в одном модуле из другого модуля. Это можно делать при помощи указания абсолютных или относительных путей. Пути можно подключить в область видимости оператором `use`, поэтому вы можете пользоваться более короткими путями для многократного использования элементов в области видимости. Код модуля по умолчанию является приватным, но можно сделать определения публичными, добавив ключевое слово `pub`.

В следующей главе мы познакомим вас с некоторыми коллекциями (особыми структурами данных) представленными в стандартной библиотеке. Завершив их изучение вы сможете использовать их в своём аккуратно организованном коде.

Коллекции стандартной библиотеки

Стандартная библиотека содержит несколько полезных структур данных, которые называются *коллекциями*. Большая часть других типов данных представляют собой хранение конкретного значения, но особенностью коллекций является хранение множества однотипных значений. В отличии от массива или кортежа данные коллекций хранятся в куче, а это значит, что размер коллекции может быть неизвестен в момент компиляции программы. Он может изменяться (увеличиваться, уменьшаться) во время работы программы. Каждый вид коллекций имеет свои возможности и отличается по производительности, так что выбор конкретной коллекции зависит от ситуации и является умением разработчика, вырабатываемым со временем. В этой главе будет рассмотрено несколько коллекций:

- *Вектор (vector)* - позволяет нам сохранять различное количество последовательно хранящихся значений,
- *Строка (string)* - это последовательность символов. Мы же упоминали тип **String** ранее, но в данной главе мы поговорим о нем подробнее.
- *Хеш таблица (hash map)* - коллекция которая позволяет хранить перечень ассоциаций значения с ключом (перечень пар ключ:значение). Является конкретной реализацией более общей структуры данных называемой *map*.

Для того, чтобы узнать о других видах коллекций предоставляемых стандартной библиотекой смотрите [документацию](#).

Мы обсудим как создавать и обновлять вектора, строки и хеш таблицы, а также объясним что делает каждую из них особенной.

Сохранение списка значений с помощью вектора

Первым типом коллекции, который мы разберём, будет `Vec<T>`, также известный как *вектор* (vector). Векторы позволяют сохранять более одного значения в одной структуре данных, сохраняющей элементы в памяти один за другим. Векторы могут сохранять данные только одного типа. Их удобно использовать, когда нужно сохранить список элементов, например, список текстовых строк в файле, или список цен товаров в корзине покупок.

Создание нового вектора

Чтобы создать новый пустой вектор, мы вызываем функцию `Vec::new`, как показано в листинге 8-1.

```
let v: Vec<i32> = Vec::new();
```

Листинг 8-1: Создание нового пустого вектора для хранения значений типа `i32`

Обратите внимание, что здесь мы добавили аннотацию типа. Поскольку мы не вставляем никаких значений в этот вектор, Rust не знает, какие элементы мы собираемся хранить. Это важный момент. Векторы реализованы с использованием обобщённых типов; мы рассмотрим, как использовать обобщённые типы с вашими собственными типами в Главе 10. А пока знайте, что тип `Vec<T>` предоставляемый стандартной библиотекой, может содержать любой тип, и когда конкретный вектор содержит определённый тип, тип указан в угловых скобках. В листинге 8-1 мы сообщили Rust, что `Vec<T>` в `v` будет содержать элементы типа `i32`.

В более реальном коде, Rust часто может вывести тип сохраняемых вами значений, как только вы вставите значения в вектор. Так что вам довольно редко нужна данная аннотация типа. Более общим является создание `Vec<T>` имеющего начальные значения: для удобства Rust предоставляет макрос `vec!` для этой цели. Макрос создаст новый вектор, содержащий указанные значения. Листинг 8-2 создаёт новый `Vec<i32>`, содержащий значения `1`, `2` и `3`. Числовым типом является `i32`, потому что это числовой тип по умолчанию для целочисленных значений, о чём упоминалось в разделе [“Типы данных”] ignore Главы 3.

```
let v = vec![1, 2, 3];
```

Листинг 8-2: Создание нового вектора, содержащего значения

Поскольку мы указали начальные значения `i32`, Rust может сделать вывод, что тип переменной `v` это `Vec<i32>` и аннотация типа здесь не нужна. Далее мы посмотрим как изменять вектор.

Изменение вектора

Чтобы создать вектор и затем добавить к нему элементы, можно использовать метод `push` показанный в листинге 8-3.

```
let mut v = Vec::new();

v.push(5);
v.push(6);
v.push(7);
v.push(8);
```

Листинг 8-3: Использование метода `push` для добавления значений в вектор

Как и с любой переменной, если мы хотим изменить её значение, нам нужно сделать её изменяемой с помощью ключевого слова `mut`, что обсуждалось в Главе 3. Все числа которые мы помещаем в вектор имеют тип `i32` по этому Rust с лёгкостью выводит тип вектора, по этой причине нам не нужна здесь аннотация типа вектора `Vec<i32>`.

Удаление элементов из вектора

Подобно структурам `struct`, вектор высвобождает свою память когда выходит из области видимости функции в которой он определён, данное поведение прокомментировано в листинге 8-4.

```
{
    let v = vec![1, 2, 3, 4];

    // do stuff with v
} // <- v goes out of scope and is freed here
```

Листинг 8-4. Показано, где удаляется вектор и его элементы

Когда вектор удаляется, всё его содержимое также удаляется: удаление вектора означает и удаление значений, которые он содержит. Это может показаться

простой концепцией, но все становится немного сложнее, когда вы начинаете вводить ссылки на элементы вектора. Давайте займёмся этим далее!

Чтение данных вектора

Есть два способа сослаться на значение, хранящееся в векторе: с помощью индекса или метода `get`. В следующих примерах мы аннотировали типы значений, возвращаемых этими функциями для большей ясности.

В листинге 8-5 показаны оба метода доступа к значению в векторе: либо с помощью синтаксиса индексации, либо с помощью метода `get`.

```
let v = vec![1, 2, 3, 4, 5];

let third: &i32 = &v[2];
println!("The third element is {}", third);

match v.get(2) {
    Some(third) => println!("The third element is {}", third),
    None => println!("There is no third element."),
}
```

Листинг 8-5. Использование синтаксиса индексации или метода `get` для доступа к элементу в векторе

Обратите внимание здесь на пару деталей. Во-первых, используется значение индекса `2` для получения третьего элемента: векторы индексируются начиная с нуля. Во-вторых, есть два способа получения третьего элемента: либо используя `&` и `[]` возвращающего ссылку на элемент, либо с помощью метода `get` содержащего индекс, переданный в качестве аргумента, который возвращает тип `Option<&T>`.

Причина, по которой Rust предоставляет два способа ссылки на элемент, заключается в том, что вы можете выбрать, как программа будет себя вести, когда вы попытаетесь использовать значение индекса за пределами диапазона существующих элементов. В качестве примера давайте посмотрим, что происходит, когда у нас есть вектор из пяти элементов, а затем мы пытаемся получить доступ к элементу с индексом 100 с помощью каждого метода, как показано в листинге 8-6.

В Rust есть два способа ссылаться на элемент, поэтому вы можете выбрать, как будет вести себя программа, когда вы попытаетесь использовать значение индекса,

для которого в векторе нет элемента. В качестве примера давайте посмотрим, что будет делать программа, если в ней определён вектор, содержащий пять элементов, но она пытается получить доступ к элементу с индексом 100, как показано в листинге 8-6.

```
let v = vec![1, 2, 3, 4, 5];  
  
let does_not_exist = &v[100];  
let does_not_exist = v.get(100);
```



Листинг 8-6. Попытка доступа к элементу с индексом 100 в векторе, содержащем пять элементов

Когда мы запускаем этот код, первая строка с `&v[100]` вызовет панику программы, потому что происходит попытка получить ссылку на несуществующий элемент. Такой подход лучше всего использовать, когда вы хотите, чтобы ваша программа аварийно завершила работу при попытке доступа к элементу за пределами вектора.

Когда методу `get` передаётся индекс, который находится за пределами вектора, он без паники возвращает `None`. Вы могли бы использовать такой подход, если доступ к элементу за пределами диапазона вектора происходит время от времени при нормальных обстоятельствах. Тогда ваш код будет иметь логику для обработки наличия `Some(&element)` или `None`, как обсуждалось в Главе 6. Например, индекс может исходить от человека, вводящего число. Если пользователь случайно введёт слишком большое число, то программа получит значение `None` и у вас будет возможность сообщить пользователю, сколько элементов находится в текущем векторе, и дать ему ещё один шанс ввести допустимое значение. Такое поведение было бы более дружелюбным для пользователя, чем внезапный сбой программы из-за опечатки!

Когда у программы есть действительная ссылка, borrow checker (средство проверки заимствований), обеспечивает соблюдение правил владения и заимствования (описанные в Главе 4), чтобы гарантировать, что эта ссылка и любые другие ссылки на содержимое вектора остаются действительными. Вспомните правило, которое гласит, что у вас не может быть изменяемых и неизменяемых ссылок в одной и той же области. Это правило применяется в листинге 8-7, где мы храним неизменяемую ссылку на первый элемент вектора и затем пытаемся добавить элемент в конец вектора. Данная программа не будет работать, если мы также попробуем сослаться на данный элемент позже в функции:

```
let mut v = vec![1, 2, 3, 4, 5];
let first = &v[0];
v.push(6);
println!("The first element is: {}", first);
```



Листинг 8-7. Попытка добавить некоторый элемент в вектор, в то время когда есть ссылка на элемент вектора

Компиляция этого кода приведёт к ошибке:

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

For more information about this error, try `rustc --explain E0502`.
error: could not compile `collections` due to previous error
```

Код в листинге 8-7 может выглядеть так, как будто он должен работать. Почему ссылка на первый элемент должна заботиться об изменениях в конце вектора? Эта ошибка возникает из-за особенности того, как работают векторы: поскольку векторы размещают значения в памяти друг за другом, добавление нового элемента в конец вектора может потребовать выделения новой памяти и копирования старых элементов в новое пространство, если нет недостаточного места, чтобы разместить все элементы друг за другом там, где в данный момент хранится вектор. В этом случае ссылка на первый элемент будет указывать на освобождённую память. Правила заимствования предотвращают попадание программ в такую ситуацию.

Примечание: Дополнительные сведения о реализации типа `Vec<T>` смотрите в

разделе "The Rustonomicon".

Перебор значений в векторе

Для доступа к каждому элементу вектора по очереди, мы итерируем все элементы, вместо использования индексов для доступа к одному за раз. В листинге 8-8 показано, как использовать цикл `for` для получения неизменяемых ссылок на каждый элемент в векторе значений типа `i32` и их вывода.

```
let v = vec![100, 32, 57];
for i in &v {
    println!("{}", i);
}
```

Листинг 8-8. Печать каждого элемента векторе, при помощи итерирования по элементам вектора с помощью цикла `for`

Мы также можем итерировать изменяемые ссылки на каждый элемент изменяемого вектора, чтобы вносить изменения во все элементы. Цикл `for` в листинге 8-9 добавит `50` к каждому элементу.

```
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}
```

Листинг 8-9. Итерирование и изменение элементов вектора по изменяемым ссылкам

Чтобы изменить значение на которое ссылается изменяемая ссылка, мы должны использовать оператор разыменования ссылки `*` для получения значения по ссылке в переменной `i` прежде чем использовать оператор `+=`. Мы поговорим подробнее об операторе разыменования в разделе ["Следуя указателю на значение с помощью оператора разыменования"] Главы 15.

Использование перечислений для хранения множества разных типов

Векторы могут хранить значения только одинакового типа. Это может быть

неудобно; определённо есть варианты использования для хранения списка элементов разных типов. К счастью, варианты перечисления определены для одного и того же типа перечисления, поэтому, когда нам нужен один тип для представления элементов разных типов, мы можем определить и использовать перечисление!

Например, мы хотим получить значения из строки в электронной таблице где некоторые столбцы строки содержат целые числа, некоторые числа с плавающей точкой, а другие - строковые значения. Можно определить перечисление, варианты которого будут содержать разные типы значений и тогда все варианты перечисления будут считаться одними и тем же типом: типом самого перечисления. Затем можно создать вектор, который содержит в себе значения этого перечисления и таким образом в конечном счёте добиться того, что в векторе будут сохраняться значения разного типа. Мы продемонстрировали это в листинге 8-10.

```
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}

let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("blue")),
    SpreadsheetCell::Float(10.12),
];
```

Листинг 8-10: Определение `enum` для хранения значений разных типов в одном векторе

Rust должен знать, какие типы будут в векторе во время компиляции, чтобы точно знать сколько памяти в куче потребуется для хранения каждого элемента. Мы также должны чётко указать, какие типы разрешены в этом векторе. Если бы Rust позволял вектору содержать любой тип, то был бы шанс что один или несколько типов вызовут ошибки при выполнении операций над элементами вектора. Использование перечисления вместе с выражением `match` означает, что во время компиляции Rust гарантирует, что все возможные случаи будут обработаны, как обсуждалось в главе 6.

Если вы не знаете исчерпывающий набор типов, которые программа получит во время выполнения для хранения в векторе, то техника использования перечисления не сработает. Вместо этого вы можете использовать типаж-объект, который мы рассмотрим в главе 17.

Теперь, когда мы обсудили некоторые из наиболее распространённых способов использования векторов, обязательно ознакомьтесь [с документацией по API вектора](#) для всего множества полезных методов, определённых в `Vec<T>` стандартной библиотеки. Например, в дополнение к методу `push`, существует метод `pop`, который удаляет и возвращает последний элемент. Давайте перейдём к следующему типу коллекции: `String` !

Сохранение текста с UTF-8 кодировкой в строках

Мы говорили о строках в главе 4, но сейчас мы рассмотрим их более подробно. Новички в Rust обычно застревают на строках из-за комбинации трёх причин: склонность Rust компилятора к выявлению возможных ошибок, более сложная структура данных чем считают многие программисты и UTF-8. Эти факторы объединяются таким образом, что тема может показаться сложной, если вы пришли из других языков программирования.

Полезно обсуждать строки в контексте коллекций, потому что строки реализованы в виде набора байтов, плюс некоторые методы для обеспечения полезной функциональности, когда эти байты интерпретируются как текст. В этом разделе мы поговорим об операциях над `String`, которые есть у каждого типа коллекций такие как создание, обновление и чтение. Мы также обсудим какими особенностями `String` отличается от других коллекций, а именно, как индексирование в `String` осложняется различием между тем, как люди и компьютеры интерпретируют данные заключённые в `String`.

Что же такое строка?

Сначала мы определим, что мы подразумеваем под термином *строка* (string). В Rust есть только один строковый тип в ядре языка - срез строки `str`, обычно используемый в заимствованном виде как `&str`. В Главе 4 мы говорили о срезах строк, *string slices*, которые являются ссылками на некоторые строковые данные в кодировке UTF-8. Например, строковые литералы хранятся в двоичном файле программы и поэтому являются срезами строк.

Тип `String` предоставляемый стандартной библиотекой Rust, не встроен в ядро языка и является расширяемым, изменяемым, владеющим, строковым типом в UTF-8 кодировке. Когда Rust разработчики говорят о "строках" то, они обычно имеют ввиду типы `String` и строковые срезы `&str`, а не просто один из них. Хотя этот раздел в основном посвящён `String`, оба типа интенсивно используются в стандартной библиотеке Rust, оба, и `String`, и строковые срезы, кодируются в UTF-8.

Стандартная библиотека Rust также включает ряд других строковых типов, таких как `OsString`, `OsStr`, `CString` и `CStr`. Библиотечные крейты могут предоставить даже большее количество возможностей для хранения строковых данных. Видите, как все имена этих типов заканчиваются на `String` или `Str`? Они относятся к

собственным и заимствованным вариантам, так же как типы `String` и `str` которые вы видели ранее. Эти типы строк могут хранить текст в различных кодировках или, например, быть по-другому представлены в памяти. Мы не будем обсуждать эти другие типы строк в данной главе; посмотрите документацию API для получения дополнительной информации о том как их использовать и когда каждый тип уместен.

Создание новых строк

Многие из тех же операций, которые доступны `Vec<T>`, доступны также в `String`, начиная с `new` функции для создания строки, показанной в листинге 8-11.

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

Листинг 8-11. Создание новой пустой `String` строки

Эта строка создаёт новую пустую строковую переменную с именем `s`, в которую мы можем затем загрузить данные. Часто у нас есть некоторые начальные данные, которые мы хотим назначить строке. Для этого мы используем метод `to_string` доступный для любого типа, который реализует типаж `Display`, как у строковых литералов. Листинг 8-12 показывает два примера.

```
let data = "initial contents";  
  
let s = data.to_string();  
  
// the method also works on a literal directly:  
let s = "initial contents".to_string();
```

Листинг 8-12. Использование метода `to_string` для создания экземпляра типа `String` из строкового литерала

Эти выражения создают строку с `"initial contents"`.

Мы также можем использовать функцию `String::from` для создания `String` из строкового литерала. Код листинга 8-13 является эквивалентным коду из листинга 8-12, который использует функцию `to_string`:

```
let s = String::from("initial contents");
```

Листинг 8-13. Использование функции `String::from` для создания экземпляра типа `String` из строкового литерала

Поскольку строки используются для очень многих вещей, можно использовать множество API для строк, предоставляющих множество возможностей. Некоторые из них могут показаться избыточными, но все они занимаются своим делом! В данном случае `String::from` и `to_string` делают одно и тоже, поэтому выбор зависит от стиля который вам больше импонирует.

Запомните, что строки хранятся в кодировке UTF-8, поэтому можно использовать любые правильно кодированные данные в них, как показано в листинге 8-14:

```
let hello = String::from("السلام عليكم");  
let hello = String::from("Dobrý den");  
let hello = String::from("Hello");  
let hello = String::from("ହୋଲ୍ଡିନ୍ଗ୍");  
let hello = String::from("ନମସ୍କାର");  
let hello = String::from("こんにちわ");  
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");
```

Листинг 8-14. Хранение приветствий в строках на разных языках

Все это допустимые `String` значения.

Обновление строковых данных

Строка `String` может увеличиваться в размере, а её содержимое может меняться, по аналогии как содержимое `Vec<T>` при вставке в него большего количества данных. Кроме того, можно использовать оператор `+` или макрос `format!` для объединения значений `String`.

Присоединение к строке с помощью `push_str` и `push`

Мы можем нарастить `String` используя метод `push_str` который добавит в исходное значение новый строковый срез, как показано в листинге 8-15.

```
let mut s = String::from("foo");
s.push_str("bar");
```

Листинг 8-15: Добавление среза строки к `String` используя метод `push_str`

После этих двух строк кода `s` будет содержать `foobar`. Метод `push_str` принимает строковый срез, потому что мы не всегда хотим владеть входным параметром. Например, код в листинге 8-16 показывает вариант, когда будет не желательно поведение, при котором мы не сможем использовать `s2` после его добавления к содержимому значения переменной `s1`.

```
let mut s1 = String::from("foo");
let s2 = "bar";
s1.push_str(s2);
println!("s2 is {}", s2);
```

Листинг 8-16. Использование фрагмента строки после его добавления в состав другого `String`

Если метод `push_str` стал бы владельцем переменной `s2`, мы не смогли бы напечатать его значение в последней строке. Однако этот код работает так, как мы ожидали!

Метод `push` принимает один символ в качестве параметра и добавляет его к `String`. В листинге 8-17 показан код, добавляющий букву "l" к `String`, используя метод `push`.

```
let mut s = String::from("lo");
s.push('l');
```

Листинг 8-17. Добавление одного символа в `String` значение используя `push`

После этого переменная `s` будет содержать `lol`.

Объединение строк с помощью оператора `+` или макроса `format!`

Часто хочется объединять две существующие строки. Один из возможных способов - это использование оператора `+` из листинга 8-18:

```
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // note s1 has been moved here and can no longer be
used
```

Листинг 8-18. Использование оператора `+` для объединения двух значений `String` в новое `String` значение

Строка `s3` будет содержать `Hello, world!` как результат выполнения этого кода. Причина того, что `s1` после добавления больше недействительна и причина, по которой мы использовали ссылку на `s2` имеют отношение к сигнатуре вызываемого метода при использовании оператора `+`. Оператор `+` использует метод `add`, чья сигнатура выглядит примерно так:

```
fn add(self, s: &str) -> String {
```

Это не точная сигнатура из стандартной библиотеки: в стандартной библиотеке `add` определён с помощью обобщённых типов. Здесь мы видим сигнатуру `add` с конкретными типами, заменяющими обобщённый, что происходит когда вызывается данный метод со значениями `String`. Мы обсудим обобщённые типы в Главе 10. Эта сигнатура даёт нам ключ для понимания особенностей оператора `+`.

Во-первых, перед `s2` мы видим `&`, что означает что мы складываем ссылку (reference) на вторую строку с самой первой строкой. Из-за параметра `s` в функции `add`, которая может только добавлять тип `&str` к типу `String`, мы не можем складывать два значения `String` вместе. Но подождите - тип `&s2` является типом `&String`, а не типом `&str`, как указано в сигнатуре второго параметра функции `add`. Так почему код в листинге 8-18 компилируется?

Причина, по которой мы можем использовать `&s2` в вызове `add` заключается в том, что компилятор может *принудительно привести* (*coerce*) аргумент типа `&String` к типу `&str`. Когда мы вызываем метод `add` в Rust используется *принудительное приведение* (*deref coercion*), которое превращает `&s2` в `&s2[..]`. Мы подробно обсудим принудительное приведение в Главе 15. Так как `add` не забирает во владение параметр `s`, `s2` по прежнему будет действительной строкой `String` после применения операции.

Во-вторых, как можно видеть в сигнатуре, `add` забирает во владение `self`, потому что `self` не имеет `&`. Это означает, что `s1` в листинге 8-18 будет перемещён в вызов `add` и больше не будет действителен после этого вызова. Не смотря на то, что код `let s3 = s1 + &s2;` выглядит как будто он скопирует обе строки и создаёт новую, это выражение фактически забирает во владение переменную `s1`, присоединяет к ней копию содержимого `s2`, а затем возвращает владение результатом. Другими словами, это выглядит как будто код создаёт множество копий, но это не так; данная реализация более эффективна чем копирование.

Если нужно объединить несколько строк, поведение оператора `+` становится громоздким:

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = s1 + "-" + &s2 + "-" + &s3;
```

На этом этапе переменная `s` будет содержать `tic-tac-toe`. С множеством символов `+` и `"` становится трудно понять, что происходит. Для более сложного комбинирования строк можно использовать макрос `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` в значение `tic-tac-toe`. Макрос `format!` работает тем же способом что макрос `println!`, но вместо вывода на экран возвращает тип `String` с содержимым. Версия кода с использованием `format!` значительно легче читается и не забирает во владение ни один из его параметров.

Индексирование в строках

Доступ к отдельным символам в строке, при помощи ссылки на них по индексу, является допустимой и распространённой операцией во многих других языках программирования. Тем не менее, если вы попытаетесь получить доступ к частям `String`, используя синтаксис индексации в Rust, то вы получите ошибку.

Рассмотрим неверный код в листинге 8-19.

```
let s1 = String::from("hello");
let h = s1[0];
```

Листинг 8-19. Попытка использовать синтаксис индекса со строкой

Этот код приведёт к следующей ошибке:



```
$ cargo run
Compiling collections v0.1.0 (file:///projects/collections)
error[E0277]: the type `String` cannot be indexed by `'{integer}`
--> src/main.rs:3:13
  |
3 |     let h = s1[0];
  |     ^^^^^ `String` cannot be indexed by `'{integer}`
  |
= help: the trait `Index<{integer}>` is not implemented for `String`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `collections` due to previous error
```

Ошибка и примечание говорят, что в Rust строки не поддерживают индексацию. Но почему так? Чтобы ответить на этот вопрос, нужно обсудить то, как Rust хранит строки в памяти.

Внутреннее представление

Тип `String` является оболочкой над типом `Vec<u8>`. Давайте посмотрим на несколько закодированных корректным образом в UTF-8 строк из примера листинга 8-14. Начнём с этой:

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

В этом случае `len` будет 4, что означает вектор, хранит строку "Hola" длиной 4 байта. Каждая из этих букв занимает 1 байт при кодировании в UTF-8. Но как насчёт следующей строки? (Обратите внимание, что эта строка начинается с заглавной кириллической "З", а не арабской цифры 3.)

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

Отвечая на вопрос, какова длина строки, вы можете ответить 12. Однако ответ Rust - 24, что равно числу байт, необходимых для кодирования «Здравствуйте» в UTF-8, так происходит, потому что каждое скалярное значение Unicode символа в этой строке занимает 2 байта памяти. Следовательно, индекс по байтам строки не всегда был соответствовал действительному скалярному Unicode значению. Для демонстрации рассмотрим этот недопустимый код Rust:

```
let hello = "Здравствуйте";
let answer = &hello[0];
```

Каким должно быть значение переменной `answer`? Должно ли оно быть значением первой буквы `3`? При кодировке в UTF-8, первый байт значения `3` равен `208`, а второй - `151`, поэтому значение в `answer` на самом деле должно быть `208`, но само по себе `208` не является действительным символом. Возвращение `208`, скорее всего не то, что хотел бы получить пользователь: ведь он ожидает первую букву этой строки; тем не менее, это единственный байт данных, который в Rust доступен по индексу 0. Пользователи обычно не хотят получить значение байта, даже если строка содержит только латинские буквы: если `&"hello"[0]` было бы допустимым кодом, который вернул значение байта, то он вернул бы `104`, а не `h`. Чтобы предотвратить возврат непредвиденного значения, вызывающего ошибки которые не могут быть сразу обнаружены, Rust просто не компилирует такой код и предотвращает недопонимание на ранних этапах процесса разработки.

Байты, скалярные значения и кластеры графем! Боже мой!

Ещё один момент, касающийся UTF-8, заключается в том, что на самом деле существует три способа рассмотрения строк с точки зрения Rust: как байты, как скалярные значения и как кластеры графем (самая близкая вещь к тому, что мы назвали бы буквами).

Если посмотреть на слово языка хинди «नमस्ते», написанное в транскрипции Devanagari, то оно хранится как вектор значений `u8` который выглядит следующим образом:

```
[224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164, 224, 165, 135]
```

Эти 18 байт являются именно тем, как компьютеры в конечном итоге сохранят в памяти эту строку. Если мы посмотрим на 18 байт как на скалярные Unicode значения, которые являются Rust типом `char`, то байты будут выглядеть так:

```
['न', 'म', 'स', '्', 'त', 'े']
```

Здесь есть шесть значений типа `char`, но четвёртый и шестой являются не буквами: они диакритики, специальные обозначения которые не имеют смысла сами по себе. Наконец, если мы посмотрим на байты как на кластеры графем, то получим то, что человек назвал бы словом на хинди состоящем из четырёх букв:

```
["न", "म", "स", "्ते"]
```

Rust предоставляет различные способы интерпретации необработанных строковых данных, которые компьютеры хранят так, чтобы каждой программе можно было выбрать необходимую интерпретацию, независимо от того, на каком человеческом языке представлены эти данные.

Последняя причина, по которой Rust не позволяет нам индексировать `String` для получения символов является то, что программисты ожидают, что операции индексирования всегда имеют постоянное время ($O(1)$) выполнения. Но невозможно гарантировать такую производительность для `String`, потому что Rust понадобилось бы пройтись по содержимому от начала до индекса, чтобы определить, сколько было действительных символов.

Срезы строк

Индексирование строк часто является плохой идеей, потому что не ясно каким должен быть возвращаемый тип такой операции: байтовым значением, символом, кластером графем или срезом строки. Поэтому Rust просит вас быть более конкретным, если действительно требуется использовать индексы для создания срезов строк. Чтобы быть более конкретным в случае строкового среза, нужно явно указывать, что вы хотите строковый срез (а не индексирование с помощью числового индекса `[]`): вы можете использовать оператор диапазона `[]` при создании среза строки в котором содержится указание на то, срез каких байтов надо делать:

```
let hello = "Здравствуйте";  
let s = &hello[0..4];
```

Здесь переменная `s` будет типа `&str` который содержит первые 4 байта строки. Ранее мы упоминали, что каждый из этих символов был по 2 байта, что означает, что `s` будет содержать `Зд`.

Что бы произошло, если бы мы использовали `&hello[0..1]`? Ответ: Rust бы запаниковал во время выполнения точно так же, как если бы обращались к недействительному индексу в векторе:

```
$ cargo run
Compiling collections v0.1.0 (file:///projects/collections)
  Finished dev [unoptimized + debuginfo] target(s) in 0.43s
    Running `target/debug/collections`
thread 'main' panicked at 'byte index 1 is not a char boundary; it is inside
'3' (bytes 0..2) of `Здравствуйте`, src/main.rs:4:14
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Вы должны использовать диапазоны для создания срезов строк с осторожностью, потому что это может привести к сбою вашей программы.

Методы для перебора строк

Сейчас поговорим о предпочтительных способах доступа к элементам строки.

Если необходимо производить операции над *отдельными* элементами юникод-строки (не буквами, а `char` символами), то наилучший способ - использовать метод `chars`. Вызов `chars` у "намстे" разделяет и возвращает 6 значений типа `char`. Далее, вы можете перебирать результат для доступа к каждому элементу:

```
for c in "намстे".chars() {
    println!("{}", c);
}
```

Код напечатает следующее:

```
н
м
с
्
т
े
```

Метод `bytes` возвращает каждый байт, который может быть подходящим в другой предметной области:

```
for b in "намстे".bytes() {
    println!("{}", b);
}
```

Этот код напечатает 18 байтов, составляющих эту строку **String**:

```
224  
164  
// --часть байтов вырезана--  
165  
135
```

Но делая так, обязательно помните, что валидные скалярные Unicode значения могут состоять более чем из одного байта.

Извлечение кластеров графем из строк сложно, поэтому данный функционал не предоставляется в стандартной библиотеке. На [crates.io](#) есть доступные библиотеки, если Вам нужен данный функционал.

Строки не так просты

Подводя итог, становится ясно, что строки сложны. Различные языки программирования реализуют различные варианты того, как представить эту сложность для программиста. В Rust решили сделать правильную обработку данных **String** поведением по умолчанию для всех программ Rust, что означает, что программисты должны заранее продумать обработку UTF-8 данных. Этот компромисс раскрывает большую сложность строк, чем в других языках программирования, но это предотвращает от необходимости обрабатывать ошибки, связанные с не-ASCII символами которые могут появиться в ходе разработки позже.

Давайте переключимся на что-то немного менее сложное: **HashMap**!

Хранение ключей со связанными значениями в HashMap

Последняя коллекция, которую мы рассмотрим в нашей книге будет *hash map* (хэш-карта). `HashMap<K, V>` сохраняет ключи типа `K` и значения типа `V`. Данная структура организует и хранит данные с помощью функции хэширования. Во множестве языков программирования реализована данная структура, но часто с разными наименованиями: такими как `hash`, `map`, `object`, `hash table`, `dictionary` или ассоциативный массив.

Хеш-карты полезны, когда нужно искать данные не используя индекс, как это например делается в векторах, а с помощью ключа, который может быть любого типа. Например, в игре вы можете отслеживать счёт каждой команды в хеш-карте, в которой каждый ключ - это название команды, а значение - счёт команды. Имея имя команды, вы можете получить её счёт из хеш-карты.

В этом разделе мы рассмотрим базовый API хеш-карт. Остальной набор полезных функций скрывается в объявлении типа `HashMap<K, V>`. Как и прежде, советуем обратиться к документации по стандартной библиотеке для получения дополнительной информации.

Создание новой хеш-карты

Создать пустую хеш-карту можно с помощью `new`, а добавить в неё элементы - с помощью `insert`. В листинге 8-20 мы отслеживаем счёт двух команд, синей (Blue) и жёлтой (Yellow). Синяя команда стартует с 10 очками, а жёлтая команда с 50.

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

Листинг 8-20. Создание новой хеш-карты и вставка в неё некоторых ключей и начальных значений

Обратите внимание, что нужно сначала указать строку `use HashMap` для её подключения из коллекций стандартной библиотеки. Из трёх коллекций данная является наименее используемой, поэтому она не подключается в область видимости функцией автоматического импорта (`prelude`). Хеш-карты также имеют

меньшую поддержку со стороны стандартной библиотеки; например, нет встроенного макроса для их конструирования.

Подобно векторам, хеш-карты хранят свои данные в куче. Здесь тип `HashMap` имеет в качестве типа ключей `String`, а в качестве типа значений тип `i32`. Как и векторы, `HashMap` однородны: все ключи должны иметь одинаковый тип и все значения должны иметь тоже одинаковый тип.

Ещё один способ построения хеш-карты - использование метода `collect` на векторе кортежей, где каждый кортеж состоит из двух значений (первое может быть представлено как ключ, а второе как значение хеш-карты). Метод `collect` собирает данные в несколько типов коллекций, включая `HashMap`. Например, если бы у нас были названия команд и начальные результаты в двух отдельных векторах, то мы могли бы использовать метод `zip` для создания вектора кортежей, где имя "Blue" спарено с числом 10, и так далее. Тогда мы могли бы использовать метод `collect`, чтобы превратить этот вектор кортежей в `HashMap`, как показано в листинге 8-21.

```
use std::collections::HashMap;

let teams = vec![String::from("Blue"), String::from("Yellow")];
let initial_scores = vec![10, 50];

let mut scores: HashMap<_, _> =
    teams.into_iter().zip(initial_scores.into_iter()).collect();
```

Листинг 8-21. Создание `HashMap` из списка команд и списка результатов

Здесь нужна аннотация типа `HashMap<_, _>`, поскольку с помощью метода `collect` данные можно собрать во множество различных структур данных и Rust не знает, в какую именно вы хотите собрать, пока вы не укажете это явно. Для параметров типа ключа и значения, мы используем подчёркивания и Rust может вывести типы, которые хеш содержит на основе типов данных из двух векторов. В листинге 8-21, тип ключа будет `String`, а тип значения будет `i32`, так же как в листинге 8-20.

Хеш-карты и владение

Для типов, которые реализуют типаж `Copy`, например `i32`, значения копируются в `HashMap`. Для значений со владением, таких как `String`, значения будут перемещены в хеш-карту и она станет владельцем этих значений, как показано в листинге 8-22.

```
use std::collections::HashMap;

let field_name = String::from("Favorite color");
let field_value = String::from("Blue");

let mut map = HashMap::new();
map.insert(field_name, field_value);
// field_name and field_value are invalid at this point, try using them
and
// see what compiler error you get!
```

Листинг 8-22. Показывает, что ключи и значения находятся во владении HashMap, как только они были вставлены

Мы не можем использовать переменные `field_name` и `field_value` после того, как их значения были перемещены в HashMap вызовом метода `insert`.

Если мы вставим в HashMap ссылки на значения, то они не будут перемещены в HashMap. Значения, на которые указывают ссылки, должны быть действительными хотя бы до тех пор, пока хеш-карта действительна. Мы поговорим об этих вопросах подробнее в разделе "Проверка ссылок с помощью времени жизни" главы 10.

Доступ к данным в HashMap

Мы можем получить значение из HashMap по ключу, с помощью метода `get`, как показано в листинге 8-23:

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

let team_name = String::from("Blue");
let score = scores.get(&team_name);
```

Листинг 8-23. Доступ к очкам команды "Blue" сохранённой в HashMap

Здесь `score` будет иметь количество очков, связанное с командой "Blue", результат будет `Some(&10)`. Результат обёрнут в вариант перечисления `Some` потому что `get` возвращает `Option<&V>`; если для этого ключа нет значения в HashMap, `get` вернёт `None`. Из-за такого подхода программе следует обрабатывать `Option`, например

одним из способов, которые мы рассмотрели в Главе 6.

Мы можем перебирать каждую пару ключ/значение в HashMap таким же образом, как мы делали с векторами, используя цикл `for`:

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

for (key, value) in &scores {
    println!("{}: {}", key, value);
}
```

Этот код будет печатать каждую пару в произвольном порядке:

```
Yellow: 50
Blue: 10
```

Обновление данных

Хотя количество ключей и значений может увеличиваться в HashMap, каждый ключ может иметь только одно значение, связанное с ним в один момент времени. Когда вы хотите изменить данные в хеш-карте, необходимо решить, как обрабатывать случай, когда ключ уже имеет назначенное значение. Можно заменить старое значение новым, полностью игнорируя старое. Можно сохранить старое значение и игнорировать новое и добавлять новое значение, если только ключ *ещё не* имел значения. Или можно было бы объединить старое значение и новое значение. Давайте посмотрим, как сделать каждый из вариантов!

Перезапись старых значений

Если мы вставим ключ и значение в HashMap, а затем вставим тот же ключ с новым значением, то старое значение связанное с этим ключом, будет заменено на новое. Даже несмотря на то, что код в листинге 8-24 вызывает `insert` дважды, хеш-карта будет содержать только одну пару ключ/значение, потому что мы вставляем значения для одного и того же ключа - ключа команды "Blue".

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Blue"), 25);

println!("{}: {}", "Blue", scores);
```

Листинг 8-24. Замена значения, хранимого в конкретном ключе

Код напечатает `{"Blue": 25}`. Начальное значение `10` было перезаписано.

Вставка значения только в том случае, когда ключ не имеет значения

Обычно проверяют, имеется ли значение для конкретного ключа и если нет, то значение для него вставляется. Хеш-карты имеют для этого специальный API называемый `entry`, который принимает ключ для проверки в качестве входного параметра. Возвращаемое значение метода `entry` - это перечисление `Entry`, с двумя вариантами: первый представляет значение, которое может существовать, а второй говорит о том, что значение отсутствует. Допустим, мы хотим проверить, имеется ли ключ и связанное с ним значение для команды "Yellow". Если хеш-карта не имеет значения для такого ключа, то мы хотим вставить значение 50. То же самое мы хотим проделать и для команды "Blue". Используем API `entry` в коде листинга 8-25.

```
use std::collections::HashMap;

let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);

scores.entry(String::from("Yellow")).or_insert(50);
scores.entry(String::from("Blue")).or_insert(50);

println!("{}: {}", "Blue", scores);
```

Листинг 8-25. Использование метода `entry` для вставки значения только в том случае, когда ключ не имеет значения

Метод `or_insert` определён в `Entry` так, чтобы возвращать изменяемую ссылку на соответствующее значение ключа внутри варианта перечисления `Entry`, когда этот ключ существует, а если его нет, то вставлять параметр в качестве нового значения этого ключа и возвращать изменяемую ссылку на новое значение. Эта техника

намного чище, чем самостоятельное написание логики и, кроме того, она более безопасна и согласуется с правилами заимствования.

При выполнении кода листинга 8-25 будет напечатано `{"Yellow": 50, "Blue": 10}`. Первый вызов метода `entry` вставит ключ для команды "Yellow" со значением 50, потому что для жёлтой команды ещё не имеется значения в HashMap. Второй вызов `entry` не изменит хеш-карту, потому что для ключа команды "Blue" уже имеется значение 10.

Создание нового значения на основе старого значения

Другим распространённым вариантом использования хеш-карт является поиск значения по ключу, а затем обновление этого значения на основе старого значения. Например, в листинге 8-26 показан код, который подсчитывает, сколько раз определённое слово появляется в каком-либо тексте. Мы используем HashMap со словами в качестве ключей и увеличиваем соответствующее слову значение, чтобы отслеживать, сколько раз в тексте мы увидели слово. Если мы впервые увидели слово, то сначала вставляем значение 0.

```
use std::collections::HashMap;

let text = "hello world wonderful world";

let mut map = HashMap::new();

for word in text.split_whitespace() {
    let count = map.entry(word).or_insert(0);
    *count += 1;
}

println!("{}:", map);
```

Листинг 8-26. Подсчёт вхождений слов с использованием хеш-карты, которая хранит слова и количество их упоминаний в тексте

Будет напечатано `{"world": 2, "hello": 1, "wonderful": 1}`. Метод `or_insert` возвращает изменяемую ссылку (`&mut V`) на значение ключа. Мы сохраним изменяемую ссылку в переменной `count`. Для того, чтобы присвоить переменной значение, необходимо произвести разыменование с помощью звёздочки (`*`). Изменяемая ссылка удаляется сразу же после выхода из области видимости цикла `for`. Все эти изменения безопасны и согласуются с правилами заимствования.

Функция хэширования

По умолчанию **HashMap** использует "криптографически сильную" функцию хэширования [SipHash](#), которая может противостоять атакам класса *отказ в обслуживании*, *Denial of Service (DoS)*. Это не самый быстрый из возможных алгоритмов хэширования, в данном случае производительность идёт на компромисс с обеспечением лучшей безопасности. Если после профилирования вашего кода окажется, что хэш функция используемая по умолчанию очень медленная, вы можете заменить её используя другой *hasher*. *Hasher* - это тип, реализующий трейт **BuildHasher**. Подробнее о типажах мы поговорим в Главе 10. Вам совсем не обязательно реализовывать свою собственную функцию хэширования, [crates.io](#) имеет достаточное количество библиотек, предоставляющих разные реализации *hasher* с множеством общих алгоритмов хэширования.

Итоги

Векторы, строки и хэш-карты предоставляют большое количество функционала для программ, когда необходимо сохранять, получать доступ и модифицировать данные. Теперь вы готовы решить следующие учебные задания:

- Есть список целых чисел. Создайте функцию, используйте вектор и верните из списка: среднее значение; медиану (значение элемента из середины списка после его сортировки); моду списка (mode of list, то значение которое встречается в списке наибольшее количество раз; *HashMap* будет полезна в данном случае)
- Преобразуйте строку в кодировку "поросячьей латыни" (Pig Latin), где первая согласная каждого слова перемещается в конец и к ней добавляется окончание "ay". Например "first" в поросячьей латыни станет "irst-fay". Если слово начинается на гласную, то в конец слова добавляется суффикс "hay" ("apple" становится "apple-hay"). Помните о деталях работы с кодировкой UTF-8!
- Используя хеш-карту и векторы, создайте текстовый интерфейс позволяющий пользователю добавлять имена сотрудников к названию отдела компании. Например, "Add Sally to Engineering" или "Add Amir to Sales". Затем позвольте пользователю получить список всех людей из отдела или всех людей в компании отсортированным в алфавитном порядке по отделам.

Документация API стандартной библиотеки описывает методы у векторов, строк и *HashMap*. Рекомендуем воспользоваться ей при решении упражнений.

Потихоньку мы переходим к более сложным программам, в которых операции могут потерпеть неудачу. Наступило идеальное время для обсуждения обработки ошибок.

Обработка ошибок

Возникновение ошибок в ходе выполнения программ - это суровая реальность в жизни программного обеспечения, поэтому Rust имеет ряд функций для обработки ситуаций в которых что-то идёт не так. Во многих случаях Rust требует, чтобы вы признали возможность ошибки и предприняли некоторые действия, прежде чем ваш код будет скомпилирован. Это требование делает вашу программу более надёжной, гарантируя, что вы обнаружите ошибки и обработаете их надлежащим образом, прежде чем развернёте свой код в производственной среде!

В Rust ошибки группируются на две основные категории *исправимые* (recoverable) и *неисправимые* (unrecoverable). В случае исправимой ошибки, такой как *файл не найден*, мы, скорее всего, просто хотим сообщить о проблеме пользователю и повторить операцию. Неисправимые ошибки всегда являются симптомами дефектов в коде, например, попытка доступа к ячейке за пределами границ массива, и поэтому мы хотим немедленно остановить программу.

Большинство языков не различают эти два вида ошибок и обрабатывают оба вида одинаково, используя такие механизмы, как исключения. В Rust нет исключений. Вместо этого он имеет тип `Result<T, E>` для обрабатываемых (исправимых) ошибок и макрос `panic!`, который останавливает выполнение, когда программа встречает необрабатываемую (неисправимую) ошибку. Сначала эта глава расскажет про вызов `panic!`, а потом расскажет о возврате значений `Result<T, E>`. Кроме того, мы рассмотрим, что нужно учитывать при принятии решения о том, следует ли попытаться исправить ошибку или остановить выполнение.

Неустранимые ошибки с макросом `panic!`

Иногда в коде происходят плохие вещи, и вы ничего не можете с этим поделать. В этих случаях у Rust есть макрос `panic!`. Когда выполнится макрос `panic!`, ваша программа напечатает сообщение об ошибке, раскрутит и очистит стек вызовов, а затем завершится. Мы обычно вызываем панику, когда обнаруживается какая-либо ошибка, и неясно, как справиться с проблемой во время написания нашей программы.

Раскручивать стек или прерывать выполнение программы в ответ на панику?

По умолчанию, когда происходит паника, программа начинает процесс *раскрутки стека*, означающий в Rust проход обратно по стеку вызовов и очистку данных для каждой обнаруженной функции. Тем не менее, этот обратный проход по стеку и очистка генерируют много работы.

Альтернативой является немедленное *прерывание* выполнения, которое завершает программу без очистки. Память, которую использовала программа, должна быть очищена операционной системой. Если в вашем проекте нужно сделать маленьким исполняемый файл, насколько это возможно, вы можете переключиться с варианта раскрутки стека на вариант прерывания при панике, добавьте `panic = 'abort'` в раздел `[profile]` вашего *Cargo.toml* файла. Например, если вы хотите прерывать выполнение программы по панике в релизной версии программы добавьте следующее:

```
[profile.release]
panic = 'abort'
```

Давайте попробуем вызвать `panic!` в простой программе:

Файл: `src/main.rs`

```
fn main() {
    panic!("crash and burn");
}
```

При запуске программы, вы увидите что-то вроде этого:

```
$ cargo run
Compiling panic v0.1.0 (file:///projects/panic)
  Finished dev [unoptimized + debuginfo] target(s) in 0.25s
    Running `target/debug/panic`
thread 'main' panicked at 'crash and burn', src/main.rs:2:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Выполнение макроса `panic!` вызывает сообщение об ошибке, содержащееся в двух последних строках. Первая строка показывает сообщение паники и место в исходном коде, где возникла паника: `src/main.rs: 2:5` указывает, что это вторая строка, пятый символ внутри нашего файла `src/main.rs`

В этом случае указанная строка является частью нашего кода, и если мы перейдём к этой строке, мы увидим вызов макроса `panic!`. В других случаях вызов `panic!` мог бы произойти в стороннем коде, который вызывает наш код, тогда имя файла и номер строки для сообщения об ошибке будет из чужого кода, где макрос `panic!` выполнен, а не из строк нашего кода, которые в конечном итоге привели к выполнению `panic!`. Мы можем использовать обратную трассировку вызовов функций которые вызвали `panic!` чтобы выяснить, какая часть нашего кода вызывает проблему. Мы обсудим обратную трассировку более подробно далее.

Использование обратной трассировки `panic!`

Давайте посмотрим на другой пример, где, вызов `panic!` происходит в сторонней библиотеке из-за ошибки в нашем коде (а не как в примере ранее, из-за вызова макроса нашим кодом напрямую). В листинге 9-1 приведён код, который пытается получить доступ по индексу в векторе за пределами допустимого диапазона значений индекса.

Файл: `src/main.rs`

```
fn main() {
    let v = vec![1, 2, 3];
    v[99];
}
```



Листинг 9-1. Попытка доступа к элементу за пределами вектора, которая вызовет `panic!`

Здесь мы пытаемся получить доступ к 100-му элементу вектора (который находится

по индексу 99, потому что индексирование начинается с нуля), но вектор имеет только 3 элемента. В этой ситуации, Rust будет вызывать панику. Использование `[]` должно возвращать элемент, но вы передаёте неверный индекс: не существует элемента, который Rust мог бы вернуть.

В языке C, например, попытка прочесть за пределами конца структуры данных (в нашем случае векторе) приведёт к *неопределённому поведению, undefined behavior, UB*. Вы всё равно получите значение, которое находится в том месте памяти компьютера, которое соответствовало бы этому элементу в векторе, несмотря на то, что память по тому адресу совсем не принадлежит вектору (всё просто: С рассчитал бы место хранения элемента с индексом 99 и считал бы то, что там хранится, упс). Это называется *чтением за пределом буфера, buffer overread*, и может привести к уязвимостям безопасности. Если злоумышленник может манипулировать индексом таким образом, то у него появляется возможность читать данные, которые он не должен иметь возможности читать.

Чтобы защитить вашу программу от такого рода уязвимостей при попытке прочитать элемент с индексом, которого не существует, Rust остановит выполнение и откажется продолжить работу программы. Давайте попробуем так сделать и посмотрим на поведение Rust:

```
$ cargo run
Compiling panic v0.1.0 (file:///projects/panic)
  Finished dev [unoptimized + debuginfo] target(s) in 0.27s
    Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is
99', src/main.rs:4:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Следующая строка говорит, что мы можем установить переменную среды `RUST_BACKTRACE`, чтобы получить обратную трассировку того, что именно стало причиной ошибки. Обратная трассировка создаёт список всех функций, которые были вызваны до какой-то определённой точки выполнения программы. Обратная трассировка в Rust работает так же, как и в других языках. По этому предлагаем вам читать данные обратной трассировки как и везде - читать сверху вниз, пока не увидите информацию о файлах написанных вами. Это место, где возникла проблема. Другие строки, которые выше над строками с упоминанием наших файлов, - это код, который вызывается нашим кодом; строки ниже являются кодом, который вызывает наш код. Эти строки могут включать основной код Rust, код стандартной библиотеки или используемые крейты. Давайте попробуем получить обратную трассировку с помощью установки переменной среды `RUST_BACKTRACE` в

любое значение, кроме 0. Листинг 9-2 показывает вывод, подобный тому, что вы увидите.

```
$ RUST_BACKTRACE=1 cargo run
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is
99', src/main.rs:4:5
stack backtrace:
 0: rust_begin_unwind
      at
/rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/std/src/panicking.rs:48

 1: core::panicking::panic_fmt
      at
/rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/src/panicking.rs:8

 2: core::panicking::panic_bounds_check
      at
/rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/src/panicking.rs:6

 3: <usize as core::slice::index::SliceIndex<[T]>>::index
      at
/rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/src/slice/index.rs

 4: core::slice::index::<impl core::ops::index::Index<I> for [T]>::index
      at
/rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/src/slice/index.rs

 5: <alloc::vec::Vec<T> as core::ops::index::Index<I>>::index
      at
/rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/alloc/src/vec.rs:1982

 6: panic::main
      at ./src/main.rs:4
 7: core::ops::function::FnOnce::call_once
      at
/rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/src/ops/function.rs

note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose
backtrace.
```

Листинг 9-2. Обратная трассировка, сгенерированная вызовом `panic!`, когда установлена переменная окружения `RUST_BACKTRACE`

Тут много вывода! Вывод, который вы увидите, может отличаться от представленного, в зависимости от вашей операционной системы и версии Rust. Для того, чтобы получить обратную трассировку с этой информацией, должны быть включены *символы отладки, debug symbols*. Символы отладки включены по

умолчанию при использовании `cargo build` или `cargo run` без флага `--release`, как у нас в примере.

В выводе обратной трассировки листинга 9-2, строка #6 указывает на строку в нашем проекте, которая вызывала проблему: строка 4 из файла `src/main.rs`. Если мы не хотим, чтобы наша программа запаниковала, мы должны начать исследование с места, на которое указывает первая строка с упоминанием нашего файла. В листинге 9-1, где мы для демонстрации обратной трассировки сознательно написали код, который паникует, способ исправления паники состоит в том, чтобы не запрашивать элемент за пределами диапазона значений индексов вектора. Когда ваш код запаникует в будущем, вам нужно будет выяснить, какое выполняющееся кодом действие, с какими значениями вызывает панику и что этот код должен делать вместо этого.

Мы вернёмся к обсуждению макроса `panic!`, и того когда нам следует и не следует использовать `panic!` для обработки ошибок в разделе "`panic!` или НЕ `panic!`" этой главы. Далее мы рассмотрим, как восстановить выполнение программы после исправляемых ошибок, использующих тип `Result`.

Исправимые ошибки с `Result`

Многие ошибки являются не настолько критичными, чтобы останавливать выполнение программы. Иногда, когда в функции происходит сбой, необходима просто правильная интерпретация и обработка ошибки. К примеру, при попытке открыть файл может произойти ошибка из-за отсутствия файла. Вы, возможно, захотите исправить ситуацию и создать новый файл вместо остановки программы.

Вспомните раздел "Обработка потенциального сбоя с помощью типа `Result`" главы 2: мы использовали там перечисление `Result`, имеющее два варианта, `Ok` и `Err` для обработки сбоев. Само перечисление определено следующим образом:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Типы `T` и `E` являются параметрами обобщённого типа: мы обсудим обобщённые типы более подробно в Главе 10. Все что вам нужно знать прямо сейчас - это то, что `T` представляет тип значения, которое будет возвращено в случае успеха внутри варианта `Ok`, а `E` представляет тип ошибки, которая будет возвращена при сбое внутри варианта `Err`. Так как тип `Result` имеет эти типовые параметры (generic type parameters), мы можем использовать тип `Result` и его методы, которые определены в стандартной библиотеке, в ситуациях, когда тип успешного значения и значения ошибки, которые мы хотим вернуть, отличаются.

Давайте вызовем функцию, которая возвращает значение `Result`, потому что может потерпеть неудачу. В листинге 9-3 мы пытаемся открыть файл.

Файл: src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");
}
```

Листинг 9-3: Открытие файла

Откуда мы знаем, что `File::open` возвращает `Result`? Мы могли бы посмотреть [документацию по API стандартной библиотеки](#) или мы могли бы спросить

компилятор! Если мы припишем переменной `f` тип, отличный от возвращаемого типа функции, а затем попытаемся скомпилировать код, компилятор скажет нам, что типы не совпадают. Сообщение об ошибке подскажет нам, каким должен быть тип `f`. Давайте попробуем! Мы знаем, что возвращаемый тип `File::open` не является типом `u32`, поэтому давайте изменим выражение `let f` на следующее:

```
let f: u32 = File::open("hello.txt");
```

Попытка компиляции выводит сообщение:



```
$ cargo run
Compiling error-handling v0.1.0 (file:///projects/error-handling)
error[E0308]: mismatched types
--> src/main.rs:4:18
  |
4 |     let f: u32 = File::open("hello.txt");
  |             ^^^^^^^^^^^^^^^^^^^^^^^^^ expected `u32`, found enum
`Result`
  |
  |         |
  |         expected due to this
  |
= note: expected type `u32`
          found enum `Result<File, std::io::Error>`

For more information about this error, try `rustc --explain E0308`.
error: could not compile `error-handling` due to previous error
```

Ошибка говорит нам о том, что возвращаемым типом функции `File::open` является `Result<T, E>`. Типовой параметр `T` здесь равен типу успешного выполнения, `std::fs::File`, то есть дескриптору файла. Тип `E`, используемый в значении ошибки, равен `std::io::Error`.

Этот возвращаемый тип означает, что вызов `File::open` может завершиться успешно и вернуть дескриптор файла, с помощью которого можно читать из файла или писать в него. Вызов функции также может завершиться ошибкой: например, файла может не существовать или у нас может не быть прав на доступ к нему. Функция `File::open` должна иметь способ сообщить нам, был ли её вызов успешен или потерпел неудачу и одновременно возвратить либо дескриптор файла либо информацию об ошибке. Эта информация - именно то, что возвращает перечисление `Result`.

Когда вызов `File::open` успешен, значение в переменной `f` будет экземпляром `Ok`, внутри которого содержится дескриптор файла. Если вызов не успешный,

значением переменной `f` будет экземпляр `Err`, который содержит больше информации о том, какая ошибка произошла.

Необходимо дописать в код листинга 9-3 выполнение разных действий в зависимости от значения, которое вернёт вызов `File::open`. Листинг 9-4 показывает один из способов обработки `Result` - пользуясь базовым инструментом языка, таким как выражение `match`, рассмотренным в Главе 6.

Файл: `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!("Problem opening the file: {:?}", error),
    };
}
```

Листинг 9-4: Использование выражения `match` для обработки возвращаемых вариантов типа `Result`

Обратите внимание, что также как перечисление `Option`, перечисление `Result` и его варианты, входят в область видимости благодаря авто-импорту (prelude), поэтому не нужно указывать `Result::` перед использованием вариантов `Ok` и `Err` в ветках выражения `match`.

Здесь мы говорим Rust, что когда результат - это `Ok`, то надо вернуть внутреннее значение `file` из варианта `Ok`, и затем мы присваиваем это значение дескриптора файла переменной `f`. После `match` мы можем использовать дескриптор файла для чтения или записи.

Другая ветвь `match` обрабатывает случай, где мы получаем значение `Err` после вызова `File::open`. В этом примере мы решили вызвать макрос `panic!`. Если в нашей текущей директории нет файла с именем `hello.txt` и мы выполним этот код, то мы увидим следующее сообщение от макроса `panic!`:

```
$ cargo run
Compiling error-handling v0.1.0 (file:///projects/error-handling)
  Finished dev [unoptimized + debuginfo] target(s) in 0.73s
    Running `target/debug/error-handling`
thread 'main' panicked at 'Problem opening the file: Os { code: 2, kind: NotFound, message: "No such file or directory" }', src/main.rs:8:23
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Как обычно, данное сообщение точно говорит, что пошло не так.

Обработка различных ошибок с помощью match

Код в листинге 9-4 будет вызывать `panic!` независимо от того, почему вызов `File::open` не удался. Мы бы хотели предпринять различные действия для разных причин сбоя. Если открытие `File::open` не удалось из-за отсутствия файла, мы хотим создать файл и вернуть его дескриптор. Если вызов `File::open` не удался по любой другой причине (например, потому что у нас не было прав на открытие файла), то мы хотим вызвать `panic!` как у нас сделано в листинге 9-4. Посмотрите листинг 9-5, в котором мы добавили дополнительное внутреннее выражение `match`.

Файл: 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!("Problem creating the file: {:?}", e),
            },
            other_error => {
                panic!("Problem opening the file: {:?}", other_error)
            }
        },
    };
}
```

Листинг 9-5: Обработка различных ошибок разными способами

Типом значения возвращаемого функцией `File::open` внутри `Err` варианта является `io::Error`, структура из стандартной библиотеки. Данная структура имеет метод `kind`, который можно вызвать для получения значения `io::ErrorKind`.

Перечисление `io::ErrorKind` из стандартной библиотеки имеет варианты, представляющие различные типы ошибок, которые могут появиться при выполнении операций в `io` (крайне который занимается проблемами ввода/вывода данных). Вариант, который мы хотим использовать, это `ErrorKind::NotFound`. Он даёт информацию, о том, что файл который мы пытаемся открыть ещё не существует. Итак, во второй строке мы вызываем сопоставление шаблона с переменной `f` и попадаем в ветку с обработкой ошибки, но также у нас есть внутренняя проверка для сопоставления `error.kind()` ошибки.

Условие, которое мы хотим проверить во внутреннем `match` - это то, что значение, которое вернул вызов `error.kind()`, является вариантом `NotFound` перечисления `ErrorKind`. Если это так, мы пытаемся создать файл с помощью функции `File::create`. Однако, поскольку вызов `File::create` тоже может завершиться ошибкой, нам нужна обработка ещё одной ошибки теперь уже во внутреннем выражении `match` - третий вложенный `match`. Заметьте: если файл не может быть создан, выводится другое сообщение об ошибке, более специализированное. Вторая же ветка внешнего `match` (который обрабатывает вызов `error.kind()`), остаётся той же самой. В итоге программа паникует при любой ошибке, кроме ошибки отсутствия файла.

Достаточно про `match`! Код с `match` является очень удобным, но также достаточно примитивным. В Главе 13 вы узнаете про замыкания (closures); тип `Result<T, E>` имеет много методов, реализованных с помощью выражения `match` и принимающих замыкание в качестве входного значения. Использование данных методов сделает ваш код более лаконичным. Более опытные разработчики могли бы написать код как в листинге 9-5, вместо нашего:

```
 {{#rustdoc_include ../listings/ch09-error-handling/no-listing-03-
 closures/src/main.rs}}
```

Несмотря на то, что данный код имеет такое же поведение как в листинге 9-5, он не содержит ни одного выражения `match` и проще для чтения. Рекомендуем вам вернуться к примеру этого раздела после того как вы прочитаете Главу 13 и изучите метод `unwrap_or_else` по документации стандартной библиотеки. Многие из методов о которых вы узнаете в документации и Главе 13 могут очистить код от больших, вложенных выражений `match` при обработке ошибок.

Сокращённые способы обработки ошибок `unwrap` и `expect`

Использование `match` работает неплохо, однако может выглядеть несколько многословно и не всегда хорошо передаёт намерения. У типа `Result<T, E>` есть много методов для различных задач. Один из них, `unwrap`, является сокращённым методом, который реализован прямо как выражение `match` из листинга 9-4. Если значение `Result` это `Ok`, `unwrap` вернёт значение внутри `Ok`. Если же `Result` это `Err`, `unwrap` вызовет макрос `panic!`. Вот пример `unwrap` в действии:

Файл: src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").unwrap();
}
```

Если мы запустим этот код при отсутствии файла `hello.txt`, то увидим сообщение об ошибке из вызова `panic!` метода `unwrap`:

```
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
```

Другой метод, похожий на `unwrap`, это `expect`, позволяющий выбрать сообщение об ошибке для макроса `panic!`. Использование `expect` вместо `unwrap` с предоставлением хорошего сообщения об ошибке выражает ваше намерение и делает более простым отслеживание источника паники. Синтаксис метода `expect` выглядит так:

Файл: src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").expect("Failed to open hello.txt");
}
```

Мы используем `expect` таким же образом, как и `unwrap`: чтобы вернуть дескриптор файла или вызвать макрос `panic!`. Сообщением об ошибке, которое `expect` передаст в `panic!`, будет параметр функции `expect`, а не значение по умолчанию,

используемое `unwrap`. Вот как оно выглядит:

```
thread 'main' panicked at 'Failed to open hello.txt: Error { repr: Os { code: 2, message: "No such file or directory" } }', src/libcore/result.rs:906:4
```

Так как сообщение об ошибке начинается с нашего пользовательского текста: `Failed to open hello.txt`, то потом будет проще найти из какого места в коде данное сообщение приходит. Если использовать `unwrap` во множестве мест, то придётся потратить время для выяснения какой именно вызов `unwrap` вызывает "панику", так как все вызовы `unwrap` генерируют одинаковое сообщение.

Проброс ошибок

Когда вы пишете функцию, реализация которой вызывает что-то, что может завершиться ошибкой, вместо обработки ошибки в этой функции, вы можете вернуть ошибку в вызывающий код, чтобы он мог решить, что с ней делать. Такой приём известен как *распространение ошибки*, *propagating the error*. Благодаря нему мы даём больше контроля вызывающему коду, где может быть больше информации или логики, которая диктует, как ошибка должна обрабатываться, чем было бы в месте появления этой ошибки.

Например, код программы 9-6 читает имя пользователя из файла. Если файл не существует или не может быть прочтён, то функция возвращает ошибку в код, который вызвал данную функцию:

Файл: src/main.rs

```

use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let f = File::open("hello.txt");

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}

```

Листинг 9-6: Функция, которая возвращает ошибки в вызывающий код, используя оператор `match`

Данную функцию можно записать гораздо короче. Чтобы больше проникнуться обработкой ошибок, мы сначала сделаем многое самостоятельно, а в конце покажем более короткий способ. Давайте сначала рассмотрим тип возвращаемого значения: `Result<String, io::Error>`. Здесь есть возвращаемое значение функции типа `Result<T, E>` где шаблонный параметр `T` был заполнен конкретным типом `String` и шаблонный параметр `E` был заполнен конкретным типом `io::Error`. Если эта функция выполнится успешно, будет возвращено `Ok`, содержащее значение типа `String` - имя пользователя прочитанное функцией из файла. Если же при чтении файла будут какие-либо проблемы, то вызываемый код получит значение `Err` с экземпляром `io::Error`, в котором содержится больше информации об ошибке. Мы выбрали `io::Error` в качестве возвращаемого значения функции, потому что обе операции, которые мы вызываем внутри этой функции, возвращают этот тип ошибки: функция `File::open` и метод `read_to_string`.

Тело функции начинается с вызова `File::open`. Затем мы обрабатываем значение `Result` возвращённое с помощью `match` аналогично коду `match` листинга 9-4, но вместо вызова `panic!` для случая `Err` делаем ранний возврат из данной функции и передаём ошибку из `File::open` обратно в вызывающий код, как ошибку уже текущей функции. Если `File::open` выполнится успешно, мы сохраняем дескриптор файла в переменной `f` и выполнение продолжается далее.

Затем мы создаём новую `String` в переменной `s` и вызываем метод `read_to_string` у дескриптора файла в переменной `f`, чтобы считать содержимое файла в переменную `s`. Метод `read_to_string` также возвращает `Result`, потому что он может потерпеть неудачу, даже если `File::open` пройдёт успешно. Таким образом, нам нужно ещё одно выражение `match`, чтобы справиться с этим `Result`: если `read_to_string` выполнится успешно, то наша функция завершится успешно и мы вернём имя пользователя из файла, которое сейчас находится в `s`, завёрнутым в `Ok`. Если вызов `read_to_string` не успешен, мы возвращаем значение ошибки так же, как мы вернули значение ошибки в `match`, обработавшем возвращаемое значение `File::open`. Тем не менее, нам не нужно явно писать `return`, потому что это последнее выражение в функции.

Код, вызывающий данный код, будет обрабатывать либо значение `Ok`, содержащее имя пользователя, либо значение `Err`, содержащее `io::Error`. Мы не знаем, что будет делать вызывающий код с этими значениями. Если вызывающий код получает значение `Err`, он может вызвать `panic!` и завершить программу, использовать имя пользователя по умолчанию, или например, попытается получить имя пользователя из какого-то другого места. У нас недостаточно информации о том, чего пытается достичь вызывающий код, поэтому мы пробрасываем всю информацию об успехе или ошибке наверх для её правильной обработки.

Такая схема распространения ошибок настолько распространена в Rust, что Rust предоставляет оператор вопросительный знак `?` для простоты.

Сокращение для проброса ошибок: оператор `?`

Код программы 9-6 показывает реализацию функции `read_username_from_file`, функционал которой аналогичен коду программы 9-5, но реализация использует оператор `?`:

Файл: `src/main.rs`

```
use std::fs::File;
use std::io;
use std::io::Read;

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)
}
```

Листинг 9-7: Функция, которая возвращает ошибки в вызывающий код, используя оператор `?`

Оператор `?`, помещаемый после значения типа `Result`, работает почти таким же образом, как выражение `match`, которое мы определили для обработки значений типа `Result` в листинге 9-6. Если значение `Result` равно `Ok`, значение внутри `Ok` будет возвращено из этого выражения и программа продолжит выполнение. Если значение является `Err`, то `Err` будет возвращено из всей функции, как если бы мы использовали ключевое слово `return`, таким образом ошибка передаётся в вызывающий код.

Имеется разница между тем, что делает выражение `match` листинга 9-6 и оператор `?`. Ошибочные значения при выполнении методов с оператором `?` возвращаются через функцию `from`, определённую в типаже `From` стандартной библиотеки. Данный типаж используется для конвертирования ошибок одного типа в ошибки другого типа. Когда оператор `?` вызывает функцию `from`, то полученный тип ошибки конвертируется в тип ошибки, который определён для возврата в текущей функции. Это удобно, когда функция возвращает один тип ошибки для представления всех возможных вариантов, из-за которых она может не завершиться успешно, даже если части кода функции могут не выполниться по разным причинам. Если каждый тип ошибки реализует функцию `from` определяя, как конвертировать себя в возвращаемый тип ошибки, то оператор `?` позаботится об этой конвертации автоматически.

В коде примера 9-7 оператор `?` в конце вызова функции `File::open` возвращает значения содержимого `Ok` в переменную `f`. Если же в при работе этой функции произошла ошибка, оператор `?` произведёт ранний возврат из функции со значением `Err`. То же касается `?` на конце вызова `read_to_string`.

Использование оператора `?` позволяют уменьшить количество строк кода и сделать реализацию проще. Написанный в предыдущем примере код можно

сделать ещё короче с помощью сокращения промежуточных переменных и конвейерного вызова нескольких методов подряд, как показано в листинге 9-8:

Файл: src/main.rs

```
use std::fs::File;
use std::io;
use std::io::Read;

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)
}
```

Листинг 9-8. Цепочка вызовов методов после оператора `?`

Мы перенесли в начало функции создание новой переменной `s` типа `String`; эта часть не изменилась. Вместо создания переменной `f` мы добавили вызов `read_to_string` непосредственно к результату `File::open("hello.txt")?`. У нас **ещё есть `?`** в конце вызова `read_to_string`, и мы по-прежнему возвращаем значение `Ok`, содержащее имя пользователя в `s` когда оба метода: `File::open` и `read_to_string` успешны, а не возвращают ошибки. Функциональность снова такая же, как в листинге 9-6 и листинге 9-7; это просто другой, более эргономичный способ решения той же задачи.

Продолжая рассматривать разные способы записи данной функции, листинг 9-9 показывает способ сделать её ещё короче.

Файл: src/main.rs

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

fn read_username_from_file() -> Result<String, io::Error> {
    fs::read_to_string("hello.txt")
}
```

Листинг 9-9: Использование `fs::read_to_string` вместо открытия и чтения файла

Чтение файла в строку довольно распространённая операция, так что Rust

предоставляет удобную функцию `fs::read_to_string`, которая открывает файл, создаёт новую `String`, читает содержимое файла, размещает его в `String` и возвращает её. Конечно, использование функции `fs::read_to_string` не даёт возможности объяснить обработку всех ошибок, поэтому мы сначала изучили длинный способ.

Оператор `?` можно использовать для функций возвращающих `Result`

Оператор `?` может использоваться в функциях, которые имеют возвращаемый тип `Result`, потому что он работает так же, как выражение `match`, определённое в листинге 9-6. Той частью `match`, которая требует возвращаемый тип `Result`, является код `return Err(e)`, таким образом возвращаемый тип функции может быть `Result`, чтобы быть совместимым с этим `return`.

Посмотрим что происходит, если использовать оператор `?` в теле функции `main`, которая, как вы помните, имеет возвращаемый тип `()`:

```
{#rustdoc_include ../../listings/ch09-error-handling/no-listing-06-question-mark-in-main/src/main.rs}
```

При компиляции этого кода, мы получим следующее сообщение об ошибке:

```
{#include ../../listings/ch09-error-handling/no-listing-06-question-mark-in-main/output.txt}
```

Эта ошибка указывает на то, что разрешено использовать оператор `?` только в функциях, которые возвращают `Result` или `Option` или другой тип, который реализует типаж `std::ops::Try`. Если вы пишете функцию, которая не возвращает один из этих типов, и хотите использовать `?` при вызове других функций, возвращающих `Result<T, E>`, у вас есть два варианта решения этой проблемы. Один из методов - изменить тип возвращаемого значения вашей функции на `Result<T, E>`, при условии что у вас нет ограничений, препятствующих этому. Другая техника заключается в использовании `match` или одного из методов `Result<T, E>` для обработки `Result<T, E>` любым подходящим способом.

Функция `main` является специальной и имеются ограничение на то, какой должен быть её возвращаемый тип. Один из допустимых типов для `main` это `()`, другой - `Result<T, E>`, как в примере:

```
{#{rustdoc_include ../listings/ch09-error-handling/no-listing-07-main-returning-result/src/main.rs}}
```

Тип `Box<dyn Error>` называется типаж объектом, о котором мы поговорим в разделе "Использование типаж объектов, которые допускают значения различных типов" Главы 17. А пока вы можете читать обозначение `Box<dyn Error>` как "любая ошибка". Использование `?` в `main` функции с этим возвращаемым типом также разрешено.

Теперь, когда мы обсудили детали вызова `panic!` или возврата `Result`, давайте вернёмся к тому, как решить, какой из случаев подходит для какой ситуации.

panic! или не panic!

Итак, как принимается решение о том, когда следует вызывать `panic!`, а когда вернуть `Result`? При панике код не имеет возможности восстановить своё выполнение. Можно было бы вызывать `panic!` для любой ошибочной ситуации, независимо от того, имеется ли способ восстановления или нет, но с другой стороны, вы принимаете решение от имени вызывающего вас кода, что ситуация необратима. Когда вы возвращаете значение `Result`, вы делегируете принятие решения вызывающему коду. Вызывающий код может попытаться выполнить восстановление способом, который подходит в данной ситуации, или же он может решить, что из ошибки в `Error` нельзя восстановиться и вызовет `panic!`, превратив вашу исправимую ошибку в неисправимую. Поэтому возвращение `Result` является хорошим выбором по умолчанию для функции, которая может дать сбой.

В таких ситуациях как примеры, прототипы и тесты, более уместно писать код, который паникует вместо возвращения `Result`. Давайте рассмотрим почему, а затем мы обсудим ситуации, в которых компилятор не может доказать, что ошибка невозможна, но вы, как человек, можете это сделать. Глава будет заканчиваться некоторыми общими руководящими принципами о том, как решить, стоит ли паниковать в коде библиотеки.

Примеры, прототипирование и тесты

Когда вы пишете пример, иллюстрирующий некоторую концепцию, наличие хорошего кода обработки ошибок может сделать пример менее понятным. Понятно, что в примерах вызов метода `unwrap`, который может привести к панике, является лишь обозначением способа обработки ошибок в приложении, который может отличаться в зависимости от того, что делает остальная часть кода.

Точно так же методы `unwrap` и `expect` являются очень удобными при создании прототипа, прежде чем вы будете готовы решить, как обрабатывать ошибки. Они оставляют чёткие маркеры в коде до момента, когда вы будете готовы сделать программу более надёжной.

Если в тесте происходит сбой при вызове метода, то вы бы хотели, чтобы весь тест не прошёл, даже если этот метод не является тестируемой функциональностью. Поскольку вызов `panic!` это способ, которым тест помечается как провалившийся, использование `unwrap` или `expect` - именно то, что нужно.

Случаи, в которых у вас больше информации, чем у компилятора

Также было бы целесообразно вызывать `unwrap` когда у вас есть какая-то другая логика, которая гарантирует, что `Result` будет иметь значение `Ok`, но вашу логику не понимает компилятор. У вас по-прежнему будет значение `Result` которое нужно обработать: любая операция, которую вы вызываете, все ещё имеет возможность неудачи в целом, хотя это логически невозможно в вашей конкретной ситуации. Если, проверяя код вручную, вы можете убедиться, что никогда не будет вариант с `Err`, то вполне допустимо вызывать `unwrap`. Вот пример:

```
use std::net::IpAddr;  
  
let home: IpAddr = "127.0.0.1".parse().unwrap();
```

Мы создаём экземпляр `IpAddr`, анализируя жёстко закодированную строку. Можно увидеть, что `127.0.0.1` является действительным IP-адресом, поэтому здесь допустимо использование `unwrap`. Однако наличие жёстко закодированной допустимой строки не меняет тип возвращаемого значения метода `parse`: мы все ещё получаем значение `Result` и компилятор все также заставляет нас обращаться с `Result` так, будто возможен вариант `Err`. Это потому, что компилятор недостаточно умён, чтобы увидеть, что эта строка всегда действительный IP-адрес. Если строка IP-адреса пришла от пользователя, то она не является жёстко запрограммированной в программе и, следовательно, может привести к ошибке, мы определённо хотели бы обработать `Result` более надёжным способом.

Руководство по обработке ошибок

Желательно, чтобы код паниковал, если он может оказаться в некорректном состоянии. В этом контексте *некорректное состояние* это когда некоторое допущение, гарантия, контракт или инвариант были нарушены. Например, когда недопустимые, противоречивые или пропущенные значения передаются в ваш код - плюс один или несколько пунктов из следующего перечисленного в списке:

- Не корректное состояние — это что-то неожиданное, отличается от того, что может происходить время от времени, например, когда пользователь вводит данные в неправильном формате.
- Ваш код после этой точки должен полагаться на то, что он не находится в не корректном состоянии, вместо проверок наличия проблемы на каждом этапе.
- Нет хорошего способа закодировать данную информацию в типах, которые вы

используете. Мы рассмотрим пример того, что мы имеем в виду в разделе “[Кодирование состояний и поведения на основе типов](#)” главы 17.

Если кто-то вызывает ваш код и передаёт значения, которые не имеют смысла, лучшим выбором может быть вызов `panic!` для оповещения пользователя библиотеки, что в его коде есть ошибка и он может её исправить. Также `panic!` подходит, если вы вызываете внешний, неподконтрольный вам код, и он возвращает недопустимое состояние, которое вы не можете исправить.

Однако, когда ожидается сбой, лучше вернуть `Result`, чем выполнить вызов `panic!`. В качестве примера можно привести синтаксический анализатор, которому передали неправильно сформированные данные, или HTTP-запрос, возвращающий статус указывающий на то, что вы достигли ограничения на частоту запросов. В этих случаях возврат `Result` означает, что ошибка является ожидаемой и вызывающий код должен решить, как её обрабатывать.

Когда код выполняет операции над данными, он должен проверить, что они корректны, и паниковать, если это не так. Так рекомендуется делать в основном из соображений безопасности: попытка оперировать некорректными данными может подвергнуть ваш код уязвимости. Это основная причина, по которой стандартная библиотека будет вызывать `panic!`, если попытаться получить доступ к памяти вне границ массива: доступ к памяти, не относящейся к текущей структуре данных, является известной проблемой безопасности. Функции часто имеют *контракты*: их поведение гарантируется, только если входные данные отвечают определённым требованиям. Паника при нарушении контракта имеет смысл, потому что это всегда указывает на дефект со стороны вызывающего кода, и это не ошибка, которую вы хотели бы, чтобы вызывающий код явно обрабатывал. На самом деле, нет разумного способа для восстановления вызывающего кода; *программисты*, вызывающие ваш код, должны исправить свой. Контракты для функций, особенно когда нарушение вызывает панику, следует описать в документации по API функции.

Тем не менее, наличие множества проверок ошибок во всех ваших функциях было бы многословным и раздражительным. К счастью, можно использовать систему типов Rust (следовательно и проверку типов компилятором), чтобы она сделала множество проверок вместо вас. Если ваша функция имеет определённый тип в качестве параметра, вы можете продолжить работу с логикой кода зная, что компилятор уже обеспечил правильное значение. Например, если используется обычный тип, а не тип `Option`, то ваша программа ожидает наличие *чего-то* вместо *ничего*. Ваш код не должен будет обрабатывать оба варианта `Some` и `None`: он будет

иметь только один вариант для определённого значения. Код, пытающийся ничего не передавать в функцию, не будет даже компилироваться, поэтому ваша функция не должна проверять такой случай во время выполнения. Другой пример - это использование целого типа без знака, такого как `u32`, который гарантирует, что параметр никогда не будет отрицательным.

Создание пользовательских типов для проверки

Давайте разовьём идею использования системы типов Rust чтобы убедиться, что у нас есть корректное значение, и рассмотрим создание пользовательского типа для валидации. Вспомним игру угадывания числа из Главы 2, в которой наш код просил пользователя угадать число между 1 и 100. Мы никогда не проверяли, что предположение пользователя лежит между этими числами, перед сравнением предположения с загаданным нами числом; мы только проверяли, что оно положительно. В этом случае последствия были не очень страшными: наши сообщения «Слишком много» или «Слишком мало», выводимые в консоль, все равно были правильными. Но было бы лучше подталкивать пользователя к правильным догадкам и иметь различное поведение для случаев, когда пользователь предлагает число за пределами диапазона, и когда пользователь вводит, например, буквы вместо цифр.

Один из способов добиться этого - пытаться разобрать введённое значение как `i32`, а не как `u32`, чтобы разрешить потенциально отрицательные числа, а затем добавить проверку для нахождение числа в диапазоне, например, так:

```
loop {
    // --snip--

    let guess: i32 = match guess.trim().parse() {
        Ok(num) => num,
        Err(_) => continue,
    };

    if guess < 1 || guess > 100 {
        println!("The secret number will be between 1 and 100.");
        continue;
    }

    match guess.cmp(&secret_number) {
        // --snip--
    }
}
```

Выражение `if` проверяет, находится ли наше значение вне диапазона, сообщает пользователю о проблеме и вызывает `continue`, чтобы начать следующую итерацию цикла и попросить ввести другое число. После выражения `if` мы можем продолжить сравнение значения `guess` с загаданным числом, зная, что `guess` лежит в диапазоне от 1 до 100.

Однако это не идеальное решение: если бы было чрезвычайно важно, чтобы программа работала только со значениями от 1 до 100, существовало бы много функций, требующих этого, то такая проверка в каждой функции была бы утомительной (и могла бы отрицательно повлиять на производительность).

Вместо этого можно создать новый тип и поместить проверки в функцию создания экземпляра этого типа, не повторяя их везде. Таким образом, функции могут использовать новый тип в своих сигнатурах и быть уверены в значениях, которые им передают. Листинг 9-13 показывает один из способов, как определить тип `Guess`, чтобы экземпляр `Guess` создавался только при условии, что функция `new` получает значение от 1 до 100.

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess { value }
    }

    pub fn value(&self) -> i32 {
        self.value
    }
}
```

Листинг 9-13. Тип `Guess`, который будет создавать экземпляры только для значений от 1 до 100

Сначала мы определяем структуру с именем `Guess`, которая имеет поле с именем `value` типа `i32`, в котором будет храниться число.

Затем мы реализуем ассоциированную функцию `new`, создающую экземпляры значений типа `Guess`. Функция `new` имеет один параметр `value` типа `i32`, и

возвращает `Guess`. Код в теле функции `new` проверяет, что значение `value` находится между 1 и 100. Если `value` не проходит эту проверку, мы вызываем `panic!`, которая оповестит программиста, написавшего вызывающий код, что в его коде есть ошибка, которую необходимо исправить, поскольку попытка создания `Guess` со значением `value` вне заданного диапазона нарушает контракт, на который полагается `Guess::new`. Условия, в которых `Guess::new` паникует, должны быть описаны в документации к API; мы рассмотрим соглашения о документации, указывающие на возможность появления `panic!` в документации API, которую вы создадите в Главе 14. Если `value` проходит проверку, мы создаём новый экземпляр `Guess`, у которого значение поля `value` равно значению параметра `value`, и возвращаем `Guess`.

Затем мы реализуем метод с названием `value`, который заимствует `self`, не имеет других параметров, и возвращает значение типа `i32`. Этот метод иногда называют извлекатель (getter), потому что его цель состоит в том, чтобы извлечь данные из полей структуры и вернуть их. Этот публичный метод является необходимым, поскольку поле `value` структуры `Guess` является приватным. Важно, чтобы поле `value` было приватным, чтобы код, использующий структуру `Guess`, не мог устанавливать `value` напрямую: код снаружи модуля должен использовать функцию `Guess::new` для создания экземпляра `Guess`, таким образом гарантируя, что у `Guess` нет возможности получить `value`, не проверенное условиями в функции `Guess::new`.

Функция, которая принимает или возвращает только числа от 1 до 100, может объявить в своей сигнатуре, что она принимает или возвращает `Guess`, вместо `i32`, таким образом не будет необходимости делать дополнительные проверки в теле такой функции.

Итоги

Функции обработки ошибок в Rust призваны помочь написанию более надёжного кода. Макрос `panic!` сигнализирует, что ваша программа находится в состоянии, которое она не может обработать, и позволяет сказать процессу чтобы он прекратил своё выполнение, вместо попытки продолжить выполнение с некорректными или неверными значениями. Перечисление `Result` использует систему типов Rust, чтобы сообщить, что операции могут завершиться неудачей, и ваш код мог восстановиться. Можно использовать `Result`, чтобы сообщить

вызывающему коду, что он должен обрабатывать потенциальный успех или потенциальную неудачу. Использование `panic!` и `Result` правильным образом сделает ваш код более надёжным перед лицом неизбежных проблем.

Теперь, когда вы увидели полезные способы использования обобщённых типов `Option` и `Result` в стандартной библиотеке, мы поговорим о том, как работают обобщённые типы и как вы можете использовать их в своём коде.

Обобщённые типы, типажи и время жизни

Каждый язык программирования имеет в своём арсенале эффективные средства борьбы с дублированием кода. В Rust одним из таких инструментов являются обобщённые типы данных - *generics*. Это абстрактные подставные типы на место которых возможно поставить какой-либо конкретный тип или другое свойство. Когда мы пишем код, мы можем выразить поведение обобщённых типов или их связь с другими обобщёнными типами, не зная какой тип будет использован на их месте при компиляции и запуске кода.

Это подобно тому, как функция принимает на вход параметры с разными заранее неизвестными значениями и запускает на них одинаковый код. Функции могут принимать параметры некоторого "обобщённого" типа вместо конкретного типа, вроде `i32` или `String`. Мы уже использовали такие типы данных в Главе 6 (`Option<T>`), в Главе 8 (`Vec<T>` и `HashMap<K, V>`) и в Главе 9 (`Result<T, E>`). В этой главе мы рассмотрим, как определить наши собственные типы данных, функции и методы, используя возможности обобщённых типов.

Прежде всего, мы рассмотрим как для уменьшения дублирования кода извлечь некоторую общую функциональность из кода. Далее, мы будем использовать тот же механизм для создания обобщённой функции из двух функций, которые отличаются только типом их параметров. Мы также объясним, как использовать обобщённые типы данных при определении структур и перечислений.

Затем вы изучите как использовать *типажи* (*traits*) для определения поведения в обобщённом виде. Можно комбинировать типажи с обобщёнными типами для ограничения обобщённого типа только теми типами, которые имеют определённое поведение, в отличии от любых типов.

В конце мы обсудим *времена жизни* (*lifetimes*), вариации обобщённых типов, которые дают компилятору информацию о том, как сроки жизни ссылок относятся друг к другу. Времена жизни позволяют одолживать (borrow) значения во многих ситуациях, предоставляя возможность компилятору удостовериться, что ссылки являются корректными.

Удаление дублирования кода с помощью выделения общей функциональности

Прежде чем перейти к рассмотрению синтаксиса обобщённых типов, предлагаем рассмотреть технику устранения дублирования кода, которая не использует обобщённые типы. Позднее мы применим эту технику для определения и извлечения функциональности которая может быть представлена отдельной функцией принимающей на вход обобщённый тип данных. Точно так же, как вы распознаете дублированный код для извлечения в функцию, вы начнёте распознавать дублированный код, который может использовать обобщённые типы.

Рассмотрим небольшую программу, которая ищет наибольшее число в списке, как показано в листинге 10-1.

Файл: 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!("The largest number is {}", largest);
}
```

Листинг 10-1: Код поиска наибольшего числа в списке

Программа сохраняет вектор целых чисел в переменной `number_list` и помещает первое значение из списка в переменную `largest`. Далее, итератор проходит по всем элементам списка. Если текущий элемент больше числа сохранённого в переменной `largest`, то его значение заменяет предыдущее значение в этой переменной. Если текущий элемент меньше или равен "наибольшему" найденному ранее, то значение переменной не изменяется. После полного перебора всех элементов, переменная `largest` должна содержать наибольшее значение, которое в нашем случае будет равно 100.

Чтобы найти наибольшее число в двух различных списках, мы можем дублировать

код листинга 10-1 и использовать такую же логику в двух различных местах программы, как показано в листинге 10-2:

Файл: 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!("The largest number is {}", 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!("The largest number is {}", largest);
}
```

Листинг 10-2: Программа поиска наибольшего числа в двух списках

Несмотря на то, что код программы работает, дублирование кода утомительно и подвержено ошибкам. Мы должны будем обновить дублируемый код в остальных местах, если захотим изменить его в каком-то одном месте, чтобы достичь единобразия.

Для устранения дублирования мы можем создать дополнительную абстракцию с помощью функции которая сможет работать с любым списком целых чисел переданным ей в качестве входного параметра и находить для этого списка наибольшее число. Данное решение делает код более ясным и позволяет абстрактным образом выразить концепцию поиска наибольшего числа в списке.

В листинге 10-3, мы извлекли код, который находит наибольшее число, в отдельную функцию с именем `largest`. В отличии от кода в листинге 10-1, который находит

наибольшее число только в одном конкретном списке, новая программа может искать наибольшее число в двух разных списках.

Файл: src/main.rs

```
fn largest(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let result = largest(&number_list);
    println!("The largest number is {}", result);
}
```

Листинг 10-3: Абстрактный код для поиска наибольшего числа в двух списках.

Функция `largest` имеет параметр с именем `list`, который представляет срез любых значений типа `i32`, которые мы можем передать в неё. В результате вызова функции, код выполнится с конкретными, переданными в неё значениями. Не беспокойтесь о синтаксисе цикла `for` на данный момент. Мы не ссылаемся здесь на ссылку на `i32`; мы сопоставляем шаблон и деструктурируем каждый `&i32` который получает цикл `for` по этой причине `item` будет типа `i32` внутри тела цикла. Мы подробно рассмотрим сопоставление с образцом в [Главе 18](#).

Итак, вот шаги выполненные для изменения кода из листинга 10-2 в листинг 10-3:

1. Определить дублирующийся код.
2. Извлечь дублирующийся код и поместить в тело функции, определяя входные и выходные значения сигнатуры функции.
3. Обновить и заменить два участка дублирующегося кода вызовом одной

функции.

Далее, мы воспользуемся этими же шагами для обобщённых типов, чтобы различными способами уменьшить дублирование кода. Обобщённые типы позволяют работать над абстрактными типами тем же образом, как тело функции может работать над абстрактным списком `list` вместо конкретных значений.

Например, у нас есть две функции: одна ищет наибольший элемент внутри среза значений типа `i32`, а другая внутри среза значений типа `char`. Как уменьшить такое дублирование? Давайте выяснить!

Обобщённые типы данных

Мы можем использовать обобщённые типы данных для функций или структур, которые затем можно использовать с различными конкретными типами данных. Давайте сначала посмотрим, как объявлять функции, структуры, перечисления и методы, используя обобщённые типы данных. Затем мы обсудим, как обобщённые типы данных влияют на производительность кода.

В объявлении функций

Когда мы объявляем функцию с обобщёнными типами, мы размещаем обобщённые типы в сигнатуре функции, где мы обычно указываем типы данных аргументов и возвращаемое значение. Используя обобщённые типы, мы делаем код более гибким, и предоставляем большую функциональность при вызове нашей функции, предотвращая дублирование кода.

Рассмотрим пример с функцией `largest`. Листинг 10-4 показывает две функции, каждая из которых находит самое большое значение в срезе своего типа.

Файл: `src/main.rs`

```

fn largest_i32(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> char {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&char_list);
    println!("The largest char is {}", result);
}

```

Листинг 10-4: Две функции, отличающихся только именем и типом обрабатываемых данных

Функция `largest_i32` уже встречалась нам: мы извлекли её в листинге 10-3, когда боролись с дублированием кода, она находит наибольшее значение типа `i32` в срезе. Функция `largest_char` находит самое большое значение типа `char` в срезе. Тело у этих функций одинаковое, поэтому давайте избавимся от дублируемого кода, добавив обобщённые типы данных.

Для параметризации типов данных в новой объявляемой функции, нам нужно дать имя обобщённому типу, также как мы это делаем для аргументов функций. Можно использовать любой идентификатор для имени параметра типа. Но мы будем

использовать **T**, потому что, по соглашению, имена параметров в Rust должны быть короткими (обычно длиной в один символ) и именование типов в Rust делается в нотации CamelCase. Сокращение слова "type" до одной буквы **T** является стандартным выбором большинства программистов использующих язык Rust.

Когда мы используем параметр в теле функции, мы должны объявить имя параметра в сигнатуре, так компилятор будет знать, что означает имя. Аналогично, когда мы используем имя параметра в сигнатуре функции, мы должны объявить имя параметра раньше, чем мы его используем. Чтобы определить обобщённую функцию **largest**, поместим объявление имён параметров в треугольные скобки, **<>**, между именем функции и списком параметров, как здесь:

```
fn largest<T>(list: &[T]) -> T {
```

Объявление читается так: функция **largest** является обобщённой по типу **T**. Эта функция имеет один параметр с именем **list**, который является срезом значений с типом данных **T**. Функция **largest** возвращает данные такого же типа **T**.

Листинг 10-5 показывает определение функции **largest** с использованием обобщённых типов данных в её сигнатуре. Листинг также показывает, как мы можем вызвать функцию со срезом данных типа **i32** или **char**. Данный код пока не будет компилироваться, но мы исправим это к концу раздела.

Файл: src/main.rs

```

fn largest<T>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}

```



Листинг 10-5: определение функции `largest` с использованием обобщённых типов, но код пока не компилируется

Если мы скомпилируем программу сейчас, мы получим следующую ошибку:

```

$ cargo run
Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0369]: binary operation `>` cannot be applied to type `T`
--> src/main.rs:5:17
   |
5 |         if item > largest {
   |             ^ ----- T
   |             |
   |             T
   |
help: consider restricting type parameter `T`
|
1 | fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> T {
   | ++++++-----+
For more information about this error, try `rustc --explain E0369`.
error: could not compile `chapter10` due to previous error

```

В подсказке упоминается `std::cmp::PartialOrd`, который является *тиражом*. Мы

поговорим про типажи в следующей секции. Сейчас, ошибка в функции `largest` указывает, что функция не будет работать для всех возможных типов `T`. Так как мы хотим сравнивать значения типа `T` в теле функции, то можно использовать только те типы, данные которых можно упорядочить: можем упорядочить, значит можем и сравнить. Для возможности сравнения, стандартная библиотека имеет типаж `std::cmp::PartialOrd`, который вы можете реализовать для типов (смотрите Дополнение C для большей информации про данный типаж). Вы узнаете, как потребовать чтобы обобщённый тип реализовывал определённый типаж в секции "Типажи как параметры", но сначала давайте рассмотрим другие варианты использования обобщённых типов.

В определении структур

Также можно определять структуры с использованием обобщённых типов в одном или нескольких полях структуры с помощью синтаксиса `<T>`. Листинг 10-6 показывает как определить структуру `Point<T>`, чтобы хранить поля координат `x` и `y` любого типа данных.

Файл: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

Листинг 10-6: структура `Point` содержащая поля `x` и `y` типа `T`

Синтаксис использования обобщённых типов в определении структуры такой же как и в определении функции. Сначала мы объявляем имена параметров внутри треугольных скобок сразу после имени структуры. Затем мы можем использовать обобщённые типы в определении структуры на местах, где ранее мы бы указывали конкретные типы.

Так как мы используем только один обобщённый тип данных для определения структуры `Point<T>`, это определение означает, что структура `Point<T>` является обобщённой с типом `T`, и оба поля `x` и `y` имеют одинаковый тип, каким бы он

типов не являлся. Если мы создадим экземпляр структуры `Point<T>` со значениями разных типов, как показано в Листинге 10-7, наш код не компилируется.

Файл: `src/main.rs`

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}
```



Листинг 10-7: поля `x` и `y` должны быть одного типа, так как они имеют один и тот же обобщённый тип `T`

В этом примере, когда мы присваиваем целочисленное значение 5 переменной `x`, мы сообщаем компилятору, что обобщённый тип `T` будет целым числом для этого экземпляра `Point<T>`. Затем, когда мы указываем значение 4.0 (имеющее тип отличный от целого числа) для `y`, который мы определили имеющим тот же тип, что и `x`, мы получим ошибку несоответствия типов:

```
$ cargo run
Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0308]: mismatched types
--> src/main.rs:7:38
   |
7 |     let wont_work = Point { x: 5, y: 4.0 };
   |                           ^^^ expected integer, found
floating-point number

For more information about this error, try `rustc --explain E0308`.
error: could not compile `chapter10` due to previous error
```

Чтобы определить структуру `Point` где оба `x` и `y` являются обобщёнными, но могут иметь различные типы, можно использовать несколько параметров обобщённого типа. Например, в листинге 10-8 мы можем изменить определение `Point`, чтобы оно было общим для типов `T` и `U` где `x` имеет тип `T` а `y` имеет тип `U`.

Файл: `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 };
}
```

Листинг 10-8: структура `Point<T, U>` обобщена для двух типов, так что `x` и `y` могут быть значениями разных типов

Теперь разрешены все показанные экземпляры типа `Point`! В объявлении можно использовать столько много обобщённых параметров типа, сколько хочется, но использование более чем несколько типов делает код трудно читаемым. Когда вам нужно много обобщённых типов в коде, это может указывать на то, что ваш код нуждается в реструктуризации на более мелкие части.

В определениях перечислений

Как и в случае со структурами, можно определить перечисления для хранения обобщённых типов в их вариантах. Давайте ещё раз посмотрим на перечисление `Option<T>` предоставленное стандартной библиотекой, которое мы использовали в Главе 6:

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

Это определение теперь должно иметь больше смысла. Как видите, перечисление `Option<T>`, которое является обобщённым по типу `T` и имеет два варианта: `Some`, который содержит одно значение типа `T` и вариант `None`, который не содержит никакого значения. Используя перечисление `Option<T>`, можно выразить абстрактную концепцию необязательного значения и так как `Option<T>` является обобщённым, можно использовать эту абстракцию независимо от того, каким будет тип для необязательного значения.

Перечисления также могут использовать в определении несколько обобщённых

типов. Определение перечисления `Result`, которое мы использовали в Главе 9, является таким примером:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Перечисление `Result` имеет два обобщённых типа `T` и `E` и два варианта: `Ok`, которое содержит тип `T`, и `Err`, которое содержит тип `E`. Такое определение позволяет использовать перечисление `Result` везде, где операции могут быть выполнены успешно (возвращая значение типа данных `T`) или неуспешно (возвращая значение типа данных `E`). Это то что мы делали в коде листинга 9-2, где при открытии файла заполнялись данные типа `T`, в примере тип `std::fs::File` или `E` тип `std::io::Error` при ошибке, при каких-либо проблемах открытия файла.

Когда вы в коде распознаете ситуации с несколькими структурами или определениями перечислений, которые отличаются только типами содержащих значений, вы можете избежать дублирования, используя обобщённые типы.

В определении методов

Также, как и в Главе 5, можно реализовать методы структур и перечислений с помощью обобщённых типов и их объявлений. Код листинга 10-9 демонстрирует пример добавления метода с названием `x` в структуру `Point<T>`, которую мы ранее описали в листинге 10-6.

Файл: 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());
}

```

Листинг 10-9. Реализация метода с именем `x` у структуры `Point`, которая будет возвращать ссылку на поле `x` типа `T`

Здесь мы определили метод с именем `x` у `Point<T>` который возвращает ссылку на данные в поле `x`.

Обратите внимание, что нужно объявить `T` сразу после `impl`, чтобы можно было использовать его для указания, что мы реализуем методы для типа `Point<T>`.

Объявляя `T` как обобщённый тип после `impl`, Rust может определить, что тип в угловых скобках у `Point` - это обобщённый, а не конкретный тип.

Мы могли бы, например, реализовать методы только для экземпляров типа `Point<f32>` вместо остальных экземпляров `Point<T>` где используется какой-то другой обобщённый тип. В листинге 10-10 мы реализуем код для конкретного типа `f32`: здесь мы не объявляем иных блоков `impl` для других вариантов обобщённого типа после.

Файл: src/main.rs

```

impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}

```

Листинг 10-10: блок `impl` который применяется только к структуре с конкретным типом для параметра обобщённого типа `T`

Этот код означает, что тип `Point<f32>` будет иметь метод с именем `distance_from_origin`, а другие экземпляры `Point<T>` где `T` имеет тип отличный от `f32` не будут иметь этого метода. Метод измеряет, насколько далеко наша точка находится от точки с координатами (0,0,0,0) и использует математические операции, доступные только для типов с плавающей запятой.

Обобщённые типы в определении структуры не всегда являются теми же, которые вы используете в сигнатуре методов этой же структуры. Чтобы сделать пример более понятным, в листинге 10-11 используются обобщённые типы `X1` и `Y1` для структуры `Point` и `X2` `Y2` для метода `mixup`. Метод создаёт новый `Point` со значением `x` из `self` `Point` (типа `X1`) и значением `y` из другой `Point` (типа `Y2`), переданной в качестве параметра.

Файл: src/main.rs

```
struct Point<X1, Y1> {
    x: X1,
    y: Y1,
}

impl<X1, Y1> Point<X1, Y1> {
    fn mixup<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1, Y2> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

Листинг 10-11: метод, использующий разные обобщённые типы из определения структуры для которой он определён

В функции `main`, мы определили тип `Point`, который имеет `i32` для `x` (со значением `5`) и тип `f64` для `y` (со значением `10.4`). Переменная `p2` является структурой `Point` которая имеет строковый срез для `x` (со значением `"Hello"`) и `char` для `y` (со значением `'c'`). Вызов `mixup` на `p1` с аргументом `p2` создаст для

нас экземпляр структуры `p3`. Новый экземпляр `p3` будет иметь для `x` тип `i32` (потому что `x` взят из `p1`), а для `y` тип `char` (потому что `y` взят из `p2`). Вызов макроса `println!` выведет `p3.x = 5, p3.y = c.`

Цель этого примера продемонстрировать ситуацию, в которой одни обобщённые параметры объявлены в `impl`, а другие в определении метода. Здесь обобщённые параметры `X1` и `Y1` объявляются после `impl`, потому что они идут вместе с определением структуры. Обобщённые параметры типа `X2` и `Y2` объявляются после `fn mixup`, потому что они относятся только к методу.

Производительность кода использующего обобщённые типы

Вы могли бы задаться вопросом, появляются ли дополнительные вычисления во время выполнения кода использующего параметры обобщённого типа. Хорошей новостью является то, что Rust реализует обобщённые типы таким способом, что ваш код не работает медленнее при их использовании, чем если бы это было с конкретными типами.

Rust достигает этого благодаря выполнению мономорфизации кода использующего обобщения. *Мономорфизация* - это процесс превращения обобщённого кода в конкретный код во время компиляции, при котором из кода с обобщёнными типами генерируется код содержащий конкретные типы которые могут встретиться в вашем приложении.

В этом процессе компилятор выполняет противоположные шаги, которые обычно используются для создания обобщённой функции в листинге 10-5: компилятор просматривает все места, где вызывается обобщённый код и генерирует код для конкретных типов, с которыми вызван обобщённый код.

Давайте посмотрим, как это работает, на примере, который 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 считывает значения, которые были использованы у экземпляра `Option<T>` и определяет два вида `Option<T>`: один для `i32`, а другой для `f64`. Таким образом, он расширяет общее определение `Option<T>` в

`Option_i32` и `Option_f64`, тем самым заменяя обобщённое определение на конкретное.

Мономорфизированная версия кода выглядит следующим образом. Обобщённый `Option<T>` заменяется конкретными определениями, созданными компилятором:

Файл: 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);
}
```

Так как Rust компилирует обобщённый код, в код указывающий тип в каждом экземпляре, то мы не платим временем выполнения за использование обобщённых типов. Когда код выполняется, он работает так же, как если бы мы дублировали каждое определение вручную. Процесс мономорфизации делает обобщённые типы Rust чрезвычайно эффективными во время выполнения.

Типажи: определение общего поведения

Типаж сообщает компилятору Rust о функциональности, которой обладает определённый тип и которой он может поделиться с другими типами. Можно использовать типажи, чтобы определять общее поведение абстрактным способом. Можно использовать типажи для ограничения обобщённого типа: указать, что обобщённым типом может быть любой тип который реализует определённое поведение.

Примечание: Типажи похожи на функциональность часто называемую *интерфейсами* в других языках, хотя и с некоторыми различиями.

Определение типажа

Поведение типа определяется теми методами, которые мы можем вызвать у данного типа. Различные типы разделяют одинаковое поведение, если мы можем вызвать одни и те же методы у этих типов. Определение типажей - это способ сгруппировать сигнатуры методов вместе для того, чтобы описать общее поведение, необходимое для достижения определённой цели.

Например, скажем есть несколько структур, которые имеют различный тип и различное количество текста: структура `NewsArticle`, которая содержит новости, напечатанные в различных местах в мире; структура `Tweet`, которая содержит 280 символьную строку твита и мета-данные, обозначающие является ли твит новым или ответом на другой твит.

Мы хотим создать библиотеку медиа-агрегатора, которая может отображать сводку данных сохранённых в экземплярах структур `NewsArticle` или `Tweet`. Чтобы этого достичь, нам необходимо иметь возможность для каждой структуры сделать короткую сводку на основе имеющихся данных: надо, чтобы обе структуры реализовали общее поведение. Мы можем делать такую сводку вызовом метода `summarize` у экземпляра объекта. Пример листинга 10-12 иллюстрирует определение типажа `Summary`, который выражает данное поведение:

Файл: `src/lib.rs`

```
pub trait Summary {  
    fn summarize(&self) -> String;  
}
```

Листинг 10-12: Определение типажа `Summary`, который содержит поведение предоставленное методом `summarize`

Здесь мы объявляем типаж с использованием ключевого слова `trait`, а затем его название, которым является `Summary` в данном случае. Внутри фигурных скобок объявляются сигнатуры методов, которые описывают поведения типов, реализующих данный типаж, в данном случае поведение определяется только одной сигнатурой метода: `fn summarize(&self) -> String`.

После сигнатуры метода, вместо предоставления реализации в фигурных скобках, мы используем точку с запятой. Каждый тип, реализующий данный типаж, должен предоставить своё собственное поведение для данного метода. Компилятор обеспечит, что любой тип содержащий типаж `Summary`, будет также иметь и метод `summarize` объявленный с точно такой же сигнатурой.

Типаж может иметь несколько методов в описании его тела: сигнатуры методов перечисляются по одной на каждой строке и должны заканчиваться символом `;`.

Реализация типажа у типа

Теперь, после того как мы определили желаемое поведение используя типаж `Summary`, можно реализовать его у типов в нашем медиа-агрегаторе. Листинг 10-13 показывает реализацию типажа `Summary` у структуры `NewsArticle`, которая использует для создания сводки в методе `summarize` заголовок, автора и место публикации статьи. Для структуры `Tweet` мы определяем реализацию `summarize` используя пользователя и полный текст твита, полагая содержание твита уже ограниченным 280 символами.

Файл: `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!("{} by {} ({})", 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)
    }
}
```

Код программы 10-13: Реализация типажа `Summary` для структур `NewsArticle` и `Tweet`

Реализация типажа у типа аналогична реализации обычных методов. Разница в том, что после `impl` мы ставим имя типажа, который мы хотим реализовать, затем используем ключевое слово `for`, а затем указываем имя типа, для которого мы хотим сделать реализацию типажа. Внутри блока `impl` мы помещаем сигнатуру метода объявленную в типаже. Вместо добавления точки с запятой в конце, после каждой сигнатуры используются фигурные скобки и тело метода заполняется конкретным поведением, которое мы хотим получить у методов типажа для конкретного типа.

После того, как мы реализовали типаж, можно вызвать его методы у экземпляров `NewsArticle` и `Tweet` тем же способом, что и вызов обычных методов, например так:

```
use aggregator::{Summary, Tweet};

fn main() {
    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
}
```

Данный код напечатает: `1 new tweet: horse_ebooks: of course, as you probably already know, people.`

Обратите внимание, что поскольку мы определили типаж `Summary` и типы `NewsArticle` и `Tweet` в одном и том же файле `lib.rs` примера 10-13, все они находятся в одной области видимости. Допустим, что `lib.rs` предназначен для крейта, который мы назвали `aggregator` и кто-то ещё хочет использовать функциональность нашего крейта для реализации типажа `Summary` у структуры, определённой в области видимости внутри их библиотеки. Им нужно будет сначала подключить типаж в их область видимости. Они сделали бы это, указав `use aggregator::Summary;`, что позволит реализовать `Summary` для их типа. Типажу `Summary` также необходимо быть публичным для реализации в других крейтах, потому мы поставили ключевое слово `pub` перед `trait` в листинге 10-12.

Одно ограничение, на которое следует обратить внимание при реализации типажей это то, что мы можем реализовать типаж для типа, только если либо типаж, либо тип являются локальным для нашего крейта. Например, можно реализовать типажи из стандартной библиотеки, такие как `Display` для пользовательского типа `Tweet` являющимся частью функциональности крейта `aggregator`, потому что тип `Tweet` является локальным в крейте `aggregator`. Мы также можем реализовать типаж `Summary` для `Vec<T>` в нашем крейте `aggregator`, потому что типаж `Summary` является локальным для крейта `aggregator`.

Но мы не можем реализовать внешние типажи для внешних типов. Например, мы не можем реализовать функцию `Display` для `Vec<T>` в нашем крейте `aggregator`, потому что и типаж `Display` и тип `Vec<T>` определены в стандартной библиотеке, а не локально в нашем крейте `aggregator`. Это ограничение является частью

свойства программы называемое *согласованность*, а точнее *сиротское правило* (*orphan rule*), называемое так, потому что родительский тип не представлен. Это правило гарантирует, что код других людей не может сломать ваш код и наоборот. Без этого правила два крейта могли бы реализовать один типаж для одинакового типа и Rust не будет знать, какой реализацией пользоваться.

Реализация поведения по умолчанию

Иногда полезно иметь поведение по умолчанию для некоторых или всех методов в типаже вместо того, чтобы требовать реализации всех методов в каждом типе, реализующим данный типаж. Затем, когда мы реализуем типаж для определённого типа, можно сохранить или переопределить поведение каждого метода по умолчанию уже внутри типов.

В примере 10-14 показано, как указать строку по умолчанию для метода `summarize` из типажа `Summary` вместо определения только сигнатуры метода, как мы сделали в примере 10-12.

Файл: `src/lib.rs`

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
```

Листинг 10-14. Определение типажа `Summary` с реализацией метода `summarize` по умолчанию

Для использования реализации по умолчанию при создании сводки у экземпляров `NewsArticle` вместо определения пользовательской реализации, мы указываем пустой блок `impl` с `impl Summary for NewsArticle {}`.

Хотя мы больше не определяем метод `summarize` непосредственно в `NewsArticle`, мы предоставили реализацию по умолчанию и указали, что `NewsArticle` реализует типаж `Summary`. В результате мы всё ещё можем вызвать метод `summarize` у экземпляра `NewsArticle`, например так:

```
let article = NewsArticle {
    headline: String::from("Penguins win the Stanley Cup Championship!"),
    location: String::from("Pittsburgh, PA, USA"),
    author: String::from("Iceburgh"),
    content: String::from(
        "The Pittsburgh Penguins once again are the best \
        hockey team in the NHL.",
    ),
};

println!("New article available! {}", article.summarize());
```

Этот код печатает `New article available! (Read more...)`.

Создание реализации по умолчанию для метода `summarize` не требует от нас изменений чего-либо в реализации `Summary` для типа `Tweet` в листинге 10-13. Причина заключается в том, что синтаксис для переопределения реализации по умолчанию является таким же, как синтаксис для реализации метода типажа, который не имеет реализации по умолчанию.

Реализации по умолчанию могут вызывать другие методы в том же типаже, даже если эти другие методы не имеют реализации по умолчанию. Таким образом, типаж может предоставить много полезной функциональности и только требует от разработчиков указывать небольшую его часть. Например, мы могли бы определить типаж `Summary` имеющий метод `summarize_author`, реализация которого требуется, а затем определить метод `summarize` который имеет реализацию по умолчанию, которая внутри вызывает метод `summarize_author`:

```
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Read more from {}...)", self.summarize_author())
    }
}
```

Чтобы использовать такую версию типажа `Summary`, нужно только определить метод `summarize_author`, при реализации типажа для типа:

```
impl Summary for Tweet {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}
```

После того, как мы определим `summarize_author`, можно вызвать `summarize` для экземпляров структуры `Tweet` и реализация по умолчанию метода `summarize` будет вызывать определение `summarize_author` которое мы уже предоставили. Так как мы реализовали метод `summarize_author` типажа `Summary`, то типаж даёт нам поведение метода `summarize` без необходимости писать код.

```
let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from(
        "of course, as you probably already know, people",
    ),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summarize());
```

Этот код печатает `1 new tweet: (Read more from @horse_ebooks...)`.

Обратите внимание, что невозможно вызвать реализацию по умолчанию из переопределённой реализации того же метода.

Типажи как параметры

Теперь, когда вы знаете, как определять и реализовывать типажи, можно изучить, как использовать типажи, чтобы определить функции, которые принимают много различных типов.

Например, в листинге 10-13 мы реализовали типаж `Summary` для типов структур `NewsArticle` и `Tweet`. Можно определить функцию `notify` которая вызывает метод `summarize` с параметром `item`, который имеет тип реализующий типаж `Summary`. Для этого можно использовать синтаксис `&impl Trait`, например так:

```
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

Вместо конкретного типа у параметра `item` указывается ключевое слово `impl` и имя типажа. Этот параметр принимает любой тип, который реализует указанный типаж. В теле `notify` мы можем вызывать любые методы у экземпляра `item`, которые должны быть определены при реализации типажа `Summary`, например

можно вызвать метод `summarize`. Мы можем вызвать `notify` и передать в него любой экземпляр `NewsArticle` или `Tweet`. Код, который вызывает данную функцию с любым другим типом, таким как `String` или `i32`, не будет компилироваться, потому что эти типы не реализуют типаж `Summary`.

Синтаксис ограничения типажа

Синтаксис `impl Trait` работает для простых случаев, но на самом деле является синтаксическим сахаром для более длинной формы, которая называется *ограничением типажа*; это выглядит так:

```
pub fn notify<T: Summary>(item: &T) {  
    println!("Breaking news! {}", item.summarize());  
}
```

Эта более длинная форма эквивалентна примеру в предыдущем разделе, но она более многословна. Мы помещаем объявление параметра обобщённого типа с ограничением типажа после двоеточия внутри угловых скобок.

Синтаксис `impl Trait` удобен и делает более выразительным код в простых случаях. Синтаксис ограничений типажа может выразить большую сложность в других случаях. Например, у нас может быть два параметра, которые реализуют типаж `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 иметь `item1` и `item2` разных типов, то использование `impl Trait` было бы уместно (до тех пор, пока оба типа реализуют `Summary`). Если мы хотим форсировать, чтобы оба параметра имели одинаковый тип, то это можно выразить только с использованием ограничения типажа, например так:

```
pub fn notify<T: Summary>(item1: &T, item2: &T) {
```

Обобщённый тип `T` указан для типов параметров `item1` и `item2` и ограничивает функцию так, что конкретные значения типов переданные аргументами в `item1` и `item2` должны быть одинаковыми.

Задание нескольких границ типажей с помощью синтаксиса +

Также можно указать более одного ограничения типажа. Скажем, мы хотели бы использовать в методе `notify` для параметра `item` с форматированием отображения, также как метод `summarize`: для этого мы указываем в определении `notify`, что `item` должен реализовывать как типаж `Display` так и `Summary`. Мы можем сделать это используя синтаксис `+`:

```
pub fn notify(item: &(impl Summary + Display)) {
```

Синтаксис `+` также допустим с ограничениями типажа для обобщённых типов:

```
pub fn notify<T: Summary + Display>(item: &T) {
```

При наличии двух ограничений типажа, тело метода `notify` может вызывать метод `summarize` и использовать `{}` для форматирования `item` при его печати.

Более ясные границы типажа с помощью `where`

Использование слишком большого количества ограничений типажа имеет свои недостатки. Каждый обобщённый тип имеет свои границы типажа, поэтому функции с несколькими параметрами обобщённого типа могут содержать много информации об ограничениях между названием функции и списком её параметров затрудняющих чтение сигнатуры. По этой причине в Rust есть альтернативный синтаксис для определения ограничений типажа внутри предложения `where` после сигнатуры функции. Поэтому вместо того, чтобы писать так:

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {
```

можно использовать предложение `where`, например так:

```
fn some_function<T, U>(t: &T, u: &U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{
```

Сигнатура этой функции менее загромождена: название функции, список параметров, и возвращаемый тип находятся рядом, а сигнатура не содержит в себе множество ограничений типажа.

Возврат значений типа реализующего определённый типаж

Также можно использовать синтаксис `impl Trait` в возвращаемой позиции, чтобы вернуть значение некоторого типа реализующего типаж, как показано здесь:

```
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    }
}
```

Используя `impl Summary` для возвращаемого типа, мы указываем, что функция `returns_summarizable` возвращает некоторый тип, который реализует типаж `Summary` без обозначения конкретного типа. В этом случае `returns_summarizable` возвращает `Tweet`, но код, вызывающий эту функцию, этого не знает.

Возможность возвращать тип, который определяется только реализуемым им признаком, особенно полезна в контексте замыканий и итераторов, которые мы рассмотрим в Главе 13. Замыкания и итераторы создают типы, которые знает только компилятор или типы, которые очень долго указывать. Синтаксис `impl Trait` позволяет кратко указать, что функция возвращает некоторый тип, который реализует типаж `Iterator` без необходимости писать очень длинный тип.

Однако, `impl Trait` возможно использовать, если возвращаете только один тип. Например, данный код, который возвращает значения или типа `NewsArticle` или типа `Tweet`, но в качестве возвращаемого типа объявляет `impl Summary`, не будет работать:



```
fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        NewsArticle {
            headline: String::from(
                "Penguins win the Stanley Cup Championship!",
            ),
            location: String::from("Pittsburgh, PA, USA"),
            author: String::from("Iceburgh"),
            content: String::from(
                "The Pittsburgh Penguins once again are the best \
                hockey team in the NHL.",
            ),
        }
    } else {
        Tweet {
            username: String::from("horse_ebooks"),
            content: String::from(
                "of course, as you probably already know, people",
            ),
            reply: false,
            retweet: false,
        }
    }
}
```

Возврат либо `NewsArticle` либо `Tweet` не допускается из-за ограничений того, как реализован синтаксис `impl Trait` в компиляторе. Мы рассмотрим, как написать функцию с таким поведением в разделе ["Использование объектов типажей, которые разрешены для значений или разных типов"](#) Главы 17.

Исправление кода функции `largest` с помощью ограничений типажа

Теперь, когда вы знаете, как указать поведение, которое вы хотите использовать для ограничения параметра обобщённого типа, давайте вернёмся к листингу 10-5 и исправим определение функции `largest`. В прошлый раз мы пытались запустить этот код, но получили ошибку:

```
$ cargo run
Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0369]: binary operation `>` cannot be applied to type `T`
--> src/main.rs:5:17
5 |         if item > largest {
   |             ^ ----- T
   |
   |             T
   |
help: consider restricting type parameter `T`

1 | fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> T {
   | ++++++  
For more information about this error, try `rustc --explain E0369`.
error: could not compile `chapter10` due to previous error
```

В теле функции `largest` мы хотели сравнить два значения типа `T` используя оператор больше чем (`>`). Так как этот оператор определён у типажа `std::cmp::PartialOrd` из стандартной библиотеки как метод по умолчанию, то нам нужно указать `PartialOrd` в качестве ограничения для типа `T`: благодаря этому функция `largest` сможет работать со срезами любого типа, значения которого мы можем сравнить. Нам не нужно подключать `PartialOrd` в область видимости, потому что он есть в авто-импорте. Изменим сигнатуру `largest`, чтобы она выглядела так:

```
fn largest<T: PartialOrd>(list: &[T]) -> T {
```

На этот раз при компиляции кода мы получаем другой набор ошибок:

```
$ cargo run
Compiling chapter10 v0.1.0 (file:///projects/chapter10)
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
|         move occurs because `list[_]` has type `T`, which
does not implement the `Copy` trait
|             help: consider borrowing here: `&list[0]`

error[E0507]: cannot move out of a shared reference
--> src/main.rs:4:18
|
4 |     for &item in list {
|     ----  ^^^^
|
|         ||
|         |data moved here
|         |move occurs because `item` has type `T`, which does not
implement the `Copy` trait
|             help: consider removing the `&`: `item`


Some errors have detailed explanations: E0507, E0508.
For more information about an error, try `rustc --explain E0507`.
error: could not compile `chapter10` due to 2 previous errors
```

Ключевая строка в этой ошибке `cannot move out of type [T], a non-copy slice`. В нашей необобщённой версии функции `largest` мы пытались найти самый большой элемент только для типа `i32` или `char`. Как обсуждалось в разделе "Данные только для стека: Копирование" Главы 4, типы подобные `i32` и `char`, имеющие известный размер, могут храниться в стеке, поэтому они реализуют типаж `Copy`. Но когда мы сделали функцию `largest` обобщённой, для параметра `list` стало возможным иметь типы, которые не реализуют типаж `Copy`. Следовательно, мы не сможем переместить значение из переменной `list[0]` в переменную `largest`, в результате чего появляется эта ошибка.

Чтобы вызывать этот код только с теми типами, которые реализуют типаж `Copy`, можно добавить типаж `Copy` в список ограничений типа `T`! Листинг 10-15 показывает полный код обобщённой функции `largest`, которая будет компилироваться, пока типы значений среза передаваемых в функцию, реализуют одновременно типажи `PartialOrd` и `Copy`, как это делают `i32` и `char`.

Файл: src/main.rs

```

fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}

```

Листинг 10-15: Объявление функции `largest` работающей с любыми обобщёнными типами, которые реализуют типажи `PartialOrd` и `Copy`

Если мы не хотим ограничить функцию `largest` типами, которые реализуют типаж `Copy`, мы можем указать, что `T` имеет ограничение типажа `Clone` вместо `Copy`. Затем мы могли бы клонировать каждое значение в срезе, если бы хотели чтобы функция `largest` забирала владение. Использование функции `clone` означает, что потенциально делается больше операций выделения памяти в куче для типов, которые владеют данными в куче, например для `String`. В то же время стоит помнить о том, что выделение памяти в куче может быть медленным, если мы работаем с большими объёмами данных.

Ещё один способ, который мы могли бы реализовать в `largest` - это создать функцию возвращающую ссылку на значение `T` из среза. Если мы изменим возвращаемый тип на `&T` вместо `T`, то тем самым изменим тело функции, чтобы она возвращала ссылку, тогда нам были бы не нужны ограничения входных значений типажами `Clone` или `Copy` и мы могли бы избежать выделения памяти в куче. Попробуйте реализовать эти альтернативные решения самостоятельно!

Использование ограничений типажа для условной реализации методов

Используя ограничение типажа с блоком `impl`, который использует параметры обобщённого типа, можно реализовать методы условно, для тех типов, которые реализуют указанный типаж. Например, тип `Pair<T>` в листинге 10-16 всегда реализует функцию `new`. Но `Pair<T>` реализует метод `cmp_display` только если его внутренний тип `T` реализует типаж `PartialOrd` (позволяющий сравнивать) и типаж `Display` (позволяющий выводить на печать).

Файл: src/lib.rs

```
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!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

Листинг 10-17: Условная реализация методов у обобщённых типов в зависимости от ограничений типажа

Мы также можем условно реализовать типаж для любого типа, который реализует другой типаж. Реализации типажа для любого типа, который удовлетворяет ограничениям типажа называются *общими реализациями* и широко используются в стандартной библиотеке Rust. Например, стандартная библиотека реализует типаж `ToString` для любого типа, который реализует типаж `Display`. Блок `impl` в стандартной библиотеке выглядит примерно так:

```
impl<T: Display> ToString for T {  
    // --snip--  
}
```

Поскольку стандартная библиотека имеет эту общую реализацию, то можно вызвать метод `to_string` определённый типажом `ToString` для любого типа, который реализует типаж `Display`. Например, мы можем превратить целые числа в их соответствующие `String` значения, потому что целые числа реализуют типаж `Display`:

```
let s = 3.to_string();
```

Общие реализации приведены в документации к типажу в разделе "Implementors".

Типажи и ограничения типажей позволяют писать код, который использует параметры обобщённого типа для уменьшения дублирования кода, а также указывая компилятору, что мы хотим обобщённый тип, чтобы иметь определённое поведение. Затем компилятор может использовать информацию про ограничения типажа, чтобы проверить, что все конкретные типы, используемые с нашим кодом, обеспечивают правильное поведение. В динамически типизированных языках мы получили бы ошибку во время выполнения, если бы вызвали метод для типа, который не реализует тип определяемый методом. Но Rust перемещает эти ошибки на время компиляции, поэтому мы вынуждены исправить проблемы, прежде чем наш код начнёт работать. Кроме того, мы не должны писать код, который проверяет своё поведение во время выполнения, потому что это уже проверено во время компиляции. Это повышает производительность без необходимости отказываться от гибкости обобщённых типов.

Другой тип обобщения, который мы уже использовали, называется *временами жизни* (*lifetimes*). Вместо гарантирования того, что тип ведёт себя так, как нужно, время жизни гарантирует что ссылки действительны до тех пор, пока они нужны. Давайте посмотрим, как времена жизни это делают.

Валидация ссылок при помощи времён жизни

Когда мы говорили о ссылках в разделе "Ссылки и заимствование" Главы 4, мы опустили весьма важную деталь: каждая ссылка в Rust имеет время жизни (*lifetime*), определяющее область действия, в которой ссылка является действительной. В большинстве случаев, времена жизни выводятся неявно также как у типов. Мы должны явно аннотировать типы, когда возможно выведение нескольких типов. Аналогичным образом, мы должны аннотировать времена жизни, когда времена жизни ссылок могут быть соотнесены несколькими различными способами. Rust требует, чтобы мы аннотировали отношения, используя обобщённые параметры времени жизни для гарантирования того, что реальные ссылки используемые во время выполнения, будут однозначно действительными.

Аннотирование времени жизни — это концепция отсутствующая в большинстве других языков программирования, так что она может показаться незнакомой. Хотя в этой главе мы не будем полностью рассматривать времена жизни, мы обсудим распространённые способы, с помощью которых вы можете столкнуться с синтаксисом времени жизни, чтобы вы могли познакомиться с этой концепцией.

Времена жизни предотвращают появление недействительных ссылок

Основная цель времён жизни состоит в том, чтобы предотвратить недействительные ссылки (dangling references), которые приводят к тому, что программа ссылается на данные отличные от данных на которые она должна ссылаться. Рассмотрим программу из листинга 10-17, которая имеет внешнюю и внутреннюю области видимости.

```
{  
    let r;  
  
    {  
        let x = 5;  
        r = &x;  
    }  
  
    println!("r: {}", r);  
}
```



Листинг 10-17: Попытка использования ссылки, значение которой вышло из области видимости

Примечание: примеры в листингах 10-17, 10-18 и 10-24 объявляют переменные без предоставления им начального значения, поэтому переменные существуют во внешней области видимости. На первый взгляд может показаться, что это противоречит отсутствию нулевых (null) значений. Однако, если мы попытаемся использовать переменную, прежде чем дать ей значение, мы получим ошибку во время компиляции, которая показывает, что Rust действительно не позволяет использование нулевых (null) значений.

Внешняя область видимости объявляет переменную с именем `r` без начального значения, а внутренняя область объявляет переменную с именем `x` с начальным значением `5`. Во внутренней области мы пытаемся установить значение `r` как ссылку на `x`. Затем внутренняя область видимости заканчивается и мы пытаемся напечатать значение из `r`. Этот код не будет скомпилирован, потому что значение на которое ссылается `r` исчезает из области видимости, прежде чем мы попробуем использовать его. Вот сообщение об ошибке:

```
$ cargo run
Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `x` does not live long enough
--> src/main.rs:7:17
|
7 |         r = &x;
|             ^^^ borrowed value does not live long enough
8 |     }
|     - `x` dropped here while still borrowed
9 |
10 |     println!("r: {}", r);
|                 - borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` due to previous error
```

Переменная `x` «не живёт достаточно долго». Причина в том, что `x` выйдет из области видимости, когда эта внутренняя область закончится в строке 7. Но `r` все ещё является действительной во внешней области видимости; поскольку её охват больше, мы говорим, что она «живёт дольше». Если бы Rust позволил такому коду работать, то переменная `r` бы смогла ссылаться на память, которая была освобождена (в тот момент, когда `x` вышла из внутренней области видимости) и всё что мы попытались бы сделать с `r` не работало бы правильно. Так как же Rust определяет, что этот код неверен? Он использует анализатор заимствований.

Анализатор заимствований

Компилятор Rust имеет в своём составе *анализатор заимствований*, который сравнивает области видимости для определения являются ли все заимствования действительными. В листинге 10-18 показан тот же код, что и в листинге 10-17, но с аннотациями, показывающими времена жизни переменных.

```
{
    let r; // -----+-- 'a
           //   |
{
    let x = 5; // -+-- 'b |
    r = &x; //   | |
}
println!("r: {}", r); //   |
} // -----+
```



Пример 10-18: Описание времён жизни переменных `r` и `x`, с помощью идентификаторов времени жизни `'a` и `'b`

Здесь мы описали время жизни для `r` с помощью `'a` и время жизни `x` с помощью `'b`. Как видите, внутренний блок времени жизни `'b` гораздо меньше времени жизни внешнего блока `'a`. Во время компиляции Rust сравнивает размер двух времён жизни и видит, что `r` имеет время жизни `'a`, но ссылается на память со временем жизни `'b`. Программа отклоняется, потому что `'b` короче, чем `'a`: объект ссылки не живёт так же долго как сама ссылка на него.

Листинг 10-19 исправляет код так, что в нём нет проблем с недействительными ссылками: он компилируется без ошибок.

```
{
    let x = 5; // -----+-- 'b
               //   |
    let r = &x; // --+-- 'a |
               //   | |
    println!("r: {}", r); //   | |
                         // --+
} // -----+
```

Листинг 10-19: Все ссылки действительны, поскольку данные имеют большее время жизни, чем ссылка на эти данные

Здесь переменная `x` имеет время жизни `'b`, которое больше, чем время жизни

'**a**'. Это означает, что переменная **r** может ссылаться на переменную **x** потому что Rust знает, что ссылка в переменной **r** будет всегда действительной до тех пор, пока переменная **x** является действительной.

После того, как мы на примерах рассмотрели времена жизни ссылок и обсудили как Rust их анализирует, давайте поговорим об общённых временах жизни входных параметров и возвращаемых значений функций.

Обобщённые времена жизни в функциях

Давайте напишем функцию, которая возвращает наиболее длинный срез строки из двух. Эта функция принимает два среза строки и вернёт один срез строки. После того как мы реализовали функцию **longest**, код в листинге 10-20 должен вывести **The longest string is abcd**.

Файл: src/main.rs

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}
```

Листинг 10-20: Функция **main** вызывает функцию **longest** для поиска наибольшей строки

Обратите внимание, что мы хотим чтобы функция принимала строковые срезы, которые являются ссылками, потому что мы не хотим, чтобы функция **longest** забирала во владение параметры. Обратитесь к разделу "[Строковые фрагменты как параметры](#)" Главы 4 для более подробного обсуждения того, почему параметры используемые в листинге 10-20 выбраны именно таким образом.

Если мы попробуем реализовать функцию **longest** так, как это показано в листинге 10-22, то программа не будет скомпилирована:

Файл: src/main.rs

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```



Листинг 10-21: Реализация функции `longest`, которая возвращает наибольший срез строки, но пока не компилируется

Вместо этого мы получим следующую ошибку, сообщающую об ошибке в определении времени жизни возвращаемого параметра:

```
$ cargo run
Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0106]: missing lifetime specifier
--> src/main.rs:9:33
9 | fn longest(x: &str, y: &str) -> &str {
|         ----      ^ expected named 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`
help: consider introducing a named lifetime parameter
|
9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
|         +++++     ++          ++          ++
For more information about this error, try `rustc --explain E0106`.
error: could not compile `chapter10` due to previous error
```

Текст показывает, что возвращаемому типу нужен обобщённый параметр времени жизни, потому что Rust не может определить, относится ли возвращаемая ссылка к `x` или к `y`. На самом деле, мы тоже не знаем, потому что блок `if` в теле функции возвращает ссылку на `x`, а блок `else` возвращает ссылку на `y`!

Когда мы определяем функцию `longest` таким образом, то мы не знаем конкретных значений передаваемых в неё. Поэтому мы не знаем какая из ветвей оператора `if` или `else` будет выполнена. Мы также не знаем конкретных времён жизни ссылок, передаваемых в функцию, из-за чего не можем посмотреть на их области видимости, как мы делали в примерах 10-19 и 10-20, чтобы убедиться в том, что возвращаемая ссылка всегда действительна. Анализатор заимствований тоже не может этого определить, потому что не знает как времена жизни переменных `x` и

y соотносятся с временем жизни возвращаемого значения. Мы добавим обобщённый параметр времени жизни, который определит отношения между ссылками, чтобы анализатор зависимостей мог провести анализ ссылок с помощью проверки заимствования.

Синтаксис аннотации времени жизни

Аннотации времени жизни не меняют продолжительность жизни каких-либо ссылок. Так же как функции могут принимать любой тип, когда в сигнатуре указан параметр обобщённого типа, функции могут принимать ссылки с любым временем жизни с помощью добавления обобщённого параметра времени жизни. Аннотации времени жизни описывают отношения времён жизни нескольких ссылок друг к другу, не влияя на само время жизни.

Аннотации времени жизни имеют немного необычный синтаксис: имена параметров времени жизни должны начинаться с апострофа `'`, они обычно очень короткие и пишутся в нижнем регистре. Обычно, по умолчанию, большинство людей использует имя `'a`. Аннотации параметров времени жизни следуют после символа `&` и отделяются пробелом от названия ссылочного типа.

Приведём несколько примеров: у нас есть ссылка на `i32` без указания времени жизни, ссылка на `i32`, с временем жизни имеющим имя `'a` и изменяемая ссылка на `i32`, которая тоже имеет время жизни `'a`.

```
&i32          // ссылка
&'a i32      // ссылка с явным временем жизни
&'a mut i32 // изменяемая ссылка с явным временем жизни
```

Одна аннотация времени жизни сама по себе не имеет большого смысла, потому что эти аннотации призваны сообщить компилятору Rust как соотносятся между собой несколько обобщённых параметров времени жизни. Предположим, что у нас есть функция с параметром `first`, имеющим ссылочный тип данных `&i32` и временем жизни `'a`, и вторым параметром `second`, который также имеет ссылочный тип `&i32` со временем жизни `'a`. Аннотации времени жизни этих параметров имеют одинаковое имя, что говорит о том, что обе ссылки `first` и `second` должны жить одинаково долго.

Аннотации времени жизни в сигнтурах функций

Теперь давайте рассмотрим аннотации времён жизни в контексте функции `longest`. Как в случае с обобщёнными параметрами типов, необходимо объявить параметры времени жизни внутри угловых скобок между именем функции и списком параметров. Ограничение, которое мы хотим выразить в этой сигнатуре, заключается в том, что времена жизни обоих параметров и времена жизни возвращаемой ссылки связаны таким образом, что возвращаемая ссылка будет действительной до тех пор, пока верны оба параметра. Мы назовём времена жизни `'a` а затем добавим его к каждой ссылке, как показано в листинге 10.22.

Файл: src/main.rs

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Листинг 10-23: В определении функции `longest` указано, что все ссылки должны иметь одинаковое время жизни обозначенное как `'a`

Этот код должен компилироваться и давать желаемый результат, когда мы вызовем его в `main` функции листинга 10-20.

Сигнатура функции теперь сообщает Rust, что для некоторого времени жизни `'a` функция принимает два параметра, оба из которых являются фрагментами строк, которые живут не меньше, чем времена жизни `'a`. Сигнатура функции также сообщает Rust, что фрагмент строки, возвращаемый функцией, будет жить как минимум столько, сколько длится времена жизни `'a`. На практике это означает, что времена жизни ссылки, возвращаемой самой функцией `longest`, равно меньшему времени жизни передаваемых в неё ссылок. Именно эти отношения мы хотим, чтобы Rust использовал при анализе этого кода.

Помните, когда мы указываем параметры времени жизни в этой сигнатуре функции, мы не меняем времена жизни каких-либо переданных или возвращённых значений. Скорее, мы указываем, что проверка заимствований должна отклонять любые значения, которые не соответствуют этим ограничениям. Обратите внимание, что самой функции `longest` не нужно точно знать, как долго будут жить `x` и `y`, только то, что некоторая область может быть заменена на `'a`, которая будет удовлетворять этой сигнатуре.

При аннотировании времени жизни функций, аннотации помещаются в сигнатуру функции, а не в тело функции. Аннотации времени жизни становятся частью контракта функции, как и типы в сигнатуре. Наличие сигнатур функций, содержащих контракт времени жизни, означает, что анализ который делает компилятор Rust, может быть проще. Если есть проблема с тем, как функция аннотируется или как она вызывается, ошибки компилятора могут указывать на часть нашего кода и ограничения более точно. Если вместо этого компилятор Rust сделает больше выводов о том, какими мы предполагали отношения времени жизни, компилятор сможет указать только на использование нашего кода за много шагов от причины проблемы.

Когда мы передаём конкретные ссылки в `longest`, время жизни, которое заменено на `'a`, будет привязано к времени жизни которое является пересечением времени жизни области видимости `x` с временем жизни области видимости `y`. Другими словами, обобщённое время жизни `'a` получит конкретное время жизни: время равное меньшему из времён жизни `x` и `y`. Так как мы аннотировали возвращаемую ссылку тем же параметром времени жизни `'a`, то возвращённая ссылка также будет действительна в течение меньшего из времён жизни `x` и `y`.

Давайте посмотрим, как аннотации времени жизни ограничивают функцию `longest` передавая внутрь ссылки, которые имеют разные конкретные времена жизни. Листинг 10-23 является простым примером.

Файл: src/main.rs

```
fn main() {
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("The longest string is {}", result);
    }
}
```

Листинг 10-23: Использование функции `longest` со ссылками на значения типа `String`, которые имеют разное время жизни

В этом примере переменная `string1` действительна до конца внешней области, `string2` действует до конца внутренней области видимости и `result` ссылается на что-то, что является действительным до конца внутренней области видимости. Запустите этот код, и вы увидите что анализатор заимствований разрешает такой

код; он скомпилирует и напечатает `The longest string is long string is long.`

Далее, давайте попробуем пример, который показывает, что время жизни ссылки `result` должно быть меньшим временем жизни одного из двух аргументов. Мы переместим объявление переменной `result` наружу из внутренней области видимости, но оставим присвоение значения переменной `result` в области видимости `string2`. Затем мы переместим `println!`, который использует `result` за пределы внутренней области видимости, после того как внутренняя область видимости закончилась. Код в листинге 10-24 не будет компилироваться.

Файл: src/main.rs

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
```



Листинг 10-24: Попытка использования переменной `result` после выхода `string2` за пределы области видимости

Когда мы попытаемся скомпилировать этот код, мы получим ошибку:

```
$ cargo run
Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `string2` does not live long enough
--> src/main.rs:6:44
   |
6 |         result = longest(string1.as_str(), string2.as_str());
   |                     ^^^^^^^^^^^^^^^^^ borrowed
value does not live long enough
7 |     }
   |     - `string2` dropped here while still borrowed
8 |     println!("The longest string is {}", result);
   |                         ----- borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` due to previous error
```

Эта ошибка говорит о том, что если мы хотим использовать `result` в `println!`, переменная `string2` должна быть действительной до конца внешней области

видимости. Rust знает об этом, потому что мы аннотировали параметры функции и её возвращаемое значение одинаковым временем жизни '`a`'.

Как люди, мы можем увидеть, что `string1` живёт дольше, чем `string2` и следовательно, `result` будет содержать ссылку на `string1`. Поскольку `string1` ещё не вышла из области видимости, ссылка на `string1` будет все ещё действительной в выражении `println!`. Однако компилятор не видит, что ссылка действительная в этом случае. Мы сказали Rust, что время жизни ссылки, возвращаемой из функции `longest`, равняется меньшему из времён жизни переданных в неё ссылок. Таким образом, проверка заимствования запрещает код в листинге 10-24, как возможно имеющий недействительную ссылку.

Попробуйте поэкспериментировать с различными значениями и временами жизни передаваемыми в функцию `longest`. Перед компиляцией делайте предположения о том, пройдёт ли ваш код проверку заимствования, затем проверяйте, чтобы увидеть насколько вы были правы.

Мышление в терминах времён жизни

Правильный способ определения времён жизни зависит от того, что функция делает. Например, если мы изменим реализацию функции `longest` таким образом, чтобы она всегда возвращала свой первый аргумент вместо самого длинного среза строки, то и не придётся указывать время жизни для параметра `y`. Этот код компилируется:

Файл: src/main.rs

```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

В этом примере мы указали параметр времени жизни '`a`' для параметра `x` и возвращаемого значения, но не для параметра `y`, поскольку параметр `y` никак не соотносится с параметром `x` и возвращаемым значением.

При возврате ссылки из функции, параметр времени жизни для возвращаемого типа должен соответствовать параметру времени жизни одного из аргументов. Если возвращаемая ссылка *не ссылается* на один из параметров то, она должна ссылаться на значение, созданное внутри функции, что приведёт к недействительной ссылке, поскольку значение, на которое она ссылается, выйдет

из области видимости в конце функции. Посмотрите на пример реализации функции `longest`, который не компилируется:

Файл: src/main.rs

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```



Здесь, несмотря на то, что мы указали параметр времени жизни `'a` для возвращаемого типа, реализация не будет скомпилирована, потому что возвращаемое значение времени жизни совсем не связано с временем жизни параметров. Получаемое сообщение об ошибке:

```
$ cargo run
Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0515]: cannot return reference to local variable `result`
--> src/main.rs:11:5
   |
11 |     result.as_str()
   |     ^^^^^^^^^^^^^^ returns a reference to data owned by the current
function

For more information about this error, try `rustc --explain E0515`.
error: could not compile `chapter10` due to previous error
```

Проблема заключается в том, что `result` выходит за область видимости и очищается в конце функции `longest`. Мы также пытаемся вернуть ссылку на `result` из функции. Мы не можем указать параметры времени жизни, которые могли бы изменить недействительную ссылку, а Rust не позволит нам создать недействительную ссылку. В этом случае лучшим решением будет вернуть данные во владение вызывающей функции, а не ссылку: так вызывающая функция понесёт ответственность за очистку полученного в её распоряжение значения.

В конечном итоге, синтаксис времён жизни реализует связывание времён жизни различных аргументов функций и их возвращаемых значений. Описывая времена жизни, мы даём Rust достаточно информации, чтобы разрешить безопасные операции с памятью и запретить операции, которые могли бы создать недействительные ссылки или иным способом нарушить безопасность памяти.

Определение времён жизни при объявлении структур

До сих пор мы объявляли структуры, которые содержали не ссылочные типы данных. Структуры могут содержать и ссылочные типы данных, но при этом необходимо добавить аннотацию времени жизни для каждой ссылки в определение структуры. Листинг 10-25 описывает структуру `ImportantExcerpt`, содержащую срез строковых данных:

Файл: src/main.rs

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().expect("Could not find a
'.');");
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}
```

Листинг 10-25. Структура, которая содержит ссылку, поэтому её объявление требует аннотации времени жизни

У структуры имеется одно поле `part`, хранящее ссылку на срез строки. Как в случае с обобщёнными типами данных, объявляется имя обобщённого параметра времени жизни внутри угловых скобок после имени структуры, чтобы иметь возможность использовать его внутри тела определения структуры. Данная аннотация означает, что экземпляр `ImportantExcerpt` не может пережить ссылку, которую он содержит в своём поле `part`.

Функция `main` здесь создаёт экземпляр структуры `ImportantExcerpt`, который содержит ссылку на первое предложение типа `String` принадлежащее переменной `novel`. Данные в `novel` существуют до создания экземпляра `ImportantExcerpt`. Кроме того, `novel` не выходит из области видимости до тех пор, пока `ImportantExcerpt` выходит за область видимости, поэтому ссылка внутри экземпляра `ImportantExcerpt` является действительной.

Правила неявного выводения времени жизни

Вы изучили, что у каждой ссылки есть время жизни и что нужно указывать

параметры времени жизни для функций или структур, которые используют ссылки. Однако в Главе 4 у нас была функция в листинге 4-9, которая снова показана в листинге 10-26, где код собран без аннотаций времени жизни.

Файл: 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[..]
}
```

Листинг 10-26: Функция, которую мы определили в листинге 4-9 компилируется без описания времени жизни параметров, несмотря на то, что входной и возвращаемый тип параметров являются ссылками

Причина, по которой этот код компилируется — историческая. В первых (pre-1.0) версиях Rust этот код не скомпилировался бы, поскольку каждой ссылке нужно было явно назначать время жизни. В те времена, сигнатура функции была бы написана примерно так:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

После написания большого количества кода на Rust разработчики языка обнаружили, что в определённых ситуациях программисты описывают одни и те же аннотации времён жизни снова и снова. Эти ситуации были предсказуемы и следовали некоторым детерминированным шаблонным моделям. Команда Rust решила запрограммировать эти шаблоны в код компилятора Rust, чтобы анализатор заимствований мог вывести времена жизни в таких ситуациях без необходимости явного указания аннотаций программистами.

Мы упоминаем этот фрагмент истории Rust, потому что возможно, что в будущем может появиться и будет добавлено больше шаблонов для автоматического выведения времён жизни, которые могут быть добавлены в компилятор и понадобится меньшее количество аннотаций.

Шаблоны анализа ссылок, запрограммированные в анализаторе Rust, называются

правилами неявного выводения времени жизни. Это не правила, которым должны следовать программисты; а набор частных случаев, которые рассмотрит компилятор и если ваш код попадает в эти случаи, вам не нужно будет указывать время жизни явно.

Правила выводения не предоставляют полного вывода. Если Rust детерминировано применяет правила, но все ещё остаётся неясность относительно времён жизни у ссылок, то компилятор не может догадаться, какими должны быть времена жизни оставшихся ссылок. В этом случае вместо угадывания компилятор выдаст ошибку, которую вы можете устранить, добавив аннотации, указывающие на то, как ссылки относятся друг с другом.

Времена жизни параметров функции или метода называются *временем жизни ввода*, а времена жизни возвращаемых значений называются *временем жизни вывода*.

Компилятор использует три правила, чтобы выяснить времена жизни имеющиеся у ссылок, когда нет явных аннотаций. Первое правило относится ко времени жизни ввода, второе и третье правила применяются ко временам жизни вывода. Если компилятор доходит до конца проверки трёх правил и всё ещё есть ссылки для которых он не может выяснить время жизни, то компилятор остановится с ошибкой. Эти правила применяются к объявлениям `fn`, а также `impl` блоков.

Первое правило говорит, что каждый параметр являющийся ссылкой, получает свой собственный параметр времени жизни. Другими словами, функция с одним параметром получит один параметр времени жизни: `fn foo<'a>(x: &'a i32)`; функция с двумя аргументами получит два различных параметра времени жизни: `fn foo<'a, 'b>(x: &'a i32, y: &'b i32)`, и так далее.

Второе правило говорит, что если существует точно один входной параметр времени жизни, то его время жизни назначается всем выходным параметрам: `fn foo<'a>(x: &'a i32) -> &'a i32`.

Третье правило о том, что если есть множество входных параметров времени жизни, но один из них является ссылкой `&self` или `&mut self` при условии что эта функция является методом структуры или перечисления, то время жизни `self` назначается временем жизни всем выходным параметрам метода. Это третье правило делает методы намного приятнее для чтения и записи, потому что требуется меньше символов.

Давайте представим, что мы компилятор и применим эти правила, чтобы вывести

времена жизни ссылок в сигнатуре функции `first_word` листинга 10-26. Сигнатура этой функции начинается без объявления времён жизни ссылок:

```
fn first_word(s: &str) -> &str {
```

Теперь мы (в качестве компилятора) применим первое правило, утверждающее, что каждый параметр функции получает своё собственное время жизни. Как обычно, мы собираемся назвать его `'a` и сигнатура выглядит так:

```
fn first_word<'a>(s: &'a str) -> &str {
```

Далее применяем второе правило, поскольку в функции указан только один входной параметр времени жизни. Второе правило гласит, что время жизни единственного входного параметра назначается выходным параметрам, как показано в сигнатуре:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

Теперь все ссылки в этой функции имеют параметры времени жизни и компилятор может продолжить свой анализ без необходимости получения аннотаций времён жизни у сигнатуры этой функции.

Давайте рассмотрим ещё один пример: заголовок функции `longest`, в котором не было параметров времени жизни в начале работы с листингом 10-21:

```
fn longest(x: &str, y: &str) -> &str {
```

Применим первое правило: каждому параметру назначается собственное время жизни. На этот раз у функции есть два параметра, поэтому есть два времени жизни:

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

Видно, что второе правило не применимо, потому что в сигнатуре указано больше одного входного параметра. Третье правило также не применимо, так как `longest` — функция, а не метод, следовательно, в ней нет параметра `self`. Итак, мы прошли все три правила, но так и не смогли вычислить время жизни выходного параметра. Вот почему мы получили ошибку при попытке скомпилировать код листинга 10-21: компилятор работал по правилам неявного выведения времён жизни, но не мог выяснить все времена жизни ссылок в сигнатуре.

Так как третье правило применяется только к методам, мы рассмотрим времена

жизни в этом контексте и поймём почему в сигнатурах методов нам часто не нужно аннотировать времена жизни.

Аннотация времён жизни в определении методов

Когда мы реализуем методы для структур с временами жизни, синтаксис аннотаций снова схож с аннотациями обобщённых типов данных, как было показано в листинге 10-11. Место объявления времён жизни зависит от того, с чем оно связано — с полем структуры или с аргументами методов и возвращаемыми значениями.

Имена переменных времени жизни для полей структур всегда описываются после ключевого слова `impl` и затем используются после имени структуры, поскольку эти имена жизни являются частью типа структуры.

В сигнатурах методов внутри блока `impl` ссылки могут быть привязаны ко времени жизни ссылок в полях структуры или могут быть независимыми. Вдобавок, правила неявного выводения времён жизни часто делают так, что аннотации переменных времён жизни являются необязательными в сигнатурах методов. Рассмотрим несколько примеров использования структуры с названием `ImportantExcerpt`, которую мы определили в листинге 10-25.

Сначала, воспользуемся методом с именем `level` где входной параметр является ссылкой на `self`, а возвращаемое значение `i32`, не является ссылкой:

```
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}
```

Объявление параметра времени жизни находится после `impl` и его использование после типа структуры является обязательным, но нам не нужно аннотировать время жизни ссылки у `self`, благодаря первому правилу неявного выводения времён жизни.

Пример применения третьего правила неявного выводения времён жизни:

```
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) -> &str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

В этом методе имеется два входных параметра, поэтому Rust применяет первое правило и назначает обоим параметрам `&self` и `announcement` собственные времена жизни. Далее, поскольку один из параметров является `&self`, то возвращаемое значение получает время жизни переменной `&self` и все времена жизни выведены.

Статическое время жизни

Существует ещё одно особенное время жизни, которое мы должны обсудить это `'static`, которое означает, что данная ссылка *может* жить всю продолжительность работы программы. Все строковые литералы по умолчанию имеют время жизни `'static`, но мы можем указать его явным образом:

```
let s: &'static str = "I have a static lifetime.;"
```

Содержание этой строки сохраняется внутри бинарного файла вашей программы и всегда доступно для использования. Следовательно, время жизни всех строковых литералов равно `'static`.

Сообщения компилятора об ошибках в качестве решения проблемы могут предлагать вам использовать `'static`. Но прежде чем указывать времена жизни для ссылки как `'static`, подумайте, должна ли данная ссылка всегда быть доступна во время всей работы программы. Большинство таких проблем появляются при попытках создания недействительных ссылок или несовпадения времён жизни. В таких случаях, она может быть решена без указания статического времени жизни `'static`.

Обобщённые типы параметров, ограничения типажей и время жизни вместе

Давайте кратко рассмотрим синтаксис задания параметров обобщённых типов, ограничений типажа и времён жизни 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!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Это функция `longest` из листинга 10.22, которая возвращает наибольший из двух фрагментов строки. Но теперь у неё есть дополнительный параметр с именем `ann` обобщённого типа `T`, который может быть заполнен любым типом, реализующим типаж `Display`, как указано в предложении `where`. Этот дополнительный параметр будет напечатан с использованием `{}`, поэтому привязка типажа `Display` необходима. Поскольку время жизни является обобщённым типом, то объявления параметра времени жизни `'a` и параметра обобщённого типа `T` помещаются в один список внутри угловых скобок после имени функции.

Итоги

В этой главе мы рассмотрели много важного материала! Теперь вы знакомы с параметрами обобщённого типа, типажами и ограничениями типажа, обобщёнными параметрами времени жизни. Вы готовы писать программы и не дублировать создаваемый вами код во множестве других случаев. Параметры обобщённого типа позволяют использовать код для различных типов данных. Типажи и ограничения типажа помогают гарантировать, что несмотря на обобщённые типы, они будут иметь поведение соответствующее потребностям кода. Вы изучили, как использование аннотаций времени жизни гарантирует, что код не будет иметь недействительных ссылок. Весь этот анализ происходит в

момент компиляции и не влияет на производительность программы!

Верите или нет, но в рамках этой темы всё есть ещё чему поучиться: в Главе 17 обсуждаются типажи-объекты, что является ещё одним способом использовать типажи. Существуют также более сложные сценарии с аннотациями времени жизни, которые вам понадобятся только в очень сложных случаях; для этого вам следует прочитать [Rust Reference](#). Далее вы узнаете, как писать тесты на Rust, чтобы убедиться, что ваш код работает так, как должен.

Написание автоматизированных тестов

В своём эссе 1972 года “The Humble Programmer,” Edsger W. Dijkstra сказал, что «Тестирование программы может быть очень эффективным способом показать наличие ошибок, но это безнадёжно неадекватно для показа их отсутствия». Это не значит, что мы не должны пытаться тестировать столько, сколько мы можем!

Правильность в наших программах - это степень, в которой наш код выполняет то для чего он предназначен. Rust разработан с высокой степенью заботы о правильности программ, но правильность сложна и её не легко доказать. Система типов Rust несёт огромную часть этого бремени, но система типов не может обнаружить каждый вид некорректности. Таким образом, Rust включает в себя поддержку написания автоматизированных программных тестов внутри языка.

В качестве примера, скажем, мы пишем функцию с именем `add_two` которая добавляет 2 к любому числу передаваемому в неё. Сигнатура этой функции принимает параметром целое число и возвращает в результате целое число. Когда мы реализуем и компилируем такую функцию, Rust выполняет все проверки типов и заимствований про которые вы уже узнали, чтобы убедиться, например, что мы не передаём значение `String` или неверную ссылку в эту функцию. Но Rust *не может* проверить, что данная функция будет делать именно то, что мы намереваемся получить, что она возвращает входной параметр плюс 2, а не скажем, параметр плюс 10 или параметр минус 50! Это то, где тесты приходят на помощь.

Мы можем написать тесты, которые утверждают, например, что когда мы передаём `3` в функцию `add_two`, возвращаемое значение будет `5`. Мы можем запускать эти тесты всякий раз, когда мы вносим изменения в наш код, чтобы убедиться, что любое существующее правильное поведение не изменилось.

Тестирование - сложный навык: мы не сможем охватить все детали написания хороших тестов в одной главе, но мы обсудим основные подходы к тестированию в Rust. Мы поговорим об аннотациях и макросах, доступных вам для написания тестов, о поведении по умолчанию и параметрах, предусмотренных для запуска тестов, а также о том, как организовать тесты в модульные тесты и интеграционные тесты.

Как писать тесты

Тесты - это функции Rust, которые проверяют, что не тестовый код работает ожидаемым образом. Содержимое тестовых функций обычно выполняет следующие три действия:

1. Установка любых необходимых данных или состояния.
2. Запуск кода, который вы хотите проверить.
3. Утверждение, что результаты являются теми, которые вы ожидаете.

Давайте рассмотрим функции, предоставляемые в Rust специально для написания тестов, которые выполняют все эти действия, включая атрибут `test`, несколько макросов и атрибут `should_panic`.

Структура тестирующей функции

В простейшем случае в Rust тест - это функция, аннотированная атрибутом `test`. Атрибуты представляют собой метаданные о фрагментах кода Rust; один из примеров атрибут `derive`, который мы использовали со структурами в главе 5. Чтобы изменить функцию в тестирующую функцию добавьте `#[test]` в строку перед `fn`. Когда вы запускаете тесты командой `cargo test`, Rust создаёт бинарный модуль выполняющий функции аннотированные атрибутом `test` и сообщающий о том, прошла успешно или не прошла каждая тестирующая функция.

Когда мы создаём новый проект библиотеки с помощью Cargo, то в нём автоматически генерируется тестовый модуль с тест функцией для нас. Этот модуль поможет вам начать написание ваших тестов, так что вам не нужно искать точную структуру и синтаксис тестовых функций каждый раз, когда вы начинаете новый проект. Вы можете добавить как большее количество дополнительных тестовых функций так и несколько тестовых модулей!

Мы исследуем некоторые аспекты работы тестов, экспериментируя с шаблонным тестом сгенерированным для нас, без реального тестирования любого кода. Затем мы напишем некоторые реальные тесты, которые вызывают некоторый написанный код и убедимся в его правильном поведении.

Давайте создадим новый проект библиотеки под названием `adder`:

```
$ cargo new adder --lib
    Created library `adder` project
$ cd adder
```

Содержимое файла `src/lib.rs` вашей библиотеки `adder` должно выглядеть как в листинге 11-1.

Файл: `src/lib.rs`

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

Листинг 11-1: Тестовый модуль и функция, сгенерированные автоматически с помощью `cargo new`

Сейчас проигнорируем первые две строчки кода и сосредоточимся на функции, чтобы увидеть как она работает. Обратите внимание на синтаксис аннотации `#[test]` перед ключевым словом `fn`. Этот атрибут сообщает компилятору, что это является заголовком тестирующей функции, так что функционал запускающий тесты на выполнение теперь знает, что это тестирующая функция. Также в составе модуля тестов `tests` могут быть вспомогательные функции, помогающие настроить и выполнить общие подготовительные операции, поэтому специальная аннотация важна для указания объявления функций тестами с использованием атрибута `#[test]`.

Тело функции использует макрос `assert_eq!`, чтобы утверждать, что $2 + 2$ равно 4. Это утверждение служит примером формата для типичного теста. Давайте запустим, чтобы увидеть, что этот тест проходит.

Команда `cargo test` выполнит все тесты в выбранном проекте и сообщит о результатах как в листинге 11-2:

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.57s
Running unitests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Листинг 11-2: Вывод информации о работе автоматически сгенерированных тестов

Cargo скомпилировал и выполнил тест. После строк `Compiling`, `Finished` и `Running` мы видим строку `running 1 test`. Следующая строка показывает имя созданной тест функции с названием `it_works` и результат её выполнения - `ok`. Далее вы видите обобщённую информацию о работе всех тестов. Текст `test result: ok.` означает, что все тесты пройдены успешно и часть вывода `1 passed; 0 failed` сообщает общее количество тестов, которые прошли или были ошибочными.

Поскольку у нас нет тестов, которые мы пометили как игнорируемые, в сводке отображается `0 ignored`. Мы также не отфильтровывали тесты для выполнения, поэтому конец сводки пишет `0 filtered out`. Мы поговорим про игнорирование и фильтрацию тестов в следующем разделе "[Контролирование хода выполнения тестов](#)".

Статистика `0 measured` предназначена для тестов производительности. На момент написания этой статьи такие тесты доступны только в ночной сборке Rust. Посмотрите [документацию о тестах производительности](#), чтобы узнать больше.

Следующая часть вывода тестов начинается с `Doc-tests adder` - это информация о тестах в документации. У нас пока нет тестов документации, но Rust может компилировать любые примеры кода, которые находятся в API документации. Такая возможность помогает поддерживать документацию и код в синхронизированном состоянии. Мы поговорим о написании тестов документации в секции "[Комментарии документации как тесты](#)" Главы 14. Пока просто проигнорируем

часть **Doc-tests** вывода.

Давайте поменяем название нашего теста и посмотрим что же измениться в строке вывода. Назовём нашу функцию **it_works** другим именем - **exploration**:

Файл: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }
}
```

Снова выполним команду **cargo test**. Вывод показывает наименование нашей тест функции - **exploration** вместо **it_works**:

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.59s
Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::exploration ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Добавим ещё один тест, но в этот раз специально сделаем так, чтобы этот новый тест не отработал. Тест терпит неудачу, когда что-то паникует в тестируемой функции. Каждый тест запускается в новом потоке и когда главный поток видит, что тестовый поток упал, то помечает тест как завершившийся аварийно. Мы говорили о простейшем способе вызвать панику в главе 9, используя для этого известный макрос **panic!**. Введём код тест функции **another**, как в файле *src/lib.rs* из листинга 11-3.

Файл: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }

    #[test]
    fn another() {
        panic!("Make this test fail");
    }
}
```



Листинг 11-3: Добавление второго теста, который завершится ошибкой, потому что мы вызываем `panic!` макрос

Запустим команду `cargo test`. Вывод результатов показан в листинге 11-4, который сообщает, что тест `exploration` пройден, а `another` нет:

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.72s
Running unitests (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test tests::another ... FAILED
test tests::exploration ... ok

failures:

---- tests::another stdout ----
thread 'main' panicked at 'Make this test fail', src/lib.rs:10:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
tests::another

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Листинг 11-4. Результаты теста, когда один тест пройден, а другой нет

Вместо `ok`, строка `test tests::another` сообщает `FAILED`. У нас есть два новых

раздела между результатами и итогами. Первый раздел показывает детальную причину ошибки каждого теста. В данном случае тест `another` не сработал, потому что `panicked at 'Make this test fail'`, произошло в строке 10 файла `src/lib.rs`. В следующем разделе перечисляют имена всех не пройденных тестов, что удобно, когда тестов очень много и есть много деталей про аварийное завершение. Мы можем использовать имя не пройденного теста для его дальнейшей отладки; мы больше поговорим о способах запуска тестов в разделе "[Контролирование хода выполнения тестов](#)".

Итоговая строка отображается в конце: общий результат нашего тестирования `FAILED`. У нас один тест пройден и один тест завершён аварийно.

Теперь, когда вы увидели, как выглядят результаты теста при разных сценариях, давайте рассмотрим другие макросы полезные в тестах, кроме `panic!`.

Проверка результатов с помощью макроса `assert!`

Макрос `assert!` доступен из стандартной библиотеки и является удобным, когда вы хотите проверить что некоторое условие в teste вычисляется в значение `true`.

Внутри макроса `assert!` переданный аргумент вычисляется в логическое значение. Если оно `true`, то `assert!` в teste ничего не делает и он считается пройденным. Если же значение вычисляется в `false`, то макрос `assert!` вызывает макрос `panic!`, что делает тест аварийным. Использование макроса `assert!` помогает проверить, что код функционирует как ожидалось.

В главе 5, листинга 5-15, мы использовали структуру `Rectangle` и метод `can_hold`, который повторён в листинге 11-5. Давайте поместим этот код в файл `src/lib.rs` и напишем несколько тестов для него используя `assert!` макрос.

Файл: `src/lib.rs`

```

#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}

```

Листинг 11-5: Использование структуры `Rectangle` и её метода `can_hold` из главы 5

Метод `can_hold` возвращает логическое значение, что означает, что она является идеальным вариантом использования в макросе `assert!`. В листинге 11-6 мы пишем тест, который выполняет метод `can_hold` путём создания экземпляра `Rectangle` шириной 8 и высотой 7 и убеждаемся, что он может содержать другой экземпляр `Rectangle` имеющий ширину 5 и высоту 1.

Файл: `src/lib.rs`

```

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

    #[test]
    fn larger_can_hold_smaller() {
        let larger = Rectangle {
            width: 8,
            height: 7,
        };
        let smaller = Rectangle {
            width: 5,
            height: 1,
        };

        assert!(larger.can_hold(&smaller));
    }
}

```

Листинг 11-6: Тесты для метода `can_hold`, который проверяет что больший прямоугольник действительно может содержать меньший

Также, в модуле `tests` обратите внимание на новую добавленную строку `use super::*;

super::*`. Модуль `tests` является обычным и подчиняется тем же правилам

видимости, которые мы обсуждали в главе 7 "Пути для ссылки на элементы внутри дерева модуля". Так как этот модуль `tests` является внутренним, нужно подключить тестируемый код из внешнего модуля в область видимости внутреннего модуля с тестами. Для этого используется глобальное подключение, так что все что определено во внешнем модуле становится доступным внутри `tests` модуля.

Мы назвали наш тест `larger_can_hold_smaller` и создали два нужных экземпляра `Rectangle`. Затем вызвали макрос `assert!` и передали результат вызова `larger.can_hold(&smaller)` в него. Это выражение должно возвращать `true`, поэтому наш тест должен пройти. Давайте выясним!

```
$ cargo test
Compiling rectangle v0.1.0 (file:///projects/rectangle)
Finished test [unoptimized + debuginfo] target(s) in 0.66s
Running unitests (target/debug/deps/rectangle-6584c4561e48942e)

running 1 test
test tests::larger_can_hold_smaller ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests rectangle

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Тест проходит. Теперь добавим другой тест, в этот раз мы попытаемся убедиться, что меньший прямоугольник не может содержать больший прямоугольник:

Файл: `src/lib.rs`

```

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

    #[test]
    fn larger_can_hold_smaller() {
        // --snip--
    }

    #[test]
    fn smaller_cannot_hold_larger() {
        let larger = Rectangle {
            width: 8,
            height: 7,
        };
        let smaller = Rectangle {
            width: 5,
            height: 1,
        };

        assert!(!smaller.can_hold(&larger));
    }
}

```

Поскольку правильный результат функции `can_hold` в этом случае `false`, то мы должны инвертировать этот результат, прежде чем передадим его в `assert!` макро. Как результат, наш тест пройдёт, если `can_hold` вернёт `false`:

```

$ cargo test
Compiling rectangle v0.1.0 (file:///projects/rectangle)
Finished test [unoptimized + debuginfo] target(s) in 0.66s
Running unitests (target/debug/deps/rectangle-6584c4561e48942e)

running 2 tests
test tests::larger_can_hold_smaller ... ok
test tests::smaller_cannot_hold_larger ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests rectangle

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

```

Два теста работают. Теперь проверим, как отреагируют тесты, если мы добавим ошибку в код. Давайте изменим реализацию метода `can_hold` заменив одно из логических выражений знак сравнения с "больше чем" на противоположный "меньше чем" при сравнении ширины:

```
// --snip--
impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width < other.width && self.height > other.height
    }
}
```



Запуск тестов теперь производит следующее:

```
$ cargo test
Compiling rectangle v0.1.0 (file:///projects/rectangle)
Finished test [unoptimized + debuginfo] target(s) in 0.66s
Running unitests (target/debug/deps/rectangle-6584c4561e48942e)

running 2 tests
test tests::larger_can_hold_smaller ... FAILED
test tests::smaller_cannot_hold_larger ... ok

failures:

---- tests::larger_can_hold_smaller stdout ----
thread 'main' panicked at 'assertion failed: larger.can_hold(&smaller)', 
src/lib.rs:28:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
    tests::larger_can_hold_smaller

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Наши тесты нашли ошибку! Так как в teste `larger.width` равно 8 и `smaller.width` равно 5 сравнение ширины в методе `can_hold` возвращает результат `false`, поскольку число 8 не меньше чем 5.

Проверка на равенство с помощью макросов `assert_eq!` и `assert_ne!`

Общим способом проверки функциональности является использование сравнения результата тестируемого кода и ожидаемого значения, чтобы убедиться в их равенстве. Для этого можно использовать макрос `assert!`, передавая ему выражение с использованием оператора `==`. Важно также знать, что кроме этого стандартная библиотека предлагает пару макросов `assert_eq!` и `assert_ne!`, чтобы сделать тестирование более удобным. Эти макросы сравнивают два аргумента на равенство или неравенство соответственно. Макросы также печатают два значения входных параметров, если тест завершился ошибкой, что позволяет легче увидеть *почему* тест ошибочен. Противоположно этому, макрос `assert!` может только отобразить, что он вычислил значение `false` для выражения `==`, но не значения, которые привели к результату `false`.

В листинге 11-7, мы напишем функцию `add_two`, которая прибавляет к входному параметру `2` и возвращает значение. Затем, протестируем эту функцию с помощью макроса `assert_eq!`:

Файл: 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));
    }
}
```

Листинг 11-7: Тестирование функции `add_two` с помощью макроса `assert_eq!`

Проверим, что тесты проходят!

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.58s
Running unitests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Первый аргумент, который мы передаём в макрос `assert_eq!` число `4` чей результат вызова равен `add_two(2)`. Стока для этого теста - `test tests::it_adds_two ... ok`, а текст `ok` означает, что наш тест пройден!

Давайте введём ошибку в код, чтобы увидеть, как она выглядит, когда тест, который использует `assert_eq!` завершается ошибкой. Измените реализацию функции `add_two`, чтобы добавлять `3`:

```
pub fn add_two(a: i32) -> i32 {
    a + 3
}
```



Попробуем выполнить данный тест ещё раз:

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.61s
Running unitests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_adds_two ... FAILED

failures:

---- tests::it_adds_two stdout ----
thread 'main' panicked at 'assertion failed: `!(left == right)`
  left: `4`,
  right: `5`, src/lib.rs:11:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
    tests::it_adds_two

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Наш тест нашёл ошибку! Тест `it_adds_two` не выполнился, отображается сообщение `assertion failed: `!(left == right)`` и показывает, что `left` было `4`, а `right` было `5`. Это сообщение полезно и помогает начать отладку: это означает `left` аргумент `assert_eq!` имел значение `4`, но `right` аргумент для вызова `add_two(2)` был со значением `5`.

Обратите внимание, что в некоторых языках (таких как Java) в библиотеках кода для тестирования принято именовать входные параметры проверочных функций как "ожидаемое" (`expected`) и "фактическое" (`actual`). В Rust приняты следующие обозначения `left` и `right` соответственно, а порядок в котором определяются ожидаемое значение и производимое тестируемым кодом значение не имеют значения. Мы могли бы написать выражение в teste как `assert_eq!(add_two(2), 4)`, что приведёт к отображаемому сообщению об ошибке `assertion failed: `!(left == right)``, слева `left` было бы `5`, а справа `right` было бы `4`.

Макрос `assert_ne!` сработает успешно, если входные параметры не равны друг другу и завершится с ошибкой, если значения равны. Этот макрос наиболее полезен в тех случаях, когда мы не знаем заранее, каким значение будет, но знаем точно, каким оно не может быть. К примеру, если тестируется функция, которая

гарантирует изменять входные данные определенным образом, но способ изменения входного параметра зависит от дня недели, в который запускаются тесты, что лучший способ проверить правильность работы такой функции - это сравнить и убедиться, что выходное значение функции не должно быть равным входному значению.

С своей работе макросы `assert_eq!` и `assert_ne!` неявным образом используют операторы `==` и `!=` соответственно. Когда проверка не срабатывает, макросы печатают значения аргументов с помощью отладочного форматирования и это означает, что значения сравниваемых аргументов должны реализовать типажи `PartialEq` и `Debug`. Все примитивные и большая часть типов стандартной библиотеки Rust реализуют эти типажи. Для структур и перечислений, которые вы реализуете сами будет необходимо реализовать типаж `PartialEq` для сравнения значений на равенство или неравенство. Для печати отладочной информации в виде сообщений в строку вывода консоли необходимо реализовать типаж `Debug`. Так как оба типажа являются выводимыми типажами, как упоминалось в листинге 5-12 главы 5, то эти типажи можно реализовать добавив аннотацию `#[derive(PartialEq, Debug)]` к определению структуры или перечисления. Смотрите больше деталей в Appendix C "Выводимые типажи" про эти и другие выводимые типажи.

Создание сообщений об ошибках

Также можно добавить пользовательское сообщение для печати в сообщении об ошибке теста как дополнительный аргумент макросов `assert!`, `assert_eq!`, and `assert_ne!`. Любые аргументы, указанные после одного обязательного аргумента в `assert!` или после двух обязательных аргументов в `assert_eq!` и `assert_ne!` передаются в макрос `format!` (он обсуждается в разделе "Конкатенация с помощью оператора `+` или макроса `format!`" главы 8), так что вы можете передать форматированную строку, которая содержит символы `{}` для заполнителей и значения, заменяющие эти заполнители. Пользовательские сообщения полезны для пояснения, что означает утверждение, когда тест не пройден. У вас будет лучшее представление о том, какая проблема в коде.

Например, есть функция, которая приветствует человека по имени и мы хотим протестировать эту функцию. Мы хотим чтобы передаваемое ей имя выводилось в консоль:

Файл: `src/lib.rs`

```
pub fn greeting(name: &str) -> String {
    format!("Hello {}!", name)
}

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

    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(result.contains("Carol"));
    }
}
```

Требования к этой программе ещё не были согласованы и мы вполне уверены, что текст `Hello` в начале приветствия ещё изменится. Мы решили, что не хотим обновлять тест при изменении требований, поэтому вместо проверки на точное равенство со значением возвращённым из `greeting`, мы просто будем проверять, что вывод содержит текст из входного параметра.

Давайте внесём ошибку в этот код, изменив `greeting` так, чтобы оно не включало `name` и увидим, как выглядит сбой этого теста:

```
pub fn greeting(name: &str) -> String {
    String::from("Hello!")
}
```



Запуск этого теста выводит следующее:

```
$ cargo test
Compiling greeter v0.1.0 (file:///projects/greeter)
Finished test [unoptimized + debuginfo] target(s) in 0.91s
Running unitests (target/debug/deps/greeter-170b942eb5bf5e3a)

running 1 test
test tests::greeting_contains_name ... FAILED

failures:

---- tests::greeting_contains_name stdout ----
thread 'main' panicked at 'assertion failed: result.contains(\"Carol\"}',
src/lib.rs:12:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
tests::greeting_contains_name

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Сообщение содержит лишь информацию о том что сравнение не было успешным и в какой строке это произошло. В данном случае, более полезный текст сообщения был бы, если бы также выводилось значение из функции `greeting`. Изменим тестирующую функцию так, чтобы выводились пользовательское сообщение форматированное строкой с заменителем и фактическими данными из кода `greeting` :

```
#[test]
fn greeting_contains_name() {
    let result = greeting("Carol");
    assert!(
        result.contains("Carol"),
        "Greeting did not contain name, value was `{}`",
        result
    );
}
```

После того, как выполним тест ещё раз мы получим подробное сообщение об ошибке:

```
$ cargo test
Compiling greeter v0.1.0 (file:///projects/greeter)
Finished test [unoptimized + debuginfo] target(s) in 0.93s
Running unitests (target/debug/deps/greeter-170b942eb5bf5e3a)

running 1 test
test tests::greeting_contains_name ... FAILED

failures:

---- tests::greeting_contains_name stdout ----
thread 'main' panicked at 'Greeting did not contain name, value was
`Hello!`, src/lib.rs:12:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
    tests::greeting_contains_name

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Мы можем увидеть значение, которое мы на самом деле получили в тестовом выводе, что поможет нам отлаживать произошедшее, а не то, что мы ожидали.

Проверка с помощью макроса `should_panic`

В дополнение к проверке того, что наш код возвращает правильные, ожидаемые значения, важным также является проверить, что наш код обрабатывает ошибки, которые мы ожидаем. Например, рассмотрим тип `Guess` который мы создали в главе 9, листинга 9-10. Другой код, который использует `Guess` зависит от гарантии того, что `Guess` экземпляры будут содержать значения только от 1 до 100. Мы можем написать тест, который гарантирует, что попытка создать экземпляр `Guess` со значением вне этого диапазона вызывает панику.

Реализуем это с помощью другого атрибута тест функции `#[should_panic]`. Этот атрибут сообщает системе тестирования, что тест проходит, когда метод генерирует ошибку. Если ошибка не генерируется - тест считается не пройденным.

Листинг 11-8 показывает тест, который проверяет, что условия ошибки `Guess::new` произойдут, когда мы их ожидаем их.

Файл: src/lib.rs

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess { value }
    }
}

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

    #[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Листинг 11-8: Проверка того, что условие вызовет макрос `panic!`

Атрибут `#[should_panic]` следует после `#[test]` и до объявления текстовой функции. Посмотрим на вывод результата, когда тест проходит:

```
$ cargo test
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished test [unoptimized + debuginfo] target(s) in 0.58s
    Running unitests (target/debug/deps/guessing_game-57d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests guessing_game

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Выглядит хорошо! Теперь давайте внесём ошибку в наш код, убрав условие о том, что функция `new` будет паниковать если значение больше 100:

```
// --snip--
impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess { value }
    }
}
```



Когда мы запустим тест в листинге 11-8, он потерпит неудачу:

```
$ cargo test
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Finished test [unoptimized + debuginfo] target(s) in 0.62s
Running unitests (target/debug/deps/guessing_game-57d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... FAILED

failures:

---- tests::greater_than_100 stdout ----
note: test did not panic as expected

failures:
tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Мы получаем не очень полезное сообщение в этом случае, но когда мы смотрим на тестирующую функцию, мы видим, что она `#[should_panic]`. Аварийное выполнение, которое мы получили означает, что код в тестирующей функции не вызвал паники.

Тесты, которые используют `should_panic` могут быть неточными, потому что они только указывают, что код вызвал панику. Тест с атрибутом `should_panic` пройдёт, даже если тест паникует по причине, отличной от той, которую мы ожидали. Чтобы сделать тесты с `should_panic` более точными, мы можем добавить необязательный параметр `expected` для атрибута `should_panic`. Такая детализация теста позволит удостовериться, что сообщение об ошибке содержит предоставленный текст. Например, рассмотрим модифицированный код для `Guess` в листинге 11-9, где `new` функция паникует с различными сообщениями в зависимости от того, является ли значение слишком маленьким или слишком большим.

Файл: src/lib.rs

```
// --snip--  
impl Guess {  
    pub fn new(value: i32) -> Guess {  
        if value < 1 {  
            panic!(  
                "Guess value must be greater than or equal to 1, got {}.",  
                value  
            );  
        } else if value > 100 {  
            panic!(  
                "Guess value must be less than or equal to 100, got {}.",  
                value  
            );  
        }  
  
        Guess { value }  
    }  
}  
  
#[cfg(test)]  
mod tests {  
    use super::*;

    #[test]
    #[should_panic(expected = "Guess value must be less than or equal to
100")]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Листинг 11-9: Проверка того, что условие вызовет макрос `panic!` с сообщением

Этот тест пройдёт, потому что значение, которое мы поместили для `should_panic` в параметр атрибута `expected` является подстрокой сообщения, с которым функция `Guess::new` вызывает панику. Мы могли бы указать полное, ожидаемое сообщение для паники, в этом случае это будет `Guess value must be less than or equal to 100, got 200`. То что вы выберите для указания как ожидаемого параметра у `should_panic` зависит от того, какая часть сообщения о панике уникальна или динамична, насколько вы хотите, чтобы ваш тест был точным. В этом случае достаточно подстrokes из сообщения паники, чтобы гарантировать выполнение кода в тестовой функции `else if value > 100`.

Чтобы увидеть, что происходит, когда тест `should_panic` неуспешно завершается с сообщением `expected`, давайте снова внесём ошибку в наш код, поменяв местами `if value < 1` и `else if value > 100` блоки:

```

if value < 1 {
    panic!(
        "Guess value must be less than or equal to 100, got {}.",
        value
    );
} else if value > 100 {
    panic!(
        "Guess value must be greater than or equal to 1, got {}.",
        value
    );
}

```



На этот раз, когда мы выполним `should_panic` тест, он потерпит неудачу:

```

$ cargo test
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Finished test [unoptimized + debuginfo] target(s) in 0.66s
Running unitests (target/debug/deps/guessing_game-57d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... FAILED

failures:

---- tests::greater_than_100 stdout ----
thread 'main' panicked at 'Guess value must be greater than or equal to 1,
got 200.', src/lib.rs:13:13
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
note: panic did not contain expected string
      panic message: `"Guess value must be greater than or equal to 1, got
200."`,
      expected substring: `"Guess value must be less than or equal to 100"`

failures:
tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

error: test failed, to rerun pass '--lib'

```

Сообщение об ошибке указывает, что этот тест действительно вызвал панику, как мы и ожидали, но сообщение о панике не включено ожидаемую строку `'Guess value must be less than or equal to 100'`. Сообщение о панике, которое мы получили в этом случае, было `Guess value must be greater than or equal to 1, got 200.` Теперь мы можем начать выяснение, где ошибка!

Использование `Result<T, E>` в тестах

Пока что мы написали тесты, которые паникуют, когда терпят неудачу. Мы также можем написать тесты которые используют `Result<T, E>`! Вот тест из листинга 11-1, переписанный с использованием `Result<T, E>` и возвращающий `Err` вместо паники:

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() -> Result<(), String> {
        if 2 + 2 == 4 {
            Ok(())
        } else {
            Err(String::from("two plus two does not equal four"))
        }
    }
}
```

Функция `it_works` теперь имеет возвращаемый тип `Result<(), String>`. В теле функции, вместо вызова `assert_eq!` макроса, мы возвращаем `Ok(())` когда тест успешно выполнен и `Err` со `String` внутри, когда тест не проходит.

Написание тестов так, чтобы они возвращали `Result<T, E>` позволяет использовать оператор "вопросительный знак" в теле тестов, который может быть удобным способом писать тесты, которые должны выполниться не успешно, если какая-либо операция внутри них возвращает вариант ошибки `Err`.

Вы не можете использовать аннотацию `#[should_panic]` в тестах, использующих `Result<T, E>`. Чтобы утверждать, что операция возвращает вариант `Err`, не используйте оператор вопросительного знака для значения `Result<T, E>`. Вместо этого используйте `assert!(value.is_err())`.

Теперь, когда вы знаете несколько способов написания тестов, давайте взглянем на то, что происходит при запуске тестов и исследуем разные опции используемые с командой `cargo test`.

Контролирование хода выполнения тестов

Подобно тому, как `cargo run` компилирует ваш код и затем запускает полученный двоичный файл, `cargo test` компилирует ваш код в тестовом режиме и запускает полученный тестовый двоичный файл. Вы можете указать параметры командной строки, чтобы изменить поведение `cargo test` по умолчанию. Например, по умолчанию двоичный файл, созданный с помощью `cargo test` запускает все тесты параллельно и фиксирует выходные данные, созданные во время тестовых запусков, предотвращая отображение выходных данных и упрощая чтение выходных данных, связанных с результатами тестирования.

Опции команды `cargo test` могут быть добавлены после, опции для тестов должны устанавливаться дополнительно (следовать далее). Для разделения этих двух типов аргументов используется разделитель `--`. Чтобы узнать подробнее о доступных опциях команды `cargo test` - используйте опцию `--help`. Для того, чтобы узнать о доступных опциях, непосредственно для тестов, используйте команду `cargo test -- --help`. Обратите внимание, что данную команду необходимо запускать внутри cargo-проекта (пакета).

Выполнение тестов параллельно или последовательно

Когда вы запускаете несколько тестов, по умолчанию они выполняются параллельно с использованием потоков. Это означает, что тесты завершатся быстрее, и вы сможете быстрее получить обратную связь о том, работает ли ваш код. Поскольку тесты выполняются одновременно, убедитесь, что ваши тесты не зависят друг от друга или от какого-либо общего состояния, включая общую среду, такую как текущий рабочий каталог или переменные среды.

Например, когда тесты создают в одном и том же месте на диске файл с одним и тем же названием, читают из него данные, записывают их - вероятность ошибки в работе таких тестов (из-за конкурирования доступа к ресурсу, некорректных данных в файле) весьма высока. Решением будет использование уникальных имён создаваемых и используемых файлов каждым тестом в отдельности, либо выполнение таких тестов последовательно.

Если вы не хотите запускать тесты параллельно или хотите более детальный контроль над количеством используемых потоков, можно установить флаг `--test-threads` и то количество потоков, которое вы хотите использовать для теста.

Взгляните на следующий пример:

```
$ cargo test -- --test-threads=1
```

Мы устанавливаем количество тестовых потоков равным **1**, указывая программе не использовать параллелизм. Выполнение тестов с использованием одного потока займёт больше времени, чем их параллельное выполнение, но тесты не будут мешать друг другу, если они совместно используют состояние.

Демонстрация результатов работы функции

По умолчанию, если тест пройден, система управления запуска тестов блокирует вывод на печать, т.е. если вы вызовете макрос **println!** внутри кода теста и тест будет пройден, вы не увидите вывода на консоль результатов вызова **println!**. Если же тест не был пройден, все информационные сообщение, а также описание ошибки будет выведено на консоль.

Например, в коде (11-10) функция выводит значение параметра с поясняющим текстовым сообщением, а также возвращает целочисленное константное значение **10**. Далее следует тест, который имеет правильный входной параметр и тест, который имеет ошибочный входной параметр:

Файл: src/lib.rs

```
fn prints_and_returns_10(a: i32) -> i32 {
    println!("I got the value {}", 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);
    }
}
```



Listing 11-10: Тест функции, которая использует макрос `println!`

Результат вывода на консоль команды `cargo test`:

```
$ cargo test
Compiling silly-function v0.1.0 (file:///projects/silly-function)
  Finished test [unoptimized + debuginfo] target(s) in 0.58s
    Running unitests (target/debug/deps/silly_function-160869f38cff9166)

running 2 tests
test tests::this_test_will_fail ... FAILED
test tests::this_test_will_pass ... ok

failures:

---- tests::this_test_will_fail stdout ----
I got the value 8
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `5`,
  right: `10`', src/lib.rs:19:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
  tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Обратите внимание, что нигде в этом выводе мы не видим сообщения `I got the value 4`, которое печатается при выполнении пройденного теста. Этот вывод был записан. Результат неудачного теста, `I got the value 8`, появляется в разделе итоговых результатов теста, который также показывает причину неудачного теста.

Для того, чтобы всегда видеть вывод на консоль корректно работающих программ, используйте флаг `--show-output`:

```
$ cargo test -- --show-output
```

Когда мы снова запускаем тесты из Листинга 11-10 с флагом `--show-output`, мы видим следующий результат:

```
$ cargo test -- --show-output
Compiling silly-function v0.1.0 (file:///projects/silly-function)
Finished test [unoptimized + debuginfo] target(s) in 0.60s
Running unitests (target/debug/deps/silly_function-160869f38cff9166)

running 2 tests
test tests::this_test_will_fail ... FAILED
test tests::this_test_will_pass ... ok

successes:

---- tests::this_test_will_pass stdout ----
I got the value 4

successes:
    tests::this_test_will_pass

failures:

---- tests::this_test_will_fail stdout ----
I got the value 8
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `5`,
  right: `10`, src/lib.rs:19:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Запуск подмножества тестов по имени

Бывают случаи, когда в запуске всех тестов нет необходимости и нужно запустить только несколько тестов. Если вы работаете над функцией и хотите запустить тесты, которые исследуют её работу - это было бы удобно. Вы можете это сделать, используя команду `cargo test`, передав в качестве аргумента имена тестов.

Для демонстрации, как запустить группу тестов, мы создадим группу тестов для функции `add_two` (код программы 11-11) и постараемся выбрать интересующие нас тесты при их запуске:

Filename: 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));
    }
}
```

Код программы 11-11: Три теста с различными именами

Если вы выполните команду `cargo test` без уточняющих аргументов, все тесты выполняются параллельно:

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.62s
Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 3 tests
test tests::add_three_and_two ... ok
test tests::add_two_and_two ... ok
test tests::one_hundred ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Запуск одного теста

Мы можем запустить один тест с помощью указания его имени в команде `cargo test`:

```
$ cargo test one_hundred
Compiling adder v0.1.0 (file:///projects/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.69s
Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::one_hundred ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out;
finished in 0.00s
```

Был запущен только тест с названием `one_hundred`; имена остальных тестов отличались. Стока `2 filtered out` в конце тестового вывода позволяет нам понять, что были ещё и другие тесты.

Таким образом мы не можем указать имена нескольких тестов; будет использоваться только первое значение, указанное для `cargo test`. Но есть способ запустить несколько тестов.

Использование фильтров для запуска нескольких тестов

Мы можем указать часть имени теста, и будет запущен любой тест, имя которого соответствует этому значению. Например, поскольку имена двух наших тестов содержат `add`, мы можем запустить эти два, запустив `cargo test add`:

```
$ cargo test add
Compiling adder v0.1.0 (file:///projects/adder)
  Finished test [unoptimized + debuginfo] target(s) in 0.61s
    Running unitests (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test tests::add_three_and_two ... ok
test tests::add_two_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out;
finished in 0.00s
```

Эта команда запускала все тесты с `add` в имени и отфильтровывала тест с именем `one_hundred`. Также обратите внимание, что модуль, в котором появляется тест, становится частью имени теста, поэтому мы можем запускать все тесты в модуле, фильтруя имя модуля.

Игнорирование тестов

Бывает, что некоторые тесты требуют продолжительного времени для своего исполнения, и вы хотите исключить их из исполнения при запуске `cargo test`. Вместо перечисления в командной строке всех тестов, которые вы хотите запускать, вы можете аннотировать тесты, требующие много времени для прогона, атрибутом `ignore`, чтобы исключить их, как показано здесь:

Файл: `src/lib.rs`

```
#[test]
fn it_works() {
    assert_eq!(2 + 2, 4);
}

#[test]
#[ignore]
fn expensive_test() {
    // code that takes an hour to run
}
```

После `#[test]` мы добавляем строку `#[ignore]` в тест, который хотим исключить. Теперь, когда мы запускаем наши тесты, `it_works` запускается, а `expensive_test` игнорируется:

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.60s
Running unit tests (target/debug/deps/adder-92948b65e88960b4)

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;
finished in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Функция `expensive_test` помечена как `ignored`. Если вы хотите выполнить только проигнорированные тесты, вы можете воспользоваться командой `cargo test -- --ignored`:

```
$ cargo test -- --ignored
Compiling adder v0.1.0 (file:///projects/adder)
  Finished test [unoptimized + debuginfo] target(s) in 0.61s
    Running unitests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test expensive_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out;
finished in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Управляя тем, какие тесты запускать, вы можете быть уверены, что результаты вашего `cargo test` будут быстрыми. Вы можете фильтровать тесты по имени при запуске. Вы также можете указать какие тесты должны быть проигнорированы при помощи `ignored`, а также отдельно запускать проигнорированные тесты при помощи `cargo test -- --ignored`.

Организация тестов

Как упоминалось в начале главы, тестирование является сложной дисциплиной и разные люди используют разную терминологию и организацию. Сообщество Rust думает о тестах с точки зрения двух основных категорий: *модульные тесты* и *интеграционные тесты*. Модульные тесты это небольшие и более сфокусированные на тестировании одного модуля в отдельности или могут тестироваться приватные интерфейсы. Интеграционные тесты являются полностью внешними по отношению к вашей библиотеке и используют код библиотеки так же, как любой другой внешний код, используя только общедоступные интерфейсы и потенциально выполняя тестирование нескольких модулей в одном тесте.

Написание обоих видов тестов важно для обеспечения того, чтобы кусочки вашей библиотеки по отдельности и вместе делали то, что вы ожидаете.

Модульные тесты

Целью модульных тестов является тестирование каждого блока кода, изолированное от остального функционала, чтобы можно было быстро понять, что работает некорректно или не так как ожидается. Мы разместим модульные тесты в папке `src`, в каждый тестируемый файл. Но в Rust принято создавать тестирующий модуль `tests` и код теста сохранять в файлы с таким же именем, как компоненты которые предстоит тестировать. Также необходимо добавить аннотацию `cfg(test)` к этому модулю.

Модуль тестов и аннотация `#[cfg(test)]`

Аннотация `#[cfg(test)]` у модуля с тестами указывает Rust компилировать и запускать только код тестов, когда выполняется команда `cargo test`, а не когда запускается `cargo build`. Это экономит время компиляции, если вы только хотите собрать библиотеку и сэкономить место для результирующих скомпилированных артефактов, потому что тесты не будут включены. Вы увидите что, по причине того, что интеграционные тесты помещаются в другой каталог им не нужна аннотация `#[cfg(test)]`. Тем не менее, так как модульные тесты идут в тех же файлах что и основной код, вы будете использовать `#[cfg(test)]` чтобы указать, что они не должны быть включены в скомпилированный результат.

Напомним, что когда мы генерировали новый проект `adder` в первом разделе этой

главы, то Cargo сгенерировал для нас код ниже:

Файл: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

Этот код является автоматически сгенерированным тестовым модулем. Атрибут `cfg` предназначен для *конфигурации* и говорит Rust, что следующий элемент должен быть включён только учитывая определённую опцию конфигурации. В этом случае опцией конфигурации является `test`, который предоставлен в Rust для компиляции и запуска текущих тестов. Используя атрибут `cfg`, Cargo компилирует только тестовый код при активном запуске тестов командой `cargo test`. Это включает в себя любые вспомогательные функции, которые могут быть в этом модуле в дополнение к функциям помеченным `#[test]`.

Тестирование приватных функций (private)

Сообщество программистов не имеет однозначного мнения по поводу тестировать или нет приватные функции. В некоторых языках весьма сложно или даже невозможно тестировать такие функции. Независимо от того, какой технологии тестирования вы придерживаетесь, в Rust приватные функции можно тестировать. Рассмотрим листинг 11-12 с приватной функцией `internal_adder`.

Файл: 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));
    }
}
```

Листинг 11-12: Тестирование приватных функций

Обратите внимание, что функция `internal_adder` не помечена как `pub`. Тесты — это просто Rust код, а модуль `tests` — это ещё один модуль. Как мы обсуждали в разделе “[Пути для ссылки на элемент в дереве модулей](#)”, элементы в дочерних модулях могут использовать элементы из своих родительских модулей. В этом тесте мы помещаем все элементы родительского модуля `test` в область видимости с помощью `use super::*` и затем тест может вызывать `internal_adder`. Если вы считаете, что приватные функции не нужно тестировать, то Rust не заставит вас это сделать.

Интеграционные тесты

В Rust интеграционные тесты являются полностью внешними по отношению к вашей библиотеке. Они используют вашу библиотеку так же, как любой другой код, что означает, что они могут вызывать только функции, которые являются частью публичного API библиотеки. Их целью является проверка, много ли частей вашей библиотеки работают вместе правильно. У модулей кода правильно работающих самостоятельно, могут возникнуть проблемы при интеграции, поэтому тестовое покрытие интегрированного кода также важно. Для создания интеграционных тестов сначала нужен каталог `tests`.

Каталог `tests`

Для создания интеграционных тестов, мы создаём папку *tests* в корневой папке вашего проекта, рядом с папкой *src*. Cargo знает, что файлы с интеграционными тестами будут храниться в этой директории. В этой директории можно создать столько интеграционных тестов, сколько нужно. Каждый такой файл будет компилироваться в отдельный крейт.

Давайте создадим интеграционный тест. Рядом с кодом из листинга 11-12, который в файле *src/lib.rs*, создайте каталог *tests*, создайте новый файл с именем *tests/integration_test.rs* и введите код из листинга 11-13.

Файл: *tests/integration_test.rs*

```
use adder;

#[test]
fn it_adds_two() {
    assert_eq!(4, adder::add_two(2));
}
```

Листинг 11-13: Интеграционная тест функция из крейта `adder`

Мы добавили `use adder` в начале кода, который был не нужен в модульных тестах. Причина его присутствия в том, что каждый файл в каталоге `tests` является отдельным крейтом, поэтому нужно подключить нашу библиотеку в область видимости каждого интеграционного тест крейта.

Нам не нужно комментировать код в *tests/integration_test.rs* с помощью `# [cfg(test)]`. Cargo специальным образом обрабатывает каталог `tests` и компилирует файлы в этом каталоге только тогда, когда мы запускаем команду `cargo test`. Запустите `cargo test` сейчас:

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
  Finished test [unoptimized + debuginfo] target(s) in 1.31s
    Running unittests (target/debug/deps/adder-1082c4b063a8fbe6)

running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

    Running tests/integration_test.rs (target/debug/deps/integration_test-
1082c4b063a8fbe6)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Три раздела вывода включают в себя модульные тесты, интеграционные тесты и док тесты. Первый раздел для модульных тестов такой же, как мы уже видели: одна строка для каждого модульного теста (одна с именем `internal` которой мы добавили в листинге 11-12) и затем итоговая строка для модульных тестов.

Раздел интеграционных тестов начинается со строки `Running` `target/debug/deps/integration_test-1082c4b063a8fbe6` (хэш в конце вашего вывода будет другим). Далее идёт строка для каждой тестовой функции в этом интеграционном тесте и итоговая строка для результатов интеграционного теста непосредственно перед началом раздела `Doc-tests adder`.

Подобно тому, как добавление большего количества функций модульных тестов добавляет большее количество строк вывода в разделе модульных тестов, добавление дополнительных функций тестирования в файл интеграционных тестов добавляет дополнительные строки вывода в разделе интеграционных тестов. Каждый файл интеграционных тестов имеет свой собственный раздел, поэтому, если мы добавим больше файлов в каталог `tests`, то будет длиннее раздел вывода

результатов интеграционных тестов.

Мы всё ещё можем запустить определённую функцию в интеграционных тестах, указав имя тест функции в качестве аргумента в `cargo test`. Чтобы запустить все тесты в конкретном файле интеграционных тестов, используйте аргумент `--test` сопровождаемый именем файла у команды `cargo test`:

```
$ cargo test --test integration_test
Compiling adder v0.1.0 (file:///projects/adder)
  Finished test [unoptimized + debuginfo] target(s) in 0.64s
    Running tests/integration_test.rs (target/debug/deps/integration_test-
82e7799c1bc62298)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Эта команда запускает только тесты в файле `tests/integration_test.rs`.

Подмодули в интеграционных тестах

По мере добавления большего количества интеграционных тестов, можно создать более одного файла в каталоге `tests`, чтобы легче организовывать их; например, вы можете сгруппировать функции тестирования по функциональности, которую они проверяют. Как упоминалось ранее, каждый файл в каталоге `tests` скомпилирован как отдельный крейт.

Рассматривая каждый файл интеграционных тестов как отдельный крейт, полезно создать отдельные области видимости, которые больше похожи на то, как конечные пользователи будут использовать ваш крейт. Тем не менее, это означает, что файлы в каталоге `tests` не разделяют поведение как это делают файлы в `src`, что вы изучили в главе 7 о том, как разделить код на модули и файлы.

Различное поведение файлов в каталоге `tests` наиболее заметно, когда у вас есть набор вспомогательных функций, которые будут полезны в нескольких интеграционных тест файлах и вы пытаетесь выполнить действия, описанные в разделе «[Разделение модулей в разные файлы](#)» главы 7, чтобы извлечь их в общий модуль. Например, если мы создадим `tests/common.rs` и поместим в него функцию с именем `setup`, то можно добавить некоторый код в `setup`, который мы хотим

вызывать из нескольких тестовых функций в нескольких тестовых файлах

Файл: tests/common.rs

```
pub fn setup() {  
    // setup code specific to your library's tests would go here  
}
```

Когда мы снова запустим тесты, мы увидим новый раздел в результатах тестов для файла `common.rs`, хотя этот файл не содержит ни тестовых функций, ни того, что мы явно вызывали функцию `setup` откуда либо:

```
$ cargo test  
Compiling adder v0.1.0 (file:///projects/adder)  
Finished test [unoptimized + debuginfo] target(s) in 0.89s  
Running unitests (target/debug/deps/adder-92948b65e88960b4)  
  
running 1 test  
test tests::internal ... ok  
  
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;  
finished in 0.00s  
  
    Running tests/common.rs (target/debug/deps/common-92948b65e88960b4)  
  
running 0 tests  
  
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;  
finished in 0.00s  
  
    Running tests/integration_test.rs (target/debug/deps/integration_test-  
92948b65e88960b4)  
  
running 1 test  
test it_adds_two ... ok  
  
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;  
finished in 0.00s  
  
Doc-tests adder  
  
running 0 tests  
  
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;  
finished in 0.00s
```

Появление файла `common` и появление сообщения в результатах выполнения тестов типа `running 0 tests` это не то, чтобы мы хотели. Мы только хотели использовать некоторый общий код с другими интеграционными файлами тестов.

Чтобы избежать появления `common` в тестовом выводе, вместо создания `tests/common.rs`, мы создадим `tests/common/mod.rs`. Это альтернатива соглашению об именах, которое Rust также понимает. Именование файла таким образом говорит, что Rust не следует рассматривать `common` модуль как файл интеграционных тестов. Когда мы перемещаем код функции `setup` в файл `tests/common/mod.rs` и удаляем файл `tests/common.rs`, то он больше не будет отображаться в результатах тестов. Файлы в подкаталогах каталога `tests` не компилируются как отдельные крейты и не появляются в выводе выполнения тестов.

После того, как вы создали модуль `tests/common/mod.rs`, можно использовать его в любых интеграционных файлах тестов как модуль. Вот пример вызова функции `setup` из теста `it_adds_two` в файле `tests/integration_test.rs`:

Файл: `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 объявлением модуля, которое продемонстрировано в листинге 7-21. Затем в тестовой функции мы можем вызвать функцию `common::setup()`.

Интеграционные тесты для бинарных крейтов

Если наш проект является бинарным крейтом, который содержит только `src/main.rs` и не содержит `src/lib.rs`, то в таком случае, мы не можем создать интеграционные тесты в папке `tests` и подключить функции определённые в файле `src/main.rs` в область видимости с помощью выражения `use`. Только библиотечные крейты могут предоставлять функции, которые можно использовать в других крейтах; бинарные крейты предназначены только для самостоятельного запуска.

Это одна из причин того, что Rust проекты для выполняемой программы имеют просто файл `src/main.rs`, который вызывает логику, которая находится в файле `src/lib.rs`. Используя такую структуру, интеграционные тесты *могут* протестировать библиотечный крейт с помощью `use`, чтобы подключить важную функциональность и сделать её доступной. Если важная функциональность работает, то небольшое количество кода в файле `src/main.rs` также будет работать, и этот небольшой объем кода не нужно проверять.

Итоги

Функции тестирования в Rust позволяют указать, как должен функционировать код, чтобы убедиться, что он продолжает работать как этого вы ожидаете, даже если вы вносите изменения. Модульные тесты проверяют разные части библиотеки по отдельности и могут тестировать частные детали реализации. Интеграционные тесты проверяют, что части библиотеки работают вместе правильно и они используют открытый API библиотеки для тестирования кода таким же образом, как внешний код будет использовать его. Хотя система типов Rust и правила владения помогают предотвратить некоторые виды ошибок, тесты по-прежнему важны для уменьшения количества логических ошибок, связанных с поведением вашего кода.

Давайте объединим знания полученные в этой и предыдущей главах, чтобы поработать над проектом!

Проект с вводом/выводом (I/O): создание консольного приложения

В этой главе вы примените многие знания, полученные ранее, а также познакомитесь с ещё неизученными API стандартной библиотеки. Мы создадим инструмент командной строки, который взаимодействует с файлом и с вводом / выводом командной строки, чтобы попрактиковаться в некоторых концепциях Rust, с которыми вы уже знакомы.

Скорость, безопасность, компиляция в один исполняемый файл и кроссплатформенность делают Rust идеальным языком для создания инструментов командной строки, так что в нашем проекте мы создадим свою собственную версию классической утилиты `grep`, что расшифровывается, как "глобальное средство поиска и печати" (globally search a regular expression and print). В простейшем случае `grep` используется для поиска в указанном файле указанного текста. Для этого утилиты `grep` получает имя файла и текст в качестве аргументов. Далее она читает файл, находит и выводит строки, содержащие искомый текст.

По пути мы покажем, как сделать так, чтобы наш инструмент использовал возможности терминала, которые используются многими инструментами командной строки. Мы будем читать значение переменной окружения, чтобы позволить пользователю настроить поведение нашего инструмента. Мы также будем печатать сообщения об ошибках в стандартный консольный поток ошибок (`stderr`) вместо стандартного вывода (`stdout`), чтобы, к примеру, пользователь мог перенаправить успешный вывод в файл, в то время, как сообщения об ошибках останутся на экране.

Один из участников Rust-сообщества, Andrew Gallant уже реализовал полнофункциональный, очень быстрый аналог программы `grep` и назвал его `ripgrep`. Создаваемая нами версия `grep` будет, конечно, намного проще, но эта глава даст вам знания, необходимые для понимания этапов создания реальных проектов, таких как `ripgrep`.

Наш проект `grep` будет использовать ранее изученные концепции:

- Организация кода (используя то, что вы узнали о модулях в [главе 7](#))
- Использование векторов и строк (коллекции, [глава 8](#))
- Обработка ошибок ([Глава 9](#))
- Использование типажей и времени жизни там, где это необходимо ([глава 10](#))

- Написание тестов ([Глава 11](#))

Мы также кратко представим замыкания, итераторы и объекты типажи, которые будут объяснены подробно в главах [13](#) и [17](#).

Принятие аргументов командной строки

Создадим новый проект консольного приложения как обычно с помощью команды `cargo new`. Мы назовём проект `minigrep`, чтобы различать наше приложение от `grep`, которое возможно уже есть в вашей системе.

```
$ cargo new minigrep
     Created binary (application) `minigrep` project
$ cd minigrep
```

Первой задачей для реализации `minigrep` приложения будет принятие двух переменных командной строки: имени файла и строки для поиска. Мы хотим запускать нашу программу командой `cargo run`, со строкой поиска и путём к файлу для поиска в нём, как в примере:

```
$ cargo run searchstring example-filename.txt
```

В данный момент программа сгенерированная `cargo new` не может обрабатывать аргументы, которые мы ей передаём. Некоторые существующие библиотеки на [crates.io](#) могут помочь с написанием программы, которая принимает аргументы командной строки, но так как вы просто изучаете эту концепцию, давайте реализуем эту возможность сами.

Чтение значений аргументов

Чтобы сделать возможным в `minigrep` чтение передаваемых значений аргументов командной строки, нам понадобится функция, предоставляемая в стандартной библиотеке Rust `std::env::args`. Эта функция возвращает итератор аргументов командной строки, которые были переданы в `minigrep`. Мы рассмотрим итераторы полностью в [Главе 13](#), в данный момент вам нужно знать про итераторы только две детали: итераторы производят серию значений и мы можем вызвать `collect` метод на итераторе, чтобы превратить его в коллекцию, такую как вектор, содержащий все элементы, которые создаёт итератор.

Используйте код в листинге 12-1, чтобы позволить программе `minigrep` читать любые аргументы командной строки передающиеся ей, а затем собирать эти значения в вектор.

Файл: `src/main.rs`

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    println!("{}: {}", args);
}
```

Листинг 12-1: Сбор аргументов командной строки в вектор и печать этих данных в консоль

Сначала подключаем модуль `std::env` в область видимости с помощью оператора `use`, чтобы иметь возможность использовать у неё функцию `args`. Обратите внимание, что функция `std::env::args` вложена на двух уровнях модулей. Как мы обсуждали в [главе 7](#) в случаях, когда желаемая функция вложена в более чем один модуль, то обычно удобно подключить в область видимости родительский модуль, а не функцию. Таким образом, мы можем легко использовать другие функции из `std::env`. Это также менее двусмысленно, чем добавление `use std::env::args`, а затем вызов функции только с `args`, потому что `args` может быть легко принята за функцию, определённую в текущем модуле.

Функция `args` и недействительный Юникод символ (Unicode)

Обратите внимание, что `std::env::args` будет паниковать, если какой-либо переданный аргумент содержит недопустимый Unicode символ. Если ваша программа должна принимать аргументы, содержащие недопустимые Unicode символы, используйте вместо этого `std::env::args_os`. Эта функция возвращает итератор, который создаёт значения `OsString` вместо значений `String`. Мы выбрали использование `std::env::args` для простоты, потому что `OsString` значения отличаются на разных платформах и более сложны для обработки, чем значения `String`.

В первой строке кода функции `main` мы вызываем `env::args` и сразу используем метод `collect`, чтобы превратить итератор в вектор содержащий все полученные значения. Мы можем использовать функцию `collect` для создания многих видов коллекций, поэтому мы явно аннотируем тип `args` чтобы указать, что мы хотим вектор строк. Хотя нам очень редко нужно аннотировать типы в Rust, `collect` - это одна из функций, с которой вам часто нужна аннотация типа, потому что Rust не может сам вывести какую коллекцию вы хотите.

В конце мы печатаем вектор с использованием средства форматированной отладки `:?`. Давай попробуем выполнить код сначала без аргументов, а затем с двумя аргументами:

```
$ cargo run
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.61s
    Running `target/debug/minigrep`
["target/debug/minigrep"]
```

```
$ cargo run needle haystack
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 1.57s
    Running `target/debug/minigrep needle haystack`
["target/debug/minigrep", "needle", "haystack"]
```

Обратите внимание, что первое значение в векторе `"target/debug/minigrep"` является названием нашего двоичного файла. Это соответствует поведению списка аргументов в Си, позволяя программам использовать название из которой они были вызваны при выполнении. Часто бывает удобно иметь доступ к имени программы, если вы хотите распечатать его в сообщениях или изменить поведение программы в зависимости от того, какой псевдоним командной строки был использован для вызова программы. Но для целей этой главы, мы проигнорируем его и сохраним только два аргумента, которые нам нужны.

Сохранения значений аргументов в переменные

Вывод на печать значений аргументов командной строки - это простой тест возможности программы иметь доступ к аргументам командной строки. Далее, нам надо сохранить значение аргументов в переменные, чтобы иметь возможность их использования далее в программе. Пример реализации в листинге 12-2.

Файл: `src/main.rs`

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    let query = &args[1];
    let filename = &args[2];

    println!("Searching for {}", query);
    println!("In file {}", filename);
}
```

Листинг 12-2: Создание переменных для хранения шаблона поиска и аргумента имени файла

Как мы уже знаем из наших предыдущих упражнений, первый аргумент вектора хранит полное имя бинарного файла в векторе `args[0]`, поэтому мы начинаем с индекса `1`. Первый аргумент принимаемый `minigrep` является строкой, которую 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`.

Для проверки корректности работы нашей программы, значения переменных выводятся в консоль. Далее, запустим нашу программу со следующими 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 0.0s
    Running `target/debug/minigrep test sample.txt`
Searching for test
In file sample.txt
```

Отлично, программа работает! Нам нужно чтобы значения аргументов были сохранены в правильных переменных. Позже мы добавим обработку ошибок с некоторыми потенциальными ошибочными ситуациями, например, когда пользователь не предоставляет аргументы; сейчас мы проигнорируем эту ситуацию и поработаем над добавлением возможности чтения файла.

Чтение файла

Теперь добавим возможность чтения файла, указанного как аргумент командной строки `filename`. Во-первых, нам нужен пример файла для тестирования: лучший тип файла для проверки работы `minigrep` это файл с небольшим количеством текста в несколько строк с несколькими повторяющимися словами. В листинге 12-3 представлено стихотворение Эмили Дикинсон, которое будет хорошо работать! Создайте файл с именем `poem.txt` в корне вашего проекта и введите стихотворение "I'm nobody! Who are you?"

Файл: `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!
```

Листинг 12-3: Стихотворение Эмили Дикинсон "I'm nobody! Who are you?"

Текст на месте, отредактируйте `src/main.rs` и добавьте код для чтения файла, как показано в листинге 12-4.

Файл: `src/main.rs`

```
use std::env;  
use std::fs;  
  
fn main() {  
    // --snip--  
    println!("In file {}", filename);  
  
    let contents = fs::read_to_string(filename)  
        .expect("Something went wrong reading the file");  
  
    println!("With text:\n{}", contents);  
}
```

Листинг 12-4: Чтение содержимого файла указанного во втором аргументе

Во-первых, мы добавляем ещё одно объявление `use` чтобы подключить

соответствующую часть стандартной библиотеки: нам нужен `std::fs` для обработки файлов.

В `main` мы добавили новый оператор: функция `fs::read_to_string` принимает `filename`, открывает этот файл и возвращает содержимое файла как `Result<String>`.

После этого выражения мы снова добавили временный вывод `println!` для печати значения `contents` после чтения файла, поэтому мы можем проверить, что программа работает.

Давайте запустим этот код с любой строкой в качестве первого аргумента командной строки (потому что мы ещё не реализовали поисковую часть) и файл `poem.txt` как второй аргумент:

```
$ cargo run the poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
    Running `target/debug/minigrep the poem.txt`
Searching for the
In file poem.txt
With text:
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! Этот код прочитал и затем печатал содержимое файла. Хотя наша программа решает поставленную задачу, она не лишена недостатков. Прежде всего, функция `main` решает множество задач. Такую функцию неудобно тестировать. Далее, не отслеживаются возможные ошибки ввода данных. Пока наша программа небольшая, то данными недочётами можно пренебречь. При увеличении размеров программы, такую программу будет всё сложнее и сложнее поддерживать. Хорошей практикой программирования является ранний рефакторинг кода по мере усложнения. Поэтому, далее мы улучшим наш код с помощью улучшения его структуры.

Рефакторинг для улучшения модульности и обработки ошибок

Для улучшения программы мы исправим 4 имеющихся проблемы, связанных со структурой программы и тем как обрабатываются потенциальные ошибки.

Во-первых, функция `main` на данный момент решает две задачи - анализирует переменные командной строки и читает файлы. Для небольшой функции это не является проблемой. Тем не менее, при увеличении функционала внутри `main`, количество отдельных обрабатываемых задач в функции `main` будет расти. Поскольку эта функция получает больше обязанностей, то становится все труднее понимать, труднее тестировать и труднее изменять, не сломав одну из её частей. Лучше всего разделить функциональность, чтобы каждая функция отвечала за одну задачу.

Эта проблема также связана со второй проблемой: хотя переменные `query` и `filename` являются переменными конфигурации нашей программы, переменные типа `contents` используются для выполнения логики программы. Чем длиннее `main` становится, тем больше переменных нам нужно будет добавить в область видимости; чем больше у нас переменных, тем сложнее будет отслеживать назначение каждой переменной. Лучше всего сгруппировать переменные конфигурации в одну структуру, чтобы сделать их назначение понятным.

Третья проблема заключается в том, что мы используем `expect` для вывода информации об ошибке при проблеме с чтением файла, но сообщение об ошибке просто выведет текст `Something went wrong reading the file`. Чтение файла может не сработать по разным причинам, например: файл не найден или у нас может не быть разрешения на его чтение. В существующем коде, независимо от ситуации, мы напечатаем `Something went wrong reading the file`, что не даст пользователю никакой информации!

В четвёртых, мы используем `expect` неоднократно для обработки различных ошибок и если пользователь запускает нашу программу без указания достаточного количества аргументов он получит ошибку `index out of bounds` из Rust, что не совсем понятно описывает проблему. Было бы лучше, если бы весь код обработки ошибок был в одном месте. Это позволило бы тем, кто будет поддерживать наш код в дальнейшем, при необходимости изменения логики обработки ошибок, вносить нужные изменения только в одном месте. Наличие всего кода обработки ошибок в одном месте гарантирует, что мы напечатаем сообщения, которые будут иметь

смысл для наших конечных пользователей.

Давайте решим эти четыре проблемы путём рефакторинга нашего проекта.

Разделение ответственности для бинарных проектов

Организационная проблема распределения ответственности за выполнение нескольких задач функции `main` является общей для многих выполняемых проектов. В результате Rust сообщество разработало процесс для использования в качестве руководства по разделению ответственности бинарной программы, когда код в `main` начинает увеличиваться. Процесс имеет следующие шаги:

- Разделите код программы на два файла `main.rs` и `lib.rs`. Перенесите всю логику работы программы в файл `lib.rs`.
- Пока ваша логика синтаксического анализа командной строки мала, она может оставаться в файле `main.rs`.
- Когда логика синтаксического анализа командной строки становится сложной, извлеките её из `main.rs` и переместите в `lib.rs`.

Функциональные обязанности, которые остаются в функции `main` после этого процесса должно быть ограничено следующим:

- Вызов логики разбора командной строки со значениями аргументов
- Настройка любой другой конфигурации
- Вызов функции `run` в `lib.rs`
- Обработка ошибки, если `run` возвращает ошибку

Этот шаблон о разделении ответственности: `main.rs` занимается запуском программы, а `lib.rs` обрабатывает всю логику задачи. Поскольку нельзя проверить функцию `main` напрямую, то такая структура позволяет проверить всю логику программы путём перемещения её в функции внутри `lib.rs`. Единственный код, который остаётся в `main.rs` будет достаточно маленьким, чтобы проверить его корректность прочитав код. Давайте переработаем нашу программу, следуя этому процессу.

Извлечение парсера аргументов

Мы извлечём функциональность для разбора аргументов в функцию, которую вызовет `main` для подготовки к перемещению логики разбора командной строки в

файл `src/lib.rs`. Листинг 12-5 показывает новый запуск `main`, который вызывает новую функцию `parse_config`, которую мы определим сначала в `src/main.rs`.

Файл: `src/main.rs`

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let (query, filename) = parse_config(&args);

    // --snip--
}

fn parse_config(args: &[String]) -> (&str, &str) {
    let query = &args[1];
    let filename = &args[2];

    (query, filename)
}
```

Листинг 12-5. Извлечение функции `parse_config` из `main`

Мы все ещё собираем аргументы командной строки в вектор, но вместо присваивания значение аргумента с индексом 1 переменной `query` и значение аргумента с индексом 2 переменной с именем `filename` в функции `main`, мы передаём весь вектор в функцию `parse_config`. Функция `parse_config` затем содержит логику, которая определяет, какой аргумент идёт в какую переменную и передаёт значения обратно в `main`. Мы все ещё создаём переменные `query` и `filename` в `main`, но `main` больше не несёт ответственности за определение соответствия аргумента командной строки и соответствующей переменной.

Эта доработка может показаться излишней для нашей маленькой программы, но мы проводим рефакторинг небольшими, постепенными шагами. После внесения этого изменения снова запустите программу и убедитесь, что анализ аргументов все ещё работает. Также хорошо часто проверять прогресс, чтобы помочь определить причину проблем, когда они возникают.

Группировка конфигурационных переменных

Мы можем сделать ещё один маленький шаг для улучшения функции `parse_config`. На данный момент мы возвращаем кортеж, но затем мы немедленно разделяем его снова на отдельные части. Это признак того, что, возможно, пока у нас нет правильной абстракции.

Ещё один индикатор, который показывает, что есть место для улучшения, это часть `config` из `parse_config`, что подразумевает, что два значения, которые мы возвращаем, связаны друг с другом и оба являются частью одного конфигурационного значения. В настоящее время мы не отражаем этого смысла в структуре данных, кроме группировки двух значений в кортеж; мы могли бы поместить оба значения в одну структуру и дать каждому из полей структуры понятное имя. Это облегчит будущую поддержку этого кода, чтобы понять, как различные значения относятся друг к другу и какое их назначение.

В листинге 12-6 показаны улучшения функции `parse_config`.

Файл: src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = parse_config(&args);

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    let contents = fs::read_to_string(config.filename)
        .expect("Something went wrong reading the file");

    // --snip--
}

struct Config {
    query: String,
    filename: String,
}

fn parse_config(args: &[String]) -> Config {
    let query = args[1].clone();
    let filename = args[2].clone();

    Config { query, filename }
}
```

Листинг 12-6: Рефакторинг функции `parse_config`, чтобы возвращать экземпляр структуры `Config`

Мы добавили структуру с именем `Config` объявленную с полями названными как `query` и `filename`. Сигнатура `parse_config` теперь указывает, что она возвращает значение `Config`. В теле `parse_config`, где мы возвращали срезы строк, которые ссылаются на значения `String` в `args`, теперь мы определяем `Config` как

содержащие собственные `String` значения. Переменная `args` в `main` является владельцем значений аргумента и позволяют функции `parse_config` только одалживать их, что означает, что мы бы нарушили правила заимствования Rust, если бы `Config` попытался бы взять во владение значения в `args`.

Мы можем управлять данными `String` разным количеством способов, но самый простой, хотя и отчасти неэффективный это вызвать метод `clone` у значений. Он сделает полную копию данных для экземпляра `Config`, для владения, что занимает больше времени и памяти, чем сохранение ссылки на строку данных. Однако клонирование данных также делает наш код очень простым, потому что нам не нужно управлять временем жизни ссылок; в этом обстоятельстве, отказ от небольшой производительности, чтобы получить простоту, стоит небольших компромисса.

Компромиссы при использовании метода `clone`

Существует тенденция в среде программистов Rust избегать использования `clone`, т.к. это понижает эффективность работы кода. В [Глава 13](#), вы изучите более эффективные методы, которые могут подойти в подобной ситуации. Но сейчас можно копировать несколько строк, чтобы продолжить работу, потому что вы сделаете эти копии только один раз, а ваше имя файла и строка запроса будут очень маленькими. Лучше иметь рабочую программу, которая немного неэффективна, чем пытаться заранее оптимизировать код при первом написании. По мере приобретения опыта работы с Rust вам будет проще начать с наиболее эффективного решения, но сейчас вполне приемлемо вызвать `clone`.

Мы обновили код в `main` поэтому он помещает экземпляр `Config` возвращённый из `parse_config` в переменную с именем `config`, и мы обновили код, в котором ранее использовались отдельные переменные `query` и `filename`, так что теперь он использует вместо этого поля в структуре `Config`.

Теперь наш код более чётко передаёт то, что `query` и `filename` связаны и что цель из использования состоит в том, чтобы настроить, как программа будет работать. Любой код, который использует эти значения знает, что может найти их в именованных полях экземпляра `config` по их назначению.

Создание конструктора для структуры Config

Пока что мы извлекли логику, отвечающую за синтаксический анализ аргументов командной строки из `main` и поместили его в функцию `parse_config`. Это помогло нам увидеть, что значения `query` и `filename` были связаны и что их отношения должны быть отражены в нашем коде. Затем мы добавили структуру `Config` в качестве названия связанных общей целью `query` и `filename` и чтобы иметь возможность вернуть именованные значения как имена полей структуры из функции `parse_config`.

Итак, теперь целью функции `parse_config` является создание экземпляра `Config`, мы можем изменить `parse_config` из простой функции на функцию названную `new`, которая связана со структурой `Config`. Выполняя это изменение мы сделаем код более идиоматичным. Можно создавать экземпляры типов в стандартной библиотеке, такие как `String` с помощью вызова `String::new`. Точно так же изменив название `parse_config` на название функции `new`, связанную с `Config`, мы будем уметь создавать экземпляры `Config`, вызывая `Config::new`. Листинг 12-7 показывает изменения, которые мы должны сделать.

Файл: src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args);

    // --snip--
}

// --snip--


impl Config {
    fn new(args: &[String]) -> Config {
        let query = args[1].clone();
        let filename = args[2].clone();

        Config { query, filename }
    }
}
```

Листинг 12-7. Изменение имени с `parse_config` на `Config::new`

Мы обновили `main` где вызывали `parse_config`, чтобы вместо этого вызывалась `Config::new`. Мы изменили имя `parse_config` на `new` и перенесли его внутрь блока

`impl`, который связывает `new` функцию с `Config`. Попробуйте снова скомпилировать код, чтобы убедиться, что он работает.

Исправление ошибок обработки

Теперь мы поработаем над исправлением обработки ошибок. Напомним, что попытки получить доступ к значениям в векторе `args` с индексом 1 или индексом 2 приведут к панике, если вектор содержит менее трёх элементов. Попробуйте запустить программу без каких-либо аргументов; это будет выглядеть так:

```
$ cargo run
Compiling minigrep v0.1.0 (file:///projects/minigrep)
  Finished dev [unoptimized + debuginfo] target(s) in 0.0s
    Running `target/debug/minigrep`
thread 'main' panicked at 'index out of bounds: the len is 1 but the index is
1', src/main.rs:27:21
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Строка `index out of bounds: the len is 1 but the index is 1` является сообщением об ошибке предназначено для программистов. Она не поможет нашим конечным пользователям понять, что случилось и что они должны сделать вместо этого. Давайте исправим это сейчас.

Улучшение сообщения об ошибке

В листинге 12-8 мы добавляем проверку в функцию `new`, которая будет проверять, что срез достаточно длинный, перед попыткой доступа по индексам 1 и 2. Если срез не достаточно длинный, программа паникует и отображает улучшенное сообщение об ошибке, чем сообщение `index out of bounds`.

Файл: src/main.rs

```
// --snip--
fn new(args: &[String]) -> Config {
    if args.len() < 3 {
        panic!("not enough arguments");
    }
// --snip--
```

Листинг 12-8. Добавление проверки на число аргументы

Этот код похож на функцию `Guess::new` написанную в листинге 9-13 где мы вызывали `panic!`, когда `value` аргумента вышло за пределы допустимых значений. Здесь вместо проверки на диапазон значений, мы проверяем, что длина `args` не менее 3 и остальная часть функции может работать при условии, что это условие было выполнено. Если в `args` меньше трёх элементов, это условие будет правдивым и мы вызываем макрос `panic!` для немедленного завершения программы.

Имея нескольких лишних строк кода в `new`, давайте запустим программу снова без аргументы, чтобы увидеть, как выглядит ошибка:

```
$ cargo run
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
    Running `target/debug/minigrep`
thread 'main' panicked at 'not enough arguments', src/main.rs:26:13
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Этот вывод лучше: у нас теперь есть разумное сообщение об ошибке. Тем не менее, мы также имеем постороннюю информацию, которую мы не хотим предоставлять нашим пользователям. Возможно, использованная техника, которую мы использовали в листинге 9-10, не является лучшей для использования: вызов `panic!` больше подходит для программирования проблемы, чем решения проблемы, как обсуждалось в главе 9. Вместо этого мы можем использовать другую технику, о которой вы узнали в главе 9 [возвращая `Result`], которая указывает либо на успех, либо на ошибку.

Возвращение `Result` из `new` вместо вызова `panic!`

Мы можем вернуть значение `Result`, которое будет содержать экземпляр `Config` в успешном случае и опишет проблему в случае ошибки. Когда `Config::new` взаимодействует с `main`, мы можем использовать тип `Result` как сигнал возникновения проблемы. Затем мы можем изменить `main`, чтобы преобразовать вариант `Err` в более практичную ошибку для наших пользователей без окружающего текста вроде `thread 'main'` и `RUST_BACKTRACE`, что происходит при вызове `panic!`.

Листинг 12-9 показывает изменения, которые нужно внести для возвращения значения из `Config::new` и в тело функции, необходимые для возврата типа `Result`. Заметьте, что этот код не скомпилируется, пока мы не обновим `main`, что

мы и сделаем в следующем листинге.

Файл: src/main.rs

```
impl Config {
    fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        Ok(Config { query, filename })
    }
}
```



Листинг 12-9. Возвращение типа `Result` из `Config::new`

Наша функция `new` теперь возвращает `Result` с экземпляром `Config` в случае успеха и `&'static str` в случае ошибки. Значения ошибок всегда будут строковыми литералами, которые имеют время жизни `'static`.

Мы внесли два изменения в тело функции `new`: вместо вызова `panic!`, когда пользователь не передаёт достаточно аргументов, мы теперь возвращаем `Err` значение и мы завернули возвращаемое значение `Config` в `Ok`. Эти изменения заставят функцию соответствовать своей новой сигнатуре типа.

Возвращение значения `Err` из `Config::new` позволяет функции `main` обработать значение `Result` возвращённое из функции `new` и выйти из процесса более чисто в случае ошибки.

Вызов `Config::new` и обработка ошибок

Чтобы обработать ошибку и вывести более дружественное сообщение об ошибке, нам нужно обновить код `main` для обработки `Result`, возвращаемого из `Config::new` как показано в листинге 12-10. Мы также возьмём на себя ответственность за выход из программы командной строки с ненулевым кодом ошибки `panic!` и реализуем это вручную. Не нулевой статус выхода - это соглашение, которое сигнализирует процессу, который вызывает нашу программу, что программа завершилась с ошибкой.

Файл: src/main.rs

```
use std::process;

fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args).unwrap_or_else(|err| {
        println!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    // --snip--
```

Листинг 12-10. Выход с кодом ошибки если создание новой `Config` терпит неудачу

В этом листинге мы использовали метод, который мы ещё не рассматривали детально: `unwrap_or_else`, который в стандартной библиотеке определён как `Result<T, E>`. Использование `unwrap_or_else` позволяет нам определить некоторые пользовательские ошибки обработки, не содержащие `panic!`. Если `Result` является значением `Ok`, поведение этого метода аналогично `unwrap`: возвращает внутреннее значение из обёртки `Ok`. Однако, если значение значение `Err`, то этот метод вызывает код замыкания, которое является анонимной функцией определённую заранее и передаваемую в качестве аргумента в `unwrap_or_else`. Мы рассмотрим замыкания более подробно в [главе 13](#). В данный момент, вам просто нужно знать, что `unwrap_or_else` передаст внутреннее значение `Err`, которое в этом случае является статической строкой `not enough arguments`, которое мы добавили в листинге 12-9, в наше замыкание как аргумент `err` указанное между вертикальными линиями. Код в замыкании может затем использовать значение `err` при выполнении.

Мы добавили новую строку `use`, чтобы подключить `process` из стандартной библиотеки в область видимости. Код в замыкании, который будет запущен в случае ошибки содержит только две строчки: мы печатаем значение `err` и затем вызываем `process::exit`. Функция `process::exit` немедленно остановит программу и вернёт номер, который был передан в качестве кода состояния выхода. Это похоже на обработку с помощью макроса `panic!`, которую мы использовали в листинге 12-8, но мы больше не получаем весь дополнительный вывод. Давай попробуем:

```
$ cargo run
Compiling minigrep v0.1.0 (file:///projects/minigrep)
  Finished dev [unoptimized + debuginfo] target(s) in 0.48s
    Running `target/debug/minigrep`
Problem parsing arguments: not enough arguments
```

Замечательно! Этот вывод намного удобнее для наших пользователей.

Извлечение логики из `main`

Теперь, когда мы закончили рефакторинг разбора конфигурации, давайте обратимся к логике программы. Как мы указали в разделе «[Разделение ответственности в бинарных проектах](#)»<!-- -->, мы извлечём функцию с именем `run`, которая будет содержать всю логику, присутствующую в настоящее время в функции `main` и которая не связана с настройкой конфигурации или обработкой ошибок. Когда мы закончим, то `main` будет краткой, легко проверяемой и мы сможем написать тесты для всей остальной логики.

Код 12-11 демонстрирует извлечённую логику в функцию `run`. Мы делаем маленькое, инкрементальное приближение к извлечению функции. Код всё ещё сосредоточен в файле `src/main.rs`:

Файл: `src/main.rs`

```
fn main() {
    // --snip--

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    run(config);
}

fn run(config: Config) {
    let contents = fs::read_to_string(config.filename)
        .expect("Something went wrong reading the file");

    println!("With text:\n{}", contents);
}

// --snip--
```

Листинг 12-11. Извлечение функции `run`, содержащей остальную логику программы

Функция `run` теперь содержит всю оставшуюся логику из `main`, начиная от чтения файла. Функция `run` принимает экземпляр `Config` как аргумент.

Возврат ошибок из функции `run`

Оставшаяся логика программы выделена в функцию `run`, где мы можем улучшить обработку ошибок как мы уже делали с `Config::new` в листинге 12-9. Вместо того, чтобы позволить программе паниковать с помощью вызова `expect`, функция `run` вернёт `Result<T, E>`, если что-то пойдёт не так. Это позволит далее консолидировать логику обработки ошибок в `main` удобным способом. Листинг 12-12 показывает изменения, которые мы должны внести в сигнатуру и тело `run`.

Файл: src/main.rs

```
use std::error::Error;

// --snip--

fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.filename)?;
    println!("With text:\n{}", contents);
    Ok(())
}
```

Листинг 12-12. Изменение функции `run` для возврата `Result`

Здесь мы сделали три значительных изменения. Во-первых, мы изменили тип возвращаемого значения функции `run` на `Result<(), Box<dyn Error>>`. Эта функция ранее возвращала тип `()` и мы сохраняли его как значение, возвращаемое в случае `Ok`.

Для типа ошибки мы использовали *объект типаж* `Box<dyn Error>` (и вверху мы подключили тип `std::error::Error` в область видимости с помощью оператора `use`). Мы рассмотрим типажи объектов в [главе 17](#). Сейчас просто знайте, что `Box<dyn Error>` означает, что функция будет возвращать тип реализующий типаж `Error`, но не нужно указывать, какой именно будет тип возвращаемого значения. Это даёт возможность возвращать значения ошибок, которые могут быть разных типов в разных случаях. Ключевое слово `dyn` сокращение для слова «динамический».

Во-вторых, мы убрали вызов `expect` в пользу использования оператора `?`, как мы обсудили в [главе 9](#). Скорее, чем вызывать `panic!` в случае ошибки, оператор `?` вернёт значение ошибки из текущей функции для вызывающего, чтобы он её обработал.

В-третьих, функция `run` теперь возвращает значение `Ok` в случае успеха. В сигнатуре функции `run` объявлен успешный тип как `()`, который означает, что нам нужно обернуть значение единичного типа в значение `Ok`. Данный синтаксис `Ok(())` поначалу может показаться немного странным, но использование `()` выглядит как идиоматический способ указать, что мы вызываем `run` для его побочных эффектов; он не возвращает значение, которое нам нужно.

Когда вы запустите этот код, он скомпилируется, но отобразит предупреждение:

```
$ cargo run the poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
warning: unused `Result` that must be used
--> src/main.rs:19:5
   |
19 |     run(config);
   |     ^^^^^^^^^^
   |
   = note: #[warn(unused_must_use)] on by default
   = note: this `Result` may be an `Err` variant, which should be handled

warning: `minigrep` (bin "minigrep") generated 1 warning
Finished dev [unoptimized + debuginfo] target(s) in 0.71s
Running `target/debug/minigrep the poem.txt`
Searching for the
In file poem.txt
With text:
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!
```

Rust говорит, что наш код проигнорировал `Result` значение и значение `Result` может указывать на то, что произошла ошибка. Но мы не проверяем, была ли ошибка и компилятор напоминает нам, что мы, вероятно, хотели здесь выполнить

некоторый код обработки ошибок! Давайте исправим эту проблему сейчас.

Обработка ошибок, возвращённых из `run` в `main`

Мы будем проверять и обрабатывать ошибки используя методику, аналогичную той, которую мы использовали для `Config::new` в листинге 12-10, но с небольшой разницей:

Файл: `src/main.rs`

```
fn main() {
    // --snip--

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    if let Err(e) = run(config) {
        println!("Application error: {}", e);

        process::exit(1);
    }
}
```

Мы используем `if let` вместо `unwrap_or_else` чтобы проверить, возвращает ли `run` значение `Err` и вызывается `process::exit(1)`, если это так. Функция `run` не возвращает значение, которое мы хотим развернуть методом `unwrap`, таким же образом как `Config::new` возвращает экземпляр `Config`. Так как `run` возвращает `()` в случае успеха и мы заботимся только об обнаружении ошибки, то нам не нужно вызывать `unwrap_or_else`, чтобы вернуть развернутое значение, потому что оно будет только `()`.

Тело функций `if let` и `unwrap_or_else` одинаковы в обоих случаях: мы печатаем ошибку и выходим.

Разделение кода на библиотечный крейт

Наш проект `minigrep` пока выглядит хорошо! Теперь мы разделим файл `src/main.rs` и поместим некоторый код в файл `src/lib.rs`, чтобы мы могли его тестировать и чтобы в файле `src/main.rs` было меньшее количество функциональных обязанностей.

Давайте перенесём весь код не относящийся к функции `main` из файла `src/main.rs` в новый файл `src/lib.rs`:

- Определение функции `run`.
- Соответствующие инструкции `use`.
- Определение структуры `Config`.
- Определение функции `Config::new`.

Содержимое `src/lib.rs` должно иметь сигнатуры, показанные в листинге 12-13 (мы опустили тела функций для краткости). Обратите внимание, что код не будет компилироваться пока мы не изменим `src/main.rs` в листинге 12-14.

Файл: `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> {
        // --snip--
    }
}

pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    // --snip--
}
```



Листинг 12-13. Перемещение `Config` и `run` в `src/lib.rs`

Мы добавили спецификатор доступа `pub` к структуре `Config`, а также её полям, к методу `new` и функции `run`. Теперь у нас есть API, функционал которого мы сможем протестировать.

Теперь нам нужно подключить код, который мы переместили в `src/lib.rs`, в область видимости бинарного крейта внутри `src/main.rs`, как показано в листинге 12-14.

Файл: `src/main.rs`

```
use std::env;
use std::process;

use minigrep::Config;

fn main() {
    // --snip--
    if let Err(e) = minigrep::run(config) {
        // --snip--
    }
}
```

Листинг 12-14. Использование крейта библиотеки `minigrep` внутри `src/main.rs`

Мы добавляем `use minigrep::Config` для подключения типа `Config` из крейта библиотеки в область видимости бинарного крейта и добавляем к имени функции `run` префикс нашего крейта. Теперь все функции должны быть подключены и должны работать. Запустите программу с `cargo run` и убедитесь, что все работает правильно.

Уф! Было много работы, но мы настроены на будущий успех. Теперь проще обрабатывать ошибки и мы сделали код более модульным. С этого момента почти вся наша работа будет выполняться внутри `src/lib.rs`.

Давайте воспользуемся этой новой модульностью, сделав что-то, что было бы трудно со старым кодом, но легко с новым кодом: мы напишем несколько тестов!

Развитие функциональности библиотеки разработкой на основе тестов

Теперь, когда мы извлекли логику в `src/lib.rs` и оставили разбор аргументов командной строки и обработку ошибок в `src/main.rs`, стало гораздо проще писать тесты для основной функциональности нашего кода. Мы можем вызывать функции напрямую с различными аргументами и проверить возвращаемые значения без необходимости вызова нашего двоичного файла из командной строки.

В этом разделе в программу `minigrep` мы добавим логику поиска с использованием процесса разработки через тестирование (TDD). Это техника разработки программного обеспечения следует этим шагам:

1. Напишите тест, который не прошёл и запустите его, чтобы убедиться, что он не прошёл по той причине, которую вы ожидаете.
2. Пишите или изменяйте ровно столько кода, чтобы успешно выполнился новый тест.
3. Модифицируйте код, который вы только что добавили или изменили и убедитесь, что тесты продолжают проходить.
4. Повторите с шага 1!

Этот процесс является лишь одним из многих способов написания программного обеспечения, но TDD также может помочь дизайну кода. Написание теста перед написанием кода, который делает тестовый прогон помогает поддерживать высокое покрытие тестированием в течение всего процесса.

Мы протестируем реализацию функциональности, которая делает поиск строки запроса в содержимом файла и создание списка строк, соответствующих запросу. Мы добавим эту функциональность в функцию под названием `search`.

Написание теста с ошибкой

Поскольку они нам больше не нужны, давайте удалим строки с `println!`, которые мы использовали для проверки поведения программы в `src/lib.rs` и `src/main.rs`. Затем в `src/lib.rs` мы добавим модуль `tests` с тестовой функцией, как делали это в [главе 11](#). Тестовая функция определяет поведение, которое мы хотим проверить в функции `search`. Она должна принимать запрос и текст для поиска, а возвращать только те строки из текста, которые содержат запрос. В листинге 12-15 показан этот тест, он пока не компилируется.

Файл: 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));
    }
}
```



Листинг 12-15. Создание теста с ошибкой для функции `search`, которую мы хотели бы получить

Этот тест ищет строку `"duct"`. Текст, который мы ищем состоит из трёх строк, только одна из которых содержит `"duct"` (обратите внимание, что обратная косая черта после открывающей двойной кавычки говорит Rust не помещать символ новой строки в начало содержимого этого строкового литерала). Мы утверждаем, что значение, возвращаемое функцией `search`, содержит только ожидаемую нами строку.

Мы не можем запустить этот тест и увидеть сбой, потому что тест даже не компилируется: функции `search` ещё не существует! Так что мы добавим код, чтобы тест компилировался и запускался, написав определение функции `search`, которая всегда возвращает пустой вектор, как показано в листинге 12-16. Потом тест должен скомпилироваться и потерпеть неудачу при запуске, потому что пустой вектор не равен вектору, содержащему строку `"safe, fast, productive."`

Файл: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    vec![]
}
```

Листинг 12-16. Определение функции `search`, достаточное, чтобы тест скомпилировался

Заметьте, что в сигнатуре `search` нужно явно указать время жизни `'a` для аргумента `contents` и возвращаемого значения. Напомним из [Главы 10](#), что

параметры времени жизни указывают с временем жизни какого аргумента связано время жизни возвращаемого значения. В данном случае мы говорим, что возвращаемый вектор должен содержать срезы строк, ссылающиеся на содержимое аргумента **contents** (а не аргумента **query**).

Другими словами, мы говорим Rust, что данные, возвращаемые функцией **search**, будут жить до тех пор, пока живут данные, переданные в функцию **search** через аргумент **contents**. Это важно! Чтобы ссылки были действительными, данные, на которые ссылаются с помощью срезов тоже должны быть действительными; если компилятор предполагает, что мы делаем строковые срезы переменной **query**, а не переменной **contents**, он неправильно выполнит проверку безопасности.

Если мы забудем аннотации времени жизни и попробуем скомпилировать эту функцию, то получим следующую ошибку:

```
$ cargo build
Compiling minigrep v0.1.0 (file:///projects/minigrep)
error[E0106]: missing lifetime specifier
--> src/lib.rs:28:51
  |
28 | pub fn search(query: &str, contents: &str) -> Vec<&str> {
  |           -----           -----           ^ expected named
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`
help: consider introducing a named lifetime parameter
  |
28 | pub fn search<'a>(query: &'a str, contents: &'a str) -> Vec<&'a str> {
  |           +++++           ++           ++           ++
For more information about this error, try `rustc --explain E0106`.
error: could not compile `minigrep` due to previous error
```

Rust не может понять, какой из двух аргументов нам нужен, поэтому нужно сказать ему об этом. Так как **contents** является тем аргументом, который содержит весь наш текст, и мы хотим вернуть части этого текста, которые совпали при поиске, мы понимаем, что **contents** является аргументом, который должен быть связан с возвращаемым значением временем жизни.

Другие языки программирования не требуют от вас связывания в сигнатуре аргументов с возвращаемыми значениями. Хотя сейчас это может показаться странным, со временем станет понятнее. Можете сравнить этот пример с разделом «[Проверка ссылок с временами жизни](#)» главы 10.

Запустим тест:

```
$ cargo test
Compiling minigrep v0.1.0 (file:///projects/minigrep)
  Finished test [unoptimized + debuginfo] target(s) in 0.97s
    Running unitests (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 1 test
test tests::one_result ... FAILED

failures:

---- tests::one_result stdout ----
thread 'main' panicked at 'assertion failed: `!(left == right)`
  left: `["safe, fast, productive."]`,
  right: `[]`', src/lib.rs:44:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
  tests::one_result

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Отлично. Наш тест не сработал, как мы и ожидали. Давайте сделаем так, чтобы он срабатывал!

Написание кода для прохождения теста

Сейчас наш тест не проходит, потому что мы всегда возвращаем пустой вектор. Чтобы исправить это и реализовать **search**, наша программа должна выполнить следующие шаги:

- Итерироваться по каждой строке содержимого.
- Проверить, содержит ли данная строка искомую.
- Если это так, добавить её в список значений, которые мы возвращаем.
- Если это не так, ничего не делать.
- Вернуть список результатов.

Давайте проработаем каждый шаг, начиная с перебора строк.

Перебор строк с помощью метода `lines`

В Rust есть полезный метод для построчной итерации строк, удобно названный `lines`, как показано в листинге 12-17. Обратите внимание, код пока не компилируется.

Файл: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        // do something with line
    }
}
```



Листинг 12-17: Итерация по каждой строке из `contents`

Метод `lines` возвращает итератор. Мы поговорим об итераторах в разделе [Глава 13](#), но вспомните, что вы видели этот способ использования итератора в [Листинге 3-5](#), где мы использовали цикл `for` с итератором, чтобы выполнить некоторый код для каждого элемента в коллекции.

Поиск в каждой строке текста запроса

Далее мы проверяем, содержит ли текущая строка нашу искомую строку. К счастью, у строк есть полезный метод `contains`, который именно это и делает! Добавьте вызов метода `contains` в функции `search`, как показано в листинге 12-18. Обратите внимание, что это все ещё не компилируется.

Файл: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        if line.contains(query) {
            // do something with line
        }
    }
}
```



Листинг 12-18. Добавление проверки, содержит ли `query` в строке

Сохранение совпавшей строки

Нам также нужен способ хранить строки, содержащие искомую строку. Для этого мы можем создать изменяемый вектор перед циклом `for` и вызывать метод `push` для сохранения `line` в векторе. После цикла `for` мы возвращаем вектор, как показано в листинге 12-19.

Файл: 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
}
```

Листинг 12-19. Сохранение совпадающих строк, чтобы вернуть их

Теперь функция `search` должна возвратить только строки, содержащие `query`, и тест должен пройти. Запустим его:

```
$ cargo test
Compiling minigrep v0.1.0 (file:///projects/minigrep)
  Finished test [unoptimized + debuginfo] target(s) in 1.22s
    Running unitests (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 1 test
test tests::one_result ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

    Running unitests (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests minigrep

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Наш тест пройден, значит он работает!

На этом этапе мы могли бы рассмотреть возможности изменения реализации функции поиска, сохраняя прохождение тестов и поддерживая имеющуюся функциональность. Код в функции поиска не так уж плох, но он не использует некоторые полезные функции итераторов. Вернёмся к этому примеру в [главе 13](#), где будем исследовать итераторы подробно, и посмотрим как его улучшить.

Использование функции `search` в функции `run`

Теперь, когда функция `search` работает и протестирована, нужно вызвать `search` из нашей функции `run`. Нам нужно передать значение `config.query` и `contents`, которые `run` читает из файла, в функцию `search`. Тогда `run` напечатает каждую строку, возвращаемую из `search`:

Файл: src/lib.rs

```
pub fn run(config: Config) -> Result<(), Box

```

Мы все ещё используем цикл `for` для возврата каждой строки из функции `search` и её печати.

Теперь вся программа должна работать! Давайте попробуем сначала запустить её со словом, которое должно вернуть только одну строчку из стихотворения Эмили Дикинсон «frog»:

```
$ cargo run frog poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.38s
Running `target/debug/minigrep frog poem.txt`
How public, like a frog
```

Здорово! Теперь давайте попробуем слово, которое будет соответствовать нескольким строкам, например «body»:

```
$ cargo run body poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
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 удостоверимся, что мы не получаем никаких строк, когда ищем слово, отсутствующее в стихотворении, например «monomorphization»:

```
$ cargo run monomorphization poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
Running `target/debug/minigrep monomorphization poem.txt`
```

Отлично! Мы создали собственную мини-версию классического инструмента и научились тому, как структурировать приложения. Мы также немного узнали о файловом вводе и выводе, временах жизни, тестировании и разборе аргументов

командной строки.

Чтобы завершить этот проект, мы кратко продемонстрируем пару вещей: как работать с переменными окружения и как печатать в стандартный поток ошибок, обе из которых полезны при написании консольных программ.

Работа с переменными окружения

Мы улучшим `minigrep`, добавив дополнительную функцию: опцию для поиск без учёта регистра, которую пользователь может включить с помощью переменной среды окружения. Мы могли бы сделать эту функцию параметром командной строки и потребовать, чтобы пользователи вводили бы её каждый раз при её применении, но вместо этого мы будем использовать переменную среды окружения. Это позволит нашим пользователям устанавливать переменную среды один раз и все поиски будут не чувствительны к регистру в этом терминальном сеансе.

Написание ошибочного теста для функции `search` с учётом регистра

Мы хотим добавить новую функцию `search_case_insensitive`, которую мы будем вызывать, когда переменная окружения включена. Мы продолжим следовать процессу TDD, поэтому первый шаг - это снова написать не проходящий тест. Мы добавим новый тест для новой функции `search_case_insensitive` и переименуем наш старый тест из `one_result` в `case_sensitive`, чтобы прояснить различия между двумя тестами, как показано в листинге 12-20.

Файл: `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)
        );
    }
}
```

Листинг 12-20. Добавление нового не проходящего теста для функции поиска нечувствительной к регистру, которую мы собираемся добавить

Обратите внимание, что мы также отредактировали содержимое переменной `contents` из старого теста. Мы добавили новую строку с текстом `"Duct tape."`, используя заглавную D, которая не должна соответствовать запросу `"duct"` при поиске с учётом регистра. Такое изменение старого теста помогает избежать случайного нарушения функциональности поиска чувствительного к регистру, который мы уже реализовали. Этот тест должен пройти сейчас и должен продолжать выполняться успешно, пока мы работаем над поиском без учёта регистра.

Новый тест для поиска нечувствительного к регистру использует `"rUsT"` качестве строки запроса. В функции `search_case_insensitive`, которую мы собираемся

реализовать, запрос `"rUsT"` должен соответствовать строке содержащей `"Rust:"` с большой буквы R и соответствовать строке `"Trust me."`, хотя обе имеют разные регистры из запроса. Это наш не проходящий тест, он не компилируется, потому что мы ещё не определили функцию `search_case_insensitive`. Не стесняйтесь добавлять скелет реализация, которая всегда возвращает пустой вектор, аналогично тому, как мы это делали для функции `search` в листинге 12-16, чтобы увидеть компиляцию теста и его сбой.

Реализация функции `search_case_insensitive`

Функция `search_case_insensitive`, показанная в листинге 12-21, будет почти такая же, как функция `search`. Разница лишь в том, что текст будет в нижнем регистре для `query` и для каждой `line`, так что для любого регистра входных аргументов это будет тот же случай, когда мы проверяем, содержит ли строка запрос.

Файл: src/lib.rs

```
pub fn search_case_insensitive<'a>(
    query: &str,
    contents: &'a str,
) -> Vec<&'a str> {
    let query = query.to_lowercase();
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.to_lowercase().contains(&query) {
            results.push(line);
        }
    }

    results
}
```

Листинг 12-21. Определение функции `search_case_insensitive` с уменьшением регистра строки запроса и строки содержимого перед их сравнением

Сначала преобразуем в нижний регистр строку `query` и сохраняем её в затенённой переменной с тем же именем. Вызов `to_lowercase` для строки запроса необходим, так что независимо от того, будет ли пользовательский запрос `"rust"`, `"RUST"`, `"Rust"` или `"rUsT"`, мы будем преобразовывать запрос к `"rust"` и делать значение нечувствительным к регистру. Хотя `to_lowercase` будет обрабатывать Unicode, он не будет точным на 100%. Если бы мы писали реальное приложение, мы бы хотели

проделать здесь немного больше работы, но этот раздел посвящён переменным среды, а не Unicode, поэтому мы оставим это здесь.

Обратите внимание, что `query` теперь имеет тип `String`, а не срез строки, потому что вызов `to_lowercase` создаёт новые данные, а не ссылается на существующие. К примеру, запрос: `"rUsT"` это срез строки не содержащий строчных букв `u` или `t`, которые мы можем использовать, поэтому мы должны выделить новую `String`, содержащую `«rust»`. Когда мы передаём запрос `query` в качестве аргумента метода `contains`, нам нужно добавить амперсанд, поскольку сигнатура `contains`, определена для приёмы среза строки.

Затем мы добавляем вызов `to_lowercase` у каждой строки `line`, прежде чем проверять, содержит ли она `query` из всех строчных символов. Теперь, когда мы преобразовали `line` и `query` в нижний регистр, мы найдём совпадения независимо от того, в каком регистре находится переменная с запросом.

Давайте посмотрим, проходит ли эта реализация тесты:

```
$ cargo test
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished test [unoptimized + debuginfo] target(s) in 1.33s
Running unitests (target/debug/deps/minigrep-9cd200e5fac0fc94)

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;
finished in 0.00s

Running unitests (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests minigrep

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Отлично! Тесты прошли. Теперь давайте вызовем новую функцию

`search_case_insensitive` из функции `run`. Во-первых, мы добавим параметр конфигурации в структуру `Config` для переключения между поиском с учётом регистра и без учёта регистра. Добавление этого поля приведёт к ошибкам компилятора, потому что мы ещё нигде не инициализируем это поле:

Файл: src/lib.rs

```
pub struct Config {
    pub query: String,
    pub filename: String,
    pub ignore_case: bool,
}
```



Обратите внимание, что мы добавили поле `case_sensitive`, которое содержит логическое значение. Далее нам нужна функция `run`, чтобы проверить значение поля `case_sensitive` и использовать его, чтобы решить, вызывать ли функцию `search` или функцию `search_case_insensitive`, как показано в листинге 12-22. Обратите внимание, что код все ещё не компилируется.

Файл: src/lib.rs

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.filename)?;

    let results = if config.ignore_case {
        search_case_insensitive(&config.query, &contents)
    } else {
        search(&config.query, &contents)
    };

    for line in results {
        println!("{}", line);
    }

    Ok(())
}
```



Листинг 12-22. Вызов либо `search`, либо `search_case_insensitive` на основе значения в

`config.case_sensitive`

Наконец, нам нужно проверить переменную среды. Функции для работы с переменными среды находятся в модуле `env` стандартной библиотеки, поэтому мы хотим подключить этот модуль в область видимости с помощью строки `use std::env;` в верхней части `src/lib.rs`. Затем мы будем использовать функцию `var` из

модуля `env` для проверки переменной среды с именем `CASE_INSENSITIVE`, как показано в листинге 12-23.

Файл: `src/lib.rs`

```
use std::env;
// --snip--

impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            filename,
            ignore_case,
        })
    }
}
```

Листинг 12-23. Проверка переменной среды с именем `CASE_INSENSITIVE`

Здесь мы создаём новую переменную `case_sensitive`. Чтобы установить её значение, мы вызываем функцию `env::var` и передаём ей имя переменной окружения `CASE_INSENSITIVE`. Функция `env::var` возвращает `Result`, который будет успешным вариантом `Ok` содержащий значение переменной среды, если переменная среды установлена. Он вернёт вариант `Err`, если переменная окружения не установлена.

Мы используем метод `is_err` у `Result`, чтобы проверить возвращается ли ошибка и следовательно, переменная среды не установлена, что означает, что *должен* выполняться чувствительный к регистру поиск. Если для переменной среды `CASE_INSENSITIVE` что-либо задано, то `is_err` вернёт значение `false` и программа выполнит поиск без учёта регистра. Мы не заботимся о *значении* переменной среды, нас интересует только установлена она или нет, поэтому мы проверяем `is_err`, а не используем `unwrap`, `expect` или любой другой метод, который мы видели у `Result`.

Мы передаём значение переменной `case_sensitive` экземпляру `Config`, чтобы функция `run` могла прочитать это значение и решить, следует ли вызывать `search` или `search_case_insensitive`, как мы реализовали в листинге 12-22.

Давайте попробуем! Во-первых, мы запустим нашу программу без установленной переменной среды и с помощью значения запроса `to`, который должен соответствовать любой строке, содержащей слово «to» в нижнем регистре:

```
$ cargo run to poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
  Finished dev [unoptimized + debuginfo] target(s) in 0.0s
    Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
```

Похоже, все ещё работает! Теперь давайте запустим программу с `CASE_INSENSITIVE`, установленным в `1`, но с тем же значением запроса `to`.

Если вы используете PowerShell, вам нужно установить переменную среды и запустить программу двумя командами, а не одной:

```
PS> $Env:CASE_INSENSITIVE=1; cargo run to poem.txt
```

Это заставит переменную окружения `CASE_INSENSITIVE` сохраниться до конца сеанса работы консоли. Переменную можно отключить с помощью команды `Remove-Item`:

```
PS> Remove-Item Env:CASE_INSENSITIVE
```

Мы должны получить строки, содержащие «to», которые могут иметь заглавные буквы:

```
$ CASE_INSENSITIVE=1 cargo run to poem.txt
  Finished dev [unoptimized + debuginfo] target(s) in 0.0s
    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, мы также получили строки, содержащие «To»! Наша программа `minigrep` теперь может выполнять поиск без учёта регистра, управляемая переменной среды. Теперь вы знаете, как управлять параметрами, заданными с помощью

аргументов командной строки или переменных среды.

Некоторые программы допускают использование аргументов и переменных среды для одной и той же конфигурации. В таких случаях программы решают, что из них имеет больший приоритет. Для другого самостоятельного упражнения попробуйте управлять нечувствительностью к регистру с помощью аргумента командной строки или переменной окружения. Решите, должен ли приоритет иметь аргумент командной строки или переменная среды, если программа запускается с установленным параметром для нечувствительного к регистру поиска и установленного для поиска с учётом регистра.

Модуль `std::env` содержит много других полезных функций для работы с переменными среды: ознакомьтесь с его документацией, чтобы узнать доступные.

Запись сообщений ошибок в поток ошибок вместо стандартного потока вывода

В данный момент мы записываем весь наш вывод в терминал, используя функцию `println!`. В большинстве терминалов предоставлено два вида вывода: *стандартный поток вывода* (`stdout`) для общей информации и *стандартный поток ошибок* (`stderr`) для сообщений об ошибках. Это различие позволяет пользователям выбирать, направлять ли успешный вывод программы в файл, но при этом выводить сообщения об ошибках на экран.

Функция `println!` может печатать только в стандартный вывод, поэтому мы должны использовать что-то ещё для печати в стандартный поток ошибок.

Проверка, куда записываются ошибки

Во-первых, давайте посмотрим, как содержимое, напечатанное из `minigrep` в настоящее время записывается в стандартный вывод, включая любые сообщения об ошибках, которые мы хотим вместо этого записать в стандартный поток ошибок. Мы сделаем это, перенаправив стандартный поток вывода в файл, а также намеренно вызовем ошибку. Мы не будем перенаправлять стандартный поток ошибок, поэтому любой контент, отправленный в поток стандартных ошибок будет продолжать отображаться на экране.

Ожидается, что программы командной строки будут отправлять сообщения об ошибках в стандартный поток ошибок, поэтому мы все равно можем видеть сообщения об ошибках на экране, даже если мы перенаправляем стандартный поток вывода в файл. Наша программа в настоящее время не ведёт себя правильно: мы увидим, что она сохраняет вывод сообщения об ошибке в файл!

Способ продемонстрировать это поведение - запустите программу с `>` и именем файла `output.txt` в который мы хотим перенаправить стандартный поток вывода. Мы не передадим никаких аргументов, что должно вызвать внутри ошибку:

```
$ cargo run > output.txt
```

Синтаксис `>` указывает оболочке записывать содержимое стандартного вывода в `output.txt` вместо экрана. Мы не увидели сообщение об ошибке, которое мы ожидали увидеть на экране, так что это означает, что оно должно быть в файле. Вот что

содержит `output.txt`:

```
Problem parsing arguments: not enough arguments
```

Да, наше сообщение об ошибке выводится в стандартный вывод. Гораздо более полезнее, чтобы подобные сообщения об ошибках печатались в стандартной поток ошибок, поэтому в файл попадают только данные из успешного запуска. Мы поменяем это.

Печать ошибок в поток ошибок

Мы будем использовать код в листинге 12-24, чтобы изменить способ вывода сообщений об ошибках. Из-за рефакторинга, который мы делали ранее в этой главе, весь код, который печатает сообщения об ошибках, находится в одной функции: `main`. Стандартная библиотека предоставляет макрос `eprintln!`, который печатает в стандартный поток ошибок, поэтому давайте изменим два места, где мы вызывали `println!` для печати ошибок, чтобы использовать `eprintln!` вместо этого.

Файл: `src/main.rs`

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    if let Err(e) = minigrep::run(config) {
        eprintln!("Application error: {}", e);

        process::exit(1);
    }
}
```

Листинг 12-24. Запись сообщений об ошибках в стандартный поток ошибок вместо потока стандартного вывода используя макрос `eprintln!`

После изменения `println!` на `eprintln!`, давайте снова запустим программу без каких-либо аргументов и перенаправим стандартный вывод с помощью `>`:

```
$ cargo run > output.txt  
Problem parsing arguments: not enough arguments
```

Теперь мы видим ошибку на экране и *output.txt* не содержит ничего, что мы ожидаем от программы командной строки.

Давайте снова запустим программу с аргументами, которые не вызывают ошибку, но все же перенаправляют стандартный вывод в файл, например так:

```
$ cargo run to poem.txt > output.txt
```

Мы не увидим никакого вывода в терминал, а *output.txt* будет содержать наши результаты:

Файл: *output.txt*

```
Are you nobody, too?  
How dreary to be somebody!
```

Это демонстрирует, что в зависимости от ситуации мы теперь используем стандартный поток вывода для успешного текста и стандартный поток ошибок для вывода ошибок.

Итоги

В этой главе были повторены некоторые основные концепции, которые вы изучили до сих пор и было рассказано, как выполнять обычные операции ввода-вывода в Rust. Используя аргументы командной строки, файлы, переменные среды и макрос `eprintln!` для печати ошибок и вы теперь готовы писать приложения командной строки. Благодаря использованию концепций из предыдущих главах ваш код будет хорошо организован, будет эффективно хранить данные в соответствующих структурах, хорошо обрабатывать ошибки и хорошо тестироваться.

Далее мы рассмотрим некоторые возможности Rust, на которые повлияли функциональные языки: замыкания и итераторы.

Функциональные возможности языка: Итераторы и Замыкания

Дизайн языка Rust получил вдохновение из многих существующих языков и техник, и одним из существенных влияний является *функциональное программирование*. Программирование в функциональном стиле часто включает использование функций в качестве значений, передаваемых в аргументах, возвращаемых из других функций, назначаемых переменным для последующего выполнения и так далее.

В этой главе мы не будем обсуждать вопрос о том, чем является функциональное программирование или не является, вместо этого мы обсудим некоторые функции Rust, которые похожи на функции во многих языках, часто называемых функциональными.

Более подробно мы поговорим про:

- *замыкания*, функциональная конструкция, которую вы можете хранить в переменной,
- *итераторы* - способ обработки последовательности элементов,
- как с помощью этих двух языковых конструкций можно улучшить операции ввода/вывода в Главе 12,
- производительности этих конструкций (они быстрее, чем вы думаете!).

На другие функции Rust, такие как сопоставление с образцом и перечисления, о которых мы говорили в других главах, также влияет функциональный стиль. Освоение замыканий и итераторов является важной частью написания идиоматического, быстрого кода на Rust, поэтому мы посвятим им всю эту главу.

Замыкания: анонимные функции, которые могут захватывать окружение

Замыкания Rust - это анонимные функции, которые вы можете сохранить в переменной или передать в качестве аргументов другим функциям. Вы можете создать замыкание в одном месте, а затем вызвать замыкание, чтобы вычислить его в другом контексте. В отличие от функций, замыкания могут захватывать значения из области видимости где они определены. Мы продемонстрируем, как эти возможности замыканий позволяют повторно использовать код и настраивать поведение.

Создание обобщённого поведения используя замыкания

Рассмотрим пример демонстрирующий ситуацию, где сохранение замыкания удобно для его более позднего выполнения. Мы также поговорим про синтаксис замыканий, выведение типов и типажи.

Рассмотрим гипотетическую ситуацию, что мы работаем в стартапе, где создаём приложение для генерации индивидуальных планов тренировок. Серверная часть приложения создаётся на Rust и алгоритм генерирующий план тренировки, учитывает многие различные факторы, такие как возраст пользователя приложения, индекс массы тела, предпочтительные задания, последние тренировки и индекс интенсивности, которые указываются. При проектировании приложения конкретные алгоритмы реализаций не важны. Важно, чтобы различные расчёты не занимали много времени. Мы хотим использовать этот алгоритм только когда нам нужно, и делать это только один раз, чтобы не заставлять пользователя ждать больше, чем требуется.

Мы будем эмулировать работу алгоритма расчёта параметров с помощью функции `simulated_expensive_calculation` листинга 13-1, которая печатает `calculating slowly...`, ждёт две секунды и возвращает любое переданное ей число как результат эмулированного расчёта.

Файл: src/main.rs

Листинг 13-1: Функция которая выполняет гипотетические расчёты длительностью в 2 секунды

Теперь рассмотрим содержание функции `main`, которая содержит важные для этого

примера части нашего приложения. Данная функция представляет код, который будет вызван, когда пользователь запросит свой план занятий. Так как взаимодействие с клиентской частью программы не связано с использованием замыканий, мы жёстко закодируем входные данные и выводимые на печать результаты.

Требуемые входные данные следующие:

- индекс интенсивности (intensity) пользователя, которое указывается пользователем, когда он запрашивает тренировку: показывает хотят ли они тренировку низкой интенсивности или тренировку высокой интенсивности,
- случайное число, которое будет генерировать разнообразие в планах тренировок.

В результате программа напечатает рекомендованный план занятий. Листинг 13-2 показывает код использованной функции `main`.

Файл: `src/main.rs`

```
let expensive_closure = |num: u32| -> u32 {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num
};
```

Листинг 13-2: Функция `main` содержащая симуляцию пользовательского ввода данных и генерацию случайного числа

Мы для простоты жёстко закодировали в коде значение переменной `simulated_user_specified_value` равным 10 и переменной `simulated_random_number` равным 7. В реальном приложении, мы бы получали значение интенсивности от пользователя и использовали бы пакет `rand` для генерации случайного числа, как мы делали в примере игры «Угадай число» из Главы 2. Функция `main` вызывает функцию `generate_workout` с эмулированными входными значениями.

Теперь когда есть контекст в котором мы будем работать, давайте займёмся алгоритмом. Функция `generate_workout` в листинге 13-3 содержит всю основную логику работы программы, которая наиболее важна в примере. Остальные изменения в коде будут сделаны внутри этой функции:

Файл: `src/main.rs`

```
let example_closure = |x| x;  
  
let s = example_closure(String::from("hello"));  
let n = example_closure(5);
```

Листинг 13-3: Логика программы печатающей план тренировки на основании введённых данных и вызова функции `simulated_expensive_calculation`

Код листинга 13-3 несколько раз вызывает функцию с медленными расчётами. Первый блок `if` дважды вызывает `simulated_expensive_calculation`, блок `if` внутри внешнего `else` вообще не вызывает её, а код внутри второго `else` вызывает её один раз.

Желаемое поведение функции `generate_workout` состоит в том, чтобы сначала проверить, хочет ли пользователь тренировку с низкой интенсивностью (обозначается числом менее 25) или тренировку с высокой интенсивностью (число от 25 или более).

Планы тренировок низкой интенсивности будут рекомендовать несколько отжиманий и приседаний на основе сложного алгоритма, который мы моделируем.

Если пользователь хочет высокую интенсивность тренировок, то выполняется дополнительная логика. Если случайный образом выбираемое число равно 3, то предлагается сделать перерыв и освежиться. Иначе на основании сложного алгоритма пользователь получит задание бегать несколько минут.

Данный код работает, так как этого хочет заказчик сейчас. Но допустим, что команда специалистов по анализу данных в будущем решит, что нам нужно внести некоторые изменения в то, как мы вызываем функцию

`simulated_expensive_calculation`. Чтобы упростить обновление кода, когда возникнут подобного рода желания, нам стоит переделать код так, чтобы функцию `simulated_expensive_calculation` вызывали только раз. Мы также хотим избавиться от места, где мы вызываем эту функцию дважды и при этом не добавлять какие-либо другие вызовы этой функции в коде. Иными словами, мы не хотим вызывать функцию, если её результат не нужен, и одновременно мы всё равно хотим вызвать её только один раз.

Рефакторинг используя функции

Можно было бы реструктуризовать нашу программу разными способами. Сначала, мы попробуем извлечь повторные вызовы функции

`simulated_expensive_calculation` в переменную, как показано в листинге 13-4.

Файл: src/main.rs

Листинг 13-4: Извлечение вызова функции `simulated_expensive_calculation` в одно место и сохранение результата в переменной `expensive_result`

Это изменение объединяет все вызовы `simulated_expensive_calculation` и решает проблему первого `if` блока, который вызывает функцию дважды без необходимости. К сожалению, сейчас мы вызываем эту функцию и ждём результат во всех случаях, включая внутренний блок `if`, который вообще не использует значение результата.

Мы хотим сослаться на `simulated_expensive_calculation` только один раз в `generate_workout`, но отложить дорогостоящие вычисления до того момента, где нам действительно понадобится результат. Этот случай как раз для замыканий!

Рефакторинг с помощью замыкания для сохранение кода, который может быть запущен позднее

Вместо того, чтобы всегда выполнять функцию `simulated_expensive_calculation` перед блоком `if`, мы можем определить замыкание и сохранить это замыкание в переменной вместо того, чтобы сохранять результат вызова функции, как показано в листинге 13-5. Можно переместить все тело `simulated_expensive_calculation` в замыкание представленное здесь.

Файл: src/main.rs

Листинг 13-5: Определение замыкания и сохранение его в переменной `expensive_closure`

Определение замыкания начинается после символа `=`, который присваивает его переменной `expensive_closure`. Замыкание мы начинаем с пары палочек (vertical pipes (`|`)). Внутри этой конструкции мы определяем входные параметры замыкания. Такой синтаксис был выбран под влиянием языков Ruby и Smalltalk. Данное замыкание имеет параметр `num`. Если нужно несколько параметров, то они разделяются запятыми: `| param1, param2 |`.

После параметров замыкания, в фигурных скобках идёт тело функции замыкания.

Фигурные скобки могут не использоваться, если код функции состоит только из одной строчки кода. После закрытия фигурных скобок необходим символ `;` для завершения выражения. Значение возвращаемое последней строчкой тела замыкания (`num`) будет являться значением, которое будет возвращено из замыкания, когда оно будет вызвано, поэтому данная строка не содержит точку с запятой (`;`) как и в теле любой функции.

Обратите внимание, что выражение `let` означает, что `expensive_closure` содержит *определение* анонимной функции, а не *значение результата выполнения* анонимной функции. Напомним, что мы используем замыкание, потому что хотим определить код для вызова в одной точке, сохранить этот код и фактически вызвать его на более позднем этапе. Код, который мы хотим вызвать, теперь хранится в переменной `expensive_closure`.

Теперь, после определения замыкания можно изменить код в блоках `if`: вызвать код замыкания чтобы его выполнить и получить результирующее значение. Вызов замыкания очень похож на вызов функции. Мы определяем имя переменной, которая содержит определение замыкания и в скобках указываем аргументы, которые мы хотим использовать для вызова, как показано в листинге 13-6.

Файл: `src/main.rs`

```
{{{#rustdoc_include ../../listings/ch13-functional-features/listing-13-06/src/main.rs:here}}}
```

Листинг 13-6: Вызов объявленного замыкания `expensive_closure`

Теперь дорогостоящие вычисления определены только в одном месте, мы знаем как они должны выполняться, но мы выполняем их только тогда, когда нам нужен результат.

Тем не менее, у нас осталась ещё одна проблема в листинге 13-3: мы все ещё дважды вызываем замыкание в первом блоке `if`, что дважды вызовет медленный код и заставит пользователя ждать дважды выполнение функции. Мы могли бы решить эту проблему, создав локальную переменную для этого блока `if`, чтобы хранить результат вызова замыкания, но замыкания предоставляют нам другое решение. Мы немного поговорим об этом решении. Но сначала давайте поговорим о том, почему в определении замыкания нет аннотаций типов и признаков, связанных с замыканиями.

Выведение типа замыкания и аннотация

Замыкания не требуют аннотирования типов параметров или возвращаемого значения, как это делают функции `fn`. Аннотации типов требуются для функций, потому что они являются частью явного интерфейса, предоставляемого пользователям. Жёсткое определение интерфейса функций вытекает из их открытости и важно для обеспечения того, чтобы все понимали какие типы значений использует и возвращает функция. Но замыкания, в отличие от функций, не используются в открытых интерфейсах: они хранятся в переменных и используются без их наименования и предоставления пользователям нашей библиотеки.

Замыкания являются обычно короткими и актуальны только в узком контексте, а не в любом произвольном сценарии. В этих ограниченных контекстах компилятор может надёжно определить типы входных параметров и возвращаемый тип, подобно тому, как он может выводить типы большинства переменных.

Заставлять программистов аннотировать типы в этих небольших анонимных функциях было бы раздражающим и в значительной степени излишним при наличии информации уже имеющейся у компилятора.

Как и в случае с переменными, можно добавить аннотации типов, если мы хотим повысить явность и ясность кода за счёт большей детализации, чем это строго необходимо. Аннотирование типов для замыкания, которое мы определили в листинге 13-5, будет выглядеть как определение, показанное в листинге 13-7.

Файл: `src/main.rs`

Листинг 13-7: Добавление необязательных аннотаций типов для параметров и типов возвращаемых значений у замыкания

С добавленными аннотациями типов синтаксис замыканий выглядит более похожим на синтаксис функций. Ниже приводится сравнение синтаксиса функции, которая добавляет 1 к своему параметру и несколько эквивалентных замыканий, которые имеют аналогичное поведение. Мы добавили несколько пробелов для выравнивания соответствующих частей. Это показывает, как синтаксис замыкания похож на синтаксис функции, за исключением использования вертикальных палочек и необязательного синтаксиса:

```
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 ;
```

Первая строка показывает определение функции, а вторая строка показывает полностью аннотированное определение замыкания. В третьей строке удаляются аннотации типов из определения замыкания, а в четвёртой строке удаляются скобки, которые являются необязательными, поскольку тело замыкания имеет только одно выражение. Все это допустимые определения, которые будут вызывать одинаковое поведение при вызове. Вызов замыканий `add_one_v3` и `add_one_v4` требует, чтобы они могли скомпилироваться, поэтому при компиляции типы будут выводиться на основе их использования.

Определения замыканий будут иметь конкретные типы выведенные только один раз - для каждого из её параметров и для возвращаемого значения. Например, в листинге 13-8 показано определение короткого замыкания, которое просто возвращает значение, которое оно получает в качестве параметра. Это замыкание не очень полезно, за исключением целей этого примера. Обратите внимание, что мы не добавили никаких аннотаций типов в определение: если мы затем попытаемся дважды вызвать замыкание, используя тип `String` в качестве аргумента в первый раз и тип `u32` во второй раз, то мы получим ошибку.

Файл: src/main.rs

Листинг 13-8: Попытка вызвать замыкание, типы у которого выводятся двумя разными типами



Компилятор вернёт нам вот такую ошибку:

```
$ cargo run
Compiling rectangles v0.1.0 (file:///projects/rectangles)
error[E0507]: cannot move out of `value`, a captured variable in an `FnMut` closure
--> src/main.rs:27:30
24     let value = String::from("by key called");
25     ----- captured outer variable
26     list.sort_by_key(|r| {
27         ----- sort_operations.push(value);
28         |           ^^^^^^ move occurs because `value` has
type `String`, which does not implement the `Copy` trait
29         |           r.width
30         });
31         |----- captured by this `FnMut` closure

For more information about this error, try `rustc --explain E0507`.
error: could not compile `rectangles` due to previous error
```

Когда мы в первый раз вызываем `example_closure` со значением типа `String`, компилятор выводит тип для `x` и возвращаемого значения как `String`. Эти типы затем привязываются к замыканию в `example_closure` и мы получаем ошибку типа, если пытаемся использовать другой тип с тем же замыканием.

Сохранение замыканий используя обобщённые типы и типажи

`Fn`

Вернёмся к нашему приложению для создания тренировок. В листинге 13-6 наш код по-прежнему вызывал замыкание с дорогостоящим вычислением больше, чем это требовалось. Один из вариантов решения этой проблемы - сохранить результат дорогостоящего замыкания в переменной для повторного использования и использовать переменную в каждом месте, где нам нужен результат, вместо повторного вызова замыкания. Однако этот метод может привести к многократному повторению кода.

К счастью, нам доступно другое решение. Можно создать структуру, которая будет содержать замыкание и значение вызова замыкания. Структура будет выполнять замыкание только если нам понадобится результирующее значение, а результирующее значение будет кэшировать, поэтому остальной части нашего кода не понадобится отвечать за сохранение и повторное использование

результата. Вы можете знать этот шаблон как *memoization* (запоминание) или *lazy evaluation* (ленивое вычисление).

Чтобы создать структуру, которая содержит замыкание, нам нужно указать тип замыкания, потому что определение структуры должно описывать типы каждого из его полей. Каждый экземпляр замыкания имеет свой уникальный анонимный тип: то есть, даже если два замыкания имеют одну и ту же сигнатуру, их типы по-прежнему считаются разными. Для определения структур, перечислений или параметров функций, которые используют замыкания, мы используем обобщённые типы и ограничения типажей, как мы обсуждали в Главе 10.

Типажи `Fn` входят в состав стандартной библиотеки. Все замыкания реализуют один из типажей: `Fn`, `FnMut` или `FnOnce`. Мы поговорим о различиях между ними в разделе "Захват переменных окружения замыканиями"; в данном примере мы можем использовать типаж `Fn`.

Мы добавляем типы в описание ограничений типажа `Fn` для описания типов параметров и возвращаемого значения, которое замыкания должны иметь для того, чтобы соответствовать данному ограничению типажа. В данном случае, наше замыкание имеет тип параметра `u32` и возвращает тип `u32`, поэтому сигнатуру ограничения типажа мы описываем как `Fn(u32) -> u32`.

Листинг 13-9 показывает определение структуры `Cacher` содержащей замыкание и необязательное значение результата:

Файл: src/main.rs

Листинг 13-9: Определение структуры `Cacher` содержащей замыкание в поле `calculation` и дополнительный результат в поле `value`

Структура `Cacher` имеет поле `calculation` обобщённого типа `T`. Ограничение типажа для `T` требует, чтобы обобщённый тип соответствовал замыканию: требование определяется типажом `Fn`. Любое замыкание, которые мы хотим сохранить в поле `calculation` должно иметь один параметр типа `u32` (указанный внутри круглых скобок после `Fn`) и должно возвращать тип `u32` (указанный после `->`).

Примечание. Функции также могут реализовывать все три типажа `Fn`. Если то, что мы хотим сделать, не требует захвата значения из среды, мы можем

использовать функцию, а не замыкание, где нам нужно что-то, что реализует типаж **Fn**.

Поле **value** имеет тип **Option<u32>**. Перед выполнением замыкания, значение **value** будет **None**. Когда код, использующий **Cacher**, запрашивает *результат выполнения* замыкания, **Cacher** выполнит код замыкания и сохранит результат внутри варианта **Some** в поле **value**. Затем, если код снова запросит результат замыкания, вместо повторного выполнения замыкания, **Cacher** вернёт результат, хранящийся в варианте **Some**.

Логика вычисления поля **value**, которую мы только что описали определена в листинге 13-10.

Файл: src/main.rs

```
let v1 = vec![1, 2, 3];
let v1_iter = v1.iter();
```

Листинг 13-10: Логика кэширования в **Cacher**

Мы хотим, чтобы **Cacher** управлял значениями полей структуры, а не позволял бы вызывающему коду потенциально изменять значения в этих полях напрямую, поэтому эти поля являются закрытыми.

Функция **Cacher::new** принимает обобщённый параметр **T**, который мы определили как имеющий то же ограничения типажа, что и структура **Cacher**. Затем **Cacher::new** возвращает экземпляр **Cacher** содержащий замыкание, указанное в поле **calculation** и значение **None** в поле **value**, потому что мы ещё не выполнили замыкание.

Когда вызывающему коду требуется результат выполнения замыкания, то вместо непосредственного вызова замыкания, он вызовет метод **value**. Этот метод проверяет, есть ли у нас уже готовое значение **self.value** в **Some**; если есть, то метод возвращает значение в **Some** без повторного выполнения замыкания.

Если же поле **self.value** имеет значение **None**, то код вызывает замыкание сохранённое в поле **self.calculation** и результат работы записывается в поле **self.value** для будущего использования и, затем, полученное значение возвращается вызывающему коду.

Листинг 13-11 демонстрирует использование структуры `Cacher` в функции `generate_workout` из листинга 13-6.

Файл: src/main.rs

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();

for val in v1_iter {
    println!("Got: {}", val);
}
```

Листинг 13-11: Использование `Cacher` в функции `generate_workout` для абстрагирования логики кэширования

Вместо непосредственного сохранения замыкания в переменной мы сохраняем новый экземпляр `Cacher`, который содержит замыкание. Затем в каждом месте, где мы хотим получить результат, мы вызываем метод `value` у экземпляра `Cacher`. Мы можем вызывать метод `value` столько раз, сколько захотим или не вызывать его вообще, а дорогостоящие вычисления будут выполняться максимум один раз.

Попробуйте запустить эту программу с помощью функции `main` из листинга 13-2. Измените значения в переменных `simulated_user_specified_value` и `simulated_random_number`, чтобы убедиться, что во всех случаях в различных блоках `if` и `else` текст `calculating slowly...` появляется только один раз и только при необходимости. `Cacher` заботится о логике, необходимой для обеспечения того, чтобы мы не вызывали дорогостоящие вычисления больше, чем нужно, чтобы мы могли сосредоточиться на бизнес-логике в `generate_workout`.

Ограничения реализации `Cacher`

Кэширование значений - это обычно полезное поведение, которое мы могли бы использовать в других частях нашего кода с другими замыканиями. Однако в текущей реализации `Cacher` есть две проблемы, которые затрудняют его повторное использование в различных контекстах.

Первая проблема заключается в том, что экземпляр `Cacher` предполагает, что он всегда получит одно и то же значение параметра `arg` для метода `value`. То есть этот тест `Cacher` не пройдёт:

```
{[#rustdoc_include ../listings/ch13-functional-features/no-listing-01-failing-cacher-test/src/lib.rs:here]}
```

Этот тест создаёт новый экземпляр **Cacher** с замыканием, которое возвращает переданное в него значение. Мы вызываем метод **value** для этого экземпляра **Cacher** со значением **arg** равным 1 и затем значением **arg** равным 2 и ожидаем, что вызов **value** со значением **arg** равным 2 вернёт 2.

Запустите этот тест с реализацией **Cacher** в листинге 13-9 и листинге 13-10, тест **assert_eq!** завершится неудачно с данным сообщением:

```
{#[include ../listings/ch13-functional-features/no-listing-01-failing-cacher-test/output.txt]}
```

Проблема в том, что при первом вызове **c.value** с аргументом 1, экземпляр **Cacher** сохранит значение **Some(1)** в **self.value**. После этого, неважно какие будут входные параметры метода **value**, он всегда будет возвращать 1.

Попробуйте изменить **Cacher** так, чтобы он сохранял **HashMap**, а не единственное значение. Ключами **HashMap** будут значения **arg**, которые передаются, а значениями будут результаты вызова замыкания для соответствующего ключа. Вместо того, чтобы проверять имеет ли **self.value** значение **Some** или **None**, функция **value** будет делать поиск **arg** в **HashMap** и возвращать значение, если оно присутствует. Если оно отсутствует, то **Cacher** вызовет замыкание и сохранит полученное значение в **HashMap** связанное со значением из **arg**.

Вторая проблема с текущей реализацией **Cacher** заключается в том, что она принимает только замыкания, которые имеют один входной параметр типа **u32** и возвращают **u32**. Мы могли бы захотеть, к примеру, кэшировать результаты замыканий, которые берут строковый срез и возвращают значение типа **usize**. Чтобы решить эту проблему, попробуйте ввести обобщённые параметры, чтобы повысить гибкость функциональности **Cacher**.

Захват переменных окружения с помощью замыкания

В примере генератора тренировок мы использовали только замыкания в качестве встроенных анонимных функций. Однако у замыканий есть дополнительная возможность, которой нет у функций: они могут захватывать своё окружение и получать доступ к переменным из области видимости, в которой они определены.

В листинге 13-12 приведён пример замыкания, хранящегося в переменной `equal_to_x`, в которой используется переменная `x` из ближайшего окружения замыкания.

Файл: src/main.rs

```
{[#rustdoc_include ../../listings/ch13-functional-features/listing-13-12/src/main.rs]}
```

Листинг 13-12: Пример замыкания, которое ссылается на переменную в области видимости

Здесь, хоть `x` и не является одним из параметров `equal_to_x`, замыканию `equal_to_x` разрешено использовать переменную `x`, которая определена в той же области видимости что и `equal_to_x`.

Мы не можем сделать то же самое с функциями; если мы попробуем использовать следующий пример, наш код не скомпилируется:

Файл: src/main.rs

```
{[#rustdoc_include ../../listings/ch13-functional-features/no-listing-02-functions-cant-capture/src/main.rs]}
```

Описание ошибки:

```
{[#include ../../listings/ch13-functional-features/no-listing-02-functions-cant-capture/output.txt]}
```

Компилятор даже напоминает нам, что это работает только с замыканиями!

В случае когда замыкание захватывает значение из своего окружения, оно использует дополнительную память для хранения значений используемых в теле замыкания. Такое использование памяти является накладными расходами, которые мы не хотим платить в общих случаях, там где мы хотим выполнить код, который не захватывает переменные окружения. Поскольку функциям никогда не разрешается захватывать их окружение, то определение и использование функций никогда не повлечёт за собой таких издержек.

Замыкания могут захватывать значения из своего окружения тремя способами, которые напрямую соответствуют трём способам, которыми функция может принимать параметр: забирать во владение, получать изменяемое заимствование и получать неизменяемое заимствование. Эти способы закодированы в трёх

типажах `Fn` следующим образом:

- замыкания типажа `FnOnce` потребляют переменные, которые они захватывают из окружающего контекста, известного как *окружение (environment)* замыкания. Чтобы использовать захваченные переменные, замыкание должно стать владельцем этих переменных и переместить их в замыкание, когда оно определено. Часть имени `Once` отражает тот факт, что замыкание не может владеть одними и теми же переменными более одного раза, поэтому его можно вызывать только один раз.
- замыкания типажа `FnMut` могут изменять значения переменных из окружения, поскольку они заимствуют изменяемые значения.
- замыкания типажа `Fn` заимствуют значения из окружения без их изменения.

Когда вы создаёте замыкание Rust определяет какой типаж использовать, основываясь на том как замыкание использует значения из окружения. Все замыкания реализуют `FnOnce`, потому что все они могут быть вызваны хотя бы один раз. Замыкания, которые не перемещают захваченные переменные, также реализуют `FnMut`, а замыкания которым не требуется изменяемый доступ к захваченным переменным, также реализуют `Fn`. В листинге 13-12 замыкание `equal_to_x` заимствует `x` как неизменяемый (поэтому `equal_to_x` имеет типаж `Fn`), поскольку тело замыкания должно только читать значение в `x`.

Если вы хотите, чтобы замыкание стало владельцем значений, которые оно использует в окружении, то вы можете использовать ключевое слово `move` перед списком параметров. Этот метод в основном полезен при передаче замыкания в новый поток для перемещения данных, чтобы они принадлежали новому потоку.

Примечание: замыкания с `move` могут по-прежнему реализовывать `Fn` или `FnMut`, даже если они захватывают переменные при перемещении. Это связано с тем, что типаж, реализуемый типом замыкания, определяется тем, что замыкание делает с захваченными значениями, а не тем, как оно их захватывает. Ключевое слово `move` указывает только как замыкание захватывает значения.

У нас будет больше примеров замыканий с `move` в Главе 16, когда мы поговорим про одновременное выполнение множества задач (concurrency). А пока будем довольствоваться кодом из листинга 13-12 с ключевым словом `move`, добавленным в определение замыкания и использующим векторы вместо целых чисел, поскольку

целые числа можно копировать, а не перемещать. Обратите внимание, что этот код ещё не компилируется.

Файл: src/main.rs

```
{#{rustdoc_include .../listings/ch13-functional-features/no-listing-03-move-  
closures/src/main.rs}}
```

Мы получаем следующую ошибку:

```
{#include .../listings/ch13-functional-features/no-listing-03-move-  
closures/output.txt}}
```

Значение `x` перемещается в замыкание, когда замыкание определено, потому что мы добавили ключевое слово `move`. Замыкание становится владельцем `x` и `main` больше не может использовать `x` в макросе `println!`. Удаление `println!` исправит ошибку в этом примере.

В большинстве случаев при указании одного из ограничений типажа `Fn` можно начать с типажа `Fn` и компилятор сообщит нужен ли вам `FnMut` или `FnOnce` в зависимости от того, что происходит в теле замыкания.

Для иллюстрации ситуаций, когда замыкания захватывающие своё окружение (как параметры функций) являются полезными, давайте перейдём к следующей теме: итераторы.

Обработка группы элементов с помощью итераторов

Шаблон итератора позволяет выполнять некоторые задачи над последовательностью элементов. Итератор отвечает за логику итерации по каждому элементу и определяет, когда последовательность завершилась. Когда вы используете итераторы, вам не нужно переопределять эту логику самостоятельно.

Итераторы в Rust *ленивы*, то есть они не делают ничего, пока вы не вызовете методы, которые потребляют итератор. Например, код в листинге 13-13 создаёт итератор по элементам вектора `v1`, вызывая метод `iter`, определённый для `Vec<T>`. Сам по себе этот код не делает ничего полезного.

```
{#rustdoc_include ../listings/ch13-functional-features/listing-13-13/src/main.rs:here}
```

Листинг 13-13: Создание итератора

Создав итератор, можно использовать его различными способами. В листинге 3-5 главы 3 мы использовали итераторы с циклами `for`, чтобы выполнить некоторый код для каждого элемента, хотя только вкратце останавливались на том, что делал вызов `iter` до этих пор.

Пример в листинге 13-14 отделяет создание итератора от его использования в цикле `for`. Итератор хранится в переменной `v1_iter`, и в это время итерация не выполняется. Когда цикл `for` вызывается с использованием итератора `v1_iter`, каждый элемент итератора используется в одной итерации цикла, которая выводит значение этого элемента.

```
let v1: Vec<i32> = vec![1, 2, 3];  
v1.iter().map(|x| x + 1);
```

Листинг 13-14: Использование итератора в цикле `for`

В языках, которые не имеют итераторов в стандартной библиотеке, вы, вероятно, написали бы эту же функцию следующим образом: взять переменную со значением 0, использовать её для индексации вектора, чтобы получить значение, и увеличивать её значение в цикле, пока не будет достигнуто общее количество элементов в векторе.

Итераторы делают все эти шаги за вас, сокращая повторяющийся код, который вы потенциально могли бы испортить. Итераторы дают вам больше гибкости для использования одной и той же логики с различными типами последовательностей, а не только со структурами данных, которые можно индексировать, типа векторов. Давайте посмотрим как итераторы это делают.

Типаж `Iterator` и метод `next`

Все итераторы реализуют типаж `Iterator`, который определён в стандартной библиотеке. Его определение выглядит так:

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
    // methods with default implementations elided
}
```

Обратите внимание данное объявление использует новый синтаксис: `type Item` и `Self::Item`, которые определяют *ассоциированный тип* (associated type) с этим типажом. Мы подробнее поговорим о ассоциированных типах в главе 19. Сейчас вам нужно знать, что этот код требует от реализаций типажа `Iterator` определить требуемый им тип `Item` и данный тип `Item` используется в методе `next`. Другими словами, тип `Item` будет являться типом элемента, который возвращает итератор.

Типаж `Iterator` требует, чтобы разработчики определяли только один метод: метод `next`, который возвращает один элемент итератора за раз обёрнутый в вариант `Some` и когда итерация завершена, возвращает `None`.

Мы можем вызвать у итераторов метод `next` напрямую; В листинге 13-15 показано, какие значения возвращаются в результате повторных вызовов `next` на итераторе, созданном из вектора.

Файл: `src/lib.rs`

```
{{{#rustdoc_include ../../listings/ch13-functional-features/listing-13-15/src/lib.rs:here}}}
```

Листинг 13-15: Вызов метода `next` на итераторе

Обратите внимание, что нам нужно сделать переменную `v1_iter` изменяемой: вызов метода `next` итератора изменяет внутреннее состояние итератора, которое итератор использует для отслеживания того, где он находится в последовательности. Другими словами, этот код *потребляет* (*consumes*) или использует итератор. Каждый вызов `next` потребляет элемент из итератора. Нам не нужно было делать изменяемой `v1_iter` при использовании цикла `for`, потому что цикл забрал во владение `v1_iter` и сделал её изменяемой неявно для нас.

Заметьте также, что значения, которые мы получаем при вызовах `next` являются неизменяемыми ссылками на значения в векторе. Метод `iter` создаёт итератор по неизменяемым ссылкам. Если мы хотим создать итератор, который становится владельцем `v1` и возвращает принадлежащие ему значения, мы можем вызвать `into_iter` вместо `iter`. Точно так же, если мы хотим перебирать изменяемые ссылки, мы можем вызвать `iter_mut` вместо `iter`.

Методы, которые потребляют итератор

У типажа `Iterator` есть несколько методов, реализация которых по умолчанию предоставляется стандартной библиотекой; вы можете узнать об этих методах, просмотрев документацию API стандартной библиотеки для `Iterator`. Некоторые из этих методов вызывают `next` в своём определении, поэтому вам необходимо реализовать метод `next` при реализации типажа `Iterator`.

Методы, вызывающие `next`, называются потребляющими адаптерами (*consuming adaptors*), поскольку их вызов использует итератор. Примером потребляющего адаптера является метод `sum`. Он становится владельцем итератора и перемещается по элементам, многократно вызывая `next`, тем самым потребляя итератор. Он выполняет итерацию для каждого элемента и добавляет его к промежуточной сумме, возвращая итоговую сумму после завершения итерации. В листинге 13-16 есть тест, иллюстрирующий использование `sum`:

Файл: `src/lib.rs`

Листинг 13-16: Вызов метода `sum` для получения суммы всех элементов итератора

Мы не можем использовать `v1_iter` после вызова метода `sum`, потому что `sum` забирает во владение итератор у которого вызван метод.

Методы, которые создают другие итераторы

Другие методы, определённые в **Iterator**, известные как *адаптеры итераторов* (*iterator adaptors*), позволяют преобразовывать в разные виды итераторов. Вы можете связать в последовательность несколько вызовов адаптеров итераторов для выполнения сложных действий в удобном виде. Но поскольку все итераторы ленивы, вы должны вызвать один из потребляющих методов, чтобы получить результат работы цепочки адаптеров.

В листинге 13-17 показан пример использования адаптера **map**, который требует замыкание, чтобы применить его к каждому элементу и создать новый итератор. Замыкание здесь создаёт новый итератор, в котором каждый элемент вектора был увеличен на 1. Однако этот код выдаёт предупреждение:

Файл: src/main.rs

```
{{{#rustdoc_include ../listings/ch13-functional-features/listing-13-17/src/main.rs:here}}}
```



Листинг 13-17. Использование адаптера **map** для создания нового итератора

Мы получаем следующее предупреждение:

```
{{{#include ../listings/ch13-functional-features/listing-13-17/output.txt}}}
```

Код в листинге 13-17 ничего не делает; замыкание никогда не вызывается. Предупреждение напоминает нам, почему это так: адаптеры итераторов ленивы и мы должны поглотить итератор чтобы увидеть результат.

Чтобы исправить это и поглотить итератор, мы воспользуемся методом **collect**, который мы уже использовали в главе 12 с **env::args** в листинге 12-1. Этот метод использует итератор и собирает полученные значения в коллекцию указанного типа.

В листинге 13-18 мы собираем результаты итерации по итератору, который возвращается из вызова **map** в вектор. Этот вектор будет содержать каждый элемент из исходного вектора, увеличенный на 1.

Файл: src/main.rs

```
fn main() {
    let config = Config::new(env::args()).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    // --snip--
}
```

Листинг 13-18: Вызов метода `map` для создания нового итератора и затем вызов метода `collect` для создания и использования нового итератора, чтобы создать новый вектор с данными

Поскольку `map` принимает замыкание, мы можем указать любую операцию, которую хотим выполнить с каждым элементом. Это отличный пример того, как замыкания позволяют настраивать какое-то поведение при повторном использовании итерационного поведения, предоставляемого типажом `Iterator`.

Использование замыканий, которые захватывают переменные окружения

Теперь, когда мы представили итераторы, мы можем продемонстрировать общее использование замыканий, которые захватывают их окружение используя адаптер итератора `filter`. Метод `filter` в итераторе принимает замыкание, которое берет каждый элемент из итератора и возвращает логическое значение. Если замыкание возвращает `true`, значение будет включено в итератор, созданный методом `filter`. Если замыкание возвращает `false`, значение не будет включено в итоговый итератор.

В листинге 13-19 мы используем метод `filter` с замыканием, которое захватывает переменную `shoe_size` из своего окружения, чтобы выполнить итерацию по коллекции экземпляров структуры `Shoe`. Он вернёт только ту обувь, которая имеет указанный размер.

Файл: src/lib.rs

```
use std::env;
use std::error::Error;
use std::fs;

pub struct Config {
    pub query: String,
    pub filename: String,
```

```
    pub ignore_case: bool,
}

// ANCHOR: here
impl Config {
    pub fn new(
        mut args: impl Iterator<Item = String>,
    ) -> Result<Config, &'static str> {
    // --snip--
    // ANCHOR_END: here
    if args.len() < 3 {
        return Err("not enough arguments");
    }

    let query = args[1].clone();
    let filename = args[2].clone();

    let ignore_case = env::var("IGNORE_CASE").is_ok();

    Ok(Config {
        query,
        filename,
        ignore_case,
    })
}
}

pub fn run(config: Config) -> Result<(), Box
```

```
    }

    results
}

pub fn search_case_insensitive<'a>(
    query: &str,
    contents: &'a str,
) -> Vec<&'a str> {
    let query = query.to_lowercase();
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.to_lowercase().contains(&query) {
            results.push(line);
        }
    }

    results
}

#[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)
        );
    }
}
```

```
    }  
}
```

Листинг 13-19: Использование метода `filter` вместе с замыканием, которое захватывает параметр `shoe_size`

Функция `shoes_in_size` принимает в качестве параметров вектор с экземплярами обуви и размер обуви, а возвращает вектор, содержащий только обувь указанного размера.

В теле `shoes_in_my_size` мы вызываем `into_iter` чтобы создать итератор, который становится владельцем вектора. Затем мы вызываем `filter`, чтобы превратить этот итератор в другой, который содержит только элементы, для которых замыкание возвращает `true`.

Замыкание захватывает параметр `shoe_size` из окружения и сравнивает его с размером каждой пары обуви, оставляя только обувь указанного размера. Наконец, вызов `collect` собирает значения, возвращаемые адаптированным итератором, в вектор, возвращаемый функцией.

Тест показывает, что когда мы вызываем `shoes_in_my_size`, мы возвращаем только туфли, размер которых совпадает с указанным нами значением.

Создание собственных итераторов с помощью типажа `Iterator`

Мы показали, что можно создать итератор из вектора, вызвав `iter`, `into_iter` или `iter_mut`. Вы можете создавать итераторы из других типов коллекций в стандартной библиотеке, таких как хэш-таблица. Вы также можете создавать итераторы, которые делают все, что захотите, реализуя типаж `Iterator` для собственных типов. Как упоминалось ранее, единственный метод, который требуется реализовать, - это `next`. Как только вы это сделаете, вы сможете использовать все другие методы, реализация по умолчанию которых предоставляетя `Iterator`!

Чтобы продемонстрировать это, давайте создадим итератор, который будет считать только от 1 до 5. Сначала мы создадим структуру для хранения некоторых значений. Затем мы превратим эту структуру в итератор, реализовав для неё типаж `Iterator` и будем использовать её значения в итераторе.

Листинг 13-20 определяет структуру `Counter` и ассоцииированную функцию `new`,

создающую экземпляры структуры `Counter`:

Файл: `src/lib.rs`

```
use std::env;
use std::error::Error;
use std::fs;

pub struct Config {
    pub query: String,
    pub filename: String,
    pub ignore_case: bool,
}

// ANCHOR: here
impl Config {
    pub fn new(
        mut args: impl Iterator<Item = String>,
    ) -> Result<Config, &'static str> {
        args.next();

        let query = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get a query string"),
        };

        let filename = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get a file name"),
        };

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            filename,
            ignore_case,
        })
    }
}
// ANCHOR_END: here

pub fn run(config: Config) -> Result<(), Box
```

```
for line in results {
    println!("{}", line);
}

Ok(())
}

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
}

pub fn search_case_insensitive<'a>(
    query: &str,
    contents: &'a str,
) -> Vec<&'a str> {
    let query = query.to_lowercase();
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.to_lowercase().contains(&query) {
            results.push(line);
        }
    }

    results
}

#[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)
    );
}
}

```

Листинг 13-20. Определение структуры `Counter` и функции `new`, которая создаёт экземпляры `Counter` с начальным значением 0 для `count`

Структура `Counter` имеет одно поле с именем `count`. Это поле содержит значение `u32`, которое будет отслеживать, где мы находимся в процессе итерации от 1 до 5. Поле `count` является приватным, потому что мы хотим, чтобы реализация `Counter` управляла его значением. Функция `new` обеспечивает такое поведение, чтобы новые экземпляры создавались со значением 0 в поле `count`.

Далее, мы реализуем типаж `Iterator` для структуры `Counter`, определив тело метода `next`, реализуя то, что хотим получить при использовании этого итератора, как это показано в листинге 13-21:

Файл: `src/lib.rs`

```

{{#rustdoc_include ../listings/ch13-functional-features/listing-13-
21/src/lib.rs:here}}

```

Листинг 13-21. Реализация типажа `Iterator` для нашей структуры `Counter`

Мы указываем связанный тип `Item` для нашего итератора как `u32`, то есть итератор возвращает значения `u32`. Опять же, пока не беспокойтесь о ассоциированных типах, мы рассмотрим их в главе 19.

Мы хотим, чтобы наш итератор прибавил 1 к текущему состоянию, поэтому мы инициализировали `count` равным 0, чтобы он сначала возвращал 1. Если значение

`count` меньше 5, `next` будет увеличивать `count` и возвращать текущее значение, обёрнутое в `Some`. Как только `count` станет 5, наш итератор перестанет увеличивать `count` и всегда будет возвращать `None`.

Использование у `Counter` метода итератора `next`

Как только мы реализовали типаж `Iterator`, у нас есть итератор! В листинге 13-22 показан тест, демонстрирующий, что мы можем использовать функциональные возможности итератора нашей структуры `Counter`, напрямую вызывая на ней метод `next`, точно так же, как мы это делали с итератором, созданным из вектора в листинге 13-15.

Файл: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    contents
        .lines()
        .filter(|line| line.contains(query))
        .collect()
}
```

Листинг 13-22. Тестирование функциональной реализации метода `next`

Тест создаёт экземпляр структуры `Counter` в переменной `counter`, затем последовательно вызывает метод `next`, проверяя реализацию необходимого поведения итератора: возвращение чисел от 1 до 5.

Использование других методов типажа `Iterator`

Мы реализовали типаж `Iterator`, написав метод `next`, и можем использовать любые методы `Iterator` для которых в стандартной библиотеке есть реализация по умолчанию, поскольку все они используют функционал метода `next`.

Например, если по какой-то причине мы хотели взять значения, созданные экземпляром `Counter`, сопоставить их со значениями, созданными другим экземпляром `Counter` пропуская первое значение, перемножить каждую пару друг с другом, сохраняя только те результаты, которые делятся на 3 и складывая все полученные значения вместе, мы могли бы сделать это так, как показано в тесте листинга 13-23:

Файл: src/lib.rs

```
{{{#rustdoc_include ../listings/ch13-functional-features/listing-13-23/src/lib.rs:here}}}
```

Листинг 13-23: Использование множества методов типажа `Iterator` в пользовательском итераторе `Counter`

Обратите внимание, что `zip` возвращает только четыре пары; теоретическая пятая пара `(5, None)` никогда не создаётся, поскольку `zip` возвращает `None`, когда любой из его входных итераторов возвращает значение `None`.

Вызовы всех этих методов возможны, потому что мы определил, как работает метод `next`, а стандартная библиотека предоставляет реализации по умолчанию для других методов, которые вызывают `next`.

Улучшение проекта ввода/вывода

Обладая новыми знаниями об итераторах, можно улучшить проект ввода-вывода главы 12 используя итераторы, чтобы сделать места в коде более понятными и краткими. Давайте посмотрим, как итераторы могут улучшить нашу реализацию функции `Config::new` и функции `search`.

Удаление метода `clone` используя итератор

В листинге 12-6 мы добавили код, который использовал срез `String` и создал экземпляр структуры `Config`, взяв по индексам данные из среза и клонировав эти значения, таким образом позволив структуре `Config` владеть значениями. В листинге 13-24 мы повторили реализацию функции `Config::new`, как это было в листинге 12-23:

Файл: src/lib.rs

```
impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            filename,
            ignore_case,
        })
    }
}
```

Листинг 13-24: Воспроизведение функции `Config::new` из листинга 12-23

Ранее мы говорили, что не стоит беспокоиться о неэффективных вызовах `clone`, потому что мы удалим их в будущем. Ну что же, это время пришло!

Нам был нужен вызов `clone` потому что у нас есть срез с элементами `String` в

параметре `args`, но функция `new` не владеет `args`. Чтобы вернуть владение экземпляром `Config`, нам пришлось клонировать значения из полей `query` и `filename` структуры `Config`, поэтому экземпляр `Config` может владеть своими значениями.

Обладая новыми знаниями об итераторах, мы можем изменить функцию `new`, чтобы она стала владельцем итератора аргумента, а не заимствовала срез. Мы будем использовать функциональность итератора вместо кода, который проверяет длину среза и получает данные по индексу. Это делает более понятным, что выполняет функция `Config::new`, потому что итератор получит доступ к значениям.

Как только `Config::new` заберёт во владение итератор и перестанет использовать заимствующие операции индексирования, мы можем переместить значения `String` из итератора в `Config` вместо вызова `clone` и выполнения нового выделения памяти.

Использование возвращённого итератора напрямую

Откройте файл `src/main.rs` проекта ввода-вывода, который должен выглядеть следующим образом:

Файл: `src/main.rs`

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    // --snip--
}
```

Мы изменим начало функции `main`, которая была в листинге 12-24, на код из листинга 13-25. Он не компилируется, пока мы не обновим `Config::new`.

Файл: `src/main.rs`

```
{#rustdoc_include ../listings/ch13-functional-features/listing-13-
25/src/main.rs:here}
```

Листинг 13-25: Передача возвращаемого значения `env::args` в `Config::new`

Функция `env::args` возвращает итератор! Вместо того, чтобы собирать значения итератора в вектор и затем передавать срез в `Config::new`, мы напрямую передаём во владение итератор, возвращённый из `env::args` параметром в `Config::new`.

Далее нам нужно обновить определение `Config::new`. В файле `src/lib.rs` вашего проекта ввода/вывода давайте изменим сигнатуру `Config::new` как показано в листинге 13-26. Код все равно ещё не компилируется, потому что нам нужно обновить тело функции.

Файл: `src/lib.rs`

```
{#{rustdoc_include ../listings/ch13-functional-features/listing-13-26/src/lib.rs:here}}
```

Листинг 13-26: Обновление сигнатуры `Config::new` для ожидания итератора

Документация стандартной библиотеки для функции `env::args` показывает, что типом возвращаемого итератора является `std::env::Args`. Мы обновили сигнатуру функции `Config::new`, поэтому параметр `args` имеет тип `std::env::Args` вместо `&[String]`. Поскольку мы забираем во владение `args` и будем изменять `args` перебирая его элементы, мы можем добавить ключевое слово `mut` в спецификацию параметра `args`, чтобы сделать его изменяемым.

Использование методов типажа `Iterator` вместо индексов

Далее мы вносим изменения в код тела `Config::new`. Стандартная библиотека документации также упоминает, что `std::env::Args` реализует типаж `Iterator`, поэтому мы знаем, что можем вызвать метод `next!`! Листинг 13-27 обновляет код из листинга 12-23 с использованием метода `next!`:

Файл: `src/lib.rs`

```
{#{rustdoc_include ../listings/ch13-functional-features/listing-13-27/src/lib.rs:here}}
```

Листинг 13-27: Новое содержание функции `Config::new` с использованием методов итератора

Помните, что первое значение в возвращаемом значении типа `env::args` ЭТО ИМЯ программы. Мы хотим игнорировать его и перейти к следующему значению,

поэтому сначала мы вызываем `next` и ничего не делаем с возвращаемым значением. Во-вторых, мы вызываем `next`, чтобы получить значение, которое мы хотим поместить в поле `query` структуры `Config`. Если `next` возвращает `Some`, мы используем `match` для извлечения значения. Если он возвращает `None`, это означает, что было передано недостаточно аргументов и мы сразу же выходим со значением `Err`. Мы делаем то же самое для значения `filename`.

Делаем код понятнее с помощью адаптеров итераторов

Мы также можем использовать преимущества итераторов в функции `search` в проекте ввода/вывода, который приводится здесь в листинге 13-28, как это было в листинге 12-19:

Файл: 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
}
```

Листинг 13-28: Реализация функции `search` из листинга 12-19

Мы можем написать этот код более кратко, используя адаптерные методы итератора. Это также позволяет нам избежать использования изменяемого промежуточного вектора `result`. Стиль функционального программирования предпочитает минимизировать количество изменяемых состояний, чтобы сделать код более понятным. Удаление изменяемого состояния может позволить в будущем усовершенствовать и сделать поиск параллельным, потому что нам не нужно управлять одновременным доступом к вектору `results`. Листинг 13-29 показывает это изменение:

Файл: src/lib.rs

```
{{{#rustdoc_include ../listings/ch13-functional-features/listing-13-29/src/lib.rs:here}}}
```

Листинг 13-29: Использование адаптерных методов итератора в реализации функции `search`

Напомним, что целью функции `search` является возвращение всех строк `content`, содержащих `query`. Подобно примеру `filter` в листинге 13-19, этот код использует адаптер `filter` для выбора только тех строк, для которых код `line.contains(query)` возвращает значение `true`. Затем мы собираем совпадающие строки в другой вектор с помощью `collect`. Так намного проще! Не стесняйтесь вносить похожие изменения, чтобы использовать методы итератора в функции `search_case_insensitive`.

Следующий логический вопрос - какой стиль вы должны выбрать в своём собственном коде и почему: тот что был в исходной реализации в листинге 13-28 или версию с использованием итераторов в листинге 13-29. Большинство программистов на Rust предпочитают использовать стиль с итератором. Поначалу немного сложнее освоиться, но как только вы изучите различные адаптерные методы итераторов и то, что они делают, итераторы будет легче понять. Вместо того чтобы возиться с различными элементами цикла и создавать новые векторы, код фокусируется на высокогорневом смысле цикла. Это абстрагирует часть обычного кода, поэтому легче увидеть концепции, уникальные для этого кода, такие как условие фильтрации, которое должен пройти каждый элемент в итераторе.

Но действительно ли эти две реализации эквивалентны? Интуитивно кажется, что более низкоуровневый цикл будет быстрее. Давайте поговорим о производительности.

Сравнение производительности циклов и итераторов

Чтобы определить, что лучше использовать циклы или итераторы, нужно знать, какая реализация быстрее: версия функции `search` с явным циклом `for` или версия с итераторами.

Мы выполнили тест производительности, разместив всё содержимое книги (*"The Adventures of Sherlock Holmes"* by Sir Arthur Conan Doyle) в строку типа `String` и поискали слово *the* в её содержимом. Вот результаты теста функции `search` с использованием цикла `for` и с использованием итераторов:

```
test bench_search_for ... bench: 19,620,300 ns/iter (+/- 915,700)
test bench_search_iter ... bench: 19,234,900 ns/iter (+/- 657,200)
```

Версия с использованием итераторов была немного быстрее! Мы не будем приводить здесь непосредственно код теста, поскольку идея не в том, чтобы доказать, что решения в точности эквивалентны, а в том, чтобы получить общее представление о том, как эти две реализации близки по производительности.

Для более исчерпывающего теста, вам нужно проверить различные тексты разных размеров в качестве содержимого для `contents`, разные слова и слова различной длины в качестве `query` и всевозможные другие варианты. Дело в том, что итераторы, будучи высокоуровневой абстракцией, компилируются примерно в тот же код, как если бы вы написали его низкоуровневый вариант самостоятельно. Итераторы - это одна из *абстракций с нулевой стоимостью* (zero-cost abstractions) в Rust, под которой мы подразумеваем, что использование абстракции не накладывает дополнительных расходов во время выполнения. Аналогично тому, как Бьёрн Страуструп, дизайнер и разработчик C++, определяет *нулевые накладные расходы* (zero-overhead) в книге *"Foundations of C++"* (2012):

В целом, реализация C++ подчиняется принципу отсутствия накладных расходов: за то, чем вы не пользуетесь, платить не нужно. И далее: тот код, что вы используете, нельзя сделать ещё лучше.

В качестве другого примера приведём код, взятый из аудио декодера. Алгоритм декодирования использует математическую операцию линейного предсказания для оценки будущих значений на основе линейной функции предыдущих выборок. Код

использует комбинирование вызовов итератора для выполнения математических вычислений для трёх переменных в области видимости: срез данных `buffer`, массив из 12 коэффициентов `coefficients` и число для сдвига данных в переменной `qlp_shift`. Переменные определены в примере, но не имеют начальных значений. Хотя этот код не имеет большого значения вне контекста, он является кратким, реальным примером того, как Rust переводит идеи высокого уровня в код низкого уровня.

```
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;
}
```

Чтобы вычислить значение переменной `prediction`, этот код перебирает каждое из 12 значений в переменной `coefficients` и использует метод `zip` для объединения значений коэффициентов с предыдущими 12 значениями в переменной `buffer`. Затем, для каждой пары мы перемножаем значения, суммируем все результаты и у суммы сдвигаем биты вправо в переменную `qlp_shift`.

Для вычислений в таких приложениях, как аудио декодеры, часто требуется производительность. Здесь мы создаём итератор, используя два адаптера, впоследствии потребляющих значение. В какой ассемблерный код будет компилироваться этот код на Rust? На момент написания этой главы он компилируется в то же самое, что вы написали бы руками. Не существует цикла, соответствующего итерации по значениям в «коэффициентах» `coefficients`: Rust знает, что существует двенадцать итераций, поэтому он «разворачивает» цикл. *Разворачивание* - это оптимизация, которая устраняет издержки кода управления циклом и вместо этого генерирует повторяющийся код для каждой итерации цикла.

Все коэффициенты сохраняются в регистрах, что означает очень быстрый доступ к значениям. Нет никаких проверок границ доступа к массиву во время выполнения. Все эти оптимизации, которые может применить Rust, делают полученный код чрезвычайно эффективным. Теперь, когда вы это знаете, используйте итераторы и

замыкания без страха! Они представляют код в более высокоуровневом виде, но без потери производительности во время выполнения.

Итоги

Замыкания (closures) и итераторы (iterators) это возможности Rust, вдохновлённые идеями функциональных языков. Они позволяют Rust ясно выражать идеи высокого уровня с производительностью низкоуровневого кода. Реализации замыканий и итераторов таковы, что нет влияния на производительность выполнения кода. Это одна из целей Rust, направленных на обеспечение абстракций с нулевой стоимостью (zero-cost abstractions).

Теперь, когда мы улучшили представление кода в нашем проекте, рассмотрим некоторые возможности, которые нам предоставляет **cargo** для публикации нашего кода в репозитории.

Больше о Cargo и Crates.io

До сих пор мы использовали только самые основные возможности Cargo для сборки, запуска и тестирования нашего кода, но он может делать гораздо больше. В этой главе мы обсудим некоторые другие, более продвинутые возможности, чтобы показать вам, как сделать следующее:

- Настраивать вашу сборку с помощью профилей выпуска
- Публиковать библиотеки на [crates.io](#)
- Организовывать крупные проекты с рабочими пространствами
- Устанавливать бинарные файлы из [crates.io](#)
- Расширять Cargo с помощью пользовательских команд

Cargo может сделать даже больше, чем мы расскажем в этой главе, поэтому полное описание всех его возможностей смотрите в [его документации](#).

Настройка сборок с профилями релизов

В Rust *профили выпуска* - это предопределённые и настраиваемые профили с различными конфигурациями, которые позволяют программисту лучше контролировать различные параметры компиляции кода. Каждый профиль настраивается независимо от других.

Cargo имеет два основных профиля: профиль `dev` используемый Cargo при запуске `cargo build` и профиль `release` используемый Cargo при запуске `cargo build --release`. Профиль `dev` определён со значениями по умолчанию для разработки, а профиль `release` имеет значения по умолчанию для релиз сборок.

Эти имена профилей могут быть знакомы по результатам ваших сборок:

```
$ cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
$ cargo build --release
    Finished release [optimized] target(s) in 0.0s
```

Выводы для `dev` и `release`, показанные при сборке указывают, что компилятор использует разные профили.

Cargo имеет настройки по умолчанию для каждого из профилей, которые применяются, когда в файле проекта *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 для параметра `opt-level` для профилей `dev` и `release`:

Файл: *Cargo.toml*

```
[profile.dev]
opt-level = 0

[profile.release]
opt-level = 3
```

Параметр `opt-level` управляет количеством оптимизаций, которые Rust будет применять к вашему коду, в диапазоне от 0 до 3. Применение дополнительных оптимизаций увеличивает время компиляции, поэтому, если вы находитесь в разработке и часто компилируете свой код, вам понадобится быстрая компиляция, даже если полученный код работает медленнее. Вот почему по умолчанию `opt-`

`level` для `dev` равно значению `0`. Когда вы будете готовы выпустить свой код, лучше потратить больше времени на компиляцию. Вы будете компилировать в режиме релиза только один раз, но вы будете запускать скомпилированную программу много раз, поэтому режим релиза тратит больше времени на компиляцию кода, который работает быстрее. Вот почему по умолчанию `opt-level` для профиля `release` является значением `3`.

Вы можете переопределить любой параметр по умолчанию, добавив для него другое значение в `Cargo.toml`. Например, если мы хотим использовать уровень оптимизации 1 в профиле разработки, мы можем добавить эти две строки в файл `Cargo.toml` нашего проекта:

Файл: `Cargo.toml`

```
[profile.dev]
opt-level = 1
```

Этот код переопределяет настройку по умолчанию `0`. Теперь, когда мы запустим `cargo build`, Cargo будет использовать значения по умолчанию для профиля `dev` плюс нашу настройку для `opt-level`. Поскольку мы установили для `opt-level` значение `1`, Cargo будет применять больше оптимизаций, чем по умолчанию, но не так много, как при сборке релиза.

Полный список параметров конфигурации и значений по умолчанию для каждого профиля вы можете найти в [документации Cargo](#).

Публикация библиотеки в Crates.io

Мы использовали пакеты из [crates.io](#) в качестве зависимостей нашего проекта, но вы также можете поделиться своим кодом с другими людьми, опубликовав свои собственные пакеты. Реестр библиотек по адресу [crates.io](#) распространяет исходный код ваших пакетов, поэтому он в основном размещает код с открытым исходным кодом.

В Rust и Cargo есть возможности, которые облегчают использование и поиск вашего опубликованного пакета. Далее мы поговорим о некоторых из этих возможностей, а затем объясним, как опубликовать пакет.

Создание полезных комментариев к документации

Аккуратное документирование ваших пакетов поможет другим пользователям знать, 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 есть особый вид комментариев к документации, который обычно называется *комментарием к документации*, который генерирует документацию HTML. HTML-код отображает содержимое комментариев к документации для публичных элементов API, предназначенных для программистов, заинтересованных в знании того, как *использовать* вашу библиотеку, в отличие от того, как она *реализована*.

Комментарии к документации используют три слеша, `///` вместо двух и поддерживают нотацию Markdown для форматирования текста. Размещайте комментарии к документации непосредственно перед элементом, который они документируют. В листинге 14-1 показаны комментарии к документации для функции `add_one` в библиотеке с именем `my_crate`:

Файл: `src/lib.rs`

```
/// Adds one to the number given.  
///  
/// # Examples  
///  
/// ``  
/// let arg = 5;  
/// let answer = my_crate::add_one(arg);  
///  
/// assert_eq!(6, answer);  
/// ``  
pub fn add_one(x: i32) -> i32 {  
    x + 1  
}
```

Листинг 14-1: Комментарий к документации для функции

Здесь мы даём описание того, что делает функция `add_one`, начинаем раздел с заголовка `Examples`, а затем предоставляем код, который демонстрирует, как использовать функцию `add_one`. Мы можем сгенерировать документацию HTML из этого комментария к документации, запустив `cargo doc`. Эта команда запускает инструмент `rustdoc`, поставляемый с Rust, и помещает сгенерированную HTML-документацию в каталог `target/doc`.

Для удобства, запустив `cargo doc --open`, мы создадим HTML для документации вашей текущей библиотеки (а также документацию для всех зависимостей вашей библиотеки) и откроем результат в веб-браузере. Перейдите к функции `add_one` и вы увидите, как отображается текст в комментариях к документации, что показано на рисунке 14-1:

The screenshot shows the Rust documentation for the `my_crate::add_one` function. On the left, there's a sidebar with links for `my_crate`, `Functions`, `add_one`, `Crates`, and `my_crate`. The main content area has a search bar at the top with the placeholder "Click or press 'S' to search, '?' for more options...". Below the search bar, the title is "Function `my_crate::add_one`". To the right of the title are "[−]" and "[src]". The function signature is `pub fn add_one(x: i32) -> i32`. A note below the signature says "[−] Adds one to the number given.". A section titled "Examples" follows, containing the following Rust code:

```
let arg = 5;
let answer = my_crate::add_one(arg);

assert_eq!(6, answer);
```

Рисунок 14-1: HTML документация для функции `add_one`

Часто используемые разделы

Мы использовали Markdown заголовок `# Examples` в листинге 14-1 для создания раздела в HTML с заголовком "Examples". Вот некоторые другие разделы, которые авторы библиотек обычно используют в своей документации:

- **Panics:** Сценарии, в которых документированная функция может вызывать панику. Вызывающие функцию, которые не хотят, чтобы их программы паниковали, должны убедиться, что они не вызывают функцию в этих ситуациях.
- **Ошибки:** Если функция возвращает `Result`, описание типов ошибок, которые могут произойти и какие условия могут привести к тому, что эти ошибки могут быть возвращены, может быть полезным для вызывающих, так что они могут написать код для обработки различных типов ошибок разными способами.
- **Безопасность:** Если функция является `unsafe` для вызова (мы обсуждаем безопасность в главе 19), должен быть раздел, объясняющий, почему функция небезопасна и охватывающий инварианты, которые функция ожидает от вызывающих сторон.

Большинству комментариев к документации не нужны все эти разделы, но это

хороший контрольный список, чтобы напомнить вам об аспектах вашего кода, о которых люди, вызывающие ваш код, будут заинтересованы узнать.

Комментарии к документации как тесты

Добавление примеров кода в комментарии к документации может помочь продемонстрировать, как использовать вашу библиотеку, и это даёт дополнительный бонус: запуск `cargo test` запустит примеры кода в вашей документации как тесты! Нет ничего лучше, чем документация с примерами. Но нет ничего хуже, чем примеры, которые не работают, потому что код изменился с момента написания документации. Если мы запустим `cargo test` с документацией для функции `add_one` из листинга 14-1, мы увидим раздел результатов 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;
finished in 0.27s
```

Теперь, если мы изменим либо функцию, либо пример, так что `assert_eq!` в примере паникует, и снова запустим `cargo test`, мы увидим, что тесты документации обнаруживают, что пример и код не синхронизированы друг с другом!

Комментирование содержащихся элементов

Другой стиль комментариев к документу, `//!`, добавляет документацию к элементу, содержащему комментарии, а не добавляет документацию к элементам, следующим за комментариями. Обычно мы используем эти комментарии к документу внутри корневого файла крейта (`src/lib.rs` по соглашению) или внутри модуля для документирования крейта или модуля в целом.

Например, если мы хотим добавить документацию, описывающую назначение крейта `my_crate`, который содержит функцию `add_one`, мы можем добавить комментарии к документации, начинающиеся с `//!`, в начало `src/lib.rs`, как показано в листинге 14-2:

Файл: `src/lib.rs`

```
///! # My Crate
///
///! `my_crate` is a collection of utilities to make performing certain
///! calculations more convenient.

/// Adds one to the number given.
// --snip--
```

Листинг 14-2: Документация для крейта `my_crate` в целом

Обратите внимание, что нет кода после последней строки, начинающейся с `//!`. Поскольку мы начинали комментарии с `//!` вместо `///`, мы документируем элемент, который содержит этот комментарий, а не элемент, следующий за этим комментарием. В этом случае элементом, содержащим этот комментарий, является файл `src/lib.rs`, который является корнем крейта. Эти комментарии описывают всю крейт.

Когда мы запускаем `cargo doc --open`, эти комментарии будут отображаться на первой странице документации для `my_crate` над списком публичных элементов в библиотеке, как показано на рисунке 14-2:

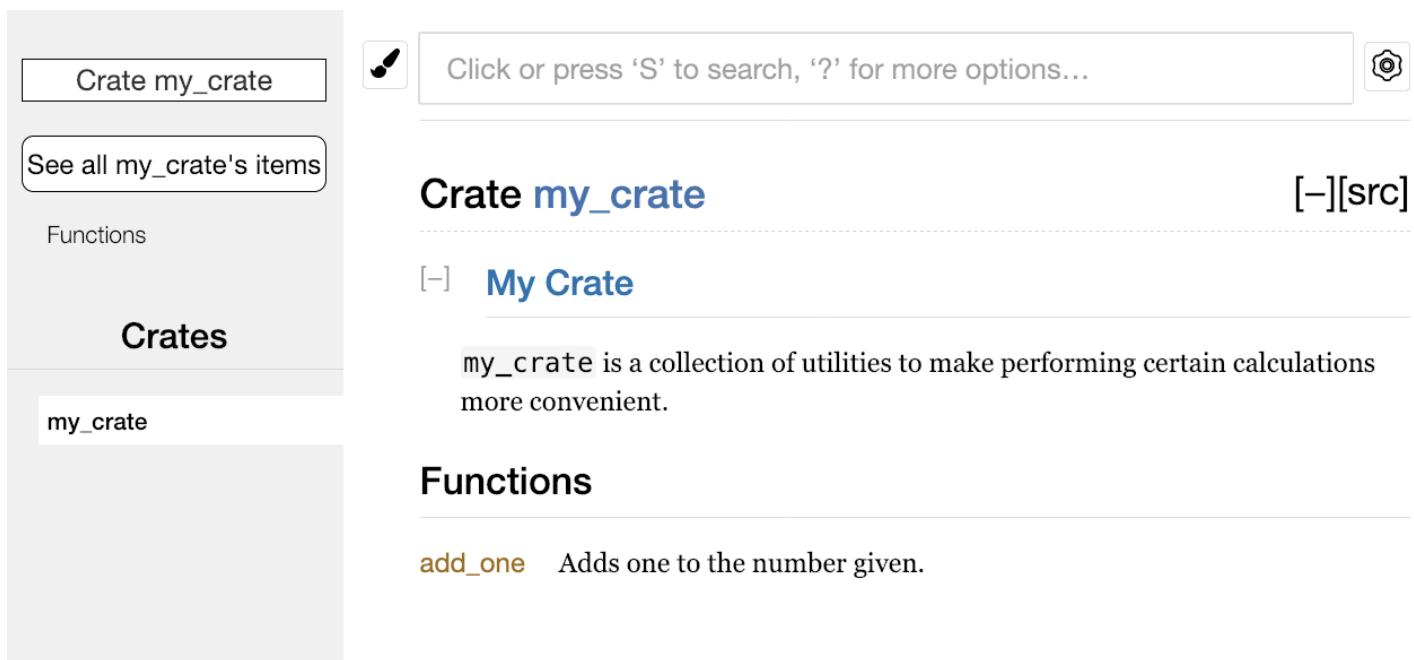


Рисунок 14-2: Предоставленная документация для `my_crate`, включая комментарий, описывающие крейт в целом

Комментарии к документации внутри элементов полезны для описания крейтов и модулей особенно. Используйте их, чтобы объяснить общую цель контейнера, чтобы помочь вашим пользователям понять организацию крейта.

Экспорт публичного API с `pub use`

В главе 7 мы рассмотрели, как организовать ваш код в модули с помощью ключевого слова `mod`, как сделать элементы публичными с помощью ключевого слова `pub` и как подключить элементы в область видимости с помощью ключевого слова `use`. Однако структура, которая имеет смысл для вас при разработке крейта, может быть не очень удобной для пользователей. Возможно, вы захотите организовать свои структуры в иерархию, содержащую несколько уровней, но тогда люди, которые хотят использовать тип, определённый вами глубоко в иерархии, могут столкнуться с трудностями при обнаружении того, что этот тип существует. Они также могут быть раздражены необходимостью ввода `use`

```
my_crate::some_module::another_module::UsefulType; вместо ввода use  
my_crate::UsefulType; .
```

Структура вашего публичного API является основным фактором при публикации крейта. Люди, которые используют вашу библиотеку, менее знакомы со структурой, чем вы и могут столкнуться с трудностями при поиске частей, которые они хотят использовать, если ваша библиотека имеет большую иерархию модулей.

Хорошей новостью является то, что если структура *не* удобна для использования другими из другой библиотеки, вам не нужно перестраивать внутреннюю организацию: вместо этого вы можете реэкспортировать элементы, чтобы сделать публичную структуру, отличную от вашей внутренней структуры, используя `pub use`. Реэкспорт берет открытый элемент в одном месте и делает его публичным в другом месте, как если бы он был определён в другом месте.

Например, скажем, мы создали библиотеку с именем `art` для моделирования художественных концепций. Внутри этой библиотеки есть два модуля: модуль `kinds` содержащий два перечисления с именами `PrimaryColor` и `SecondaryColor` и модуль `utils`, содержащий функцию с именем `mix`, как показано в листинге 14-3:

Файл: `src/lib.rs`

```
///! # Art
///
///! A library for modeling artistic concepts.

pub mod kinds {
    /// The primary colors according to the RYB color model.
    pub enum PrimaryColor {
        Red,
        Yellow,
        Blue,
    }

    /// The secondary colors according to the RYB color model.
    pub enum SecondaryColor {
        Orange,
        Green,
        Purple,
    }
}

pub mod utils {
    use crate::kinds::*;

    /// Combines two primary colors in equal amounts to create
    /// a secondary color.
    pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {
        // --snip--
    }
}
```

Листинг 14-3: Библиотека `art` с элементами, организованными в модули `kinds` и `utils`

На рисунке 14-3 показано, как будет выглядеть титульная страница документации для этого крейта, сгенерированный `cargo doc`:

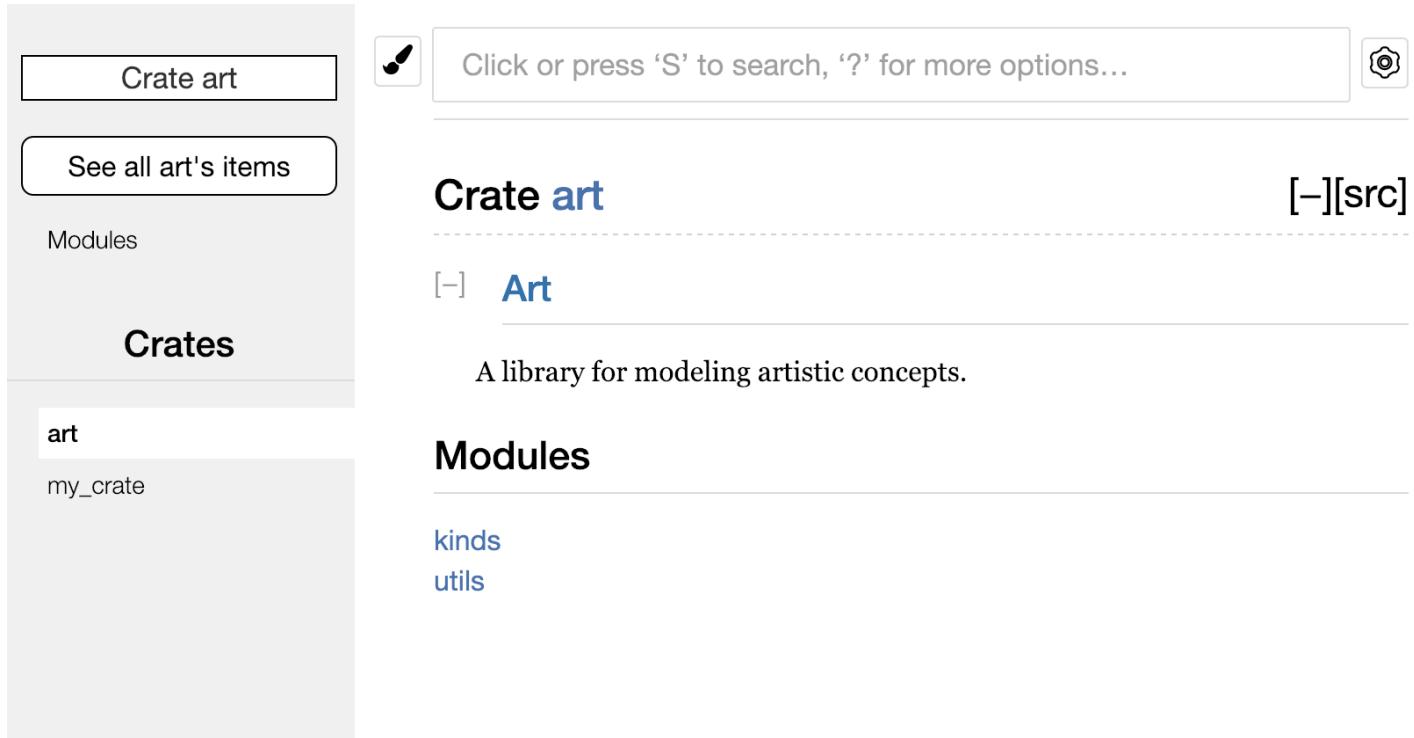


Рисунок 14-3: Первая страница документации для `art`, в которой перечислены модули `kinds` и `utils`

Обратите внимание, что типы `PrimaryColor` и `SecondaryColor` не указаны на главной странице, равно как и функция `mix`. Мы должны нажать `kinds` и `utils`, чтобы увидеть их.

В другой библиотеке, которая зависит от этой библиотеки, потребуются операторы `use`, которые подключают элементы из `art` в область видимости, определяя структуру модуля, которая определена в данный момент. В листинге 14-4 показан пример крейта, в котором используются элементы `PrimaryColor` и `mix` из крейта `art`:

Файл: src/main.rs

```
use art::kinds::PrimaryColor;
use art::utils::mix;

fn main() {
    let red = PrimaryColor::Red;
    let yellow = PrimaryColor::Yellow;
    mix(red, yellow);
}
```

Листинг 14-4: Крейт использующий элементы из крейта `art` с экспортированной внутренней структурой

Автор кода в листинге 14-4, в котором используется крейт `art`, должен был выяснить, что `PrimaryColor` находится в модуле `kinds`, а `mix` находится в модуле `utils`. Модульная структура крейта `art` больше уместна разработчикам работающим над крейтом `art`, чем разработчикам, использующим крейт `art`. Внутренняя структура, которая организует части библиотеки в модуль `kinds` и модуль `utils`, не содержит никакой полезной информации для того, кто пытается понять, как использовать крейт `art`. Вместо этого, структура модулей крейта `art` вызывает путаницу, потому что разработчики должны выяснить где делать поиск и структура неудобна, потому что разработчики должны указывать имена модулей в операторах `use`.

Чтобы удалить внутреннюю организацию из общедоступного API, мы можем изменить код крейта `art` в листинге 14-3, чтобы добавить операторы `pub use` для повторного реэкспорта элементов на верхнем уровне, как показано в листинге 14-5:

Файл: src/lib.rs

```
///! # Art
///!
///! A library for modeling artistic concepts.

pub use self::kinds::PrimaryColor;
pub use self::kinds::SecondaryColor;
pub use self::utils::mix;

pub mod kinds {
    // --snip--
}

pub mod utils {
    // --snip--
}
```

Листинг 14-5: Добавление операторов `pub use` для реэкспорта элементов

Документация API, которую `cargo doc` генерирует для этой библиотеки, теперь будет перечислять и связывать реэкспорты на главной странице, как показано на рисунке 14-4, упрощая поиск типов `PrimaryColor`, `SecondaryColor` и функции `mix`.

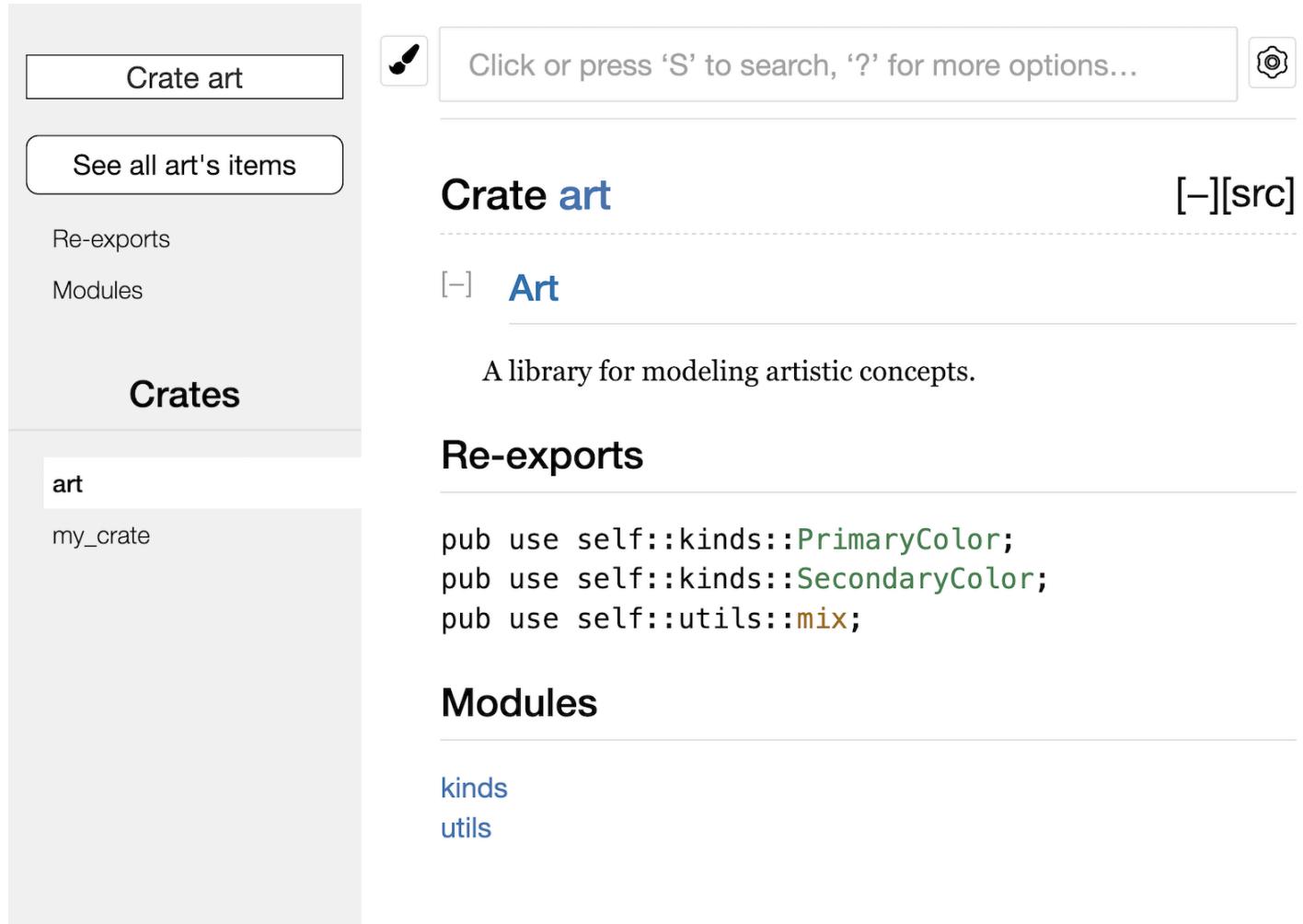


Рисунок 14-4: Первая страница документации для `art`, которая перечисляет реэкспорт

Пользователи крейта `art` могут по-прежнему видеть и использовать внутреннюю структуру из листинга 14-3, как показано в листинге 14-4, или они могут использовать более удобную структуру в листинге 14-5, как показано в листинге 14-6:

Файл: `src/main.rs`

```
use art::mix;
use art::PrimaryColor;

fn main() {
    // --snip--
}
```

Листинг 14-6: Программа, использующая реэкспортированные элементы из крейта `art`

В тех случаях, когда имеется много вложенных модулей, реэкспорт типов на верхнем уровне с помощью `pub use` может существенно изменить опыт людей,

использующих библиотеку.

Создание полезной публичной структуры API - это больше искусство чем наука, и вы можете повторять, чтобы найти API, который лучше всего подойдёт вашим пользователям. Использование `pub use` даёт вам гибкость в том, как вы структурируете свою библиотеку внутри и отделяете эту внутреннюю структуру от того, что вы предоставляете пользователям. Посмотрите на код некоторых установленных крейтов, чтобы увидеть отличается ли их внутренняя структура от их публичного API.

Настройка учётной записи [Crates.io](#)

Прежде чем вы сможете опубликовать любые библиотеки, вам необходимо создать учётную запись на [crates.io](#) и получить API токен. Для этого зайдите на домашнюю страницу [crates.io](#) и войдите в систему через учётную запись GitHub. (В настоящее время требуется наличие учётной записи GitHub, но сайт может поддерживать другие способы создания учётной записи в будущем.) Сразу после входа в систему перейдите в настройки своей учётной записи по адресу <https://crates.io/me/> и получите свой ключ API. Затем выполните команду `cargo login` с вашим ключом API, например:

```
$ cargo login abcdefghijklmnopqrstuvwxyz012345
```

Эта команда сообщает Cargo о вашем API token и сохранит его локально в `~/.cargo/credentials`. Обратите внимание, что этот токен является *секретным*: не делитесь им ни с кем другим. Если вы по какой-либо причине поделитесь им с кем-либо, вы должны отозвать его и сгенерировать новый токен на [crates.io](#).

Добавление метаданных в новую библиотеку

Теперь у вас есть аккаунт, допустим, у вас есть библиотека, которую вы хотите опубликовать. Перед публикацией вам необходимо добавить некоторые метаданные в ваш крейт, добавив их в раздел `[package]` файла `Cargo.toml`.

Вашему крейту понадобится уникальное имя. Пока вы работаете над библиотекой локально, вы можете назвать библиотеку как угодно. Однако имена крейтов на [crates.io](#) выделяются по принципу "первым пришёл - первым обслужен". Как только имя крейта занято, никто не может опубликовать крейт с таким именем. Прежде

чем пытаться опубликовать крейт, поищите имя, которое вы хотите использовать на сайте. Если имя было использовано другим крейтом, то вам нужно найти другое имя и отредактировать поле `name` в файле `Cargo.toml` в разделе `[package]`, чтобы использовать новое имя для публикации, например, так:

Файл: `Cargo.toml`

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

Даже если вы выбрали уникальное имя, когда вы запустите `cargo publish` чтобы опубликовать крейт, вы получите предупреждение, а затем ошибку:

```
$ cargo publish
    Updating crates.io index
warning: manifest has no description, license, license-file, documentation,
homepage or repository.
See https://doc.rust-lang.org/cargo/reference/manifest.html#package-metadata
for more info.
--snip--
error: failed to publish to registry at https://crates.io

Caused by:
    the remote server responded with an error: missing or empty metadata
fields: description, license. Please see https://doc.rust-
lang.org/cargo/reference/manifest.html for how to upload metadata
```

Причина в том, что вам не хватает важной информации: требуется описание и лицензия, чтобы люди знали, что делает ваша библиотека, и на каких условиях они могут её использовать. Чтобы исправить эту ошибку, вам нужно включить эту информацию в файл `Cargo.toml`.

Добавьте описание, которое из себя представляет одно или два предложения, потому что оно будет отображаться вместе с вашим крейтом в результатах поиска. В поле `license` необходимо указать значение идентификатора лицензии. В [Linux Foundation's Software Package Data Exchange \(SPDX\)](#) перечислены идентификаторы, которые можно использовать для этого значения. Например, чтобы указать, что вы лицензировали свою библиотеку с использованием лицензии MIT, добавьте идентификатор `MIT`:

Файл: `Cargo.toml`

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

Если вы хотите использовать лицензию, которая отсутствует в SPDX, вам нужно поместить текст этой лицензии в файл, включите файл в свой проект, а затем используйте `license-file`, чтобы указать имя этого файла вместо использования ключа `license`.

Руководство по выбору лицензии для вашего проекта выходит за рамки этой книги. Многие люди в сообществе Rust лицензируют свои проекты так же, как и Rust, используя двойную лицензию `MIT OR Apache 2.0`. Эта практика демонстрирует, что вы также можете указать несколько идентификаторов лицензий, разделённых `OR`, чтобы иметь несколько лицензий для вашего проекта.

С добавлением уникального имени, версии, вашего описания и лицензии, файл `Cargo.toml` для проекта, который готов к публикации может выглядеть следующим образом:

Файл: `Cargo.toml`

```
[package]
name = "guessing_game"
version = "0.1.0"
edition = "2021"
description = "A fun game where you guess what number the computer has
chosen."
license = "MIT OR Apache-2.0"

[dependencies]
```

[Документация Cargo](#) описывает другие метаданные, которые вы можете указать, чтобы другие могли легче находить и использовать ваш крейт.

Публикация на [Crates.io](#)

Теперь, когда вы создали учётную запись, сохранили свой токен API, выбрали имя для своего крейта и указали необходимые метаданные, вы готовы к публикации! Публикация библиотеки загружает определённую версию в [crates.io](#) для использования другими.

Будьте осторожны при публикации библиотеки, потому что публикация является

постоянной. Версия никогда не может быть перезаписана и код не может быть удалён. Одна из основных целей [crates.io](#) - является работа как постоянного архива кода, чтобы сборки всех проектов, которые зависят от библиотек из [crates.io](#), продолжали работать. Разрешение на удаление версий сделает достижение этой цели невозможным. Однако, количество версий библиотек, которые вы можете опубликовать, не ограничено.

Запустите команду `cargo publish` ещё раз. Сейчас эта команда должна выполниться успешно:

```
$ cargo publish
  Updating 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 0.19s
  Uploading guessing_game v0.1.0 (file:///projects/guessing_game)
```

Поздравляем! Теперь вы поделились своим кодом с сообществом Rust и любой может легко добавить вашу библиотеку в качестве зависимости их проекта.

Публикация новой версии существующей библиотеки

Когда вы внесли изменения в свой крейт и готовы выпустить новую версию, измените значение `version`, указанное в вашем файле `Cargo.toml` и повторите публикацию. Воспользуйтесь [Semantic Versioning rules](#), чтобы решить, какой номер следующей версии подходит для ваших изменений. Затем запустите `cargo publish`, чтобы загрузить новую версию.

Удаление версий из Crates.io с помощью `cargo yank`

Хотя вы не можете удалить предыдущие версии крейта, вы можете помешать любым будущим проектам добавлять его в качестве новой зависимости. Это полезно, когда версия крейта сломана по той или иной причине. В таких ситуациях Cargo поддерживает *выламывание* (*yanking*) версии крейта.

Выламывание версии не позволяет новым проектам зависеть от этой версии, а все существующие проекты, которые зависят от неё, продолжают скачивать и зависеть от этой версии. По сути, выламывание означает, что все проекты с `Cargo.lock` не

сломаются, и любые будущие сгенерированные файлы *Cargo.lock* не будут использовать выломанную версию.

Чтобы выломать версию крейт, запустите `cargo yank` и укажите, какую версию вы хотите выломать:

```
$ cargo yank --vers 1.0.1
```

Добавив в команду `--undo`, вы также можете отменить выламывание и разрешить проектам начать зависеть от версии снова:

```
$ cargo yank --vers 1.0.1 --undo
```

Выламывание *не* удаляет код. Например, функция выламывания не предназначена для удаления случайно загруженных секретов. Если это произойдёт, вы должны немедленно сбросить эти секреты.

Рабочие пространства Cargo

В главе 12 мы создали пакет, включающий бинарный крейт и крейт библиотеки. По мере развития вашего проекта вы можете обнаружить, что крейт библиотеки продолжает увеличиваться и вы хотите разделить свой пакет ещё на несколько библиотечных крейтов. В этой ситуации Cargo предлагает функцию, называемую рабочими пространствами *workspace*, которая может помочь в управлении несколькими связанными пакетами, разработанными в tandemе.

Создание рабочего пространства

Рабочее пространство является набором пакетов, которые совместно используют один и тот же файл *Cargo.lock* и папку для хранения конечных программных продуктов (будь то бинарные файлы или библиотеки). Давайте создадим проект, используя рабочее пространство, мы будем использовать простой код, чтобы сосредоточиться на структуре рабочего пространства. Есть несколько способов структурировать рабочее пространство; мы собираемся показать общий способ. У нас будет рабочее пространство, содержащее исполняемый файл и две библиотеки. Исполняемый файл, обеспечивающий основную функциональность, будет зависеть от двух библиотек. Одна библиотека будет предоставлять функцию `add_one`, а вторая функцию `add_two`. Эти три крейта будут частью одного рабочего пространства. Начнём с создания нового каталога для рабочей области:

```
$ mkdir add  
$ cd add
```

Затем в каталоге *add* мы создаём файл *Cargo.toml*, который настроит всю рабочую область. В этом файле не будет раздела `[package]` или метаданных, которые мы видели в других файлах *Cargo.toml*. Вместо этого он начнётся с раздела `[workspace]`, который позволит нам добавлять участников в рабочую область, указывая путь к нашему двоичному крейту; в этом случае этот путь *adder*:

Файл: *Cargo.toml*

```
[workspace]  
  
members = [  
    "adder",  
]
```

Затем мы создадим исполняемый крейт `adder`, запустив команду `cargo new` в каталоге `add`:

```
$ cargo new adder
   Created binary (application) `adder` package
```

На этом этапе мы можем создать рабочее пространство, запустив команду `cargo build`. Файлы в каталоге `add` должны выглядеть следующим образом:

```
└── Cargo.lock
└── Cargo.toml
└── adder
    └── Cargo.toml
        └── src
            └── main.rs
└── target
```

Рабочее пространство имеет одну целевую директорию `target` на верхнем уровне для размещения в ней скомпилированных артефактов; крейт `adder` не имеет своей собственной директории `target`. Даже если бы мы запускали `cargo build` из каталога `adder`, скомпилированные артефакты все равно оказались бы в `add/target`, а не в `add/adder/target`. Cargo структурирует каталог `target` в рабочем пространстве таким образом, потому что крейты в рабочем пространстве должны зависеть друг от друга. Если бы у каждого крейта был свой собственный каталог `target`, каждому крейту пришлось бы пере компилировать все другие крейты в рабочем пространстве, чтобы артефакты находились в собственных каталогах `target`. Используя один каталог `target`, крейты могут избежать ненужной сборки из исходных кодов.

Добавление второго крейта в рабочее пространство

Далее давайте создадим ещё одного участника пакета в рабочей области и назовём его `add_one`. Внесите изменения в `Cargo.toml` верхнего уровня так, чтобы указать путь `add_one` в списке `members`:

Файл: `Cargo.toml`

```
[workspace]
```

```
members = [
    "adder",
    "add_one",
]
```

Затем сгенерируйте новый крейт библиотеки с именем `add_one`:

```
$ cargo new add_one --lib
Created library `add_one` package
```

Ваш каталог `add` должен теперь иметь следующие каталоги и файлы:

```
└── Cargo.lock
└── Cargo.toml
└── add_one
    ├── Cargo.toml
    └── src
        └── lib.rs
└── adder
    ├── Cargo.toml
    └── src
        └── main.rs
└── target
```

В файле `add_one/src/lib.rs` добавим функцию `add_one`:

Файл: `add_one/src/lib.rs`

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

Теперь, когда у нас есть крейт библиотеки в рабочей области, мы можем получить исполняемый крейт `adder` зависящий от библиотечного крейта `add-one`. Сначала нам нужно добавить путь зависимости от `add-one` в `adder/Cargo.toml`.

Файл: `adder/Cargo.toml`

```
[dependencies]
add_one = { path = "../add_one" }
```

Cargo не предполагает, что крейты в рабочей области будут зависеть друг от друга, поэтому нам нужно чётко указать отношения зависимостей между крейтами.

Далее воспользуемся функцией `add_one` из крейта `add_one` в крейте `adder`.

Откройте файл `adder/src/main.rs` и добавьте вверху строку `use`, чтобы включить в область видимости новый крейт библиотеки `add_one`. Затем измените функцию `main`, чтобы она вызывала функцию `add_one`, как показано в листинге 14.7.

Файл: `adder/src/main.rs`

```
use add_one;

fn main() {
    let num = 10;
    println!(
        "Hello, world! {} plus one is {}!",
        num,
        add_one::add_one(num)
    );
}
```

Листинг 14-7: Использование функционала библиотечного крейта `add-one` в крейте `adder`

Давайте соберём рабочее пространство, запустив команду `cargo build` в каталоге верхнего уровня `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 0.68s
```

Чтобы запустить бинарный крейт из каталога `add`, нам нужно указать какой пакет из рабочей области мы хотим использовать с помощью аргумента `-p` и названия пакета в команде `cargo run`:

```
$ cargo run -p adder
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
Running `target/debug/adder`
Hello, world! 10 plus one is 11!
```

Запуск кода из `adder/src/main.rs`, который зависит от `add_one`.

Зависимость от внешних крейтов в рабочем пространстве

Обратите внимание, что в рабочем пространстве на верхнем уровне есть только один файл `Cargo.lock`, нет наличия `Cargo.lock` в каталоге каждого крейта. Это

гарантирует, что все крейты используют одну и ту же версию всех зависимостей. Если мы добавим крейт `rand` в `adder/Cargo.toml` и `add_one/Cargo.toml` файлы, Cargo разрешит оба из них в одну версию `rand` и запишет её в один `Cargo.lock`.

Использованием всеми крейтами одинаковых зависимостей в рабочем пространстве означает, что крейты в рабочем пространстве всегда будут совместимы с друг с другом. Давайте добавим крейт `rand` в раздел `[dependencies]` в файле `add_one/Cargo.toml`, чтобы можно было использовать крейт `rand` в крейте `add_one`:

Файл: `add_one/Cargo.toml`

```
[dependencies]
rand = "0.8.3"
```

Теперь мы можем добавить `use rand;` в файл `add_one/src/lib.rs` и сделать сборку рабочего пространства, запустив `cargo build` в каталоге `add`, что загрузит и скомпилирует `rand` крейт:

```
$ cargo build
  Updating crates.io index
Downloaded rand v0.8.3
--snip--
  Compiling rand v0.8.3
  Compiling add_one v0.1.0 (file:///projects/add/add_one)
warning: unused import: `rand`
--> add_one/src/lib.rs:1:5
  |
1 | use rand;
  |     ^
  |
= note: #[warn(unused_imports)]` on by default

warning: 1 warning emitted

  Compiling adder v0.1.0 (file:///projects/add/adder)
Finished dev [unoptimized + debuginfo] target(s) in 10.18s
```

Файл `Cargo.lock` верхнего уровня теперь содержит информацию о зависимости `add_one` к крейту `rand`. Тем не менее, несмотря на то что `rand` использован где-то в рабочем пространстве, мы не можем использовать его в других крейтах рабочего пространства, пока не добавим крейт `rand` в отдельные `Cargo.toml` файлы. Например, если мы добавим `use rand;` в файл `adder/src/main.rs` крейта `adder`, то получим ошибку:

```
$ cargo build
--snip--
Compiling adder v0.1.0 (file:///projects/add/adder)
error[E0432]: unresolved import `rand`
--> adder/src/main.rs:2:5
2 | use rand;
|     ^^^^ no external crate `rand`
```

Чтобы исправить ошибку, отредактируйте файл *Cargo.toml* крейта `adder` и укажите, что `rand` является зависимостью и для этого крейта. Сборка крейта `adder` добавит крейт `rand` в список зависимостей крейта `adder` в его *Cargo.lock*, но нет будет загружать дополнительные копии `rand`. Cargo гарантирует, что каждый крейт рабочего пространства, использующий крейт `rand`, будет использовать одну и ту же версию. Использование одной и той же версии `rand` в рабочем пространстве экономит место, потому что мы не будем иметь несколько копий и есть гарантия, что крейты в рабочем пространстве будут совместимы друг с другом.

Добавление теста в рабочее пространство

В качестве ещё одного улучшения давайте добавим тест функции `add_one::add_one` в `add_one`:

Файл: `add_one/src/lib.rs`

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

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

    #[test]
    fn it_works() {
        assert_eq!(3, add_one(2));
    }
}
```

Выполните команду `cargo test` в каталоге верхнего уровня *adder*:

```
$ cargo test
Compiling add_one v0.1.0 (file:///projects/add/add_one)
Compiling adder v0.1.0 (file:///projects/add/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.27s
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;
finished in 0.00s

Running target/debug/deps/adder-49979ff40686fa8e

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests add_one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Первый раздел вывода показывает, что тест `it_works` в крейте `add_one` прошёл. В следующем разделе показано, что в крейте `adder` было найдено ноль тестов, а затем в последнем разделе показано, что в документации крейта `add_one` также было найдено ноль тестов. Выполнение команды `cargo test` в рабочем пространстве структурированном таким образом, будет запускать тесты для всех крейтов в рабочего пространства.

Мы также можем запустить тесты для одного конкретного крейта в рабочем пространстве из каталог верхнего уровня с помощью флага `-p` и указанием имени крейта для которого мы хотим запустить тесты:

```
$ cargo test -p add_one
    Finished test [unoptimized + debuginfo] target(s) in 0.00s
        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;
finished in 0.00s

Doc-tests add_one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Эти выходные данные показывают, что выполнение `cargo test` запускает только тесты для крейта `add-one` и не запускает тесты крейта `adder`.

Если вы публикуете крейты из такого рабочего пространства на [crates.io](#), то каждый крейт должен быть опубликован отдельно. Команда `cargo publish` не имеет флага `--all` или `-p`, поэтому вы должны перейти в каталог каждого крейта и запустить `cargo publish` в каждом крейте рабочего пространства для его публикации.

Для дополнительной практики добавьте крейт `add_two` в данное рабочее пространство аналогичным способом, как делали с крейт `add_one` !

По мере роста вашего проекта рассмотрите возможность использования рабочего пространства, так как легче понимать более маленькие, отдельные компоненты, чем один большой кусок кода. Кроме того, сохраняя крейты в рабочем пространстве можно облегчить координацию между ними, если они часто меняются одновременно.

Установка исполняемых крейтов из Crates.io командой `cargo install`

Команда `cargo install` позволяет локально устанавливать и использовать исполняемые крейты. Она не предназначена для замены системных пакетов; она используется как удобный способ Rust разработчикам устанавливать инструменты, которыми другие разработчики поделились на сайте [crates.io](#). Заметьте, можно устанавливать только пакеты, имеющие исполняемые целевые крейты.

Исполняемой целью (binary target) является запускаемая программа, созданная и имеющая в составе крейта файл `src/main.rs` или другой файл, указанный как исполняемый, в отличии от библиотечных крейтов, которые не могут запускаться сами по себе, но подходят для включения в другие программы. Обычно крейт содержит информацию в файле `README`, является ли он библиотекой, исполняемым файлом или обоими вместе.

Все исполняемые файлы установленные командой `cargo install` сохранены в корневой установочной папке `bin`. Если вы установили Rust с помощью `rustup.rs` и у вас его нет в пользовательских конфигурациях, то этим каталогом будет `$HOME/.cargo/bin`. Он гарантирует, что каталог находится в вашем окружении `$PATH`, чтобы вы имели возможность запускать программы, которые вы установили командой `cargo install`.

Например, в главе 12 мы упоминали Rust реализацию инструмента `grep` под названием `ripgrep` для поиска файлов. Если мы хотим установить `ripgrep`, мы можем запустить следующее:

```
$ cargo install ripgrep
  Updating crates.io index
  Downloaded ripgrep v11.0.2
  Downloaded 1 crate (243.3 KB) in 0.88s
  Installing ripgrep v11.0.2
--snip--
  Compiling ripgrep v11.0.2
  Finished release [optimized + debuginfo] target(s) in 3m 10s
  Installing ~/.cargo/bin/rg
  Installed package `ripgrep v11.0.2` (executable `rg`)
```

Последняя строка вывода показывает местоположение и название установленного исполняемого файла, который в случае `ripgrep` называется `rg`. Если вашей установочной директорией является `$PATH`, как уже упоминалось ранее, вы можете запустить `rg --help` и начать использовать более быстрый и грубый инструмент

для поиска файлов!

Расширение Cargo пользовательскими командами

Cargo спроектирован так, что вы можете расширять его новыми субкомандами без необходимости изменения самого Cargo. Если исполняемый файл доступен через переменную окружения `$PATH` и назван по шаблону `cargo-something`, то его можно запускать как субкоманду Cargo `cargo something`. Пользовательские команды подобные этой также перечисляются в списке доступных через `cargo --list`. Возможность использовать `cargo install` для установки расширений и затем запускать их так же, как встроенные в Cargo инструменты, это очень удобное следствие продуманного дизайна Cargo!

Итоги

Совместное использование кода с Cargo и [crates.io](#) является частью того, что делает экосистему Rust полезной для множества различных задач. Стандартная библиотека Rust небольшая и стабильная, но крейты легко распространять, использовать и улучшать независимо от самого языка. Не стесняйтесь делиться кодом, который был вам полезен, через [crates.io](#); скорее всего, он будет полезен и кому-то ещё!

Умные указатели

Указатель - это общая концепция для переменной, которая содержит адрес на участок памяти. Этот адрес "относится к", или "указывает на" некоторые другие данные. Наиболее общая разновидность указателя в Rust - это ссылка, о которой вы узнали из Главы 4. Ссылки обозначаются символом `&` и заимствуют значение на которое они указывают. Они не имеют каких-либо специальных возможностей, кроме как ссылаться на данные. Кроме того, они не имеют никаких накладных расходов и являются тем указателем, который мы используем чаще всего.

Умные указатели с другой стороны, являются структурами данных, которые не только действуют как указатель, но также имеют дополнительные метаданные и дополнительные возможности. Концепция умных указателей не уникальна для Rust: умные указатели возникли в C ++ и существуют в других языках. В Rust есть разные умные указатели, определённые в стандартной библиотеке, которые обеспечивают функциональность выходящую за рамки ссылок. Одним из примеров, который мы рассмотрим в этой главе, является тип умного указателя *reference counting* (подсчёт ссылок). Этот указатель позволяет иметь несколько владельцев данных с помощью отслеживания количества владельцев и очистки данных, когда владельцев не осталось.

В Rust, который использует концепцию владения и заимствования, дополнительная разница между ссылками и умными указателями заключается в том, что ссылки являются указателями, которые только заимствуют данные; против того, что во многих случаях умные указатели *владеют* данными на которые они указывают.

Мы уже встречали несколько умных указателей в этой книге, таких как `String` и тип `Vec<T>` в главе 8, хотя мы не называли их умными указателями в тот момент. Оба этих типа считаются умными указателями, потому что они владеют некоторой областью памяти и позволяют ею манипулировать. У них также есть метаданные (например, их ёмкость) и дополнительные возможности или гарантии (например, `String` обеспечивает, что содержимое всегда будет действительными данными в кодировке UTF-8).

Умные указатели обычно реализуются с использованием структур. Характерной чертой, которая отличает умный указатель от обычной структуры является то, что для умных указателей реализованы типажи `Deref` и `Drop`. Типаж `Deref` позволяет экземпляру умной структуры указателя вести себя как ссылка, так что вы можете написать код работающий или со ссылками или с умными указателями. Типаж `Drop` позволяет настроить код запускаемый, когда экземпляр умного указателя выходит

из области видимости. В этой главе мы обсудим оба типажа и продемонстрируем почему они важны для умных указателей.

Учитывая, что шаблон умный указатель является общим шаблоном проектирования часто используемым в Rust, эта глава не описывает все существующие умные указатели. Множество библиотек имеют свои умные указатели и вы также можете написать свои. Мы охватим наиболее распространённые умные указатели из стандартной библиотеки:

- `Box<T>` для распределения значений в куче (памяти)
- `Rc<T>` тип счётчика ссылок, который допускает множественное владение
- Типы `Ref<T>` и `RefMut<T>`, доступ к которым осуществляется через тип `RefCell<T>`, который обеспечивает правила заимствования во время выполнения, вместо времени компиляции

Дополнительно мы рассмотрим шаблон *внутренняя изменчивость* (*interior mutability*), где неизменяемый тип предоставляет API для изменения своего внутреннего значения. Мы также обсудим *ссылочные зацикленности* (*reference cycles*): как они могут приводить к утечке памяти и как это предотвратить.

Приступим!

Использование `Box<T>` для ссылки на данные в куче

Наиболее простой умный указатель - это *box*, чей тип записывается как `Box<T>`. Такие переменные позволяют хранить данные в куче, а не в стеке. То, что остается в стеке, является указателем на данные в куче. Обратитесь к Главе 4, чтобы рассмотреть разницу между стеком и кучей.

У *Box* нет проблем с производительностью, кроме хранения данных в куче вместо стека. Но он также и не имеет множества дополнительных возможностей. Вы будете использовать его чаще всего в следующих ситуациях:

- Если у вас есть тип, размер которого не может быть известен во время компиляции и вы хотите использовать значение этого типа в контексте, который требует точного размера
- Когда у вас есть большой объем данных и вы хотите передать его во владение, но убедиться, что данные не будут скопированы, когда вы это сделаете
- Когда вы хотите иметь значение и вам важно только то, что это тип, который реализует конкретный типаж, а не является конкретным типом

Мы продемонстрируем первую ситуацию в разделе "[Реализация рекурсивных типов с помощью Box](#)". Во втором случае, передача владения на большой объем данных может занять много времени, потому что данные копируются через стек. Для повышения производительности в этой ситуации, мы можем хранить большое количество данных в куче с помощью *Box*. Затем только небольшое количество данных указателя копируется в стеке, в то время как данные, на которые он ссылается, остаются в одном месте кучи. Третий случай известен как *типаж объект* (trait object) и глава 17 посвящает целый раздел "[Использование типаж объектов, которые допускают значения разных типов](#)" только этой теме. Итак, то что вы узнаете здесь, вы примените снова в Главе 17!

Использование `Box<T>` для хранения данных в куче

Прежде чем мы обсудим этот вариант использования `Box<T>`, мы рассмотрим синтаксис и как взаимодействовать со значениями, хранящимися в `Box<T>`.

В листинге 15-1 показано, как использовать поле для хранения значения `i32` в куче:

Файл: src/main.rs

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

Листинг 15-1: Сохранение значения `i32` в куче с использованием `box`

Мы определяем переменную `b` как имеющую значение типа `Box`, указывающее на значение `5` в куче. Эта программа напечатает `b = 5`; в этом случае мы можем получить доступ к данным в поле, как если бы это были данные в стеке. Как и любое значение во владении, данная память будет освобождена, когда `box` выйдет из области действия, что происходит с `b` в конце `main`. Освобождается память, занимаемая `box` (хранится в стеке), и тех данных, на которые он указывает (хранятся в куче).

Размещение единственного значения в куче не очень полезно, поэтому вы не будете часто использовать `box` сам по себе таким способом. Иметь единственное значение `i32` в стеке, где они хранятся по умолчанию, подходит в большинстве ситуаций. Давайте рассмотрим случай, когда `Box` позволяет определять типы, которые были бы невозможны, если бы у нас не было `Box`.

Включение рекурсивных типов с помощью Boxes

Во время компиляции Rust должен знать, сколько места занимает тип. Некоторый тип, чей размер не может быть известен во время компиляции, является *рекурсивным типом* (recursive type), где значение может иметь в своём составе другое значение того же типа. По причине того, что это вложение значений может теоретически продолжаться бесконечно, Rust не знает, сколько пространства памяти необходимо для значений рекурсивного типа. Однако `Box` имеет известный размер, поэтому используя `Box` в определении рекурсивного типа, можно его реализовать.

Давайте рассмотрим *cons список* (`cons` - функция конструктор, создаёт объекты памяти, которые содержат два значения или указатели на значения), который является распространённым в функциональных языках программирования типом данных, как пример рекурсивного типа. Тип "cons список", который мы определим, является простым, за исключением рекурсии; поэтому концепции, используемые в примере, с которым мы будем работать, будут полезны и в более сложных

ситуациях, связанных с рекурсивными типами.

Больше информации о cons списке

cons список (*cons list*) - это структура данных, которая пришла из языка программирования Lisp и егоialectов. В Lisp, функция **cons** (сокращение от "construct function" функция-конструктор) создаёт новую пару используя два аргумента, один из которых значение, а другой - пара. Эти пары, содержащие другие пары, образуют список.

Концепция функции конструктора прошла свой путь и превратилась в более общий функциональный, программный жаргон: "to cons x onto y" неформально означает создание нового экземпляра контейнера, помещая элемент x в начале нового контейнера, за которым следует контейнер y.

Каждый элемент в cons списке содержит два элемента: значение текущего элемента и следующий элемент. Последний элемент в списке содержит только значение называемое **Nil** без следующего элемента. Cons список создаётся путём рекурсивного вызова функции **cons**. Каноничное имя для обозначения базового случая рекурсии - **Nil**. Обратите внимание, что это не то же самое, что понятие "null" или "nil" из главы 6, которая является недействительным или отсутствующим значением.

Хотя функциональные языки программирования часто используют cons списки, этот список не является широко используемой структурой данных в Rust. Большую часть времени, когда есть список элементов в Rust, лучше использовать **Vec<T>**. Более сложные рекурсивные типы данных являются полезными в различных ситуациях, но начиная изучение с cons списка, мы можем исследовать, как box-ы позволяют определить рекурсивный тип данных без особых проблем.

Листинг 15-2 содержит объявление перечисления cons списка. Обратите внимание, что этот код не будет компилироваться, потому что тип **List** не имеет известного размера, что мы и продемонстрируем.

Файл: src/main.rs

```
enum List {
    Cons(i32, List),
    Nil,
}
```



Листинг 15-2: Первая попытка определить перечисление для представления структуры данных cons списка из значений [i32](#)

Примечание: мы реализуем cons список, который для примера содержит только значения [i32](#). Чтобы определить тип cons список для хранения значений любого типа, мы могли бы использовать обобщённые типы, как обсуждалось в главе 10.

Использование типа [List](#) для хранения списка [1, 2, 3](#) будет выглядеть как код в листинге 15-3:

Файл: src/main.rs

```
use crate::List::{Cons, Nil};  
  
fn main() {  
    let list = Cons(1, Cons(2, Cons(3, Nil)));  
}
```



Листинг 15-3: Использование перечисления [List](#) для хранения списка [1, 2, 3](#)

Первое значение [Cons](#) содержит [1](#) и другой [List](#). Это значение [List](#) является следующим значением [Cons](#), которое содержит [2](#) и другой [List](#). Это значение [List](#) является еще один значением [Cons](#), которое содержит [3](#) и значение [List](#), которое наконец является [Nil](#), не рекурсивным вариантом, сигнализирующим об окончании списка.

Если мы попытаемся скомпилировать код в листинге 15-3, мы получим ошибку, показанную в листинге 15-4:

```
$ cargo run
Compiling cons-list v0.1.0 (file:///projects/cons-list)
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 some indirection (e.g., a `Box`, `Rc`, or `&`) to make `List`
representable
|
2 |     Cons(i32, Box<List>),
|             +++++      +
|
error[E0391]: cycle detected when computing drop-check constraints for `List`
--> src/main.rs:1:1
|
1 | enum List {
| ^^^^^^^^^^
|
= note: ...which immediately requires computing drop-check constraints for
`List` again
= note: cycle used when computing dropck types for `Canonical {
max_universe: U0, variables: [], value: ParamEnvAnd { param_env: ParamEnv {
caller_bounds: [], reveal: UserFacing, constness: NotConst }, value: List } }`  
Some errors have detailed explanations: E0072, E0391.
For more information about an error, try `rustc --explain E0072`.
error: could not compile `cons-list` due to 2 previous errors
```

Листинг 15-3: Ошибка, получаемая при попытке определить бесконечное рекурсивное перечисление

Ошибка сообщает, что этот тип "имеет бесконечный размер". Причина в том, что мы определили **List** с рекурсивным вариантом: он содержит другое значение самого себя. В результате Rust не может понять, сколько места ему нужно для хранения значения **List**. Давайте разберёмся, почему мы получаем эту ошибку. Во-первых, давайте посмотрим как Rust решает, сколько места ему нужно для хранения значения нерекурсивного типа.

Вычисление размера нерекурсивного типа

Вспомните перечисление **Message** определённое в листинге 6-2, когда обсуждали

объявление enum в главе 6:

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

Чтобы определить, сколько памяти выделять под значение `Message`, Rust проходит каждый из вариантов, чтобы увидеть, какой вариант требует наибольшее количество памяти. Rust видит, что для `Message::Quit` не требуется места, `Message::Move` хватает места для хранения двух значений `i32` и т.д. Так как будет использоваться только один вариант, то наибольшее пространство, которое потребуется для значения `Message`, это пространство, которое потребуется для хранения самого большого из вариантов перечисления.

Сравните это с тем, что происходит, когда Rust пытается определить, сколько места необходимо рекурсивному типу, такому как перечисление `List` в листинге 15-2. Компилятор смотрит на вариант `Cons`, который содержит значение типа `i32` и значение типа `List`. Следовательно, `Cons` нужно пространство, равное размеру `i32` плюс размер `List`. Чтобы выяснить, сколько памяти необходимо типу `List`, компилятор смотрит на варианты, начиная с `Cons`. Вариант `Cons` содержит значение типа `i32` и значение типа `List`, и этот процесс продолжается бесконечно, как показано на рисунке 15-1.

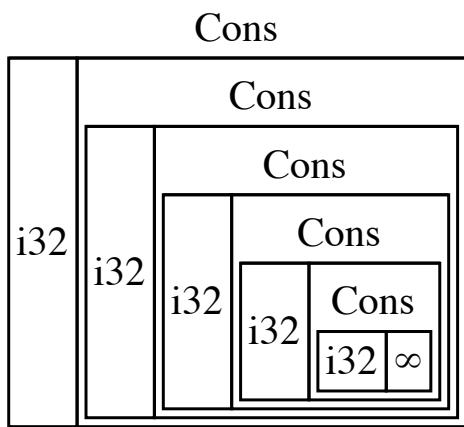


Рисунок 15-1: Бесконечный `List`, состоящий из бесконечных вариантов `Cons`

Использование `Box<T>` для получения рекурсивного типа с известным размером

Rust не может понять, сколько места выделить для типов определённых рекурсивно, поэтому компилятор выдаёт ошибку в листинге 15-4. Но ошибка включает в себя это полезное предложение:

```
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to make `List`  
representable  
|  
2 |     Cons(i32, Box<List>),  
|         ^^^^   ^
```

В этом предложении "косвенность" означает, что вместо сохранения значения напрямую, мы изменим структуру данных для хранения косвенного значения, с помощью хранения указателя на значение.

Поскольку `Box<T>` является указателем, Rust всегда знает, сколько памяти нужно для `Box<T>`: размер указателя не изменяется в зависимости от размера данных, на которые он указывает. Это значит, что мы можем поместить тип `Box<T>` в вариант перечисления `Cons` вместо помещения `List` напрямую. Поле `Box<T>` будет указывать на следующее значение `List`, которое будет в куче, а не внутри варианта `Cons`. Концептуально у нас все ещё есть список, созданный списками, "содержащими" другие списки, но эта реализация теперь больше похожа на размещение элементов рядом друг с другом, а не внутри друг друга.

Мы можем изменить определение перечисления `List` в листинге 15-2 и использование `List` в листинге 15-3 на код из листинга 15-5, который будет компилироваться:

Файл: 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))))));  
}
```

Листинг 15-5: Определение `List`, которое использует `Box`, чтобы иметь известный размер

Варианту `Cons` понадобится размер `i32` плюс место для хранения данных указателя `Box`. Вариант `Nil` не хранит значений, поэтому ему нужно меньше места, чем варианту `Cons`. Теперь мы знаем, что любое значение `List` будет занимать размер `i32` плюс размер данных указателя `Box`. Используя `Box`, мы сломали бесконечную рекурсивную цепочку, так что компилятор может определить размер, необходимый для хранения значения `List`. На рисунке 15-2 показано как теперь выглядит вариант `Cons`.

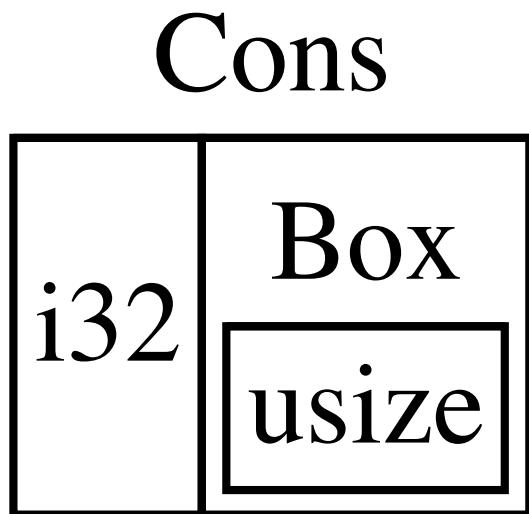


Рисунок 15-2: `List`, размер которого не ограничен, потому что `Cons` содержит `Box`

`Box`-ы обеспечивают только косвенность и выделение в куче; у них нет других специальных возможностей, таких как те, которые мы увидим у других типов умных указателей. Они также не имеют накладных расходов из-за этих специальных возможностей, поэтому могут быть полезны в случаях, похожих на `cons` список, где косвенность - единственная нужная функциональность. Мы рассмотрим ещё больше вариантов использования типа `Box` в главе 17.

Тип `Box<T>` является умным указателем, потому что он реализует типаж `Deref`, что позволяет значениям `Box<T>` обрабатываться как ссылки. Когда значение `Box<T>` выходит из области видимости, то данные кучи, на которые указывает `Box`, очищаются благодаря реализации типажа `Drop`. Давайте рассмотрим эти два типажа более подробно. Эти два типажа будут ещё более важными для функциональности, предоставляемой другими типами умных указателей, которые

мы обсудим в оставшихся частях этой главы.

Обращение с умными указателями как с обычными ссылками с помощью `Deref` типажа

Реализация типажа `Deref` позволяет настроить поведение *оператора разыменования* (dereference operator), `*` (отличается от умножения или оператора глобального подключения). Реализуя типаж `Deref` таким образом, что умный указатель может использоваться как обычная ссылка, вы можете писать код, который работает с ссылками и использовать этот код также с умными указателями.

Давайте сначала посмотрим, как работает оператор разыменования с обычными ссылками. Затем мы попытаемся определить пользовательский тип, который ведёт себя как `Box<T>` и посмотрим, почему оператор разыменования не работает как ссылка для нового объявленного типа. Мы рассмотрим, как реализация типажа `Deref` делает возможным работу умных указателей аналогично ссылкам. Затем посмотрим на *разыменованное приведение* (deref coercion) в Rust и как оно позволяет работать с любыми ссылками или умными указателями.

Примечание: есть одна большая разница между типом `MyBox<T>`, который мы собираемся создать и реальным `Box<T>`: наша версия не будет хранить свои данные в куче. В примере мы сосредоточимся на типаже `Deref`, поэтому менее важно то, где данные хранятся, чем поведение подобное указателю.

Следование по указателю к значению с помощью оператора разыменования

Обычная ссылка является типом указателя и один из способов думать про указатель, как будто это стрела в направлении к значению, хранящемуся где-то ещё. В листинге 15-6 мы создаём ссылку на значение [132](#) и затем используем оператор разыменования, чтобы следовать по ссылке к данным:

Файл: src/main.rs

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

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

Листинг 15-6: Использование оператора разыменования для следования по ссылке к значению 132

Переменная `x` содержит тип 132 со значением 5. Мы устанавливаем `y` равным ссылке на `x`. Мы можем утверждать, что `x` равно 5. Тем не менее, если мы хотим сделать утверждение о значении `y` то, мы должны использовать оператор `*y`, чтобы проследовать по ссылке к значению, на которое она указывает (следовательно, *разыменовывает* (dereference)). Как только мы разыменовываем `y`, у нас есть доступ к целочисленному значению `y`, на которое указывает ссылка и мы можем сравнить его с 5.

Если бы мы попытались написать `assert_eq!(5, y);`, то получили ошибку компиляции:

```
$ cargo run
Compiling deref-example v0.1.0 (file:///projects/deref-example)
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 `PartialEq<&'{integer}>` is not implemented for
`'{integer}`
= note: this error originates in the macro `assert_eq` (in Nightly builds,
run with -Z macro-backtrace for more info)

For more information about this error, try `rustc --explain E0277`.
error: could not compile `deref-example` due to previous error
```

Сравнение числа и ссылки на число не допускается, потому что они различных типов. Мы должны использовать оператор разыменования, чтобы перейти по ссылке на значение, на которое она указывает.

Использование Box<T> как ссылку

Можно переписать код в листинге 15-6, чтобы использовать `Box<T>` вместо ссылки; оператор разыменования будет работать, как показано в листинге 15-7:

Файл: src/main.rs

```
fn main() {
    let x = 5;
    let y = Box::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

Листинг 15-7: Использование оператора разыменования с типом `Box<i32>`

Разница между листингом 15-7 и листингом 15-6 состоит в том, что здесь мы устанавливаем `y` на экземпляр `box`, указывающий на значение `x`, а не ссылкой, указывающей на значение `x`. В последнем утверждении мы можем использовать оператор разыменования, чтобы проследовать за указателем `box`-а так же, как мы это делали когда `y` была ссылкой. Далее мы рассмотрим, что особенного у типа `Box<T>`, что позволяет нам использовать оператор разыменования, определяя наш собственный тип `Box`.

Определение собственного умного указателя

Давайте создадим умный указатель, похожий на тип `Box<T>` предоставляемый стандартной библиотекой, чтобы понять как поведение умных указателей отличается от поведения обычной ссылки. Затем мы рассмотрим вопрос, как добавить возможность использовать оператор разыменования.

Тип `Box<T>` в конечном итоге определяется как структура кортежа с одним элементом, поэтому в листинге 15-8 аналогичным образом определяется `MyBox<T>`. Мы также определим функцию `new`, чтобы она соответствовала функции `new`, определённой в `Box<T>`.

Файл: src/main.rs

```
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}
```

Листинг 15-8: Определение типа `MyBox<T>`

Мы определяем структуру с именем `MyBox` и объявляем обобщённый параметр `T`, потому что мы хотим, чтобы наш тип хранил значения любого типа. Тип `MyBox` является структурой кортежа с одним элементом типа `T`. Функция `MyBox::new` принимает один параметр типа `T` и возвращает экземпляр `MyBox`, который содержит переданное значение.

Давайте попробуем добавить функцию `main` из листинга 15-7 в листинг 15-8 и изменим её на использование типа `MyBox<T>`, который мы определили вместо `Box<T>`. Код в листинге 15-9 не будет компилироваться, потому что Rust не знает, как разыменовывать `MyBox`.

Файл: src/main.rs

```
fn main() {
    let x = 5;
    let y = MyBox::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

Листинг 15-9. Попытка использовать `MyBox<T>` таким же образом, как мы использовали ссылки и

`Box<T>`

Вот результат ошибки компиляции:

```
$ cargo run
Compiling deref-example v0.1.0 (file:///projects/deref-example)
error[E0614]: type `MyBox<{integer}>` cannot be dereferenced
--> src/main.rs:14:19
 |
14 |     assert_eq!(5, *y);
   |     ^

For more information about this error, try `rustc --explain E0614`.
error: could not compile `deref-example` due to previous error
```

Наш тип `MyBox<T>` не может быть разыменован, потому что мы не реализовали эту возможность. Чтобы включить разыменование с помощью оператора `*`, мы реализуем типаж `Deref`.

Трактование типа как ссылки реализуя типаж `Deref`

Как обсуждалось в разделе “[Реализация трейта для типа](#)” Главы 10, для реализации типажа нужно предоставить реализации требуемых методов типажа. Типаж `Deref`, предоставляемый стандартной библиотекой требует от нас реализации одного метода с именем `deref`, который заимствует `self` и возвращает ссылку на внутренние данные. Листинг 15-10 содержит реализацию `Deref` добавленную к определению `MyBox`:

Файл: `src/main.rs`

```
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}
```

Листинг 15-10: Реализация `Deref` для типа `MyBox<T>`

Синтаксис `type Target = T;` определяет связанный тип для использования у типажа `Deref`. Связанные типы - это немного другой способ объявления обобщённого параметра, но пока вам не нужно о них беспокоиться; мы рассмотрим их более подробно в главе 19.

Мы заполняем тело метода `deref` с помощью `&self.0`, поэтому `deref` возвращает ссылку на значение, к которому мы хотим получить доступ с помощью оператора `*`. Вспомним из раздела ["Использование структур кортежей без именованных полей для создания разных типов"](#). главы 5, что `.0` обращается к первому значению в структуре кортежа. Функция `main` в листинге 15-9, которая вызывает `*` для `MyBox<T>`, теперь компилируется и все утверждения проходят!

Без типажа `Deref` компилятор может только разыменовывать `&` ссылки. Метод `deref` даёт компилятору возможность принимать значение любого типа, реализующего `Deref` и вызывать метод `deref` чтобы получить ссылку `&`, которую он знает, как разыменовывать.

Когда мы ввели `*y` в листинге 15-9, Rust фактически выполнил за кулисами такой код:

```
*(y.deref())
```

Rust заменяет оператор `*` вызовом метода `deref` и затем простое разыменование, поэтому нам не нужно думать о том, нужно ли нам вызывать метод `deref`. Эта функция Rust позволяет писать код, который функционирует одинаково, независимо от того, есть ли у нас обычная ссылка или тип, реализующий типаж `Deref`.

Система владения является причиной того, что метод `deref` возвращает ссылку на значение и того что простое разыменование за круглыми скобками в коде `(*(y.deref()))` все ещё необходимо. Если метод `deref` вернёт значение напрямую, а не ссылку на значение, то значение было бы перемещено из кода `self`. Мы не хотим забирать во владение внутреннее значение внутри типа `MyBox<T>` в этом случае или в большинстве случаев, когда используем оператор разыменования.

Обратите внимание, что оператор `*` заменён вызовом метода `deref`, а затем вызовом оператора `*` только один раз, каждый раз, когда мы используем `*` в коде. Поскольку замена оператора `*` не повторяется бесконечно, мы получаем данные типа `i32`, которые соответствуют `5` в `assert_eq!` листинга 15-9.

Неявные разыменованные приведения с функциями и методами

Разыменованное приведение (Deref coercion) - это удобство, которое Rust выполняет

над аргументами функций и методов. Разыменованное приведение преобразует ссылку на исходный тип, реализующий типаж **Deref**, в ссылку на целевой тип, в который **Deref** может преобразовать исходный тип. Разыменованное приведение происходит автоматически, когда мы передаём ссылку на значение определённого типа в качестве аргумента функции или метода, который не соответствует типу параметра в определении функции или метода. Последовательность вызовов метода **deref** преобразует предоставленный исходный тип, в целевой тип необходимый параметру.

Разыменованное приведение было добавлено в Rust, так что программистам, пишущим вызовы функций и методов, не нужно добавлять множество явных ссылок и разыменований с помощью использования `&` и `*`. Функциональность разыменованного приведения также позволяет писать больше кода, который может работать как с ссылками, так и с умными указателями.

Чтобы увидеть разыменованное приведение в действии, давайте воспользуемся типом **MyBox<T>** определённым в листинге 15-8, а также реализацией **Deref** добавленную в листинге 15-10. Листинг 15-11 показывает определение функции, у которой есть параметр типа срез строки:

Файл: src/main.rs

```
fn hello(name: &str) {
    println!("Hello, {}!", name);
}
```

Листинг 15-11: Функция **hello** имеющая параметр `name` типа `&str`

Можно вызвать функцию **hello** со срезом строки в качестве аргумента, например `hello("Rust");`. Разыменованное приведение делает возможным вызов **hello** со ссылкой на значение типа **MyBox<String>**, как показано в листинге 15-12.

Файл: src/main.rs

```
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&m);
}
```

Листинг 15-12: Вызов **hello** со ссылкой на значение **MyBox<String>**, которое работает из-за разыменованного приведения

Здесь мы вызываем функцию `hello` с аргументом `&m`, который является ссылкой на значение `MyBox<String>`. Поскольку мы реализовали типаж `Deref` для `MyBox<T>` в листинге 15-10, то Rust может преобразовать `&MyBox<String>` в `&String` вызывая `deref`. Стандартная библиотека предоставляет реализацию типажа `Deref` для типа `String`, которая возвращает срез строки, это описано в документации API типажа `Deref`. Rust снова вызывает `deref`, чтобы превратить `&String` в `&str`, что соответствует определению функции `hello`.

Если бы Rust не реализовал разыменованное приведение, мы должны были бы написать код в листинге 15-13 вместо кода в листинге 15-12 для вызова метода `hello` со значением типа `&MyBox<String>`.

Файл: src/main.rs

```
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&(*m)[..]);
}
```

Листинг 15-13: Код, который нам пришлось бы написать, если бы в Rust не было разыменованного приведения ссылок

Код `(*m)` разыменовывает `MyBox<String>` в `String`. Затем `&` и `[..]` принимают строковый срез `String`, равный всей строке, чтобы соответствовать сигнатуре `hello`. Код без разыменованного приведения сложнее читать, писать и понимать со всеми этими символами. Разыменованное приведение позволяет Rust обрабатывать эти преобразования для нас автоматически.

Когда типаж `Deref` определён для задействованных типов, Rust проанализирует типы и будет использовать `Deref::deref` столько раз, сколько необходимо, чтобы получить ссылку, соответствующую типу параметра. Количество раз, которое нужно вставить `Deref::deref` определяется во время компиляции, поэтому использование разыменованного приведения не имеет накладных расходов во время выполнения!

Как разыменованное приведение взаимодействует с изменяемостью

Подобно тому, как вы используете типаж `Deref` для переопределения оператора `*` у неизменяемых ссылок, вы можете использовать типаж `DerefMut` для

переопределения оператора `*` у изменяемых ссылок.

Rust выполняет разыменованное приведение, когда находит типы и реализации типажей в трёх случаях:

- Из типа `&T` в тип `&U` когда верно `T: Deref<Target=U>`
- Из типа `&mut T` в тип `&mut U` когда верно `T: DerefMut<Target=U>`
- Из типа `&mut T` в тип `&U` когда верно `T: Deref<Target=U>`

Первые два случая одинаковы за исключением изменяемости. В первом случае говорится, что если у вас есть тип `&T`, а `T` реализует типаж `Deref` для некоторого типа `U`, вы можете прозрачно получить `&U`. Во втором случае утверждается, что такое же разыменованное приведение происходит для изменяемых ссылок.

Третий случай хитрее: Rust также приводит изменяемую ссылку к неизменяемой. Но обратное *не* представляется возможным: неизменяемые ссылки никогда не приводятся к изменяемым ссылкам. Из-за правил заимствования, если у вас есть изменяемая ссылка, эта изменяемая ссылка должна быть единственной ссылкой на данные (в противном случае программа не будет компилироваться).

Преобразование одной изменяемой ссылки в неизменяемую ссылку никогда не нарушит правила заимствования. Преобразование неизменяемой ссылки в изменяемую ссылку потребует наличия только одной неизменяемой ссылки на эти данные, и правила заимствования не гарантируют этого. Следовательно, Rust не может сделать предположение, что преобразование неизменяемой ссылки в изменяемую ссылку возможно.

Запуск кода при очистке с помощью типажа `Drop`

Второй типаж, важный для шаблона умного указателя, - это `Drop`, который позволяет настроить то, что происходит, когда значение собирается выйти из области видимости. Вы можете предоставить реализацию `Drop` для любого типа, а указанный код можно использовать для освобождения ресурсов, таких как файлы или сетевые подключения. Мы представим `Drop` в контексте умных указателей, потому что функциональность типажа `Drop` почти всегда используется при их реализации. Например, `Box<T>` настраивает `Drop` для освобождения пространства в куче, на которое указывает `Box`.

В некоторых языках программист должен вызывать код для освобождения памяти или ресурсов каждый раз, когда они заканчивают использовать экземпляр умного указателя. Если программист забудет это сделать, то система может стать перегруженной и начать зависать. В Rust можно указать, что определённый код будет запускаться всякий раз, когда значение выходит из области видимости и компилятор вставит такой код автоматически. В результате вам не нужно заботиться о размещении кода очистки ресурсов во всей программе, где экземпляр определённого типа заканчивает существование - вы не потеряете системные ресурсы!

Укажите код, который будет запускаться, когда значение выходит из области видимости с помощью реализации типажа `Drop`. Типаж `Drop` требует, чтобы вы реализовали один метод с именем `drop`, который принимает изменяемую ссылку на `self`. Чтобы увидеть, когда Rust вызывает метод `drop`, давайте реализуем `drop` с помощью оператора `println!`.

В листинге 15-14 показана структура `CustomSmartPointer`, единственная функция которой заключается в том, что она будет печатать `Dropping CustomSmartPointer!` когда экземпляр выходит за область видимости. Этот пример демонстрирует, когда Rust выполняет функцию `drop`.

Файл: src/main.rs

```

struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer with data `{}`!", self.data);
    }
}

fn main() {
    let c = CustomSmartPointer {
        data: String::from("my stuff"),
    };
    let d = CustomSmartPointer {
        data: String::from("other stuff"),
    };
    println!("CustomSmartPointers created.");
}

```

Листинг 15-14: Структура `CustomSmartPointer`, которая реализует типаж `Drop`, где мы поместили бы код очистки

Типаж `Drop` входит в прелюдию, поэтому не нужно подключать его в область видимости. Мы реализуем типаж `Drop` у структуры `CustomSmartPointer` и предоставляем реализацию для метода `drop`, который вызывает `println!`. В теле функции `drop` можно разместить любую логику, которую хочется запустить, когда экземпляр вашего типа выходит из области видимости. Здесь мы печатаем некоторый текст, чтобы продемонстрировать, когда Rust вызовет `drop`.

В `main` мы создаём два экземпляра `CustomSmartPointer` и затем печатаем `CustomSmartPointers created`. В конце `main` наши экземпляры `CustomSmartPointer` выйдут из области видимости и Rust вызовет код, который мы добавили в метод `drop`, который и напечатает наше окончательное сообщение. Обратите внимание, что нам не нужно вызывать метод `drop` явно.

Когда мы запустим эту программу, мы увидим следующий вывод:

```

$ cargo run
Compiling drop-example v0.1.0 (file:///projects/drop-example)
Finished dev [unoptimized + debuginfo] target(s) in 0.60s
    Running `target/debug/drop-example`
CustomSmartPointers created.
Dropping CustomSmartPointer with data `other stuff`!
Dropping CustomSmartPointer with data `my stuff`!

```

Rust автоматически вызывает `drop`, когда наши экземпляры выходят из области видимости, вызывая указанный нами код. Переменные удаляются в обратном порядке их создания, поэтому `d` удалено до `c`. Этот пример даёт наглядное руководство о том, как работает метод `drop`; обычно вы указываете код очистки, который должен выполнить ваш тип, вместо печати сообщения как в этом примере.

Раннее удаление значения с помощью `std::mem::drop`

К сожалению, отключение функции автоматического удаления с помощью `drop` является не простым. Отключение `drop` обычно не требуется; весь смысл типажа `Drop` том, чтобы о функции позаботились автоматически. Иногда, однако, вы можете захотеть очистить значение рано. Одним из примеров является использование интеллектуальных указателей, которые управляют блокировками: вы могли бы потребовать принудительный вызов метода `drop` который снимает блокировку, чтобы другой код в той же области видимости мог получить блокировку. Rust не позволяет вызвать метод типажа `Drop` вручную; вместо этого вы должны вызвать функцию `std::mem::drop` предоставляемую стандартной библиотекой, если хотите принудительно удалить значение до конца области видимости.

Если попытаться вызвать метод `drop` типажа `Drop` вручную, изменяя функцию `main` листинга 15-14 так, как показано в листинге 15-15, мы получим ошибку компилятора:

Файл: src/main.rs

```
fn main() {
    let c = CustomSmartPointer {
        data: String::from("some data"),
    };
    println!("CustomSmartPointer created.");
    c.drop();
    println!("CustomSmartPointer dropped before the end of main.");
}
```



Листинг 15-15: Попытка вызвать метод `drop` из типажа `Drop` вручную для ранней очистки

Когда мы попытаемся скомпилировать этот код, мы получим ошибку:

```
$ cargo run
Compiling drop-example v0.1.0 (file:///projects/drop-example)
error[E0040]: explicit use of destructor method
--> src/main.rs:16:7
16 |     c.drop();
   |     ^^^^^^
   |     |
   |     explicit destructor calls not allowed
   |     help: consider using `drop` function: `drop(c)`
For more information about this error, try `rustc --explain E0040`.
error: could not compile `drop-example` due to previous error
```

Это сообщение об ошибке говорит, что мы не можем явно вызывать **drop**. В сообщении об ошибке используется термин *деструктор* (*destructor*), который является общим термином программирования для функции, которая очищает экземпляр. *Деструктор* аналогичен *конструктору*, который создаёт экземпляр. Функция **drop** в Rust является определённым деструктором.

Rust не позволяет явно вызывать **drop**, потому что Rust всё равно будет автоматически вызывать **drop** для значения в конце **main**. Это приведёт к ошибке *двойного освобождения*, потому что Rust будет пытаться очистить одно и то же значение дважды.

Мы не можем отключить автоматическую вставку **drop** кода, когда значение выходит из области видимости и мы не можем явно вызвать метод **drop**. Таким образом, если нам нужно принудительно очистить значение, мы можем использовать функцию **std::mem::drop**.

Функция **std::mem::drop** отличается от метода **drop** типажа **Drop**. Мы вызываем её, передавая значение, которое мы хотим принудительно освободить в качестве аргумента. Функция находится в прелюдии, поэтому можно изменить **main** из листинга 15-15 так, чтобы вызвать функцию **drop**, как показано в листинге 15-16:

Файл: src/main.rs

```
fn main() {
    let c = CustomSmartPointer {
        data: String::from("some data"),
    };
    println!("CustomSmartPointer created.");
    drop(c);
    println!("CustomSmartPointer dropped before the end of main.");
}
```

Листинг 15-16: Вызов `std::mem::drop` для явного удаления значения до его выхода из области видимости

Выполнение данного кода выведет следующий результат::

```
$ cargo run
Compiling drop-example v0.1.0 (file:///projects/drop-example)
Finished dev [unoptimized + debuginfo] target(s) in 0.73s
Running `target/debug/drop-example`
CustomSmartPointer created.
Dropping CustomSmartPointer with data `some data`!
CustomSmartPointer dropped before the end of main.
```

Текст `Dropping CustomSmartPointer with data some data!`, напечатанный между `CustomSmartPointer created.` и текстом `CustomSmartPointer dropped before the end of main.`, показывает, что код метода `drop` вызывается для удаления `c` в этой точке.

Вы можете использовать код, указанный в реализации типажа `Drop`, чтобы сделать очистку удобной и безопасной: например, вы можете использовать её для создания своего собственного менеджера памяти! С помощью типажа `Drop` и системы владения Rust не нужно специально заботиться о том, чтобы освобождать ресурсы, потому что Rust делает это автоматически.

Также не нужно беспокоиться о проблемах, возникающих в результате случайной очистки значений, которые всё ещё используются: система владения, которая гарантирует, что ссылки всегда действительны, также гарантирует, что `drop` вызывается только один раз, когда значение больше не используется.

После того, как мы познакомились с `Box<T>` и характеристиками умных указателей, познакомимся с её другими умными указателями, определёнными в стандартной библиотеке.

Rc<T>, умный указатель с подсчётом ссылок

В большинстве случаев владение является понятным: вы точно знаете, какой переменной принадлежит данное значение. Однако существуют случаи, когда одно значение может иметь несколько владельцев. Например, в структурах граф данных несколько рёбер могут указывать на один и тот же узел и этот узел концептуально принадлежит всем рёбрам, указывающим на него. Узел не должен быть очищен, до тех пор пока есть ребра, указывающие на него.

Чтобы разрешить множественное владение, в Rust есть тип `Rc<T>`, который является аббревиатурой для *подсчёта ссылок* (reference counting). Тип `Rc<T>` отслеживает количество ссылок на значение, чтобы определить, используется ли это значение. Если нет ссылок на значение, значение может быть очищено так, что ссылки стали недействительными.

Представьте себе `Rc<T>` как телевизор в гостиной. Когда один человек входит, чтобы смотреть телевизор, он включает его. Другие могут войти в комнату и посмотреть телевизор. Когда последний человек покидает комнату, он выключает телевизор, потому что он больше не используется. Если кто-то выключит телевизор во время его просмотра другими, то оставшиеся телезрители устроят шум!

Тип `Rc<T>` используется, когда мы хотим разместить в куче некоторые данные для чтения несколькими частями нашей программы и не можем определить во время компиляции, какая из частей завершит использование данных последней. Если бы мы знали, какая часть завершит использование последней то, мы могли бы сделать эту часть владельцем данных и вступили бы в силу обычные правила владения, применяемые во время компиляции.

Обратите внимание, что `Rc<T>` используется только в одно поточных сценариях. Когда мы обсудим конкурентность в главе 16, мы рассмотрим, как выполнять подсчёт ссылок во много поточных программах.

Использование `Rc<T>` для совместного использования данных

Давайте вернёмся к нашему примеру с `cons` списком в листинге 15-5. Напомним, что мы определили его с помощью типа `Box<T>`. В этот раз мы создадим два списка, оба из которых будут владеть третьим списком. Концептуально это похоже на рисунок 15-3:

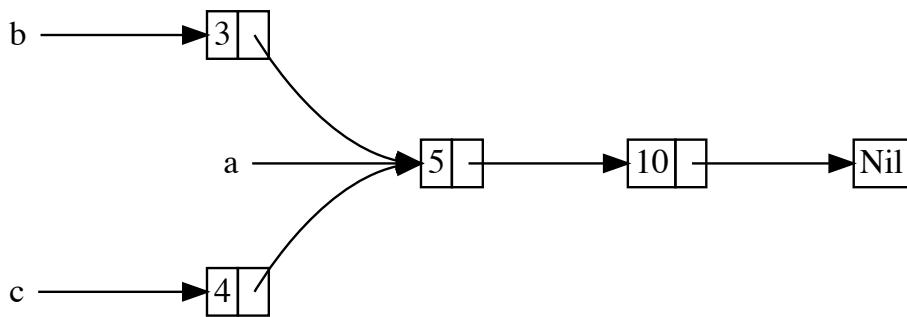


Рисунок 15-3: Два списка **b** и **c** совместно владеют третьим списком **a**

Мы создадим список **a**, содержащий 5 и затем 10. Затем мы создадим ещё два списка: **b** начинающийся с 3 и **c** начинающийся с 4. Оба списка **b** и **c** затем продолжать первый список **a**, содержащий 5 и 10. Другими словами, оба списка будут разделять первый список, содержащий 5 и 10.

Попытка реализовать этот сценарий, используя определение **List** с типом **Box<T>** не будет работать, как показано в листинге 15-17:

Файл: 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)))); // Error: Boxed list
    let b = Cons(3, Box::new(a)); // Error: Boxed list
    let c = Cons(4, Box::new(a)); // Error: Boxed list
}
  
```



Листинг 15-17: Демонстрация того, что нам не разрешено иметь два списка, использующих тип **Box<T>**, которые пытаются разделить владение третьим списком

При компиляции этого кода, мы получаем эту ошибку:

```
$ cargo run
Compiling cons-list v0.1.0 (file:///projects/cons-list)
error[E0382]: use of moved value: `a`
--> src/main.rs:11:30
|
9 |     let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));  

|         - move occurs because `a` has type `List`, which does not  

implement the `Copy` trait
10|     let b = Cons(3, Box::new(a));  

|                     - value moved here
11|     let c = Cons(4, Box::new(a));  

|                     ^ value used here after move

For more information about this error, try `rustc --explain E0382`.
error: could not compile `cons-list` due to previous error
```

Варианты `Cons` владеют данными, которые они содержат, поэтому, когда мы создаём список `b`, то `a` перемещается в `b`, а `b` становится владельцем `a`. Затем, мы пытаемся использовать `a` снова при создании `c`, но нам не разрешают, потому что `a` был перемещён.

Мы могли бы изменить определение `Cons`, чтобы вместо этого хранить ссылки, но тогда нам пришлось бы указывать параметры времени жизни. Указывая параметры времени жизни, мы бы указали, что каждый элемент в списке будет жить как минимум столько же, сколько и весь список. Это относится к элементам и спискам в листинге 15.17, но не во всех сценариях.

Вместо этого мы изменим наше определение типа `List` так, чтобы использовать `Rc<T>` вместо `Box<T>`, как показано в листинге 15-18. Каждый вариант `Cons` теперь будет содержать значение и тип `Rc<T>`, указывающий на `List`. Когда мы создадим `b` то, вместо того чтобы стал владельцем `a`, мы будем клонировать `Rc<List>` который содержит `a`, тем самым увеличивая количество ссылок с единицы до двойки и позволяя переменным `a` и `b` разделять владение на данные в типе `Rc<List>`. Мы также клонируем `a` при создании `c`, увеличивая количество ссылок с двух до трёх. Каждый раз, когда мы вызываем `Rc::clone`, счётчик ссылок на данные внутри `Rc<List>` будет увеличиваться и данные не будут очищены, если на них нет нулевых ссылок.

Файл: src/main.rs

```

enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}

```

Листинг 15-18: Определение `List`, который использует тип `Rc<T>`

Нам нужно добавить оператор `use`, чтобы подключить тип `Rc<T>` в область видимости, потому что он не входит в список автоматического импорта прелюдии. В `main`, мы создаём список владеющий 5 и 10, сохраняем его в новом `Rc<List>` переменной `a`. Затем при создании `b` и `c`, мы называем функцию `Rc::clone` и передаём ей ссылку на `Rc<List>` как аргумент `a`.

Мы могли бы вызвать `a.clone()`, а не `Rc::clone(&a)`, но в Rust принято использовать `Rc::clone` в таком случае. Внутренняя реализация `Rc::clone` не делает глубокого копирования всех данных, как это происходит в типах большинства реализаций `clone`. Вызов `Rc::clone` только увеличивает счётчик ссылок, что не занимает много времени. Глубокое копирование данных может занимать много времени. Используя `Rc::clone` для подсчёта ссылок, можно визуально различать виды клонирования с глубоким копированием и клонирования, которые увеличивают количество ссылок. При поиске в коде проблем с производительностью нужно рассмотреть только клонирование с глубоким копированием и игнорировать вызовы `Rc::clone`.

Клонирование `Rc<T>` увеличивает количество ссылок

Давайте изменим рабочий пример в листинге 15-18, чтобы увидеть как изменяется число ссылок при создании и удалении ссылок на `Rc<List>` внутри переменной `a`.

В листинге 15-19 мы изменим `main` так, чтобы она имела внутреннюю область видимости вокруг списка `c`; тогда мы сможем увидеть, как меняется счётчик ссылок при выходе `c` из внутренней области видимости.

Файл: src/main.rs

```
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("count after creating a = {}", Rc::strong_count(&a));
    let b = Cons(3, Rc::clone(&a));
    println!("count after creating b = {}", Rc::strong_count(&a));
    {
        let c = Cons(4, Rc::clone(&a));
        println!("count after creating c = {}", Rc::strong_count(&a));
    }
    println!("count after c goes out of scope = {}", Rc::strong_count(&a));
}
```

Листинг 15-19: Печать количества ссылок

В каждой точке программы, где изменяется счётчик ссылок, мы печатаем значение счётчика, которое можно получить вызовом функции `Rc::strong_count`. Эта функция называется `strong_count`, а не `count`, потому что тип `Rc<T>` также имеет функцию `weak_count`; мы увидим, для чего используется `weak_count` в разделе "Предотвращение циклических ссылок: превращением `Rc<T>` в `Weak<T>`".

Код выводит в консоль:

```
$ cargo run
Compiling cons-list v0.1.0 (file:///projects/cons-list)
Finished dev [unoptimized + debuginfo] target(s) in 0.45s
Running `target/debug/cons-list`
count after creating a = 1
count after creating b = 2
count after creating c = 3
count after c goes out of scope = 2
```

Можно увидеть, что `Rc<List>` в переменной `a` имеет начальный счётчик ссылок равный 1; затем каждый раз при вызове `clone` счётчик увеличивается на 1. Когда `c` выходит из области видимости, счётчик уменьшается на 1. Нам не нужно вызывать функцию уменьшения счётчика ссылок, как при вызове `Rc::clone` для увеличения счётчика ссылок: реализация `Drop` автоматически уменьшает счётчик ссылок, когда значение `Rc<T>` выходит из области видимости.

То чего мы не можем увидеть в этом примере является тем, что когда `b`, а затем `a` выходят из области видимости в конце функции `main`, счётчик становится равен 0 и `Rc<List>` полностью очищается в этом месте. Использование `Rc<T>` позволяет одному значению иметь нескольких владельцев, а подсчёт ссылок гарантирует, что

значение остаётся действительным до тех пор, пока существует любой из владельцев.

С помощью неизменяемых ссылок, тип `Rc<T>` позволяет обмениваться данными между несколькими частями вашей программы только для чтения данных. Если тип `Rc<T>` позволял бы иметь несколько изменяемых ссылок, вы могли бы нарушить одно из правил заимствования, описанных в главе 4: множественные изменяемые заимствования в одном и том же месте могут вызвать гонки данных (data races) и несогласованность данных. Но возможность изменять данные очень полезна! В следующем разделе мы обсудим шаблон внутренней изменчивости и тип `RefCell<T>`, который можно использовать вместе с `Rc<T>` для работы с этим ограничением.

RefCell<T> и шаблон внутренней изменяемости

Внутренняя изменяемость - это шаблон проектирования в Rust, позволяющий изменять данные даже если ссылки на эти данные неизменяемые. Обычно этого нельзя делать из-за правил владения. Для изменения данных, данный шаблон использует **unsafe** (небезопасный) код внутри структур данных, чтобы обойти обычные правила заимствования и изменяемости в Rust. Мы подробнее поговорим о небезопасном коде в Главе 19. Можно использовать типы, которые используют шаблон внутренней изменяемости, когда мы можем обеспечить соблюдение правил заимствования во время выполнения, даже если не возможно этого гарантировать при компиляции. Использованный **unsafe** небезопасный код помещается в безопасный API, а внешний тип остаётся неизменным.

Давайте изучим данную концепцию с помощью типа данных **RefCell<T>**, который реализует этот шаблон.

Применение правил заимствования во время выполнения с помощью RefCell<T>

В отличие от **Rc<T>** тип **RefCell<T>** предоставляет единоличное владение данными, которые он содержит. В чем же отличие типа **RefCell<T>** от **Box<T>**? Давайте вспомним правила заимствования из Главы 4:

- В любой момент времени у вас может быть либо (но не обе) одна изменяемая ссылка, либо любое количество неизменяемых ссылок.
- Ссылки всегда должны быть действительными.

С помощью ссылок и типа **Box<T>** инварианты правил заимствования применяются на этапе компиляции. С помощью **RefCell<T>** они применяются *во время работы программы*. Если вы нарушите эти правила, работая с ссылками, то будет ошибка компиляции. Если вы работаете с **RefCell<T>** и нарушите эти правила, то программа вызовет панику и завершится.

Преимущества проверки правил заимствования во время компиляции состоят в том, что ошибки будут обнаруживаться быстрее в процессе разработки и это не влияет на производительность во время выполнения программы, поскольку весь анализ выполняется заранее. По этим причинам проверка правил заимствования во время компиляции является лучшим выбором в большинстве случаев, именно поэтому она используется в Rust по умолчанию.

С другой стороны, преимущества проверки правил заимствования во время выполнения программы состоят в том, будут разрешены определённые сценарии безопасные с точки зрения работы с памятью, которые запрещены проверками во время компиляции. Статический анализ, как и компилятор Rust, по своей сути консервативен. Некоторые свойства кода невозможно обнаружить с помощью анализа кода: наиболее известным примером является проблема остановки, которая выходит за рамки этой книги, но представляет собой интересную тему для исследования.

Поскольку некоторый анализ невозможен, то если компилятор Rust не может быть уверен, что код соответствует правилам владения, он может отклонить корректную программу; таким образом он является консервативным. Если Rust принял некорректную программу, то пользователи не смогут доверять гарантиям, которые даёт Rust. Однако, если Rust отклонит корректную программу, то программист будет испытывать неудобства, но ничего катастрофического не произойдёт. Тип `RefCell<T>` полезен, когда вы уверены, что ваш код соответствует правилам заимствования, но компилятор не может понять и гарантировать этого.

Подобно типу `Rc<T>`, тип `RefCell<T>` предназначен только для использования в одно поточных сценариях и выдаст ошибку времени компиляции, если вы попытаетесь использовать его в много поточном контексте. Мы поговорим о том, как получить функциональность `RefCell<T>` во много поточной программе в главе 16.

Вот список причин выбора типов `Box<T>`, `Rc<T>` или `RefCell<T>`:

- Тип `Rc<T>` разрешает множественное владение одними и теми же данными; типы `Box<T>` и `RefCell<T>` разрешают иметь единственных владельцев.
- Тип `Box<T>` разрешает неизменяемые или изменяемые владения, проверенные при компиляции; тип `Rc<T>` разрешает только неизменяемые владения, проверенные при компиляции; тип `RefCell<T>` разрешает неизменяемые или изменяемые владения, проверенные во время выполнения.
- Поскольку `RefCell<T>` разрешает изменяемые заимствования, проверенные во время выполнения, можно изменять значение внутри `RefCell<T>` даже если `RefCell<T>` является неизменным.

Изменение значения внутри неизмененного значения является шаблоном *внутренней изменяемости* (interior mutability). Давайте посмотрим на ситуацию, в которой внутренняя изменяемость полезна и рассмотрим, как это возможно.

Внутренняя изменяемость: изменяемое заимствование неизменяемого значения

Следствием правил заимствования является то, что когда у вас есть неизменяемое значение, вы не можете заимствовать его с изменением. Например, этот код не будет компилироваться:

```
fn main() {  
    let x = 5;  
    let y = &mut x;  
}
```



Если вы попытаетесь скомпилировать этот код, вы получите следующую ошибку:

```
$ cargo run  
Compiling borrowing v0.1.0 (file:///projects/borrowing)  
error[E0596]: cannot borrow `x` as mutable, as it is not declared as mutable  
--> src/main.rs:3:13  
|  
2 |     let x = 5;  
|         - help: consider changing this to be mutable: `mut x`  
3 |     let y = &mut x;  
|             ^^^^^^ cannot borrow as mutable  
  
For more information about this error, try `rustc --explain E0596`.  
error: could not compile `borrowing` due to previous error
```

Однако существуют ситуации в которых было бы полезно, чтобы значение изменяло само себя в своих методах, ноказалось неизменным для другого кода. Код за пределами таких методов над значениями не мог бы изменить сами значения. Использование `RefCell<T>` является одним из способов получить внутреннюю изменяемость. Но `RefCell<T>` не обходит правила заимствования полностью: анализатор заимствования компилятора допускает эту внутреннюю изменяемость и вместо этого правила заимствования проверяются во время выполнения. Если вы нарушаете правила, вы получите `panic!` вместо ошибки компилятора.

Давайте разберём практический пример, в котором мы можем использовать `RefCell<T>` для изменения неизменяемого значения и посмотрим, почему это полезно.

Вариант использования внутренней изменяемости: мок объекты

Тест дубликаты - это общая программная концепция для типа, используемого вместо другого типа во время тестирования. *Мок объекты* - это особые типы тест дубликаторов, которые записывают то, что происходит во время теста, поэтому после прохождения теста можно утверждать, что выполнились правильные действия в мок объекте.

В Rust нет объектов в том же смысле, в каком они есть в других языках и в Rust нет функциональности мок объектов, встроенных в стандартную библиотеку, как в некоторых других языках. Однако вы определённо можете создать структуру, которая будет служить тем же целям, что и мок объект.

Вот сценарий, который мы будем тестировать: мы создадим библиотеку, которая отслеживает значение по отношению к заранее определённому максимальному значению и отправляет сообщения в зависимости от того, насколько текущее значение находится близко к такому максимальному значению. Эта библиотека может использоваться, например, для отслеживания квоты количества вызовов API пользователя, которые ему разрешено делать.

Наша библиотека будет предоставлять только функции отслеживания того, насколько близко к максимальному значению находится значение и какие сообщения должны быть внутри в этот момент. Ожидается, что приложения, использующие нашу библиотеку, предоставят механизм для отправки сообщений: приложение может поместить сообщение в приложение, отправить электронное письмо, отправить текстовое сообщение или что-то ещё. Библиотеке не нужно знать эту деталь. Все что ему нужно - это что-то, что реализует типаж, который мы предоставим с названием **Messenger**. Листинг 15-20 показывает код библиотеки:

Файл: src/lib.rs

```

pub trait Messenger {
    fn send(&self, msg: &str);
}

pub struct LimitTracker<'a, T: Messenger> {
    messenger: &'a T,
    value: usize,
    max: usize,
}

impl<'a, T> LimitTracker<'a, T>
where
    T: Messenger,
{
    pub fn new(messenger: &'a T, max: usize) -> LimitTracker<'a, T> {
        LimitTracker {
            messenger,
            value: 0,
            max,
        }
    }

    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("Error: You are over your quota!");
        } else if percentage_of_max >= 0.9 {
            self.messenger
                .send("Urgent warning: You've used up over 90% of your
quota!");
        } else if percentage_of_max >= 0.75 {
            self.messenger
                .send("Warning: You've used up over 75% of your quota!");
        }
    }
}

```

Листинг 15-20: Библиотека, которая отслеживает, насколько близко текущее значение находится по величине к максимальному значению и предупреждает, когда текущее значение находится на определённых уровнях своей величины.

Одна важная часть этого кода состоит в том, что типаж `Messenger` имеет один метод `send`, принимающий аргументами неизменяемую ссылку на `self` и текст сообщения. Он является интерфейсом, который должен иметь наш мок объект. Другой важной частью является то, что мы хотим проверить поведение метода

`set_value` у типа `LimitTracker`. Мы можем изменить значение, которое передаём параметром `value`, но `set_value` ничего не возвращает и нет основания, чтобы мы могли бы проверить утверждения о выполнении метода. Мы хотим иметь возможность сказать, что если мы создаём `LimitTracker` с чем-то, что реализует типаж `Messenger` и с определённым значением для `max`, то когда мы передаём разные числа в переменной `value` экземпляр `self.messenger` отправляет соответствующие сообщения.

Нам нужен мок объект, который вместо отправки электронного письма или текстового сообщения будет отслеживать сообщения, которые были ему поручены для отправки через `send`. Мы можем создать новый экземпляр мок объекта, создать `LimitTracker` с использованием мок объекта для него, вызвать метод `set_value` у экземпляра `LimitTracker`, а затем проверить, что мок объект имеет ожидаемое сообщение. В листинге 15-21 показана попытка реализовать мок объект, чтобы сделать именно то что хотим, но анализатор заимствований не разрешит такой код:

Файл: src/lib.rs



```

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

    struct MockMessenger {
        sent_messages: Vec<String>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger {
                sent_messages: vec![],
            }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.push(String::from(message));
        }
    }

    #[test]
    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);
    }
}

```

Листинг 15-21: Попытка реализовать `MockMessenger`, код которого не разрешён анализатором заимствований

Этот тестовый код определяет структуру `MockMessenger`, в которой есть поле `sent_messages` со значениями типа `Vec` из `String` для отслеживания сообщений, которые поручены структуре для отправки. Мы также определяем ассоциированную функцию `new`, чтобы было удобно создавать новые экземпляры `MockMessenger`, которые создаются с пустым списком сообщений. Затем мы реализуем типаж `Messenger` для типа `MockMessenger`, чтобы передать `MockMessenger` в `LimitTracker`. В сигнатуре метода `send` мы принимаем сообщение для передачи в качестве параметра и сохраняем его в `MockMessenger` внутри списка `sent_messages`.

В этом тесте мы проверяем, что происходит, когда `LimitTracker` сказано установить `value` в значение, превышающее 75 процентов от значения `max`. Сначала мы создаём новый `MockMessenger`, который будет иметь пустой список сообщений. Затем мы создаём новый `LimitTracker` и передаём ему ссылку на новый `MockMessenger` и `max` значение равное 100. Мы вызываем метод `set_value` у `LimitTracker` со значением 80, что составляет более 75 процентов от 100. Затем мы с помощью утверждения проверяем, что `MockMessenger` должен содержать одно сообщение из списка внутренних сообщений.

Однако с этим тестом есть одна проблема, показанная ниже:

```
$ cargo test
Compiling limit-tracker v0.1.0 (file:///projects/limit-tracker)
error[E0596]: cannot borrow `self.sent_messages` as mutable, as it is behind
a `&` reference
--> src/lib.rs:58:13
  |
2 |     fn send(&self, msg: &str);
  |             ----- help: consider changing that to be a mutable
reference: `&mut self`
...
58 |         self.sent_messages.push(String::from(message));
  |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ `self` is a
`&` reference, so the data it refers to cannot be borrowed as mutable

For more information about this error, try `rustc --explain E0596`.
error: could not compile `limit-tracker` due to previous error
warning: build failed, waiting for other jobs to finish...
error: build failed
```

Мы не можем изменять `MockMessenger` для отслеживания сообщений, потому что метод `send` принимает неизменяемую ссылку на `self`. Мы также не можем принять предложение из текста ошибки, чтобы использовать `&mut self`, потому что тогда сигнатура `send` не будет соответствовать сигнатуре в определении типажа `Messenger` (не стесняйтесь попробовать и посмотреть, какое сообщение об ошибке получите вы).

Это ситуация, в которой внутренняя изменяемость может помочь! Мы сохраним `sent_messages` внутри типа `RefCell<T>`, а затем в методе `send` сообщение сможет изменить список `sent_messages` для хранения сообщений, которые мы видели. Листинг 15-22 показывает, как это выглядит:

Файл: `src/lib.rs`

```

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

    use std::cell::RefCell;

    struct MockMessenger {
        sent_messages: RefCell<Vec<String>>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger {
                sent_messages: RefCell::new(vec![]),
            }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.borrow_mut().push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        // --snip--

        assert_eq!(mock_messenger.sent_messages.borrow().len(), 1);
    }
}

```

Листинг 15-22: Использование `RefCell` для изменения внутреннего значения, в то время как внешнее значение считается неизменным

Поле `sent_messages` теперь имеет тип `RefCell<Vec<String>>` вместо `Vec<String>`. В функции `new` мы создаём новый экземпляр `RefCell<Vec<String>>` для пустого вектора.

Для реализации метода `send` первый параметр по-прежнему является неизменяемым для заимствования `self`, которое соответствует определению типажа. Мы вызываем `borrow_mut` для `RefCell<Vec<String>>` в `self.sent_messages`, чтобы получить изменяемую ссылку на значение внутри `RefCell<Vec<String>>`, которое является вектором. Затем мы можем вызвать `push` у изменяемой ссылки на вектор, чтобы отслеживать сообщения, отправленные во время теста.

Последнее изменение, которое мы должны сделать, заключается в утверждении для

проверки: чтобы увидеть, сколько элементов находится во внутреннем векторе, мы вызываем метод `borrow` у `RefCell<Vec<String>>`, чтобы получить неизменяемую ссылку на внутренний вектор сообщений.

Теперь, когда вы увидели как использовать `RefCell<T>`, давайте изучим как он работает!

Отслеживание заимствований во время выполнения с помощью `RefCell<T>`

При создании неизменных и изменяемых ссылок мы используем синтаксис `&` и `&mut` соответственно. У типа `RefCell<T>`, мы используем методы `borrow` и `borrow_mut`, которые являются частью безопасного API, который принадлежит `RefCell<T>`. Метод `borrow` возвращает тип умного указателя `Ref<T>`, метод `borrow_mut` возвращает тип умного указателя `RefMut<T>`. Оба типа реализуют типаж `Deref`, поэтому мы можем рассматривать их как обычные ссылки.

Тип `RefCell<T>` отслеживает сколько умных указателей `Ref<T>` и `RefMut<T>` активны в данное время. Каждый раз, когда мы вызываем `borrow`, тип `RefCell<T>` увеличивает количество активных заимствований. Когда значение `Ref<T>` выходит из области видимости, то количество неизменяемых заимствований уменьшается на единицу. Как и с правилами заимствования во время компиляции, `RefCell<T>` позволяет иметь много неизменяемых заимствований или одно изменяемое заимствование в любой момент времени.

Если попытаться нарушить эти правила, то вместо получения ошибки компилятора, как это было бы со ссылками, реализация `RefCell<T>` будет вызывать панику во время выполнения. В листинге 15-23 показана модификация реализации `send` из листинга 15-22. Мы намеренно пытаемся создать два изменяемых заимствования активных для одной и той же области видимости, чтобы показать как `RefCell<T>` не позволяет нам делать так во время выполнения.

Файл: `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));
    }
}
```



Листинг 15-23. Создание двух изменяемых ссылок в одной и той же области видимости, чтобы увидеть как `RefCell` будет паниковать.

Мы создаём переменную `one_borrow` для умного указателя `RefMut<T>` возвращаемого из метода `borrow_mut`. Затем мы создаём другое изменяемое заимствование таким же образом в переменной `two_borrow`. Это создаёт две изменяемые ссылки в одной области видимости, что недопустимо. Когда мы запускаем тесты для нашей библиотеки, код в листинге 15-23 компилируется без ошибок, но тест завершится неудачно:

```
$ cargo test
Compiling limit-tracker v0.1.0 (file:///projects/limit-tracker)
Finished test [unoptimized + debuginfo] target(s) in 0.91s
Running unitests (target/debug/deps/limit_tracker-e599811fa246dbde)

running 1 test
test tests::it_sends_an_over_75_percent_warning_message ... FAILED

failures:

---- tests::it_sends_an_over_75_percent_warning_message stdout ----
thread 'main' panicked at 'already borrowed: BorrowMutError',
src/lib.rs:60:53
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
    tests::it_sends_an_over_75_percent_warning_message

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Обратите внимание, что код вызвал панику с сообщением `already borrowed: BorrowMutError`. Вот так тип `RefCell<T>` обрабатывает нарушения правил

заемствования во время выполнения.

Выявление ошибок заимствования во время выполнения, а не во время компиляции означает, что вы обнаружите ошибку в своём коде позже в процессе разработки, а возможно и когда ваш код будет развернут в производство. Кроме того, ваш код получает небольшое снижение производительности в результате отслеживания заимствований во время выполнения, а не во время компиляции. Тем не менее, использование `RefCell<T>` позволяет написать мок объект, который может изменять себя так, чтобы отслеживать отправляемые сообщения, когда вы используете его в контексте где разрешены только неизменяемые значения. Несмотря на его компромиссы и уступки `RefCell<T>` можно использовать, чтобы получить больше функциональности чем позволяют обычные ссылки.

Наличие нескольких владельцев изменяемых данных путём объединения типов `Rc<T>` и `RefCell<T>`

Обычный способ использования `RefCell<T>` заключается в его сочетании с типом `Rc<T>`. Напомним, что тип `Rc<T>` позволяет иметь нескольких владельцев некоторых данных, но даёт только неизменяемый доступ к этим данным. Если у вас есть `Rc<T>`, который внутри содержит тип `RefCell<T>`, вы можете получить значение, которое может иметь несколько владельцев и которое можно изменять!

Например, вспомните пример `cons` списка листинга 15-18, где мы использовали `Rc<T>`, чтобы несколько списков могли совместно владеть другим списком. Поскольку `Rc<T>` содержит только неизменяемые значения, мы не можем изменить ни одно из значений в списке после того, как мы их создали. Давайте добавим тип `RefCell<T>`, чтобы получить возможность изменять значения в списках. В листинге 15-24 показано использование `RefCell<T>` в определении `Cons` так, что мы можем изменить значение хранящееся во всех списках:

Файл: src/main.rs

```

#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));
    let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));

    *value.borrow_mut() += 10;

    println!("a after = {:?}", a);
    println!("b after = {:?}", b);
    println!("c after = {:?}", c);
}

```

Листинг 15-24: Использование `Rc<RefCell>` для создания `List`, который можно изменять.

Мы создаём значение, которое является экземпляром `Rc<RefCell<i32>>` и сохраняем его в переменной с именем `value`, чтобы получить к ней прямой доступ позже. Затем мы создаём `List` в переменной `a` с вариантом `Cons`, который содержит `value`. Нам нужно вызвать клонирование `value`, так как обе переменные `a` и `value` владеют внутренним значением `5`, а не передают владение из `value` в переменную `a` или не выполняют заимствование с помощью `a` переменной `value`.

Мы оборачиваем список у переменной `a` в тип `Rc<T>`, поэтому при создании списков в переменные `b` и `c` они оба могут ссылаться на `a`, что мы и сделали в листинге 15-18.

После того, как мы создали списки в переменных `a`, `b` и `c`, мы добавляем число 10 к значению внутри `value`. Мы делаем это, вызывая метод `borrow_mut` у `value`, которое использует функцию автоматической разыменования, обсуждавшуюся в главе 5 (см. раздел "Где находится оператор `-> ?`") для разыменования `Rc<T>` до внутреннего значения `RefCell<T>`. Метод `borrow_mut` возвращает умный указатель типа `RefMut<T>` и мы используем для него оператор разыменования и меняем

внутреннее значение.

Когда мы печатаем `a`, `b` и `c` то видим, что все они имеют изменённое значение равное 15, а не 5:

```
$ cargo run
Compiling cons-list v0.1.0 (file:///projects/cons-list)
Finished dev [unoptimized + debuginfo] target(s) in 0.63s
    Running `target/debug/cons-list`
a after = Cons(RefCell { value: 15 }, Nil)
b after = Cons(RefCell { value: 3 }, Cons(RefCell { value: 15 }, Nil))
c after = Cons(RefCell { value: 4 }, Cons(RefCell { value: 15 }, Nil))
```

Эта техника довольно изящна! Используя `RefCell<T>`, мы получаем внешне неизменяемое значение `List`. Но мы можем использовать методы типа `RefCell<T>`, которые обеспечивают доступ к его внутренней изменяемости так, что мы можем менять данные при необходимости. Проверки правил заимствования во время выполнения защищают нас от гонки данных и иногда стоит поступиться некоторой скоростью в пользу гибкости структурах данных.

Стандартная библиотека имеет другие типы, которые обеспечивают внутреннюю изменяемость, например тип `Cell<T>` аналогичен, за исключением того, что вместо предоставления ссылки на внутреннее значение, значение копируется внутрь `Cell<T>` и изнутри наружу. Есть также тип `Mutex<T>`, который предлагает внутреннюю изменяемость, которую можно безопасно использовать в разных потоках; мы обсудим его использование в главе 16. Посмотрите документацию стандартной библиотеки для получения более подробной информации о различиях между этими типами.

Ссылочные зацикливания могут приводить к утечке памяти

Гарантии безопасности памяти в Rust затрудняют, но не делают невозможным случайное выделение памяти, которое никогда не очищается (известное как *утечка памяти*). Полное предотвращение утечек памяти не является одной из гарантий Rust, а это означает, что утечки памяти безопасны в Rust. Мы видим, что Rust допускает утечку памяти с помощью `Rc<T>` и `RefCell<T>`: можно создавать ссылки, в которых элементы ссылаются друг на друга в цикле. Это создаёт утечки памяти, потому что счётчик ссылок каждого элемента в цикле никогда не достигнет 0, а значения никогда не будут удалены.

Создание ссылочного зацикливания

Давайте посмотрим, как может произойти ситуация ссылочного зацикливания и как её предотвратить, начиная с определения перечисления `List` и метода `tail` в листинге 15-25:

Файл: src/main.rs

```
use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match self {
            Cons(_, item) => Some(item),
            Nil => None,
        }
    }
}

fn main() {}
```

Листинг 15-25: Определение списка `cons`, который содержит `RefCell`, так что можно изменять то, на

что ссылается вариант `Cons`

Мы используем другой вариант определения `List` из листинга 15-5. Второй элемент в варианте `Cons` теперь типа `RefCell<Rc<List>>`, что означает, что вместо возможности изменять значение `i32` как мы делали в листинге 15-24, мы хотим изменить значение у типа `List` на которое указывает вариант `Cons`. Мы также добавляем метод `tail`, чтобы было удобно получить доступ ко второму элементу, если у нас есть вариант `Cons`.

В листинге 15-26 мы добавляем `main` функцию, которая использует определения листинга 15-25. Этот код создаёт список в переменной `a` и список `b`, который указывает на список `a`. Затем он изменяет список внутри `a` так, чтобы он указывал на `b`, создавая ссылочное зацикливание. В коде есть инструкции `println!`, чтобы показать значения счётчиков ссылок в различных точках этого процесса.

Файл: src/main.rs

```
fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    println!("a initial rc count = {}", Rc::strong_count(&a));
    println!("a next item = {:?}", a.tail());

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    println!("a rc count after b creation = {}", Rc::strong_count(&a));
    println!("b initial rc count = {}", Rc::strong_count(&b));
    println!("b next item = {:?}", b.tail());

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }

    println!("b rc count after changing a = {}", Rc::strong_count(&b));
    println!("a rc count after changing a = {}", Rc::strong_count(&a));

    // Uncomment the next line to see that we have a cycle;
    // it will overflow the stack
    // println!("a next item = {:?}", a.tail());
}
```

Листинг 15-26: Создание ссылочного зацикливания из двух значений `List` указывающих друг на друга

Мы создаём экземпляр `Rc<List>` содержащий значение `List` в переменной `a` с

начальным списком `5, Nil`. Затем мы создаём экземпляр `Rc<List>` содержащий другое значение `List` в переменной `b`, которое содержит значение 10 и указывает на список в `a`.

Мы меняем `a` так, чтобы он указывал на `b` вместо `Nil`, создавая зацикленность. Мы делаем это с помощью метода `tail`, чтобы получить ссылку на `RefCell<Rc<List>>` из переменной `a`, которую мы помещаем в переменную `link`. Затем мы используем метод `borrow_mut` из типа `RefCell<Rc<List>>`, чтобы изменить внутреннее значение типа `Rc<List>`, содержащего начальное значение `Nil` на значение типа `Rc<List>` взятое из переменной `b`.

Когда мы запускаем этот код, оставив последний `println!` закомментированным в данный момент, мы получим вывод:

```
$ cargo run
Compiling cons-list v0.1.0 (file:///projects/cons-list)
  Finished dev [unoptimized + debuginfo] target(s) in 0.53s
    Running `target/debug/cons-list`
a initial rc count = 1
a next item = Some(RefCell { value: Nil })
a rc count after b creation = 2
b initial rc count = 1
b next item = Some(RefCell { value: Cons(5, RefCell { value: Nil }) })
b rc count after changing a = 2
a rc count after changing a = 2
```

Счётчик ссылок экземпляров `Rc<List>` в обоих переменных `a` и `b` равен 2 после того, как мы изменяем список внутри `a`, чтобы он указывал на `b`. В конце `main` Rust сначала попытается удалить `b`, что уменьшит количество экземпляров `Rc<List>` с 2 на 1. Память, на которую `Rc<List>` указывает в куче, на этом этапе не будет удалена, потому что её счётчик ссылок равен 1, а не 0. Затем Rust удаляет `a`, что также уменьшает счётчик ссылок экземпляра `a` `Rc<List>` с 2 до 1. Память этого экземпляра также не может быть удалена, потому что другой экземпляр `Rc<List>` по-прежнему ссылается на неё. Память, выделенная для списка, навсегда останется не освобождённой. Чтобы наглядно представить этот эталонный цикл, мы создали диаграмму на Рисунке 15-4.

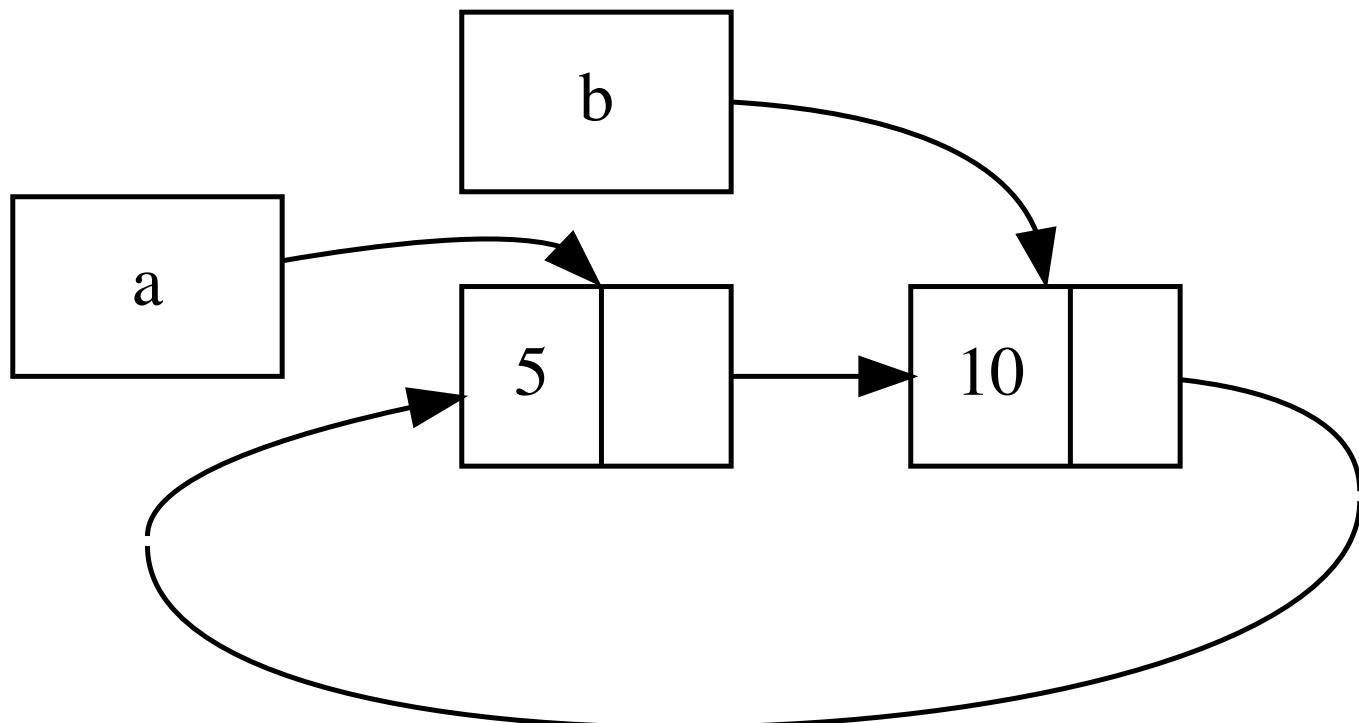


Рисунок 15-4: Ссыльочная зацикленность списков `a` и `b` указывающих друг на друга

Если вы удалите последний комментарий с `println!` и запустите программу, Rust будет пытаться печатать зацикленность в `a`, указывающей на `b`, указывающей на `a` и так далее, пока не переполниться стек.

В этом случае, сразу после создания ссылочной зацикленности, программа завершается. Последствия зацикленности не очень страшны. Однако, если более сложная программа выделяет много памяти в цикле и удерживает её в течение длительного времени, программа использует больше памяти, чем необходимо, что может перегрузить систему вызывая утечки памяти.

Вызвать образование ссылочной зацикленности не просто, но и не невозможно. Если у вас есть значения `RefCell<T>` которые содержат значения `Rc<T>` или аналогичные вложенные комбинации типов с внутренней изменчивостью и подсчётом ссылок, вы должны убедиться, что вы не создаёте зацикленность; Вы не можете полагаться на то, что Rust их обнаружит. Создание ссылочной зацикленности являлось бы логической ошибкой в программе, для которой вы должны использовать автоматические тесты, проверку кода и другие практики разработки программного обеспечения для её минимизации.

Другое решение для избежания ссылочной зацикленности - это реорганизация ваших структур данных, чтобы некоторые ссылки выражали владение, а другие -

отсутствие владения. В результате можно иметь циклы, построенные на некоторых отношениях владения и некоторые не основанные на отношениях владения, тогда только отношения владения влияют на то, можно ли удалить значение. В листинге 15-25 мы всегда хотим, чтобы варианты `Cons` владели своим списком, поэтому реорганизация структуры данных невозможна. Давайте рассмотрим пример с использованием графов, состоящих из родительских и дочерних узлов, чтобы увидеть, когда отношения владения не являются подходящим способом предотвращения ссылочной зацикленности.

Предотвращение ссылочной зацикленности: замена умного указателя `Rc<T>` на `Weak<T>`

До сих пор мы демонстрировали, что вызов `Rc::clone` увеличивает значение `strong_count` экземпляра типа `Rc<T>`, а экземпляр `Rc<T>` очищается только в том случае, если его значение `strong_count` равно 0. Вы также можете создать *слабую ссылку* (weak reference) на значение внутри экземпляра `Rc<T>` путём вызова `Rc::downgrade` и передачи ссылки на `Rc<T>`. Когда вы вызываете `Rc::downgrade`, вы получаете умный указатель типа `Weak<T>`. Вместо увеличения значения `strong_count` в экземпляре `Rc<T>` на 1, вызов `Rc::downgrade` увеличивает значение `weak_count` на 1. Тип `Rc<T>` использует `weak_count` для отслеживания того, сколько существует `Weak<T>` ссылок, аналогично `strong_count`. Разница заключается в том, что `weak_count` не должно быть равно 0, чтобы очистить такой экземпляр типа `Rc<T>`.

Сильные ссылки - это способ, которым вы можете делиться владением экземпляра `Rc<T>`. Слабые ссылки не выражают отношения владения. Они не будут вызывать ссылочную зацикленность, потому что любой цикл, включающий некоторые слабые ссылки, будет прерван, как только счётчик сильных ссылок вовлечённых значений будет равен 0.

Поскольку значение, на которое ссылается `Weak<T>` могло быть удалено, то необходимо убедиться, что это значение все ещё существует, чтобы сделать что-либо со значением на которое указывает `Weak<T>`. Делайте это вызывая метод `upgrade` у экземпляра типа `Weak<T>`, который вернёт `Option<Rc<T>>`. Вы получите результат `Some`, если значение `Rc<T>` ещё не было удалено и результат `None`, если значение `Rc<T>` было удалено. Поскольку `upgrade` возвращает тип `Option<T>`, Rust обеспечит обработку обоих случаев `Some` и `None` и не будет некорректного указателя.

В качестве примера, вместо того чтобы использовать список чей элемент знает только о следующем элементе, мы создадим дерево, чьи элементы знают о своих дочерних элементах и о своих родительских элементах.

Создание древовидной структуры данных: `Node` с дочерними узлами

Для начала мы построим дерево с узлами, которые знают о своих дочерних узлах. Мы создадим структуру с именем `Node`, которая будет содержать собственное значение `i32`, а также ссылки на его дочерние значения `Node`:

Файл: src/main.rs

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

#[derive(Debug)]
struct Node {
    value: i32,
    children: RefCell<Vec<Rc<Node>>,
}
```

Мы хотим, чтобы `Node` владел своими дочерними узлами и мы хотим поделиться этим владением с переменными так, чтобы мы могли напрямую обращаться к каждому `Node` в дереве. Для этого мы определяем внутренние элементы типа `Vec<T>` как значения типа `Rc<Node>`. Мы также хотим изменять те узлы, которые являются дочерними по отношению к другому узлу, поэтому у нас есть тип `RefCell<T>` в поле `children` обрабатывающий тип `Vec<Rc<Node>>`.

Далее мы будем использовать наше определение структуры и создадим один экземпляр `Node` с именем `leaf` со значением 3 и без дочерних элементов, а другой экземпляр с именем `branch` со значением 5 и `leaf` в качестве одного из его дочерних элементов, как показано в листинге 15-27:

Файл: 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)]),
    });
}
```

Листинг 15-27. Создание узла `leaf` без дочерних узлов и узла `branch` с `leaf` как одним дочерним узлом

Мы клонируем содержимое `Rc<Node>` из переменной `leaf` и сохраняем его в переменной `branch`, что означает, что `Node` в `leaf` теперь имеет двух владельцев: `leaf` и `branch`. Мы можем получить доступ из `branch` к `leaf` через обращение `branch.children`, но нет способа добраться из `leaf` к `branch`. Причина в том, что `leaf` не имеет ссылки на `branch` и не знает, что они связаны. Мы хотим, чтобы `leaf` знал, что `branch` является его родителем. Мы сделаем это далее.

Добавление ссылки от ребёнка к его родителю

Для того, чтобы дочерний узел знал о своём родительском узле нужно добавить поле `parent` в наше определение структуры `Node`. Проблема в том, чтобы решить, каким должен быть тип `parent`. Мы знаем, что он не может содержать `Rc<T>`, потому что это создаст ссылочную зацикленность с `leaf.parent` указывающей на `branch` и `branch.children`, указывающей на `leaf`, что приведёт к тому, что их значения `strong_count` никогда не будут равны 0.

Подумаем об этих отношениях по-другому, родительский узел должен владеть своими потомками: если родительский узел удаляется, его дочерние узлы также должны быть удалены. Однако дочерний элемент не должен владеть своим родителем: если мы удаляем дочерний узел то родительский элемент все равно должен существовать. Это случай для использования слабых ссылок!

Поэтому вместо `Rc<T>` мы сделаем так, чтобы поле `parent` использовало тип `Weak<T>`, а именно `RefCell<Weak<Node>>`. Теперь наше определение структуры `Node` выглядит так:

Файл: src/main.rs

```
use std::cell::RefCell;
use std::rc::{Rc, Weak};

#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}
```

Узел сможет ссылаться на свой родительский узел, но не владеет своим родителем. В листинге 15-28 мы обновляем `main` на использование нового определения так, чтобы у узла `leaf` был бы способ ссылаться на его родительский узел `branch`:

Файл: src/main.rs

```
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());

    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);

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
}
```

Листинг 15-28: Узел `leaf` со слабой ссылкой на свой родительский узел `branch`.

Создание узла `leaf` выглядит так же, как при создании узла `leaf` в листинге 15-27, за исключением поля `parent`: узел `leaf` начинается без родителя, поэтому мы создаём новый пустой ссылочный экземпляр `Weak<Node>`.

На этом этапе, когда мы пытаемся получить ссылку на родительский узел у узла `leaf` с помощью метода `upgrade`, мы получаем значение `None`. Мы видим это в выводе первого `println!` выражения:

```
leaf parent = None
```

Когда мы создаём узел `branch` у него также будет новая ссылка типа `Weak<Node>` в поле `parent`, потому что узел `branch` не имеет своего родительского узла. У нас все ещё есть `leaf` как один из потомков узла `branch`. Когда мы получили экземпляр `Node` в переменной `branch`, мы можем изменить переменную `leaf` чтобы дать ей `Weak<Node>` ссылку на её родителя. Мы используем метод `borrow_mut` у типа `RefCell<Weak<Node>>` поля `parent` у `leaf`, а затем используем функцию `Rc::downgrade` для создания `Weak<Node>` ссылки на `branch` из `Rc<Node>` в `branch`.

Когда мы снова напечатаем родителя `leaf` то в этот раз мы получим вариант `Some` содержащий `branch`, теперь `leaf` может получить доступ к своему родителю! Когда мы печатаем `leaf`, мы также избегаем цикла, который в конечном итоге заканчивался переполнением стека, как в листинге 15-26; ссылки типа `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 означает, что этот код не создал ссылочной зацикленности. Мы также можем сказать это, посмотрев на значения, которые мы получаем при вызове `Rc::strong_count` и `Rc::weak_count`.

Визуализация изменений в `strong_count` и `weak_count`

Давайте посмотрим, как изменяются значения `strong_count` и `weak_count` экземпляров типа `Rc<Node>` с помощью создания новой внутренней области видимости и перемещая создания экземпляра `branch` в эту область. Таким образом можно увидеть, что происходит, когда `branch` создаётся и затем удаляется при выходе из области видимости. Изменения показаны в листинге 15-29:

Файл: src/main.rs

```

fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );

    {
        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);

        println!(
            "branch strong = {}, weak = {}",
            Rc::strong_count(&branch),
            Rc::weak_count(&branch),
        );
    }

    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );
}

println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
println!(
    "leaf strong = {}, weak = {}",
    Rc::strong_count(&leaf),
    Rc::weak_count(&leaf),
);
}

```

Листинг 15-29: Создание `branch` во внутренней области и проверка сильных и слабых ссылок

После того, как `leaf` создан его `Rc<Node>` имеет значения `strong count` равное 1 и `weak count` равное 0. Во внутренней области мы создаём `branch` и связываем её с `leaf`, после чего при печати значений счётчиков `Rc<Node>` в `branch` они будут

иметь strong count 1 и weak count 1 (для `leaf.parent` указывающего на `branch` с `Weak<Node>`). Когда мы распечатаем счётчики из `leaf`, мы увидим, что они будут иметь strong count 2, потому что `branch` теперь имеет клон `Rc<Node>` переменной `leaf` хранящийся в `branch.children`, но все равно будет иметь weak count 0.

Когда заканчивается внутренняя область видимости, `branch` выходит из области видимости и strong count `Rc<Node>` уменьшается до 0, поэтому его `Node` удаляется. Weak count 1 из `leaf.parent` не имеет никакого отношения к тому, был ли `Node` удалён, поэтому не будет никаких утечек памяти!

Если мы попытаемся получить доступ к родителю переменной `leaf` после окончания области видимости, мы снова получим значение `None`. В конце программы `Rc<Node>` внутри `leaf` имеет strong count 1 и weak count 0 потому что переменная `leaf` снова является единственной ссылкой на `Rc<Node>`.

Вся логика, которая управляет счётчиками и сбросом их значений, встроена внутри `Rc<T>` и `Weak<T>` и их реализаций типажа `Drop`. Указав, что отношение из дочернего к родительскому элементу должно быть ссылкой типа `Weak<T>` в определении `Node`, делает возможным иметь родительские узлы, указывающие на дочерние узлы и наоборот, не создавая ссылочной зацикленности и утечек памяти.

Итоги

В этой главе рассказано как использовать умные указатели для обеспечения различных гарантий и компромиссов по сравнению с обычными ссылками, которые Rust использует по умолчанию. Тип `Box<T>` имеет известный размер и указывает на данные размещённые в куче. Тип `Rc<T>` отслеживает количество ссылок на данные в куче, поэтому данные могут иметь несколько владельцев. Тип `RefCell<T>` с его внутренней изменяемостью предоставляет тип, который можно использовать при необходимости неизменного типа, но необходимости изменить внутреннее значение этого типа; он также обеспечивает соблюдение правил заимствования во время выполнения, а не во время компиляции.

Мы обсудили также типажи `Deref` и `Drop`, которые обеспечивают большую функциональность умных указателей. Мы исследовали ссылочную зацикленность, которая может вызывать утечки памяти и как это предотвратить с помощью типа `Weak<T>`.

Если эта глава вызвала у вас интерес и вы хотите реализовать свои собственные умные указатели, обратитесь к "[The Rustonomicon](#)" за более полезной информацией.

Далее мы поговорим о параллелизме в Rust. Вы даже узнаете о нескольких новых умных указателях.

Многопоточность без страха

Безопасное и эффективное управление многопоточным программированием - ещё одна из основных целей Rust. *Многопоточное программирование*, когда разные части программы выполняются независимо, и *параллельное программирование*, когда разные части программы выполняются одновременно, становятся все более важными, поскольку всё больше компьютеров используют преимущества нескольких процессоров. Исторически программирование в этих условиях было сложным и подверженным ошибкам: Rust надеется изменить это.

Первоначально команда Rust считала, что обеспечение безопасности памяти и предотвращение проблем многопоточности - это две отдельные проблемы, которые необходимо решать различными методами. Со временем команда обнаружила, что системы владения и система типов являются мощным набором инструментов, помогающих управлять безопасностью памяти и проблемами многопоточного параллелизма! Используя владение и проверку типов, многие ошибки многопоточности являются ошибками времени компиляции в Rust, а не ошибками времени выполнения. Поэтому вместо того, чтобы тратить много времени на попытки воспроизвести точные обстоятельства, при которых возникает ошибка многопоточности во время выполнения, некорректный код будет отклонён с ошибкой. В результате вы можете исправить свой код во время работы над ним, а не после развёртывания на рабочем сервере. Мы назвали этот аспект Rust *бесстрашной многопоточностью*. Бесстрашная многопоточность позволяет вам писать код, который не содержит скрытых ошибок и легко реорганизуется без внесения новых ошибок.

Примечание. Для простоты мы будем называть многие проблемы как *многопоточные*, а не более точными терминами *многопоточные и / или параллельные*. Если бы эта книга была о многопоточности и / или параллелизме, мы были бы более конкретны. В этой главе, пожалуйста, всякий раз, когда мы используем термин *многопоточный*, мысленно замените на понятие *многопоточный и / или параллельный*.

Многие языки догматичны в отношении решений, которые они предлагают для решения сопутствующих проблем. Например, Erlang обладает элегантной функциональностью для многопоточности при передаче сообщений, но не определяет ясных способов совместного использования состояния между потоками. Поддержка только подмножества возможных решений является

разумной стратегией для языков более высокого уровня, поскольку язык более высокого уровня обещает выгоду при отказе от некоторого контроля над получением абстракций. Однако ожидается, что языки низкого уровня обеспечат решение с наилучшей производительностью в любой конкретной ситуации и будут иметь меньше абстракций по сравнению с аппаратным обеспечением. Поэтому Rust предлагает множество инструментов для моделирования проблем любым способом, который подходит для вашей ситуации и требований.

Вот темы, которые мы рассмотрим в этой главе:

- Как создать потоки для одновременного запуска нескольких фрагментов кода
- Многопоточность *передачи сообщений*, где каналы передают сообщения между потоками
- Многопоточность для *совместно используемого состояния*, когда несколько потоков имеют доступ к некоторому фрагменту данных
- Типажи **Sync** и **Send**, которые расширяют гарантии многопоточности в Rust для пользовательских типов, а также типов, предоставляемых стандартной библиотекой

Использование потоков для одновременного выполнения кода

В большинстве современных операционных систем, выполняющийся кода программы происходит в *процессе* и операционная система управляет множеством процессов одновременно. Внутри программы, также можно иметь независимые части выполняющиеся одновременно. Функционал позволяющий запускать эти независимые части называется *потоками*.

Разделение вычислений в программе на несколько потоков может повысить производительность, поскольку программа выполняет несколько задач одновременно, но это также добавляет сложности. Поскольку потоки могут работать одновременно то, нет внутренней гарантии порядка в котором будут выполняться части вашего кода в разных потоках. Это может привести к таким проблемам, как:

- Состояние гонок, когда потоки обращаются к данным или ресурсам в несогласованном порядке
- Взаимные блокировки, когда два потока ожидают друг друга для завершения использования ресурса занятного другим потоком, препятствуя продолжению работы обоих потоков
- Дефекты, возникающие только в определённых ситуациях, которые трудно воспроизвести и надёжно исправить

Rust пытается смягчить негативные последствия использования потоков, но программирование в многопоточном контексте все ещё требует тщательного обдумывания структуры кода, которая отличается от структуры кода программ, работающих в одном потоке.

Языки программирования реализуют потоки несколькими различными способами. Многие операционные системы предоставляют API для создания новых потоков. Эта модель, в которой язык вызывает API операционной системы для создания потоков, иногда называется *1:1*, что означает один поток операционной системы на один языковой поток. Стандартная библиотека Rust обеспечивает только реализацию потоков *1:1*; есть крейты, которые реализуют другие модели многопоточности, которые делают другие компромиссы.

Создание нового потока с помощью `spawn`

Чтобы создать новый поток, мы вызываем функцию `thread::spawn` и передаём ей замыкание (мы говорили о замыканиях в главе 13), содержащее код, который мы хотим запустить в новом потоке. Пример в листинге 16-1 печатает некоторый текст из нового потока и другой текст из основного потока:

Файл: src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

Листинг 16-1: Создание нового потока для печати в отдельном потоке чего-либо во время печати в главном потоке чего-то другого

Обратите внимание, что с помощью этой функции новый поток будет остановлен по окончании основного потока, независимо от того, завершился он или нет. Вывод этой программы может каждый раз немного отличаться, но он будет выглядеть примерно так:

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
```

Вызовы `thread::sleep` заставляют поток на короткое время останавливать своё выполнение, позволяя выполнятся другим потокам. Очередность выполнения потоков вероятно будет меняться, но это не гарантировано: это зависит от того, как

ваша операционная система планирует потоки. В этом цикле основной поток печатает первым, не смотря на то, что оператор печати из порождённого потока появляется раньше в коде. И даже несмотря на то, что мы написали код, что порождённый поток должен печатать до тех пор, пока значение `i` не достигнет числа 9, оно дошло только до 5, перед тем как основной поток закрылся.

Если вы запустите этот код и увидите вывод только из основного потока или не видите печати из других потоков, попробуйте увеличить числа в диапазонах, чтобы дать операционной системе больше возможностей для переключения между потоками.

Ожидание завершения работы всех потоков используя `join`

Код в листинге 16-1 не только преждевременно останавливает порождённый поток из-за окончания основного потока, но также не может гарантировать, что порождённый поток вообще запустится. Причина в том, что нет никакой гарантии относительно порядка выполнения потоков!

Мы можем исправить проблему того, что порождённый поток не запускается или не запускается полностью, сохранением возвращаемого значения `thread::spawn` в переменной. Возвращаемым из метода `thread::spawn` типом является `JoinHandle`.

`JoinHandle` - это собственное значение такое, что когда мы вызываем у него метод `join`, оно будет ожидать завершения этого потока. В листинге 16-2 показано как использовать `JoinHandle` потока, который мы создали в листинге 16-1 и вызвать у него `join`, чтобы убедиться, что порождённый поток завершает работу до выхода из `main`:

Файл: src/main.rs

```

use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}

```

Листинг 16-2: Сохранение значения `JoinHandle` из `thread::spawn` для гарантированного ожидания завершения работы потока

Вызов `join` у дескриптора блокирует текущий поток, пока поток, представленный дескриптором не завершится. *Блокировка* потока означает, что потоку запрещено выполнять работу или выходить из него. Поскольку мы поместили вызов `join` после цикла `for` основного потока, выполнение листинга 16-2 должно привести к выводу, подобному следующему:

```

hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 1 from the spawned thread!
hi number 3 from the main thread!
hi number 2 from the spawned thread!
hi number 4 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!

```

Два потока продолжают чередоваться, но основной поток находится в ожидании из-за вызова `handle.join()` и не завершается до тех пор, пока не завершится запущенный поток.

Но давайте посмотрим, что произойдёт, если мы вместо этого переместим `handle.join()` перед циклом `for` в `main`, например так:

Файл: src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

Основной поток будет ждать завершения порождённого потока, а затем запустит свой цикл `for`, поэтому выходные данные больше не будут чередоваться, как показано ниже:

```
hi number 1 from the spawned thread!
hi number 2 from the spawned thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 3 from the main thread!
hi number 4 from the main thread!
```

Небольшие детали, такие как где вызывается `join`, могут повлиять на то, выполняются ли ваши потоки одновременно.

Использование `move`-замыканий в потоках

Ключевое слово `move` часто используется с замыканиями, передаваемыми в `thread::spawn`, потому что замыкание затем получает право собственности на значения, которые оно использует из окружения, тем самым передавая владение этими значениями из одного потока в другой. В разделе “[Захват окружения с помощью замыканий](#)” Главы 13 мы обсуждали `move` в контексте замыканий. Теперь мы больше сосредоточимся на взаимодействии между `move` и `thread::spawn`.

Обратите внимание, что в листинге 16-1 замыкание, которое мы передаём в `thread::spawn` не принимает аргументов: мы не используем никаких данных из основного потока в коде порождённого потока. Чтобы использовать данные из основного потока в порождённом потоке, замыкание порождённого потока должно захватывать значения, которые ему необходимы. Листинг 16-3 показывает попытку создать вектор в главном потоке и использовать его в порождённом потоке. Тем не менее, это не будет работать, как вы увидите через мгновение.

Файл: src/main.rs

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```



Листинг 16-3: Попытка использования вектора в дочернем потоке, созданного в основном потоке

Замыкание использует переменную `v`, поэтому оно захватит `v` и сделает его частью окружения замыкания. Поскольку `thread::spawn` запускает это замыкание в новом потоке, мы должны иметь доступ к `v` внутри этого нового потока. Но при компиляции этого примера, мы получаем следующую ошибку:

```
$ cargo run
Compiling threads v0.1.0 (file:///projects/threads)
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!("Here's a vector: {:?}", v);
|                         - `v` is borrowed here
|
note: function requires argument type to outlive `'static`
--> src/main.rs:6:18
|
6 |     let handle = thread::spawn(|| {
|             ^
7 |     |     println!("Here's a vector: {:?}", v);
8 |     | );
|     | ^
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 || {
|             +---
```

For more information about this error, try `rustc --explain E0373`.
error: could not compile `threads` due to previous error

Rust выводит как захватить `v` и так как в `println!` нужна только ссылка на `v`, то замыкание пытается заимствовать `v`. Однако есть проблема: Rust не может определить, как долго будет работать порождённый поток, поэтому он не знает, будет ли всегда действительной ссылка на `v`.

В листинге 16-4 приведён сценарий, который с большей вероятностью будет иметь ссылку на `v`, что будет недопустимо:

Файл: src/main.rs



```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    drop(v); // oh no!

    handle.join().unwrap();
}
```

Листинг 16-4: Поток с замыканием, которое пытается захватить ссылку на `v` из главного потока, который удаляет `v`

Если бы нам разрешили запустить этот код, была бы вероятность что порождённый поток был бы немедленно помещён в фоновый режим, вообще без его запуска. Внутри порождённого потока есть ссылка на `v`, но основной поток немедленно удаляет `v` используя функцию `drop`, о которой мы говорили в главе 15. Затем когда порождённый поток начинает выполняться то `v` больше не действительна и поэтому ссылка на него также недействительным. О нет!

Чтобы исправить ошибку компилятора в листинге 16-3, мы можем использовать совет из сообщения об ошибке:

```
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 заставляем замыкание забирать используемые значения во владение, вместо того, чтобы позволить Rust вывести необходимость заимствования значения. Модификация Листинга 16-3, показанная в Листинге 16-5, будет скомпилирована и запущена так, как мы ожидаем:

Файл: src/main.rs

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

Листинг 16-5: Использование ключевого слова `move` для принуждения замыкания забрать во владение используемых им значений

Что произойдёт с кодом в листинге 16-4, где основной поток вызывает `drop` при использовании в замыкании `move`? Исправит ли `move` этот случай? К сожалению нет; мы получили бы другую ошибку, потому что то, что пытается сделать листинг 16-4 не разрешено по другой причине. Если бы мы добавили `move` в замыкание, мы бы переместили `v` в среду замыкания и больше не могли вызывать `drop` для него в главном потоке. Вместо этого мы получили бы ошибку компилятора:

```
$ cargo run
Compiling threads v0.1.0 (file:///projects/threads)
error[E0382]: use of moved value: `v`
--> src/main.rs:10:10
|
4 |     let v = vec![1, 2, 3];
|         - move occurs because `v` has type `Vec<i32>`, which does not
implement the `Copy` trait
5 |
6 |     let handle = thread::spawn(move || {
|                         ----- value moved into closure here
7 |         println!("Here's a vector: {:?}", v);
|                         - variable moved due to use in
closure
...
10 |     drop(v); // oh no!
|           ^ value used here after move

For more information about this error, try `rustc --explain E0382`.
error: could not compile `threads` due to previous error
```

Правила владения Rust снова нас спасли! Мы получили ошибку кода из листинга 16-3, потому что Rust был консервативен и только заимствовал `v` для потока, что означало, что основной поток теоретически может сделать недействительной

ссылку на порождённый поток. Сообщив Rust о передаче владения `v` в порождаемый поток, мы гарантируем Rust, что основной поток больше не будет использовать `v`. Если мы изменим Листинг 16-4 таким же образом, то мы нарушаем правила владения при попытке использовать `v` в главном потоке. Ключевое слово `move` отменяет основное консервативное поведение Rust по заимствованию, что не позволяет нам нарушать правила владения.

Имея базовое понимание потоков и API потоков, давайте посмотрим, что мы можем делать с помощью потоков.

Передача данных с помощью сообщений между потоками

Одним из все более популярных подходов к обеспечению безопасной многопоточности является *передача сообщений*, когда потоки или участники взаимодействуют друг с другом, отправляя друг другу сообщения, содержащие данные. Вот идея в слогане из [документации языка Go](#): «Не общайтесь разделяя память; делитесь памятью обмениваясь сообщениями».

Одним из основных инструментов Rust для обеспечения многопоточной отправки сообщений является *канал* (channel), концепция программирования, для которой стандартная библиотека Rust предоставляет реализацию. Вы можете представить канал в контексте программирования как канал воды, как ручей или река. Если вы поместите что-то вроде резиновой утки или лодки в ручей, она пойдёт вниз по течению до конца водного пути.

Канал в программировании имеет две части: передатчик и приёмник. Передаваемая часть - это место вверх по течению, где вы помещаете резиновых уток в реку, а приёмная часть - там, куда резиновая утка приплывает вниз по течению. Одна часть вашего кода вызывает методы у передатчика с данными, которые вы хотите отправить, а другая часть проверяет на принимающей стороне поступающие сообщения. Говорят, что канал *закрыт*, если передаваемая или приёмная сторона канала закрылась.

Здесь мы будем работать с программой, в которой есть один поток для генерации значений и отправки их в канал и другой поток, который получит значения и распечатает их. Используя канал, мы будем отправлять простые значения между потоками, чтобы проиллюстрировать этот функционал. Как только вы освоитесь с этой техникой, вы можете использовать каналы для реализации чат системы или системы, в которой множество потоков выполняют части вычисления и отправляют вычисленные части в один поток, который объединяет результаты.

Сначала в листинге 16-6 мы создадим канал, но не будем ничего с ним делать. Обратите внимание, что этот код ещё не компилируется, потому что Rust не может сказать, какой тип значений мы хотим отправить через канал.

Файл: src/main.rs

```
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();
}
```



Листинг 16-6: Создание канала и назначение двух половин для `tx` (передачи) и `rx` (приёма)

Мы создаём новый канал, используя `mpsc::channel` функцию; `mpsc` означает *несколько производителей, один потребитель* (multiple producer, single consumer). Коротко, способ которым стандартная библиотека Rust реализует каналы, означает что канал может иметь несколько *отправляющих* источников генерирующих значения, но только одну *принимающую* сторону, которая потребляет эти значения. Представьте, что несколько потоков втекают в одну большую реку: все, что идёт вниз по любому из потоков, в конце концов окажется в одной реке. Сейчас мы начнём с одного производителя, но добавим несколько производителей, когда этот пример будет работать.

Функция `mpsc::channel` возвращает кортеж, первый элемент которого является отправляющей стороной, а второй элемент - принимающей стороной. Сокращения `tx` и `rx` традиционно используются во многих областях для обозначения *передатчика* и *приёмника* соответственно, поэтому мы называем наши переменные таким образом для обозначения каждой стороны. Мы используем оператор `let` с шаблоном, который разрушает кортеж; мы обсудим использование шаблонов в операторах `let` и деструктуризацию в главе 18. Использование оператора `let` таким образом является удобным подходом для извлечения фрагментов кортежа, возвращаемого функцией `mpsc::channel`.

Давайте переместим передающую часть в порождённый поток так, чтобы он отправлял одну строку и чтобы таким образом, порождённый поток связывался с основным потоком, как показано в листинге 16-7. Это похоже на то, как если бы вы поместили резиновую утку в реку вверх по течению или отправили сообщение чата из одного потока в другой.

Файл: src/main.rs

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });
}
```

Листинг 16-7: Перемещение `tx` в порождённый поток и отправка "hi"

Опять же, мы используем вызов `thread::spawn` для создания нового потока, а затем с помощью ключевого слова `move` перемещаем `tx` в замыкание, чтобы порождённый поток завладел `tx`. Созданный поток должен владеть передающей частью канала, чтобы иметь возможность отправлять сообщения через канал.

Передающая часть имеет метод `send`, который принимает отправляемое значение. Метод `send` возвращает тип `Result<T, E>`, поэтому если принимающая часть закрылась и некуда отправить значение, то операция `send` выдаст ошибку. В этом примере мы вызываем функцию `unwrap` для паники в случае ошибки. Но в реальном приложении мы бы обработали её правильно: вернитесь в главу 9, чтобы просмотреть стратегии правильной обработки ошибок.

В листинге 16-8 мы получим значение на принимающей части канала в основном потоке. Это похоже на извлечение резиновой утки из воды в конце реки или на получение сообщения в чате.

Файл: src/main.rs

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

Листинг 16-8: Получение значения "hi" в главном потоке и его печать

Принимающая часть канала имеет два полезных метода: `recv` и `try_recv`. Мы используем `recv`, сокращение от `receive`, которое блокирует выполнение основного потока и ждёт, пока значение не будет передано по каналу. Как только значение отправлено, `recv` вернёт его внутри `Result<T, E>`. Когда передающая часть канала закрывается, вызов `recv` вернёт ошибку, сообщив что больше значений не будет.

Метод `try_recv` не блокирующий, отличается тем, что немедленно вернёт тип `Result<T, E>`: где значение `Ok` содержащее сообщение, если оно доступно и значение `Err`, если в этот раз нет сообщений. Использование `try_recv` полезно, если текущий поток выполняет другую работу во время ожидания сообщений: мы могли бы написать цикл довольно часто вызывающий метод `try_recv`, так чтобы обрабатывать сообщение, если оно доступно и в противном случае выполнять другую работу некоторое время до следующей проверки.

Мы использовали `recv` в этом примере для простоты; у нас нет никакой другой работы для основного потока, кроме как ждать сообщений, поэтому блокировка основного потока уместна.

При запуске кода листинга 16-8, мы увидим значение, напечатанное из основного потока:

```
Got: hi
```

Отлично!

Каналы и передача владения

Правила владения играют жизненно важную роль в отправке сообщений, потому что они помогают писать безопасный многопоточный код. Предотвращение ошибок в многопоточном программировании является преимуществом для размышлений о владении во всех ваших Rust программах. Давайте проведём эксперимент, чтобы показать как каналы и владение действуют совместно для предотвращения проблем: мы попытаемся использовать значение `val` в порождённом потоке *после* того как отправим его в канал. Попробуйте скомпилировать код в листинге 16-9, чтобы понять, почему этот код не разрешён:

Файл: src/main.rs

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
        println!("val is {}", val);
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```



Листинг 16-9: Попытка использовать `val` после того, как мы отправили его в канал

Здесь мы пытаемся напечатать значение `val` после того, как отправили его в канал вызвав `tx.send`. Разрешить это было бы плохой идеей: после того, как значение было отправлено в другой поток, текущий поток мог бы изменить или удалить значение, прежде чем мы попытались бы использовать значение снова. Потенциально изменения в другом потоке могут привести к ошибкам или не ожидаемым результатам из-за противоречивых или несуществующих данных. Однако Rust выдаёт нам ошибку, если мы пытаемся скомпилировать код в листинге 16-9:

```
$ cargo run
Compiling message-passing v0.1.0 (file:///projects/message-passing)
error[E0382]: borrow of moved value: `val`
--> src/main.rs:10:31
|
8 |     let val = String::from("hi");
|         --- move occurs because `val` has type `String`, which does
not implement the `Copy` trait
9 |     tx.send(val).unwrap();
|             --- value moved here
10|    println!("val is {}", val);
|                      ^^^ value borrowed here after move
|
= note: this error originates in the macro `$crate::format_args_nl` (in
Nightly builds, run with -Z macro-backtrace for more info)

For more information about this error, try `rustc --explain E0382`.
error: could not compile `message-passing` due to previous error
```

Наша ошибка для многопоточности привела к ошибке компиляции. Функция `send` вступает во владение своим параметром и когда значение перемещается, получатель становится владельцем этого параметра. Это останавливает нас от случайного использования значения снова после его отправки; анализатор заимствования проверяет, что все в порядке.

Отправка нескольких значений и ожидание получателем

Код в листинге 16-8 компилируется и выполняется, но в нем не ясно показано то, что два отдельных потока общаются друг с другом через канал. В листинге 16-10 мы внесли некоторые изменения, которые докажут, что код в листинге 16-8 работает одновременно: порождённый поток теперь будет отправлять несколько сообщений и делать паузу на секунду между каждым сообщением.

Файл: src/main.rs

```

use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];
        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
}

```

Листинг 16-10: Отправка нескольких сообщений и пауза между ними

На этот раз порождённый поток имеет вектор строк, которые мы хотим отправить основному потоку. Мы перебираем их, отправляя каждую строку по отдельности и делаем паузу между ними, вызывая функцию `thread::sleep` со значением `Duration` равным 1 секунде.

В основном потоке мы больше не вызываем функцию `recv` явно: вместо этого мы используем `rx` как итератор. Для каждого полученного значения мы печатаем его. Когда канал будет закрыт, итерация закончится.

При выполнении кода в листинге 16-10 вы должны увидеть следующий вывод с паузой в 1 секунду между каждой строкой:

```

Got: hi
Got: from
Got: the
Got: thread

```

Поскольку у нас нет кода, который приостанавливает или задерживает цикл `for` в

основном потоке, мы можем сказать, что основной поток ожидает получения значений из порождённого потока.

Создание нескольких отправителей путём клонирования передатчика

Ранее мы упоминали, что `mpsc` аббревиатура *несколько производителей, один потребитель*. Давайте подключим `mpsc` и расширим код в листинге 16-10 так, чтобы создать несколько потоков, которые все отправляют значения одному и тому же получателю. Мы можем сделать это путём клонирования передающей части канала, как показано в листинге 16-11:

Файл: src/main.rs

```
// --snip--  
  
let (tx, rx) = mpsc::channel();  
  
let tx1 = tx.clone();  
thread::spawn(move || {  
    let vals = vec![  
        String::from("hi"),  
        String::from("from"),  
        String::from("the"),  
        String::from("thread"),  
    ];  
  
    for val in vals {  
        tx1.send(val).unwrap();  
        thread::sleep(Duration::from_secs(1));  
    }  
});  
  
thread::spawn(move || {  
    let vals = vec![  
        String::from("more"),  
        String::from("messages"),  
        String::from("for"),  
        String::from("you"),  
    ];  
  
    for val in vals {  
        tx.send(val).unwrap();  
        thread::sleep(Duration::from_secs(1));  
    }  
});  
  
for received in rx {  
    println!("Got: {}", received);  
}  
  
// --snip--
```

Листинг 16-11. Отправка нескольких сообщений от нескольких отправителей

На этот раз, прежде чем мы создадим первый порождённый поток, мы вызываем **clone** на передающей части канала. Это даст нам новый дескриптор отправки, который мы можем передать первому порождённому потоку. Мы передаём исходную отправляющую часть канала второму порождённому потоку. Это даёт нам два потока, каждый из которых отправляет разные сообщения принимающей части канала.

Когда вы запустите код, вывод должен выглядеть примерно так:

```
Got: hi
Got: more
Got: from
Got: messages
Got: for
Got: the
Got: thread
Got: you
```

Вы можете увидеть эти значения в другом порядке; все зависит от вашей системы. Это то, что делает многопоточный параллелизм интересным и трудным. Если вы экспериментируете с `thread::sleep` задавая различные значения в разных потоках, то каждый прогон будет более не определённым и каждый раз будет создавать разные выходные данные.

Теперь, когда мы посмотрели, как работают каналы, давайте рассмотрим другой метод многопоточности.

Многопоточное разделяемое состояние

Передача сообщений - это прекрасный способ многопоточной работы, но он не является единственным. Вспомните часть лозунга из документации на языке Go: «не общайтесь, разделяя память».

Как бы выглядело общение, используя разделяемую память? Кроме того, почему энтузиасты передачи сообщений не используют его и делают наоборот?

В каком-то смысле каналы в любом языке программирования похожи на единоличное владение, потому что после передачи значения по каналу вам больше не следует использовать отправленное значение. Многопоточная, совместно используемая память подобна множественному владению: несколько потоков могут одновременно обращаться к одной и той же области памяти. Как вы видели в главе 15, где умные указатели сделали возможным множественное владение, множественное владение может добавить сложность, потому что нужно управлять этими разными владельцами. Система типов Rust и правила владения очень помогают в их правильном управлении. Для примера давайте рассмотрим мьютексы, один из наиболее распространённых многопоточных примитивов для разделяемой памяти.

Мьютексы предоставляют доступ к данным из одного потока (за раз)

Mutex - это сокращение от *взаимное исключение* (mutual exclusion), так как мьютекс позволяет только одному потоку получать доступ к некоторым данным в любой момент времени. Для того, чтобы получить доступ к данным в мьютексе, поток должен сначала подать сигнал, что он хочет получить доступ запрашивая блокировку (lock) мьютекса. Блокировка - это структура данных, являющаяся частью мьютекса, которая отслеживает кто в настоящее время имеет эксклюзивный доступ к данным. Поэтому мьютекс описывается как объект защищающий данные, которые он хранит через систему блокировки.

Мьютексы имеют репутацию трудных в использовании, потому что вы должны помнить два правила:

- Перед тем как попытаться получить доступ к данным необходимо получить блокировку.
- Когда вы закончили работу с данными, которые защищает мьютекс, вы

должны разблокировать данные, чтобы другие потоки могли получить блокировку.

Для понимания мьютекса, представьте пример из жизни как групповое обсуждение на конференции с одним микрофоном. Прежде чем участник дискуссии сможет говорить, он должен спросить или дать сигнал, что он хочет использовать микрофон. Когда он получает микрофон, то может говорить столько, сколько хочет, а затем передаёт микрофон следующему участнику, который попросит дать ему выступить. Если участник дискуссии забудет освободить микрофон, когда закончит с ним, то никто больше не сможет говорить. Если управление общим микрофоном идёт не правильно, то конференция не будет работать как было запланировано!

Правильное управление мьютексами может быть невероятно сложным и именно поэтому многие люди с энтузиазмом относятся к каналам. Однако, благодаря системе типов и правилам владения в Rust, вы не можете использовать блокировку и разблокировку неправильным образом.

Mutex<T> API

Давайте рассмотрим пример использования мьютекса в листинге 16-12 без использования нескольких потоков:

Файл: src/main.rs

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

    println!("m = {:?}", m);
}
```

Листинг 16-12: Изучение API `Mutex<T>` для простоты в однопоточном контексте.

Как и во многих типах, мы создаём `Mutex<T>` используя ассоциированную функцию `new`. Чтобы получить доступ к данным внутри мьютекса, мы используем метод `lock` для получения блокировки. Этот вызов блокирует текущий поток, поэтому он

не может выполнять какую-либо другую работу, пока не наступит наша очередь получить блокировку.

Вызов `lock` завершится неудачей, если запаникует другой поток, удерживающий блокировку. В этом случае никто никогда не сможет получить блокировку, поэтому мы решили вызвать `unwrap` и вызвать панику, если окажемся в такой ситуации.

После того как мы получили блокировку, мы можем рассматривать возвращаемое значение, в данном случае с именем `num`, как изменяемую ссылку на данные внутри. Система типов гарантирует, что мы получим блокировку перед использованием значения из `m: Mutex<i32>` не является типом `i32`, поэтому мы должны получить блокировку, чтобы иметь возможность использовать значение `i32`. Мы не можем забыть этого сделать; так как система типов не позволит нам получить доступ ко внутреннему `i32` значению.

Как вы наверное подозреваете, `Mutex<T>` является умным указателем. Точнее, вызов `lock` возвращает умный указатель называемый `MutexGuard`, обёрнутый в `LockResult`, который мы обработали с помощью вызова `unwrap`. Умный указатель типа `MutexGuard` реализует типаж `Deref` для указания на внутренние данные; умный указатель также имеет реализацию типажа `Drop` автоматически снимающего блокировку, когда `MutexGuard` выходит из области видимости, что происходит в конце внутренней области видимости в листинге 16-12. В результате мы не рискуем забыть снять блокировку и заблокировать мьютекс от использования другими потоками, поскольку снятие блокировки происходит автоматически.

После снятия блокировки можно напечатать значение мьютекса и увидеть, что мы смогли изменить внутреннее `i32` на 6.

Разделение `Mutex<T>` между множеством потоков

Теперь давайте попробуем с помощью `Mutex<T>` совместно использовать значение между несколькими потоками. Мы стартуем 10 потоков и каждый из них увеличивает значение счётчика на 1, поэтому счётчик изменяется от 0 до 10. Обратите внимание, что в следующих нескольких примерах будут ошибки компилятора и мы будем использовать эти ошибки, чтобы узнать больше об использовании типа `Mutex<T>` и как Rust помогает нам правильно его использовать. Листинг 16-13 содержит наш начальный пример:

Файл: `src/main.rs`

```
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    for _ in 0..10 {
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```



Листинг 16-13. Десять потоков каждый из которых увеличивает счётчик, защищённый `Mutex<T>`

`data-md-type="raw_html">`

Мы создаём переменную счётчик `counter` для хранения `i32` внутри `Mutex<T>`, как мы это делали в листинге 16-12. Далее мы создаём 10 потоков, перебирая диапазон чисел. Мы используем `thread::spawn` и передаём всем этим потокам одинаковое замыкание, которое перемещает счётчик в поток, запрашивает блокировку на `Mutex<T>`, вызывая метод `lock`, а затем добавляет 1 к значению в мьютексе. Когда поток завершит выполнение своего замыкания, `num` выйдет из области видимости и освободит блокировку, чтобы другой поток мог её получить.

В основном потоке мы собираем все дескрипторы в переменную `handles`. Затем, как мы это делали в листинге 16-2, вызываем `join` для каждого дескриптора, чтобы убедиться в завершении всех потоков. В этот момент основной поток получит доступ к блокировке и тоже напечатает результат программы.

Компилятор намекнул, что этот пример не компилируется. Теперь давайте выясним почему!

```
$ cargo run
Compiling shared-state v0.1.0 (file:///projects/shared-state)
error[E0382]: use of moved value: `counter`
--> src/main.rs:9:36
|
5 |     let counter = Mutex::new(0);
|         ----- move occurs because `counter` has type `Mutex<i32>`, which does not implement the `Copy` trait
...
9 |         let handle = thread::spawn(move || {
|             ^^^^^^^^ value moved into closure here, in previous iteration of loop
10|             let mut num = counter.lock().unwrap();
|                           ----- use occurs due to use in closure

For more information about this error, try `rustc --explain E0382`.
error: could not compile `shared-state` due to previous error
```

Ага! Первое сообщение об ошибке указывает, что `counter` перемещён в замыкание для потока, связанного с `handle`. Этот шаг не позволяет нам захватить `counter`, когда мы пытаемся вызвать `lock` и сохранить результат в `num2` во втором потоке! Так что Rust говорит нам, что мы не можем передать во владение `counter` в несколько потоков. Это было трудно увидеть ранее, потому что наши потоки были в цикле и Rust не может указывать на разные потоки в разных итерациях цикла. Давайте исправим ошибку компилятора с помощью метода множественного владения, который мы обсуждали в главе 15.

Множественное владение между множеством потоков

В главе 15 мы давали значение нескольким владельцам, используя умный указатель `Rc<T>` для создания значения подсчитанных ссылок. Давайте сделаем то же самое здесь и посмотрим, что произойдёт. Мы завернём `Mutex<T>` в `Rc<T>` в листинге 16-14 и клонируем `Rc<T>` перед передачей владения в поток. Теперь, когда мы увидели ошибки, мы также вернёмся к использованию цикла `for` и сохраним ключевое слово `move` у замыкания.

Файл: 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!("Result: {}", *counter.lock().unwrap());
}
```

Листинг 16-14: Попытка использования `Rc<T data-md-type="raw_html">`, чтобы позволить нескольким потокам владение `Mutex<T data-md-type="raw_html">`

Ещё раз, мы компилируем и получаем ... другие ошибки! Компилятор учит нас.

```
$ cargo run
Compiling shared-state v0.1.0 (file:///projects/shared-state)
error[E0277]: `Rc<Mutex<i32>>` cannot be sent between threads safely
--> src/main.rs:11:22
11 |         let handle = thread::spawn(move || {
|         |         ^^^^^^^^^^
|         |
|         |             `Rc<Mutex<i32>>` cannot be sent between threads
safely
12 |         let mut num = counter.lock().unwrap();
13 |
14 |         *num += 1;
15 |     });
|         - within this `[closure@src/main.rs:11:36: 15:10]`
|
= help: within `[closure@src/main.rs:11:36: 15:10]`, the trait `Send` is
not implemented for `Rc<Mutex<i32>>`
= note: required because it appears within the type
`[closure@src/main.rs:11:36: 15:10]`
note: required by a bound in `spawn`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `shared-state` due to previous error
```

Вау, сообщение об ошибке очень многословное! Вот некоторые важные части, на которых нужно сосредоточить внимание: первая встроенная ошибка говорит о том, что `std::rc::Rc<std::sync::Mutex>` **cannot be sent between threads safely**. Причиной этого является следующая важная часть сообщения об ошибке. Сообщение об ошибке говорит, **the trait bound Send is not satisfied**. Мы поговорим про типаж **Send** в следующем разделе: это один из типажей гарантирующих что типы, используемые потоками, предназначены для использования в многопоточных ситуациях.

К сожалению, **Rc<T>** небезопасен для совместного использования между потоками. Когда **Rc<T>** управляет счётчиком ссылок, он добавляется значение к счётчику для каждого вызова **clone** и вычитается значение из счётчика, когда каждое клонированное значение удаляется при выходе из области видимости. Но он не использует примитивы многопоточности, чтобы гарантировать, что изменения в подсчёте не могут быть прерваны другим потоком. Это может привести к неправильным подсчётам - незначительным ошибкам, которые в свою очередь, могут привести к утечкам памяти или удалению значения до того, как мы отработали с ним. Нам нужен тип точно такой же как **Rc<T>**, но который позволяет изменять счётчик ссылок безопасно из разных потоков.

Атомарный счётчик ссылок `Arc<T>`

К счастью, `Arc<T>` является типом аналогичным типу `Rc<T>`, который безопасен для использования в ситуациях многопоточности. Буква *A* означает *атомарное*, что означает тип *ссылка подсчитываемая атомарно*. Atomics - это дополнительный вид примитивов для многопоточности, который мы не будем здесь подробно описывать: дополнительную информацию смотрите в документации стандартной библиотеки для `std::sync::atomic`. На данный момент вам просто нужно знать, что atomics работают как примитивные типы, но безопасны для совместного использования между потоками.

Вы можете спросить, почему все примитивные типы не являются атомарными и почему стандартные типы библиотек не реализованы для использования вместе с типом `Arc<T>` по умолчанию. Причина в том, что безопасность потоков сопровождается снижением производительности, которое вы хотите платить только тогда, когда вам это действительно нужно. Если вы просто выполняете операции со значениями в одном потоке, то ваш код может работать быстрее, если он не должен обеспечивать гарантии предоставляемые atomics.

Давайте вернёмся к нашему примеру: типы `Arc<T>` и `Rc<T>` имеют одинаковый API, поэтому мы исправляем нашу программу, заменяя тип в строках `use`, вызове `new` и вызове `clone`. Код в листинге 16-15, наконец скомпилируется и запустится:

Файл: src/main.rs

```

use std::sync::{Arc, Mutex};
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!("Result: {}", *counter.lock().unwrap());
}

```

Листинг 16-15: Использование типа `Arc<T data-md-type="raw_html">` для обёртывания `Mutex<T data-md-type="raw_html">` для возможности совместного владения несколькими потоками

Код напечатает следующее:

`Result: 10`

Мы сделали это! Мы посчитали от 0 до 10, что может показаться не очень впечатляющим, но это позволило больше узнать про `Mutex<T>` и безопасность потоков. Вы также можете использовать структуру этой программы для выполнения более сложных операций, чем просто увеличение счётчика. Используя эту стратегию, вы можете разделить вычисления на независимые части, разделить эти части на потоки, а затем использовать `Mutex<T>`, чтобы каждый поток обновлял конечный результат своей частью кода.

Сходства `RefCell<T> / Rc<T>` и `Mutex<T> / Arc<T>`

Вы могли заметить, что `counter` теперь не изменяемый, но мы могли бы получить изменяемую ссылку на значение внутри него; это означает, что `Mutex<T>`

обеспечивает внутреннюю изменяемость как и семейство `Cell` типов. Таким же образом мы использовали `RefCell<T>` в главе 15, чтобы позволить нам изменять содержимое внутри `Rc<T>`, мы используем `Mutex<T>` для изменения содержимого внутри `Arc<T>`.

Ещё одна деталь, на которую стоит обратить внимание: Rust не может защитить вас от всевозможных логических ошибок при использовании `Mutex<T>`. Вспомните в главе 15, что использование `Rc<T>` сопряжено с риском создания ссылочной зацикленности, где два значения `Rc<T>` ссылаются друг на друга, что приводит к утечкам памяти. Аналогичным образом, `Mutex<T>` сопряжён с риском создания взаимных блокировок (deadlocks). Это происходит, когда операции необходимо заблокировать два ресурса и каждый из двух потоков получил одну из блокировок, заставляя оба потока ждать друг друга вечно. Если вам интересна тема взаимных блокировок, попробуйте создать программу Rust, которая её содержит; затем исследуйте стратегии устранения взаимных блокировок для мьютексов на любом языке и попробуйте реализовать их в Rust. Документация стандартной библиотеки для `Mutex<T>` и `MutexGuard` предлагает полезную информацию.

Мы завершим эту главу, рассказав о типажах `Send` и `Sync` и о том, как мы можем использовать их с пользовательскими типами.

Расширенная многопоточность с помощью типажей `Sync` и `Send`

Интересно, что сам язык Rust имеет очень мало возможностей для многопоточности. Почти все функции многопоточности о которых мы говорили в этой главе, были частью стандартной библиотеки, а не языка. Ваши варианты работы с многопоточностью не ограничиваются языком или стандартной библиотекой; Вы можете написать свой собственный многопоточный функционал или использовать возможности написанные другими.

Тем не менее, в язык встроены две концепции многопоточности: `std::marker` типажи `Sync` и `Send`.

Разрешение передачи во владение между потоками с помощью `Send`

Маркерный типаж `Send` указывает, что владение типом реализующим `Send`, может передаваться между потоками. Почти каждый тип Rust является типом `Send`, но есть некоторые исключения, вроде `Rc<T>`: он не может быть `Send`, потому что если вы клонировали значение `Rc<T>` и попытались передать владение клоном в другой поток, оба потока могут обновить счётчик ссылок одновременно. По этой причине `Rc<T>` реализован для использования в однопоточных ситуациях, когда вы не хотите платить за снижение производительности.

Следовательно, система типов Rust и ограничений типажа гарантируют, что вы никогда не сможете случайно небезопасно отправлять значение `Rc<T>` между потоками. Когда мы попытались сделать это в листинге 16-14, мы получили ошибку, `the trait Send is not implemented for Rc<Mutex<i32>>`. Когда мы переключились на `Arc<T>`, который является типом `Send`, то код скомпилировался.

Любой тип полностью состоящий из типов `Send` автоматически помечается как `Send`. Почти все примитивные типы являются `Send`, кроме сырых указателей, которые мы обсудим в главе 19.

Разрешение доступа из нескольких потоков с `Sync`

Маркерный типаж `Sync` указывает, что на тип реализующий `Sync` можно

безопасно ссылаться из нескольких потоков. Другими словами, любой тип `T` является типом `Sync`, если `&T` (ссылка на `T`) является типом `Send`, что означает что ссылку можно безопасно отправить в другой поток. Подобно `Send`, примитивные типы являются типом `Sync`, а типы полностью скомбинированные из типов `Sync`, также являются `Sync` типом.

Умный указатель `Rc<T>` не является `Sync` типом по тем же причинам, по которым он не является `Send`. Тип `RefCell<T>` (о котором мы говорили в главе 15) и семейство связанных типов `Cell<T>` не являются `Sync`. Реализация проверки заимствования, которую делает тип `RefCell<T>` во время выполнения программы не является поточно-безопасной. Умный указатель `Mutex<T>` является типом `Sync` и может использоваться для совместного доступа из нескольких потоков, как вы уже видели в разделе «[Совместное использование `Mutex<T>` между несколькими потоками](#)».

Реализация `Send` и `Sync` вручную небезопасна

Поскольку типы созданные из типажей `Send` и `Sync` автоматически также являются типами `Send` и `Sync`, мы не должны реализовывать эти типажи вручную. Являясь маркерными типажами у них нет никаких методов для реализации. Они просто полезны для реализации инвариантов, связанных с многопоточностью.

Ручная реализация этих типажей включает в себя реализацию небезопасного кода Rust. Мы поговорим об использовании небезопасного кода Rust в главе 19; на данный момент важная информация заключается в том, что для создания новых многопоточных типов, не состоящих из частей `Send` и `Sync` необходимо тщательно продумать гарантии безопасности. В [Rustonomicon](#) есть больше информации об этих гарантиях и о том как их соблюдать.

Итоги

Это не последний случай, когда вы увидите многопоточность в этой книге: проект в главе 20 будет использовать концепции этой главы для более реалистичного случая, чем небольшие примеры обсуждаемые здесь.

Как упоминалось ранее, поскольку в языке Rust очень мало того, с помощью чего можно управлять многопоточностью, многие решения реализованы в виде

крейтов. Они развиваются быстрее, чем стандартная библиотека, поэтому обязательно поищите в Интернете текущие современные крейты.

Стандартная библиотека Rust предоставляет каналы для передачи сообщений и типы умных указателей, такие как `Mutex<T>` и `Arc<T>`, которые можно безопасно использовать в многопоточных контекстах. Система типов и анализатор заимствований гарантируют, что код использующий эти решения не будет содержать гонки данных или недействительные ссылки. Получив компилирующийся код, вы можете быть уверены, что он будет успешно работать в нескольких потоках без ошибок, которые трудно обнаружить в других языках. Многопоточное программирование больше не является концепцией, которую стоит опасаться: иди вперёд и сделай свои программы многопоточными безбоязненно!

Далее мы поговорим об идиоматичных способах моделирования проблем и структурирования решений по мере усложнения ваших программ на Rust. Кроме того, мы обсудим как идиомы Rust связаны с теми, с которыми вы, возможно, знакомы по объектно-ориентированному программированию.

Особенности объектно-ориентированного программирования в Rust

Объектно-ориентированное программирование (ООП) является способом моделирования программ. Объекты пришли из языка Simula в 1960-х годах. Эти объекты повлияли на архитектуру программирования Алана Кея, в которой объекты передают сообщения друг другу. Он ввёл термин *объектно-ориентированное программирование* в 1967 году для описания этой архитектуры. Многие конкурирующие определения описывают, что такое ООП; некоторые определения будут классифицировать Rust как объектно-ориентированный, но другие определения не будут. В этой главе мы рассмотрим некоторые характеристики, которые обычно считаются объектно-ориентированными и как эти характеристики переводятся в идиоматический Rust. Затем мы покажем вам, как реализовать подход объектно-ориентированного проектирования в Rust и обсудим его компромиссы по сравнению с реализацией решения, использующего некоторые сильные стороны Rust.

Характеристики объектно-ориентированных языков

В сообществе разработчиков отсутствует согласие относительно того, какие особенности языка делают его объектно-ориентированным. На Rust повлияли многие парадигмы программирования, включая ООП; например, в главе 13 мы изучили вещи, присущие функциональному программированию. Возможно, ООП языки имеют некоторые общие характеристики, а именно объекты, инкапсуляцию и наследование. Давайте посмотрим, что означает каждая из этих характеристик и поддерживает ли её Rust.

Объекты содержат данные и поведение

Книга «Приёмы объектно-ориентированного проектирования. Шаблоны проектирования» (1994), называемая также «книгой банды четырёх», является каталогом объектно-ориентированных шаблонов проектирования. Объектно-ориентированные программы определяются в ней следующим образом:

Объектно-ориентированные программы состоят из объектов. *Объект* объединяет данные и процедуры, которые работают с этими данными. Эти процедуры обычно называются *методами* или *операциями*.

В соответствии с этим определением, Rust является объектно-ориентированным языком: в структурах и перечислениях содержатся данные, а в блоках `impl` определяются методы. Хотя структуры и перечисления, имеющие методы, не называются объектами, они обеспечивают такую же функциональность, которую предоставляют объекты, соответствующие определению в книге банды четырёх.

Инкапсуляция скрывающая детали реализации

Другим аспектом, обычно связанным с объектно-ориентированным программированием, является идея *инкапсуляции*: детали реализации объекта недоступны для кода, использующего этот объект. Единственный способ взаимодействия с объектом — через его публичный интерфейс; код, использующий этот объект, не должен иметь возможности взаимодействовать с внутренними

свойствами объекта и напрямую изменять его данные или поведение. Инкапсуляция позволяет изменять и реорганизовывать внутренние свойства объекта без необходимости изменять код, который использует объект.

Как мы обсудили в главе 7, мы можем использовать ключевое слово `pub` чтобы решить, какие модули, типы, функции и методы в нашем коде должны быть публичными; по умолчанию все остальное является приватным. Например, мы можем определить структуру `AveragedCollection`, которая имеет поле, содержащее вектор значений типа `i32`. Структура также может иметь поле содержащее среднее значение в векторе, так что всякий раз, когда кто-либо захочет получить среднее значение элементов вектора, нам не нужно вычислять его заново. Другими словами, `AveragedCollection` будет кэшировать рассчитанное среднее значение для нас. В примере 17-1 приведено определение структуры `AveragedCollection`:

Файл: src/lib.rs

```
pub struct AveragedCollection {
    list: Vec<i32>,
    average: f64,
}
```

Листинг 17-1: Структура `AveragedCollection` содержит список целых чисел и среднее значение элементов в коллекции.

Обратите внимание, что структура помечена ключевым словом `pub`, что позволяет другому коду её использовать, однако, поля внутри структуры остаются недоступными. Это важно, потому что мы хотим гарантировать обновление среднего значения при добавлении или удалении элемента из списка. Мы можем получить нужное поведение, определив в структуре методы `add`, `remove` и `average`, как показано в примере 17-2:

Файл: 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;
    }
}
```

Листинг 17-2: Реализация публичных методов `add`, `remove` и `average` структуры `AveragedCollection`

Публичные методы `add`, `remove` и `average` являются единственным способом получить или изменить данные в экземпляре `AveragedCollection`. Когда элемент добавляется в `list` методом `add`, или удаляется с помощью метода `remove`, код реализации каждого из этих методов вызывает приватный метод `update_average`, который позаботится об обновлении поля `average`.

Мы оставляем поля `list` и `average` недоступными, чтобы внешний код не мог добавлять или удалять элементы непосредственно в поле `list`; в противном случае `average` поле может оказаться не синхронизировано при изменении `list`. Метод `average` возвращает значение в поле `average`, что позволяет внешнему коду читать значение `average`, но не изменять его.

Поскольку мы инкапсулировали детали реализации структуры `AveragedCollection`, мы можем легко изменить такие аспекты, как структура данных, в будущем. Например, мы могли бы использовать `HashSet<i32>` вместо `Vec<i32>` для поля `list`. Благодаря тому, что сигнатуры публичных методов `add`, `remove` и `average`

остаются неизменными, код, использующий `AveragedCollection`, также не будет нуждаться в изменении. У нас бы не получилось этого достичь, если бы мы сделали поле `list` доступным внешнему коду: `HashSet<i32>` и `Vec<i32>` имеют разные методы для добавления и удаления элементов, поэтому внешний код, вероятно, должен измениться, если он модифицирует `list` напрямую.

Если инкапсуляция является обязательным аспектом для определения языка как объектно-ориентированного, то Rust соответствует этому требованию.

Возможность использования модификатора доступа `pub` для различных частей кода позволяет построить публичный интерфейс и инкапсулировать детали реализации.

Наследование как система типов и способ совместного использования кода

Наследование — это механизм, предоставляемый другими языками программирования, с помощью которого объект может быть определён, унаследовав данные и поведение от родительского объекта без необходимости их повторного определения.

Если язык должен иметь наследование для того, чтобы считаться объектно-ориентированным, тогда Rust не является таковым. Не существует способа определить структуру, которая бы наследовала поля и методы от другой структуры. Однако, если вы привыкли использовать наследование в списке своих инструментов, то в Rust можно использовать альтернативные решения.

Вы выбираете наследование по двум основным причинам. Одна из них - возможность повторного использования кода: вы можете реализовать определённое поведение для одного типа, а наследование позволяет вам повторно использовать эту реализацию для другого типа. Вместо этого в Rust можно делиться кодом, используя реализацию метода типажа по умолчанию, который вы видели в листинге 10-14, когда мы добавили реализацию по умолчанию в методе `summarize` типажа `Summary`. Любой тип, реализующий свойство `Summary` будет иметь доступный метод `summarize` без дополнительного кода. Это аналогично родительскому классу, имеющему реализацию метода и аналогично наследующему дочернему классу, также имеющему реализацию метода. Мы также можем переопределить реализацию по умолчанию для метода `summarize`, когда реализуем типаж `Summary`, что похоже на дочерний класс, переопределяющий реализацию метода, унаследованного от родительского класса.

Вторая причина использования наследования относится к системе типов: чтобы иметь возможность использовать дочерний тип в тех же места, что и родительский. Эта возможность также называется *полиморфизм* и означает возможность подменять объекты во время исполнения, если они имеют одинаковые характеристики.

Полиморфизм

Для многих людей полиморфизм является синонимом наследования. Но на самом деле это более общая концепция, которая относится к коду, который может работать с данными разных типов. Для наследования эти типы обычно являются подклассами. Вместо этого Rust использует обобщённые типы для абстрагирования от типов, и ограничения типажей (trait bounds) для указания того, какие возможности эти типы должны предоставлять. Это иногда называют *ограниченным параметрическим полиморфизмом*.

Вместо этого Rust использует обобщённые типы для абстрагирования от типов, и ограничения типажей (trait bounds) для указания того, какие возможности эти типы должны предоставлять. Это иногда называют *ограниченным параметрическим полиморфизмом*.

Наследование в последнее время утратило популярность, как подход к разработке, на многих языках программирования, поскольку часто существует риск совместного использования большего количества кода, чем необходимо. Подклассы не всегда должны иметь все общие характеристики родительского класса, но будут получать их через наследование. Это может сделать дизайн программы менее гибким. Это также даёт возможность вызова методов у подклассов, которые не имеют смысла или вызывают ошибки, потому что методы неприменимы к подклассу. Кроме того, в некоторых языках подкласс может наследоваться только от одного класса, что ещё больше ограничивает гибкость разработки программы.

По этим причинам Rust выбрал альтернативный подход с использованием типажей-объектов вместо наследования. Давайте посмотрим как типажи-объекты реализуют полиморфизм в Rust.

Использование типаж-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. Мы создали обходное решение в листинге 8-10, где мы определили перечисление `SpreadsheetCell` в котором были 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, представляющий строку из ячеек. Это очень хорошее решение, когда наши взаимозаменяемые элементы вектора являются типами с фиксированным набором, известным при компиляции кода.

Однако иногда мы хотим, чтобы пользователь нашей библиотеки мог расширить набор типов, которые допустимы в конкретной ситуации. Чтобы показать как этого добиться, мы создадим пример инструмента с графическим интерфейсом пользователя (GUI), который просматривает список элементов, вызывает метод `draw` для каждого из них, чтобы нарисовать его на экране - это обычная техника для инструментов GUI. Мы создадим библиотечный крейт с именем `gui`, содержащий структуру библиотеки GUI. Этот крейт мог бы включать некоторые готовые типы для использования, такие как `Button` или `TextField`. Кроме того, пользователи такого крейта `gui` захотят создавать свои собственные типы, которые могут быть нарисованы: например, кто-то мог бы добавить тип `Image`, а кто-то другой добавить тип `SelectBox`.

Мы не будем реализовывать полноценную библиотеку GUI для этого примера, но покажем, как её части будут подходить друг к другу. На момент написания библиотеки мы не можем знать и определить все типы, которые могут захотеть создать другие программисты. Но мы знаем, что `gui` должен отслеживать множество значений различных типов и ему нужно вызывать метод `draw` для каждого из этих значений различного типа. Ему не нужно точно знать, что произойдёт, когда вызывается метод `draw`, просто у значения будет доступен такой метод для вызова.

Чтобы сделать это на языке с наследованием, можно определить класс с именем `Component` у которого есть метод с названием `draw`. Другие классы, такие как `Button`, `Image` и `SelectBox` наследуются от `Component` и следовательно, наследуют метод `draw`. Каждый из них может переопределить реализацию метода `draw`, чтобы определить своё пользовательское поведение, но платформа может обрабатывать все типы, как если бы они были экземплярами `Component` и вызывать

`draw` у них. Но поскольку в Rust нет наследования, нам нужен другой способ структурировать `gui` библиотеку, чтобы позволить пользователям расширять её новыми типами.

Определение типажа для общего поведения

Чтобы реализовать поведение, которое мы хотим иметь в `gui`, мы определим типаж с именем `Draw`, который будет содержать один метод с названием `draw`. Затем мы можем определить вектор, который принимает *типаж-объект*. Типаж-объект указывает как на экземпляр типа, реализующего указанный типаж, так и на внутреннюю таблицу, используемую для поиска методов типажа указанного типа во время выполнения. Мы создаём типаж-объект, указывая что-то вроде указателя, такого как ссылка `&` или умный указатель `Box<T>`, затем ключевое слово `dyn`, а затем определяем соответствующий типаж. (Мы будем говорить о причине того, что типаж-объекты должны использовать указатель в главе 19 раздела "Типы динамического размера и `sized` типаж" <!-- -->). Мы можем использовать типаж-объекты вместо универсального или конкретного типа. Везде, где мы используем типаж-объект, система типов Rust гарантирует во время компиляции, что любое значение используемое в этом контексте будет реализовывать нужный типаж у типаж-объекта. Следовательно, нам не нужно знать все возможные типы во время компиляции.

Мы упоминали, что в Rust мы воздерживаемся от слова «объекты» для структур и перечислений, чтобы отличать их от объектов в других языках. В структуре или перечислении данные в полях структуры и поведение в блоках `impl` разделены, тогда как в других языках данные и поведение, объединённые в одну концепцию, часто обозначающуюся как объект. Тем не менее, типаж-объекты являются более похожими на объекты на других языках, в том смысле, что они сочетают в себе данные и поведение. Но типаж-объекты отличаются от традиционных объектов тем, что мы не можем добавлять данные к типаж-объекту. Типаж-объекты обычно не настолько полезны, как объекты в других языках: их конкретная цель - обеспечить абстракцию через обычное поведение.

В листинге 17.3 показано, как определить типаж с именем `Draw` с помощью одного метода с именем `draw`:

Файл: `src/lib.rs`

```
pub trait Draw {
    fn draw(&self);
}
```

Листинг 17-3: Определение типажа `Draw`

Этот синтаксис должен выглядеть знакомым из наших дискуссий о том, как определять типажи в главе 10. Далее следует новый синтаксис: в листинге 17.4 определена структура с именем `Screen`, которая содержит вектор с именем `components`. Этот вектор имеет тип `Box<dyn Draw>`, который и является типаж-объектом; это замена для любого типа внутри `Box` который реализует типаж `Draw`.

Файл: src/lib.rs

```
pub struct Screen {
    pub components: Vec<Box<dyn Draw>>,
}
```

Листинг 17-4: Определение структуры `Screen` с полем `components`, которое является вектором типаж-объектов, которые реализуют типаж `Draw`

В структуре `Screen`, мы определим метод `run`, который будет вызывать метод `draw` каждого элемента вектора `components`, как показано в листинге 17-5:

Файл: src/lib.rs

```
impl Screen {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

Листинг 17-5: Реализация метода `run` у структуры `Screen`, который вызывает метод `draw` каждого компонента из вектора

Это работает иначе, чем определение структуры, которая использует параметр общего типа с ограничениями типажа. Обобщённый параметр типа может быть заменён только одним конкретным типом, тогда как типаж-объекты позволяют

нескольким конкретным типам замещать типаж-объект во время выполнения. Например, мы могли бы определить структуру `Screen` используя общий тип и ограничение типажа, как показано в листинге 17-6:

Файл: 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();
        }
    }
}
```

Листинг 17-6: Альтернативная реализация структуры `Screen` и метода `run`, используя обобщённый тип и ограничения типажа

Это вариант ограничивает нас экземпляром `Screen`, который имеет список компонентов всех типов `Button` или всех типов `TextField`. Если у вас когда-либо будут только однородные коллекции, использование обобщений и ограничений типажа является предпочтительным, поскольку определения будут мономорфизированы во время компиляции для использования с конкретными типами.

С другой стороны, с помощью метода, использующего типаж-объекты, один экземпляр `Screen` может содержать `Vec<T>` который содержит `Box<Button>`, также как и `Box<TextField>`. Давайте посмотрим как это работает, а затем поговорим о влиянии на производительность во время выполнения.

Реализации типажа

Теперь мы добавим несколько типов, реализующих типаж `Draw`. Мы объявим тип `Button`. Опять же, фактическая реализация библиотеки GUI выходит за рамки этой книги, поэтому тело метода `draw` не будет иметь никакой полезной реализации.

Чтобы представить, как может выглядеть такая реализация, структура `Button` может иметь поля для `width`, `height` и `label`, как показано в листинге 17-7:

Файл: src/lib.rs

```
pub struct Button {
    pub width: u32,
    pub height: u32,
    pub label: String,
}

impl Draw for Button {
    fn draw(&self) {
        // code to actually draw a button
    }
}
```

Листинг 17-7: Структура `Button` реализует типаж `Draw`

Поля `width`, `height` и `label` структуры `Button` будут отличаться от, например, полей других компонентов вроде типа `TextField`, которая могла бы иметь те же поля плюс поле `placeholder`. Каждый из типов, который мы хотим нарисовать на экране будет реализовывать типаж `Draw`, но будет использовать отличающийся код метода `draw` для определения как рисовать конкретный тип, также как `Button` в этом примере (без фактического кода GUI, который выходит за рамки этой главы). Например, тип `Button` может иметь дополнительный блок `impl`, содержащий методы, относящиеся к тому, что происходит, когда пользователь нажимает кнопку. Эти варианты методов не будут применяться к таким типам, как `TextField`.

Если кто-то использующий нашу библиотеку решает реализовать структуру `SelectBox`, которая имеет `width`, `height` и поля `options`, он реализует также и типаж `Draw` для типа `SelectBox`, как показано в листинге 17-8:

Файл: src/main.rs

```
use gui::Draw;

struct SelectBox {
    width: u32,
    height: u32,
    options: Vec<String>,
}

impl Draw for SelectBox {
    fn draw(&self) {
        // code to actually draw a select box
    }
}
```

Листинг 17-8: Другой крейт использующий `gui` и реализующий типаж `Draw` у структуры `SelectBox`

Пользователь нашей библиотеки теперь может написать свою функцию `main` для создания экземпляра `Screen`. К экземпляру `Screen` он может добавить `SelectBox` и `Button`, поместив каждый из них в `Box<T>`, чтобы он стал типаж-объектом. Затем он может вызвать метод `run` у экземпляра `Screen`, который вызовет `draw` для каждого из компонентов. Листинг 17-9 показывает эту реализацию:

Файл: src/main.rs

```

use gui::{Button, Screen};

fn main() {
    let screen = Screen {
        components: vec![
            Box::new(SelectBox {
                width: 75,
                height: 10,
                options: vec![
                    String::from("Yes"),
                    String::from("Maybe"),
                    String::from("No"),
                ],
            }),
            Box::new(Button {
                width: 50,
                height: 10,
                label: String::from("OK"),
            }),
        ],
    };
    screen.run();
}

```

Листинг 17-9: Использование типаж-объектов для хранения значений разных типов, реализующих один и тот же типаж

Когда мы писали библиотеку, мы не знали, что кто-то может добавить тип `SelectBox`, но наша реализация `Screen` могла работать с новым типом и рисовать его, потому что `SelectBox` реализует типаж `Draw`, что означает, что он реализует метод `draw`.

Эта концепция, касающаяся только сообщений на которые значение отвечает, в отличии от конкретного типа значения, аналогична концепции *duck typing* в динамически типизированных языках: если что-то ходит как утка и крякает как утка, то она должна быть утка! В реализации метода `run` у `Screen` в листинге 17-5, `run` не нужно знать каким будет конкретный тип каждого компонента. Он не проверяет, является ли компонент экземпляром `Button` или `SelectBox`, он просто вызывает метод `draw` компонента. Указав `Box<dyn Draw>` в качестве типа значений в векторе `components`, мы определили `Screen` для значений у которых мы можем вызвать метод `draw`.

Преимущество использования типаж-объектов и системы типов Rust для написания кода, похожего на код с использованием концепции *duck typing* состоит в том, что

нам не нужно во время выполнения проверять реализует ли значение в векторе конкретный метод или беспокоиться о получении ошибок, если значение не реализует метод, мы все равно вызываем метод. Rust не скомпилирует наш код, если значения не реализуют типаж, который нужен типаж-объектам.

Например, код (17-10) демонстрирует, что случится если мы попытаемся добавить `String` в качестве компонента вектора:

Файл: `src/main.rs`

```
use gui::Screen;

fn main() {
    let screen = Screen {
        components: vec![Box::new(String::from("Hi"))],
    };

    screen.run();
}
```



Листинг 17-10: Попытка использования типа, который не реализует типаж для типаж-объекта

Мы получим ошибку, потому что `String` не реализует типаж `Draw`:

```
$ cargo run
Compiling gui v0.1.0 (file:///projects/gui)
error[E0277]: the trait bound `String: Draw` is not satisfied
--> src/main.rs:5:26
  |
5 |         components: vec![Box::new(String::from("Hi"))],
  |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `Draw` is
not implemented for `String`
  |
= note: required for the cast to the object type `dyn Draw`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `gui` due to previous error
```

Эта ошибка даёт понять, что либо мы передаём в компонент `Screen` что-то, что мы не собирались передавать и мы тогда должны передать другой тип, либо мы должны реализовать типаж `Draw` у типа `String`, чтобы `Screen` мог вызывать `draw` у него.

Типаж-объекты выполняют динамическую диспетчеризацию (связывание)

Напомним, в разделе «Производительность кода с использованием обобщений» главы 10 обсуждается процесс мономорфизации выполняемый компилятором, когда мы используем ограничения типажей для обобщённых типов: компилятор генерирует конкретные реализации функций и методов для каждого конкретного типа, который мы используем вместо параметра обобщённого типа. Код, полученный в результате мономорфизации, выполняет *статическую диспетчеризацию*, когда компилятор знает какой метод вы вызываете во время компиляции. Это противоположно подходу *динамической диспетчеризации*, когда компилятор не может сказать во время компиляции, какой метод вы вызываете. В случаях динамической диспетчеризации компилятор генерирует код, который во время выполнения определяет, какой метод необходимо вызывать.

Когда мы используем типаж-объекты, Rust должен использовать динамическую диспетчеризацию. Компилятор не знает всех типов, которые могут быть использованы с кодом, использующим типаж-объекты, поэтому он не знает, какой метод реализован для какого типа при вызове. Вместо этого во время выполнения Rust использует указатели внутри типаж-объекта, чтобы узнать какой метод вызывать. Когда происходит такой поиск, то возникают затраты времени выполнения, которые не происходят при статической диспетчеризации. Динамическая диспетчеризация также не позволяет компилятору выбрать встраивание кода метода, что в свою очередь предотвращает некоторые оптимизации. Однако мы получили дополнительную гибкость в коде, который мы написали в листинге 17-5 и смогли поддержать в листинге 17-9, так что это является компромиссом.

Безопасность объекта необходима для типаж-объектов

Вы можете превратить только *объектно-безопасные* типажи в типаж-объекты. Некоторые сложные правила управляют всеми свойствами, которые делают типаж-объект безопасным, но на практике имеют значение только два правила. Типаж является объектно-безопасным, если все методы определённые в нем имеют следующие свойства:

- Тип возвращаемого значения не является `Self`.
- Нет обобщённых параметров типа.

Ключевое слово `Self` является псевдонимом для типа, для которого мы реализуем типажи или его методы. Типаж-объекты должны быть объектно-безопасными, потому что как только вы использовали типаж-объект Rust больше не знает точный тип, реализующий типаж. Если метод типажа возвращает тип `Self`, но типаж-объект забывает чем является этот точный тип `Self`, то у метода нет возможности использовать исходный точный тип. То же самое верно для параметров обобщённого типа, которые заполняются параметрами конкретного типа при использовании типажа: конкретные типы становятся частью типа, который реализует типаж. Когда тип забыт из-за использования типаж-объекта, невозможно узнать какими типами нужно заполнять параметры обобщённого типа.

Примером типажа методы которого не являются объектно-безопасными, является типаж `Clone` из стандартной библиотеки. Сигнатура для метода `clone` в типаже `Clone` выглядит следующим образом:

```
pub trait Clone {  
    fn clone(&self) -> Self;  
}
```

Тип `String` реализует типаж `Clone` и когда мы вызываем метод `clone` у экземпляра `String`, мы получаем экземпляр `String`. Точно так же, если мы вызываем `clone` для экземпляра `Vec<T>` то, мы возвращаем экземпляр `Vec<T>`. Сигнатура `clone` должна знать, какой тип будет заменять `Self`, потому что это тип возвращаемого значения.

Компилятор укажет, когда вы пытаетесь сделать что-то, что нарушает правила безопасности объекта в отношении свойств объекта. Например, допустим, мы попытались реализовать структуру `Screen` в листинге 17-4 для хранения типов, которые реализуют типаж `Clone` вместо типажа `Draw`, например:

```
{[#rustdoc_include ../listings/ch17-oop/no-listing-01-trait-object-of-  
clone/src/lib.rs]}
```

Мы получим ошибку:

```
{[#include ../listings/ch17-oop/no-listing-01-trait-object-of-  
clone/output.txt]}
```

Эта ошибка означает, что таким способом вы не можете использовать этот типаж как типаж-объект. Если вы заинтересованы в более подробной информации о

безопасности объектов, см. [Rust RFC 255](#).

Реализация ООП шаблона проектирования

Шаблон *состояние* является объектно-ориентированным шаблоном проектирования. Суть шаблона в том, что главный объект имеет некоторое внутреннее состояние, которое представлено набором *объектов состояния* и поведение этого объекта изменяется в зависимости от внутреннего состояния. Объекты состояния имеют общую функциональность: конечно в Rust мы используем структуры и типажи, а не объекты и наследование. Каждый объект состояния отвечает за своё поведение и определяет, когда он должен перейти в другое состояние. Значение, которое содержит объект состояния, ничего не знает о различном поведении состояний или о моментах перехода между состояниями.

Использование шаблона состояния означает, что при изменении бизнес-требований программы не нужно изменять код объекта, содержащего состояние или код использующий такой объект. Нам нужно только обновить код внутри одного из значений состояния, чтобы изменить его правила или, возможно, добавить больше значений состояния. Давайте рассмотрим пример проектирования шаблона состояния и как его использовать в Rust.

Мы поэтапно реализуем процесс создания поста в блоге. Окончательная функциональность блога будет выглядеть так:

1. При создании сообщения публикации она создаётся как пустой черновик.
2. Когда черновик готов, запрашивается его рецензия.
3. После проверки происходит её публикация.
4. Только опубликованные сообщения блога возвращают контент в печать, поэтому неутверждённые сообщения не могут быть опубликованы случайно.

Любые другие изменения, сделанные в сообщении, не должны иметь никакого эффекта. Например, если мы попытаемся подтвердить черновик сообщения в блоге до того как запросим рецензию, то сообщение должно остаться неопубликованным черновиком.

В листинге 17-11 показан этот рабочий процесс в виде кода: это пример использования API, который мы реализуем в библиотеку с именем `blog`. Он ещё не компилируется, потому что мы ещё не реализовали крейт для `blog`.

Файл: `src/main.rs`

```
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");
    assert_eq!("", post.content());

    post.request_review();
    assert_eq!("", post.content());

    post.approve();
    assert_eq!("I ate a salad for lunch today", post.content());
}
```



Листинг 17-11: Код, демонстрирующий желаемое поведение, которое мы хотим получить в крейте `blog`

Мы хотим, чтобы пользователь мог создать новый черновик сообщения в блоге с помощью `Post::new`. Затем мы хотим разрешить добавление текста в сообщение блога, пока он находится в черновом состоянии. Если мы попытаемся получить содержимое сообщения до его утверждения, ничего не произойдёт, потому что сообщение все ещё является черновиком. Мы добавили `assert_eq!` в коде для демонстрационных целей. Отличным модульным тестом для этого было бы утверждение, что черновик поста блога возвращает пустую строку из метода `content`, но мы не собираемся писать тесты для этого примера.

Далее мы хотим разрешить сделать запрос на проверку публикации и хотим, чтобы `content` возвращал пустую строку в ожидании проверки. Когда сообщение получит одобрение, оно должно быть опубликовано, то есть текст сообщения будет возвращён при вызове `content`.

Обратите внимание, что единственный тип из крейта, с которым мы взаимодействуем - это тип `Post`. Этот тип будет использовать шаблон состояния и будет содержать значение, которое будет одним из трёх состояний объекта, представляющих различные состояния в которых может находиться публикация: черновик, сообщение ожидающее проверки или опубликованное сообщение. Переход из одного состояния в другое будет осуществляться внутри типа `Post`. Состояния изменяются в ответ на методы, вызываемые пользователями нашей библиотеки у экземпляра `Post`, но они не должны напрямую управлять изменениями состояния. Кроме того, пользователи не могут ошибиться с состояниями, например, опубликовать сообщение до его проверки.

Определение `Post` и создание нового экземпляра в состоянии черновика

Приступим к реализации библиотеки! Мы знаем, что нам нужна публичная структура `Post`, которая содержит некоторое содержимое, поэтому мы начнём с определения структуры и связанную с ней публичную функцию `new` для создания экземпляра `Post`, как показано в листинге 17-12. Мы также сделаем приватный типаж `State`. Затем `Post` будет содержать типаж-объект `Box<dyn State>` внутри `Option<T>` в приватном поле с названием `state`. Вскоре вы поймёте, почему `Option<T>` необходим. Файл: `src/lib.rs`

Файл: `src/lib.rs`

```
pub struct Post {
    state: Option<Box<dyn State>>,
    content: String,
}

impl Post {
    pub fn new() -> Post {
        Post {
            state: Some(Box::new(Draft {})),
            content: String::new(),
        }
    }
}

trait State {}

struct Draft {}

impl State for Draft {}
```

Листинг 17-12. Определение структуры `Post` и функции `new`, которая создаёт новый экземпляр `Post`, типаж `State` и структуру `Draft`

Типаж `State` определяет поведение совместно используемое различными состояниями в сообщениях, все типы состояний вроде `Draft`, `PendingReview` и `Published` будут реализовывать типаж `State`. Пока у этого типажа нет никаких методов и мы начнём с определения только `Draft` состояния, потому что это то состояние, в котором мы хотим сделать появление публикации.

Когда мы создаём новый экземпляр `Post`, мы устанавливаем его поле `state` в значение `Some`, содержащее `Box`. Этот `Box` указывает на новый экземпляр

структурой `Draft`. Это гарантирует, что всякий раз, когда мы создаём новый экземпляр `Post`, он появляется как черновик. Поскольку поле `state` в структуре `Post` является приватным, то нет никакого способа создать `Post` в любом другом состоянии! В функции `Post::new` мы устанавливаем в поле `content` новую пустую строку `String`.

Хранение текста содержимого публикации

В листинге 17-11 показано, что мы хотим иметь возможность вызывать метод `add_text` и передать ему `&str`, которое добавляется к текстовому содержимому публикации блога. Мы реализуем эту возможность как метод, а не предоставляем поле `content` как публично доступное `pub`. Это означает, что позже мы сможем реализовать метод, который будет контролировать, как читаются данные из поля `content`. Метод `add_text` довольно прост, поэтому давайте добавим его реализацию в листинге 17-13 в `impl Post`:

Файл: src/lib.rs

```
impl Post {
    // --snip--
    pub fn add_text(&mut self, text: &str) {
        self.content.push_str(text);
    }
}
```

Листинг 17-13. Реализация `add_text` для добавления текста к `content`

Метод `add_text` принимает изменяемую ссылку на `self`, потому что мы меняем экземпляр `Post` для которого вызываем `add_text`. Затем мы вызываем `push_str` для `String` у `content` и передаём `text` аргументом для добавления к сохранённому `content`. Это поведение не зависит от состояния, в котором находится сообщение, поэтому оно не является частью шаблона состояния. Метод `add_text` вообще не взаимодействует с полем `state`, но это часть поведения, которое мы хотим поддерживать.

Гарантирование пустого содержания черновика сообщения

Даже после того, как мы вызвали `add_text` и добавили некоторый контент в наше сообщение, мы хотим чтобы метод `content` возвращал пустой фрагмент строки,

потому что публикация находится ещё в черновом состоянии, как это показано в строке 7 листинга 17-11. А пока давайте реализуем метод `content` с самым простым функционалом, который будет выполнять это требование: всегда возвращать пустой фрагмент строки. Мы изменим код позже, как только реализуем возможность изменить состояние сообщения, чтобы оно могло быть опубликовано. Пока что сообщения могут находиться только в черновом состоянии, поэтому содержимое сообщения всегда должно быть пустым. Листинг 17-14 показывает пустую реализацию:

Файл: src/lib.rs

```
impl Post {
    // --snip--
    pub fn content(&self) -> &str {
        ""
    }
}
```

Листинг 17-14. Добавление реализации заполнителя для метода `content` в `Post`, который всегда возвращает пустой фрагмент строки.

С этим дополнением `content` все в листинге 17-11 до строки 7 работает, как задумано.

Запрос проверки публикации меняет её состояние

Далее нам нужно добавить функциональность запроса рецензии публикации, который должен изменить её состояние с `Draft` на `PendingReview`. Листинг 17-15 показывает этот код:

Файл: src/lib.rs

```

impl Post {
    // --snip--
    pub fn request_review(&mut self) {
        if let Some(s) = self.state.take() {
            self.state = Some(s.request_review())
        }
    }
}

trait State {
    fn request_review(self: Box<Self>) -> Box<dyn State>;
}

struct Draft {}

impl State for Draft {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        Box::new(PendingReview {})
    }
}

struct PendingReview {}

impl State for PendingReview {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        self
    }
}

```

Листинг 17-15. Реализация методов `request_review` в `Post` и `State`

Мы предоставляем в `Post` публичный метод с именем `request_review`, который будет принимать изменяемую ссылку на `self`. Затем мы вызываем внутренний метод `request_review` у поля для текущего состояния `Post` и этот второй метод `request_review` поглощает текущее состояние и возвращает новое состояние.

Мы добавили метод `request_review` `State`; все типы, реализующие этот типаж, теперь должны будут реализовать 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 действителен только при вызове `Box` содержащего тип. Этот синтаксис становится владельцем `Box<Self>`, делая старое состояние недействительным, поэтому значение состояния `Post` может преобразоваться в новое состояние.

Чтобы поглотить старое состояние, метод `request_review` должен стать владельцем

значения состояния. Это место где `Option` для поля `state` публикации `Post` приходит на помощь: мы называем метод `take` чтобы забрать значение `Some` из поля `state` и оставить значение `None` в поле, потому что Rust не позволяет иметь не инициализированные поля в структурах. Это позволяет перемещать значение `state` из `Post`, а не заимствовать его. Затем мы установим новое значение `state` как результат этой операции.

Нам нужно временно установить `state` в `None`, а не устанавливать его напрямую с помощью кода вроде `self.state = self.state.request_review();` для получения владения значения поля `state`. Это гарантирует, что `Post` не сможет использовать старое значение `state` после того, как мы преобразовали его в новое состояние.

Метод `request_review` в `Draft` должен вернуть новый "упакованный" экземпляр новой структуры `PendingReview`, которая предоставляет состояние, когда публикация ожидает рецензии. Структура `PendingReview` также реализует метод `request_review`, но не выполняет никаких преобразований. Она возвращает сам себя, потому что когда мы запрашиваем рецензию публикации, она уже в состоянии `PendingReview` и должна продолжать оставаться в состоянии `PendingReview`.

Теперь мы можем увидеть преимущества шаблона состояния: метод `request_review` для `Post` одинаков независимо от значения его `state`. Каждое состояние несёт ответственность за свои правила.

Мы оставим метод `content` у `Post` таким как он есть, возвращая пустой фрагмент строки. Теперь мы можем получить `Post` из состояния `PendingReview`, а также из состояния `Draft`, но нам нужно такое же поведение в состоянии `PendingReview`. Листинг 17-11 теперь работает до строки 10!

Добавление метода `approve` который изменяет поведение `content`

Метод `approve` будет аналогичен методу `request_review`: он будет устанавливать у `state` значение, которое должна иметь публикация при её одобрении, как показано в листинге 17-16:

Файл: src/lib.rs

```
impl Post {
    // --snip--
    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 {
    // --snip--
    fn approve(self: Box<Self>) -> Box<dyn State> {
        self
    }
}

struct PendingReview {}

impl State for PendingReview {
    // --snip--
    fn approve(self: Box<Self>) -> Box<dyn State> {
        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
    }
}
```

Листинг 17-16. Реализация метода `approve` для типа `Post` и типажа `State`

Мы добавляем метод `approve` в типаж `State`, добавляем новую структуру, которая реализует этот типаж `State` и структуру для состояния `Published`.

Подобно тому, как работает `request_review` для `PendingReview`, если мы вызовем метод `approve` для `Draft`, он не будет иметь никакого эффекта, потому что `approve` вернёт `self`. Когда мы вызываем для `PendingReview` метод `approve`, то он возвращает новый упакованный экземпляр структуры `Published`. Структура `Published` реализует трейт `State`, и как для метода `request_review`, так и для метода `approve` она возвращает себя, потому что в этих случаях сообщение должно оставаться в состоянии `Published`.

Теперь нам нужно обновить метод `content` для `Post`. Мы хотим, чтобы значение, возвращаемое из `content`, зависело от текущего состояния `Post`, поэтому мы собираемся делегировать `Post` в метод `content`, определённого для его `state`, как показано в листинге 17.17:

Файл: src/lib.rs

```
impl Post {
    // --snip--
    pub fn content(&self) -> &str {
        self.state.as_ref().unwrap().content(self)
    }
    // --snip--
}
```



Листинг 17-17: Обновление метода `content` в структуре `Post` для делегирования вызова методу `content` у структуры `State`

Поскольку цель состоит в том, чтобы сохранить все эти правила внутри структур, реализующих типаж `State`, мы вызываем метод `content` у значения в поле `state` и передаём экземпляр публикации (то есть `self`) в качестве аргумента. Затем мы возвращаем значение, которое возвращается при использовании метода `content` у поля `state`.

Мы вызываем метод `as_ref` у `Option` потому что нам нужна ссылка на значение внутри `Option`, а не владение значением. Поскольку `state` является типом `Option<Box<dyn State>>`, то при вызове метода `as_ref`, возвращается `Option<&Box<dyn State>>`. Если бы мы не вызывали `as_ref`, мы бы получили ошибку, потому что мы не можем переместить `state` из заимствованного `&self` параметра функции.

Затем мы вызываем метод `unwrap`, который как мы знаем в данном месте никогда не паникует, потому что мы знаем, что методы `Post` гарантируют что в `state` будет

всегда содержаться значение `Some`, после выполнения методов. Это один из случаев, о которых мы говорили в разделе "Случаи, когда у вас больше информации, чем у компилятора" главы 9, когда мы знаем, что значение `None` невозможно, даже если компилятор не может этого понять.

В этот момент, когда мы вызываем `content` у типа `&Box<dyn State>`, в действие вступает принудительное приведение (deref coercion) для `&` и `Box`, поэтому в конечном итоге метод `content` будет вызываться для типа, который реализует типаж `State`. Это означает, что нам нужно добавить метод `content` в определение типажа `State` и именно там мы поместим логику для того, какое содержание возвращать в зависимости от состояния, которое мы имеем как показано в листинге 17-18:

Файл: src/lib.rs

```
trait State {
    // --snip--
    fn content<'a>(&self, post: &'a Post) -> &'a str {
        ""
    }
}

// --snip--
struct Published {}

impl State for Published {
    // --snip--
    fn content<'a>(&self, post: &'a Post) -> &'a str {
        &post.content
    }
}
```

Листинг 17-18. Добавление метода `content` в трейт `State`

Мы добавляем реализацию по умолчанию метода `content`, который возвращает пустой фрагмент строки. Это означает, что не нужно реализовывать `content` в структурах `Draft` и `PendingReview`. Структура `Published` будет переопределять метод `content` и вернёт значение из `post.content`.

Обратите внимание, как мы обсуждали в главе 10, нам нужны аннотации времени жизни у этого метода. Мы берём ссылку на `post` в качестве аргумента и возвращаем ссылку на часть `post`, поэтому время жизни возвращённой ссылки связано с временем жизни аргумента `post`.

Мы закончили, теперь все из листинга 17-11 работает! Мы внедрили шаблон состояния с помощью правил процесса публикации блога. Логика, связанная с правилами, живёт в объектах состояний, а не разбросана по всей структуре `Post`.

Компромиссы шаблона состояния

Мы показали, что Rust способен реализовать объектно-ориентированный шаблон состояния для инкапсуляции различных типов поведения, которые должна иметь публикация в каждом состоянии. Методы в `Post` ничего не знают о различных видах поведения. При такой организации кода, мы должны смотреть только в одном месте, чтобы узнать, как может вести себя опубликованная публикация: в реализации типажа `State` у структуры `Published`.

Если бы нужно было создать альтернативную реализацию не используя шаблон состояния, то мы могли бы использовать выражения `match` в методах структуры `Post` или даже в коде `main`, который проверяет состояние публикации и изменяет поведение в этих местах. Это означало бы, что нам пришлось бы искать в нескольких местах, чтобы понять как сообщение попадает в опубликованное состояние! Это могло бы только увеличить число добавленных нами состояний: каждому из этих выражений `match` потребовались бы ещё внутренние рукава.

С помощью шаблона состояния, методы `Post` и места в которых мы используем `Post` не требуют использования `match` выражения, а для добавления нового состояния нужно было бы только добавить новую структуру и реализовать методы типажа у этой одной структуры.

Реализация с использованием шаблона состояния легко расширяется с добавлением функциональности. Чтобы увидеть простоту поддержки кода, который использует данный шаблон состояния, попробуйте выполнить предложения:

- Добавьте метод `reject`, который изменяет состояние публикации из `PendingReview` обратно в `Draft`.
- Потребуйте два вызова метода `approve` прежде чем состояние можно изменить в `Published`.
- Разрешите пользователям добавлять текстовое содержимое только тогда, когда публикация находится в состоянии `Draft`. Подсказка: наличие объекта состояния, отвечающего за то, может ли измениться содержимое, но не отвечающего за изменение `Post`.

Одним из недостатков шаблона состояния является то, что поскольку состояния реализуют переходы между ними, некоторые из состояний связаны друг с другом. Если мы добавим другое состояние между `PendingReview` и `Published`, такое как например `Scheduled`, то придётся изменить код в `PendingReview`, чтобы оно переходило в `Scheduled`. Было бы меньше работы, если бы не нужно было менять `PendingReview` при добавлением нового состояния, но это означало бы переключение на другой шаблон проектирования.

Другим недостатком является то, что мы продублировали некоторую логику. Чтобы устраниТЬ некоторое дублирование, мы могли бы попытаться сделать реализации по умолчанию для методов `request_review` и `approve` типажа `State`, которые возвращают `self`; однако это нарушило бы безопасность объекта, потому что типаж не знает, какой конкретно будет `self`. Мы хотим иметь возможность использовать `State` в качестве типаж-объекта, поэтому нам нужно, чтобы его методы были безопасны для объекта.

Другое дублирование включает в себя аналогичные реализации методов `request_review` и `approve` у `Post`. Оба метода делегируют реализации одного и того же метода значению поля `state` типа `Option` и устанавливают результатом новое значение поля `state`. Если бы у `Post` было много методов, которые следовали этому шаблону, мы могли бы рассмотреть определение макроса для устранения повторения (смотри раздел "Макросы" в главе 19).

Реализуя шаблон состояния точно так, как он определён для объектно-ориентированных языков, мы не в полной мере используем преимущества Rust, как можно было бы. Давайте посмотрим на некоторые изменения, которые мы можем внести в крейт `blog`, которые могут превратить недопустимые состояния и переходы в ошибки времени компиляции.

Кодирование состояний и поведения как типы

Мы покажем вам, как переосмыслить шаблон состояния, чтобы получить другой набор компромиссов. Вместо того, чтобы полностью инкапсулировать состояния и переходы так, чтобы внешний код не знал о них, мы будем кодировать состояния с помощью разных типов. Следовательно, система проверки типов Rust предотвратит попытки использовать черновые публикации, там где разрешены только опубликованные публикации, вызывая ошибки компиляции.

Давайте рассмотрим первую часть `main` в листинге 17-11:

Файл: src/main.rs

```
fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");
    assert_eq!("", post.content());
}
```

Мы по-прежнему разрешаем создание новых сообщений в состоянии черновик используя метод `Post::new` и возможность добавлять текст к содержимому публикации. Но вместо того, чтобы иметь метод `content` у чернового сообщения, которое возвращает пустую строку мы сделаем так, что у черновых сообщений вообще не было метода `content`. Таким образом, если мы попытаемся получить содержимое черновика, мы получим ошибку компилятора, сообщающую, что метод не существует. В результате мы не сможем случайно отобразить черновик содержимого публикации, потому что этот код даже не скомпилируется. В листинге 17-19 показано определение структур `Post` и `DraftPost`, а также методов для каждой из них:

Файл: src/lib.rs

```
pub struct Post {
    content: String,
}

pub struct DraftPost {
    content: String,
}

impl Post {
    pub fn new() -> DraftPost {
        DraftPost {
            content: String::new(),
        }
    }

    pub fn content(&self) -> &str {
        &self.content
    }
}

impl DraftPost {
    pub fn add_text(&mut self, text: &str) {
        self.content.push_str(text);
    }
}
```

Листинг 17-19: Структура `Post` с методом `content` и структура `DraftPost` без метода `content`

Обе структуры `Post` и `DraftPost` имеют приватное поле `content` в котором хранится текст сообщения блога. Структуры больше не имеют поля `state`, потому что мы перемещаем функционал состояния в типы структур. Структура `Post` будет представлять опубликованную публикацию и у неё есть метод `content`, который возвращает `content`.

У нас все ещё есть функция `Post::new`, но вместо возврата экземпляра `Post` она возвращает экземпляр `DraftPost`. Поскольку метод `content` является приватным и нет никаких функций, которые возвращают `Post`, то невозможно сразу создать экземпляр `Post`.

Структура `DraftPost` имеет метод `add_text`, поэтому мы можем добавлять текст к `content` как и раньше, но учтите, что в `DraftPost` не определён метод `content`! Так что теперь программа гарантирует, что все публикации начинаются как черновики, а черновики публикаций не имеют своего контента для отображения. Любая попытка обойти эти ограничения приведёт к ошибке компилятора.

Реализация переходов как преобразований в другие типы

Так как же получить опубликованный пост? Мы хотим обеспечить соблюдение правила, согласно которому черновик сообщения должен быть рассмотрен и утверждён до того, как он будет опубликован. Публикация находящаяся в состоянии отложенной рецензии, по-прежнему не должна отображать содержимое. Давайте реализуем эти ограничения, добавив ещё одну структуру, `PendingReviewPost`, определяя метод `request_review` у `DraftPost` возвращающий `PendingReviewPost`, определяя метод `approve` у `PendingReviewPost`, возвращающий `Post`, как показано в листинге 17-20:

Файл: src/lib.rs

```
impl DraftPost {
    // --snip--
    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,
        }
    }
}
```

Листинг 17-20: Тип `PendingReviewPost`, который создаётся путём вызова `request_review` экземпляра `DraftPost` и метод `approve`, который превращает `PendingReviewPost` в опубликованный `Post`.

Методы `request_review` и `approve` забирают во владение `self`, таким образом поглощая экземпляры `DraftPost` и `PendingReviewPost`, которые потом преобразуются в `PendingReviewPost` и опубликованное `Post` соответственно. Таким образом, у нас не будет никаких длительных экземпляров `DraftPost` после того, как мы вызвали у них `request_review` и так далее. В структуре `PendingReviewPost` не определён метод `content`, поэтому попытка прочитать его содержимое приводит к ошибке компилятора, как в случае с `DraftPost`. Так как единственным способом

получить опубликованный экземпляр `Post` у которого действительно есть объявленный метод `content`, является вызов метода `approve` у экземпляра `PendingReviewPost`, то единственный способ получить `PendingReviewPost` это вызвать метод `request_review` у экземпляра `DraftPost`. Так мы реализовали процесс публикации блога с помощью системы типов.

Но мы также должны внести небольшие изменения в `main`. Методы `request_review` и `approve` возвращают новые экземпляры, а не изменяют структуру к которой они обращаются, поэтому нам нужно добавить больше выражений `let post =` затеняя присваивания для сохранения возвращаемых экземпляров. Мы также не можем использовать утверждения для черновика и ожидающей рецензии публикации сравнивая содержимое с пустыми строками, сейчас они нам не нужны: мы больше не сможем скомпилировать код, который пытается использовать содержимое сообщений в этих состояниях. Обновлённый код в `main` показан в листинге 17-21:

Файл: src/main.rs

```
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");

    let post = post.request_review();

    let post = post.approve();

    assert_eq!("I ate a salad for lunch today", post.content());
}
```

Листинг 17-21: Изменения в `main` для использования новой реализации процесса публикации блога

Изменения, которые нам нужно было внести в `main` чтобы переназначить `post` означают, что эта реализация не совсем следует объектно-ориентированному шаблону состояний: преобразования между состояниями больше не инкапсулированы полностью внутри реализации `Post`. Тем не менее выгода в том, что недопустимые состояния теперь невозможны из-за системы типов и проверки типов, которая происходит во время компиляции! Это гарантирует, что определённые ошибки, такие как отображение содержимого неопубликованной публикации, будут обнаружены до того, как они поступят в производство.

Попробуйте в крейте **blog** сделать предложенные задачи с дополнительными требованиями, которые мы упомянули в начале этого раздела, как это происходит после листинга 17-20, чтобы увидеть и подумать о дизайне версии этого кода. Обратите внимание, что некоторые задачи могут быть выполнены уже в этом проекте.

Мы видели, что хотя Rust и способен реализовывать объектно-ориентированные шаблоны проектирования, но в нем также доступны другие шаблоны, такие как кодирование состояния с помощью системы типов. Эти модели имеют различные компромиссы. Хотя вы, возможно, очень хорошо знакомы с объектно-ориентированными шаблонами, переосмысление проблем для использования преимуществ и возможностей Rust может дать такие выгоды, как предотвращение некоторых ошибок во время компиляции. Объектно-ориентированные шаблоны не всегда будут лучшим решением в Rust из-за наличия определённых возможностей, такого как владение, которого нет у объектно-ориентированных языков.

Итоги

Независимо от того, что вы думаете о принадлежности Rust к объектно-ориентированным языкам после прочтения этой главы, вы теперь знаете, что можете использовать типаж-объекты, чтобы получить некоторые объектно-ориентированные функции в Rust. Динамическая диспетчеризация может дать вашему коду некоторую гибкость в обмен на небольшое ухудшение производительности во время выполнения. Вы можете использовать эту гибкость для реализации объектно-ориентированных шаблонов, которые могут помочь в поддержке вашего кода. В Rust также есть другие функции, такие как владение, которых нет у объектно-ориентированных языков. Объектно-ориентированный шаблон не всегда является лучшим способом использовать преимущества Rust, но является доступной опцией.

Далее мы рассмотрим шаблоны, которые являются ещё одной возможностью языка Rust, обеспечивающей больше гибкости. Мы кратко рассмотрели их на протяжении всей книги, но ещё не увидели всех этих возможностей. Приступим!

Шаблоны и сопоставление

Шаблоны - это специальный синтаксис в Rust для сопоставления со структурой типов, как сложных, так и простых. Использование шаблонов в сочетании с выражениями `match` и другими конструкциями даёт вам больший контроль над потоком управления программы. Шаблон состоит из некоторой комбинации следующего:

- Литералы
- Реструктуризованные массивы, перечисления, структуры или кортежи
- Переменные
- Специальные символы
- Заполнители

Эти компоненты описывают форму данных, с которыми мы работаем и затем сопоставляем их с указанными значениями в памяти, чтобы определить, есть ли в нашей программе правильные данные для продолжения выполнения определённого фрагмента кода.

Чтобы использовать шаблон, мы сравниваем его с некоторым значением. Если шаблон соответствует значению, мы используем части значения в нашем дальнейшем коде. Вспомните выражения `match` главы 6, в которых использовались шаблоны, например, описание машины для сортировки монет. Если значение в памяти соответствует форме шаблона, мы можем использовать именованные части шаблона. Если этого не произойдёт, то не выполнится код, связанный с шаблоном.

Эта глава - справочник по всем моментам, связанным с шаблонами. Мы расскажем о допустимых местах использования шаблонов, разнице между опровергими и неопровергими шаблонами и про различные виды синтаксиса шаблонов, которые вы можете увидеть. К концу главы вы узнаете, как использовать шаблоны для ясного выражения многих понятий.

Все случаи, где могут быть использованы шаблоны

В процессе использования языка Rust вы часто используете шаблоны, даже не осознавая этого! В этом разделе обсуждаются все случаи, где использование шаблонов является корректным.

Ветки `match`

Как обсуждалось в главе 6, мы используем шаблоны в ветках выражений `match`. Формально выражения `match` определяется как ключевое слово `match`, значение используемое для сопоставления, одна или несколько веток, которые состоят из шаблона и выражения для выполнения, если значение соответствует шаблону этой ветки, как здесь:

```
match VALUE {  
    PATTERN => EXPRESSION,  
    PATTERN => EXPRESSION,  
    PATTERN => EXPRESSION,  
}
```

Одно из требований к выражениям `match` состоит в том, что они должны быть *исчерпывающими* (*exhaustive*) в том смысле, что они должны учитывать все возможности для значения в выражении `match`. Один из способов убедиться, что вы рассмотрели каждую возможность - это иметь шаблон перехвата всех вариантов в последней ветке выражения: например, имя переменной, совпадающее с любым значением, никогда не может потерпеть неудачу и таким образом, охватывает каждый оставшийся случай.

Конкретный шаблон `_` будет соответствовать чему угодно, но он никогда не привязывается к переменной, поэтому он часто используется в последней ветке. Шаблон `_` может быть полезен, если вы, например, хотите игнорировать любое не указанное значение. Мы рассмотрим шаблон `_` более подробно в разделе "Игнорирование значений в шаблоне" позже в этой главе.

Условные выражения `if let`

В главе 6 мы обсуждали, как использовать выражения `if let` как правило в качестве более короткого способа записи эквивалента `match`, которое

обрабатывает только один случай. Дополнительно `if let` может иметь соответствующий `else`, содержащий код для выполнения, если шаблон выражения `if let` не совпадает.

В листинге 18-1 показано, что можно также смешивать и сопоставлять выражения `if let`, `else if` и `else if let`. Это даёт больше гибкости, чем `match` выражение, в котором можно выразить только одно значение для сравнения с шаблонами. Кроме того, условия в серии `if let`, `else if`, `else if let` не обязаны относиться друг к другу.

Код в листинге 18-1 показывает последовательность проверок нескольких условий, определяющих каким должен быть цвет фона. В данном примере, мы создали переменные с предопределёнными значениями, которые в реальной программе могли получить из пользовательского ввода.

Файл: src/main.rs

```
fn main() {
    let favorite_color: Option<&str> = None;
    let is_tuesday = false;
    let age: Result<u8, _> = "34".parse();

    if let Some(color) = favorite_color {
        println!("Using your favorite color, {}, as the background", color);
    } else if is_tuesday {
        println!("Tuesday is green day!");
    } else if let Ok(age) = age {
        if age > 30 {
            println!("Using purple as the background color");
        } else {
            println!("Using orange as the background color");
        }
    } else {
        println!("Using blue as the background color");
    }
}
```

Листинг 18-1: Использование условных конструкций `if let`, `else if`, `else if let`, и `else`

Если пользователь указывает любимый цвет, то этот цвет является цветом фона. Если сегодня вторник, то цвет фона - зелёный. Если пользователь указывает свой возраст в виде строки, и мы можем успешно проанализировать его и представить числом, то цвет будет либо фиолетовым, либо оранжевым, в зависимости от значения числа. Если ни одно из этих условий не выполняется, то цвет фона будет

синим.

Эта условная структура позволяет поддерживать сложные требования. С жёстко закодированными значениями, которые у нас здесь есть, этот пример напечатает **Using purple as the background color.**

Можно увидеть, что **if let** может также вводить затенённые переменные, как это можно сделать в **match** ветках: строка **if let Ok(age) = age** вводит новую затенённую переменную **age**, которая содержит значение внутри варианта **Ok**. Это означает, что нам нужно поместить условие **if age > 30** внутри этого блок: мы не можем объединить эти два условия в **if let Ok(age) = age && age > 30**. Затенённый **age**, который мы хотим сравнить с 30, не является действительным, пока не начнётся новая область видимости с фигурной скобки.

Недостатком использования **if let** выражений является то, что компилятор не проверяет полноту (exhaustiveness) всех вариантов, в то время как с помощью выражения **match** это происходит. Если мы пропустим последний блок **else** и поэтому пропустим обработку некоторых случаев, компилятор не предупредит нас о возможной логической ошибке.

Условные циклы **while let**

Аналогично конструкции **if let**, конструкция условного цикла **while let** позволяет условию цикла **while** работать до тех пор, пока шаблон продолжает совпадать. Пример в листинге 18-2 демонстрирует цикл **while let**, который использует вектор в качестве стека и печатает значения вектора в порядке обратном тому, в котором они были помещены.

```
let mut stack = Vec::new();

stack.push(1);
stack.push(2);
stack.push(3);

while let Some(top) = stack.pop() {
    println!("{}", top);
}
```

Листинг 18-2: Использование цикла **while let** для печати значений до тех пор, пока **stack.pop()** возвращает **Some**

В этом примере выводится 3, 2, а затем 1. Метод `pop` извлекает последний элемент из вектора и возвращает `Some(value)`. Если вектор пуст, то `pop` возвращает `None`. Цикл `while` продолжает выполнение кода в своём блоке, пока `pop` возвращает `Some`. Когда `pop` возвращает `None`, цикл останавливается. Мы можем использовать `while let` для удаления каждого элемента из стека.

Цикл `for`

В главе 3 мы упоминали, что цикл `for` является самой распространённой конструкцией цикла в коде Rust, но мы ещё не обсуждали шаблон, который использует ключевое слово `for`. В цикле `for`, шаблон - это значение, которое следует непосредственно за ключевым словом `for`, поэтому `for x in y x` является шаблоном.

В листинге 18-3 показано, как использовать шаблон в цикле `for` для деструктурирования или разбиения кортежа как части цикла `for`.

```
let v = vec!['a', 'b', 'c'];

for (index, value) in v.iter().enumerate() {
    println!("{} is at index {}", value, index);
}
```

Листинг 18-3: Использование шаблона в цикле `for` для деструктурирования кортежа

Код в листинге 18-3 выведет следующее:

```
$ cargo run
Compiling patterns v0.1.0 (file:///projects/patterns)
  Finished dev [unoptimized + debuginfo] target(s) in 0.52s
    Running `target/debug/patterns`
a is at index 0
b is at index 1
c is at index 2
```

Мы используем метод `enumerate` чтобы адаптировать итератор для создания значения и индекса этого значения в итераторе, помещённом в кортеж. Первый вызов `enumerate` производит кортеж `(0, 'a')`. Когда это значение сопоставляется с шаблоном `(index, value)`, `index` будет равен `0` а `value` будет равно `'a'` и будет напечатана первая строка выходных данных.

Оператор `let`

До этой главы мы подробно обсуждали только использование шаблонов с `match` и `if let`, но на самом деле, мы использовали шаблоны и в других местах, в том числе в операторах `let`. Например, рассмотрим следующее простое назначение переменной с помощью `let`:

```
let x = 5;
```

На протяжении всей этой книги мы использовали `let` сотни раз, хотя вы, возможно, не поняли, что вы использовали шаблоны! Более формально, выражение `let` выглядит так:

```
let PATTERN = EXPRESSION;
```

В выражениях типа `let x = 5;` с именем переменной в слоте `PATTERN`, имя переменной является просто отдельной, простой формой шаблона. Rust сравнивает выражение с шаблоном и присваивает любые имена, которые он находит. Так что в примере `let x = 5;`, `x` - это шаблон, который означает "привязать то, что соответствует здесь, переменной `x`". Поскольку имя `x` является полностью шаблоном, этот шаблон фактически означает "привязать все к переменной `x` независимо от значения".

Чтобы более чётко увидеть аспект сопоставления с шаблоном `let`, рассмотрим листинг 18-4, в котором используется шаблон с `let` для деструктуризации кортежа.

```
let (x, y, z) = (1, 2, 3);
```

Листинг 18-4. Использование шаблона для деструктуризации кортежа и создания трёх переменных одновременно

Здесь мы сопоставляем кортеж с шаблоном. Rust сравнивает значение `(1, 2, 3)` с шаблоном `(x, y, z)` и видит, что значение соответствует шаблону, поэтому Rust связывает `1` с `x`, `2` с `y` и `3` с `z`. Вы можете думать об этом шаблоне кортежа как о вложении в него трёх отдельных шаблонов переменных.

Если количество элементов в шаблоне не совпадает с количеством элементов в кортеже, то весь тип не будет совпадать и мы получим ошибку компилятора.

Например, в листинге 18-5 показана попытка деструктурировать кортеж с тремя элементами в две переменные, что не будет работать.

```
let (x, y) = (1, 2, 3);
```

Листинг 18-5: Неправильное построение шаблона, переменные не соответствуют количеству элементов в кортеже

Попытка скомпилировать этот код приводит к ошибке:

```
$ cargo run
Compiling patterns v0.1.0 (file:///projects/patterns)
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 tuple `({integer}, {integer}, {integer})`
        found tuple `(_, _)`  

For more information about this error, try `rustc --explain E0308`.
error: could not compile `patterns` due to previous error
```

Если бы мы хотели игнорировать одно или несколько значений в кортеже, мы могли бы использовать `_` или `..`, как вы увидите это в разделе “[Игнорирование значений в Шаблоне](#)”<!-- -->. Если проблема в том, что у нас слишком много переменных в шаблоне, решение состоит в том, чтобы сделать так что типы совпадают, удаляя некоторые переменные и уравнивая число переменных к числу элементов в кортеже.

Параметры функции

Параметры функции также могут быть образцами. Код в листинге 18-6 объявляет функцию с именем `foo`, которая принимает один параметр с именем `x` типа `i32`, к настоящему времени это должно выглядеть знакомым.

```
fn foo(x: i32) {
    // code goes here
}
```

Листинг 18-6: Сигнатура функции использует образцы в параметрах

Х это часть шаблона! Как и в случае с `let`, мы можем сопоставить кортеж в аргументах функции с образцом. Листинг 18-7 разделяет значения в кортеже при его передачи в функцию.

Файл: src/main.rs

```
fn print_coordinates(&(x, y): &(i32, i32)) {
    println!("Current location: ({}, {})", x, y);
}

fn main() {
    let point = (3, 5);
    print_coordinates(&point);
}
```

Листинг 18-7: Функция с параметрами, которая разрушает кортеж

Этот код печатает `текущие координаты: (3, 5)`. Значения `&(3, 5)` соответствуют образцу `&(x, y)`, поэтому `x` - это значение `3`, а `y` - это значение `5`.

Добавляя к вышесказанному, мы можем использовать шаблоны в списках параметров замыкания таким же образом, как и в списках параметров функции, потому что, как обсуждалось в главе 13, замыкания похожи на функции.

На данный момент вы видели несколько способов использования шаблонов, но шаблоны работают не одинаково во всех местах, где их можно использовать. В некоторых местах шаблоны должны быть неопровергнутыми; в других обстоятельствах они могут быть опровергнуты. Мы обсудим эти две концепции далее.

Возможность опровержения: может ли шаблон не совпадать

Шаблоны бывают двух форм: опровергимые и неопровергимые. Шаблоны, которые будут соответствовать любому возможному переданному значению, являются *неопровергимыми* (irrefutable). Примером может быть `x` в выражении `let x = 5;`, потому что `x` соответствует чему-либо и следовательно не может не совпадать. Шаблоны, которые могут не соответствовать некоторому возможному значению, являются *опровергимыми* (refutable). Примером может быть `Some(x)` в выражении `if let Some(x) = a_value`, потому что если значение в переменной `a_value` равно `None`, а не `Some`, то шаблон `Some(x)` не будет совпадать.

Параметры функций, операторы `let` и `for` могут принимать только неопровергимые шаблоны, поскольку программа не может сделать ничего значимого, если значения не совпадают. А выражения `if let` и `while let` принимают опровергимые и неопровергимые шаблоны, но компилятор предостерегает от неопровергимых шаблонов, поскольку по определению они предназначены для обработки возможного сбоя: функциональность условного выражения заключается в его способности выполнять разный код в зависимости от успеха или неудачи.

В общем случае, вам не нужно беспокоиться о разнице между опровергимыми (refutable) и неопровергимыми (irrefutable) образцами; тем не менее, вам необходимо ознакомиться с концепцией возможности опровержения, чтобы вы могли отреагировать на неё, увидев в сообщении об ошибке. В таких случаях вам потребуется изменить либо шаблон, либо конструкцию с которой вы используете шаблон в зависимости от предполагаемого поведения кода.

Давайте посмотрим на пример того, что происходит, когда мы пытаемся использовать опровергимый (refutable) шаблон, где Rust требует неопровергимый шаблон и наоборот. В листинге 18-8 показан оператор `let`, но для образца мы указали `Some(x)` являющийся шаблоном, который можно опровергнуть. Как и следовало ожидать, этот код не будет компилироваться.

```
let Some(x) = some_option_value;
```

Листинг 18-8: Попытка использовать опровергимый шаблон вместе с `let`



Если `some_option_value` было бы значением `None`, то оно не соответствовало бы шаблону `Some(x)`, что означает, что шаблон является опровергимым. Тем не

менее, оператор `let` может принимать только неопровергимый шаблон, потому что нет корректного кода, который может что-то сделать со значением `None`. Во время компиляции Rust будет жаловаться на то, что мы пытались использовать опровергимый шаблон для которого требуется неопровергимый шаблон:

```
$ cargo run
Compiling patterns v0.1.0 (file:///projects/patterns)
error[E0005]: refutable pattern in local binding: `None` not covered
--> src/main.rs:3:9
   |
3 |     let Some(x) = some_option_value;
      ^^^^^^ pattern `None` not covered
   |
   = note: `let` bindings require an "irrefutable pattern", like a `struct`
or an `enum` with only one variant
   = note: for more information, visit https://doc.rust-lang.org/book/ch18-
02-refutability.html
   = note: the matched value is of type `Option<i32>`
help: you might want to use `if let` to ignore the variant that isn't matched
--> src/main.rs:3:9
   |
3 |     if let Some(x) = some_option_value { /* */ }
   |

For more information about this error, try `rustc --explain E0005`.
error: could not compile `patterns` due to previous error
```

Поскольку мы не покрыли (и не могли покрыть!) каждое допустимое значение с помощью образца `Some(x)`, то Rust выдаёт ошибку компиляции.

Чтобы исправить проблему наличия опровергимого шаблона, там где нужен неопровергимый шаблон, можно изменить код использующий шаблон: вместо использования `let`, можно использовать `if let`. Затем, если шаблон не совпадает, будет пропущено выполнение кода внутри фигурных скобок, что даст ему возможность продолжить корректное выполнение. В листинге 18-9 показано, как исправить код из листинга 18-8.

```
if let Some(x) = some_option_value {
    println!("{}", x);
}
```

Листинг 18-9. Использование `if let` и блока с опровергнутыми шаблонами вместо `let`

Код сделан! Этот код совершенно корректный, хотя это означает, что мы не можем использовать неопровергимый образец без получения ошибки. Если мы используем шаблон `if let`, который всегда будет совпадать, то для примера `x`

показанного в листинге 18-10, компилятор выдаст предупреждение.

```
if let x = 5 {  
    println!("{}", x);  
};
```

Листинг 18-10. Попытка использовать неопровергимый шаблон с `if let`

Rust жалуется, что не имеет смысла использовать, `if let` с неопровергимым образцом:

```
$ cargo run  
Compiling patterns v0.1.0 (file:///projects/patterns)  
warning: irrefutable `if let` pattern  
--> src/main.rs:2:8  
|  
2 |     if let x = 5 {  
|     ^^^^^^  
|  
= note: `#[warn(irrefutable_let_patterns)]` on by default  
= note: this pattern will always match, so the `if let` is useless  
= help: consider replacing the `if let` with a `let`  
  
warning: `patterns` (bin "patterns") generated 1 warning  
Finished dev [unoptimized + debuginfo] target(s) in 0.39s  
Running `target/debug/patterns`  
5
```

По этой причине совпадающие рукава должны использовать опровергимые образцы, за исключением последнего, который должен сопоставлять любые оставшиеся значения с неопровергимым образцом. Rust позволяет нам использовать неопровергимый шаблон в `match` только с одним рукавом, но этот синтаксис не особенно полезен и может быть заменён более простым оператором `let`.

Теперь, когда вы знаете, где использовать шаблоны и разницу между опровергимыми и неопровергимыми шаблонами, давайте рассмотрим весь синтаксис, который мы можем использовать для создания шаблонов.

Синтаксис шаблонов

На протяжении всей книги вы видели примеры различных видов шаблонов. В этом разделе мы собираем весь синтаксис, допустимый в шаблонах и обсуждаем, почему вы могли бы использовать каждый из них.

Сопоставление с литералом

Как мы уже видели в главе 6, можно сопоставлять с литералами напрямую. Следующий код с примерами:

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

Этот код печатает `one`, потому что значение в `x` равно 1. Данный синтаксис полезен, когда вы хотите, чтобы ваш код предпринял действие, если он получает конкретное значение.

Сопоставление именованных переменных

Именованные переменные - это неопровергимые (irrefutable) шаблоны, которые соответствуют любому значению и мы использовали их много раз в книге. Однако при использовании именованных переменных в выражениях `match` возникает сложность. Поскольку `match` начинает новую область видимости, то переменные, объявленные как часть шаблона 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 с именем `x` со значением `Some(5)` и переменную `y` со значением `10`. Затем мы создаём выражение `match` для значения `x`. Посмотрите на шаблоны в ветках, `println!` в конце и попытайтесь выяснить, какой код будет напечатан прежде чем запускать его или читать дальше.

Файл: src/main.rs

```

let x = Some(5);
let y = 10;

match x {
    Some(50) => println!("Got 50"),
    Some(y) => println!("Matched, y = {:?}", y),
    _ => println!("Default case, x = {:?}", x),
}

println!("at the end: x = {:?}", x, y);

```

Листинг 18-1: Выражение `match` с веткой, которая объявляет затенённую переменную `y`

Давайте рассмотрим, что происходит, когда выполняется выражение `match`. Шаблон в первой ветке не соответствует определённому значению `x`, поэтому выполнение продолжается.

Шаблон во второй ветке вводит новую переменную с именем `y`, которая будет соответствовать любому значению в `Some`. Поскольку мы находимся в новой области видимости внутри выражения `match`, это новая переменная `y`, а не `y` которую мы объявили в начале со значением 10. Эта новая привязка `y` будет соответствовать любому значению из `Some`, которое находится в `x`. Следовательно, эта новая `y` связывается с внутренним значением `Some` из переменной `x`. Этим значением является `5`, поэтому выражение для этой ветки выполняется и печатает `Matched, y = 5`.

Если бы `x` было значением `None` вместо `Some(5)`, то шаблоны в первых двух ветках не совпали бы, поэтому значение соответствовало бы подчёркиванию. Мы не ввели переменную `x` в шаблоне ветки со знаком подчёркивания, поэтому `x` в выражении все ещё является внешней переменной `x`, которая не была затенена. В этом гипотетическом случае совпадение `match` выведет `Default case, x = None`.

Когда выражение `match` завершается, заканчивается его область видимости как и область действия внутренней переменной `y`. Последний `println!` печатает `at the end: x = Some(5), y = 10`.

Чтобы создать выражение `match`, которое сравнивает значения внешних `x` и `y`, вместо введения затенённой переменной нужно использовать условие в сопоставлении образца. Мы поговорим про условие в сопоставлении шаблона позже в разделе “[Дополнительные условия в сопоставлении образца](#)”.

Группа шаблонов

В выражениях `match` можно сопоставлять несколько шаблонов, используя синтаксис `|`, что означает логическое *или*. Например, следующий код сопоставляет значение `x` с ветками, первая из которых имеет опцию *или*, т.е. если значение `x` соответствует любому из значений в этой ветке, то код этой ветки будет выполняться:

```
let x = 1;

match x {
    1 | 2 => println!("one or two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

Будет напечатано `one or two`.

Сопоставление диапазонов с помощью `..=`

Синтаксис `..=` позволяет нам сопоставлять широкий диапазон значений. В следующем коде, когда шаблон соответствует любому из значений в пределах диапазона, эта ветка будет выполняться:

```
let x = 5;

match x {
    1..=5 => println!("one through five"),
    _ => println!("something else"),
}
```

Если `x` равно 1, 2, 3, 4 или 5, то совпадёт первая ветка. Этот синтаксис более удобен, чем использование оператора `|` для выражения той же идеи; вместо `1..=5` мы должны указать `1 | 2 | 3 | 4 | 5`, если бы использовали логическое `|`. Задание диапазона намного короче, особенно если мы хотим сопоставить, скажем, любое число от 1 до 1000!

Диапазоны допускаются только с числовыми значениями или значениями типа `char`, поскольку компилятор проверяет, что диапазон не является пустым во время компиляции. Единственные типы, для которых Rust может определить, является ли диапазон пустым это `char` и числовые значения.

Вот пример использования диапазонов значений `char`:

```
let x = 'c';

match x {
    'a'..'j' => println!("early ASCII letter"),
    'k'..'z' => println!("late ASCII letter"),
    _ => println!("something else"),
}
```

Rust может сказать, что `c` находится в пределах диапазона первого шаблона и печатает `early ASCII letter`.

Деструктуризация для получения значений

Мы также можем использовать шаблоны для деструктуризации структур, перечислений и кортежей, чтобы использовать разные части этих значений. Давайте пройдёмся по каждому варианту.

Деструктуризация структуры

В листинге 18-12 показана структура `Point` с двумя полями `x` и `y`, которые мы можем разделить, используя шаблон с выражением `let`.

Файл: 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);
}
```

Листинг 18-12: Деструктуризация полей структуры на отдельные переменные

Этот код создаёт переменные `a` и `b`, которые соответствуют значениям полей `x` и `y` структуры `p`. Этот пример показывает, что имена переменных в шаблоне не

обязательно должны совпадать с именами полей структуры. Но обычно желательно, чтобы имена переменных совпадали, чтобы было легче запомнить, какие переменные из каких полей появились.

Поскольку наличие имён переменных, совпадающих с полями, является распространённым явлением и поскольку запись `let Point { x: x, y: y } = p;` содержит много дубликатов, существует сокращение для шаблонов, которые соответствуют структурам полей. Вам нужно только перечислить имена полей структуры и переменные, созданные из шаблона, будут иметь те же имена. В листинге 18-13 показан код, который ведёт себя так же, как и код в листинге 18-12, но переменные, созданные в шаблоне `let` являются `x` и `y` вместо переменных `a` и `b`.

Файл: 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);
}
```

Листинг 18-13: Деструктуризация структурных полей с использованием сокращения полей структуры

Этот код создаёт переменные `x` и `y`, которые соответствуют полям `x` и `y` из переменной `p`. В результате переменные `x` и `y` содержат значения из структуры `p`.

Вместо создания переменных для всех полей мы также можем деструктурировать с помощью литеральных значений являющихся частью структуры. Это позволяет проверить некоторые поля на определённые значения при создании переменных для деструктуризации других полей.

В листинге 18-14 показано выражение `match`, которое разделяет значения `Point` на три случая: точки, которые лежат непосредственно на оси `x` (что верно, когда `y = 0`), на оси `y` (`x = 0`) или ни то, ни другое.

Файл: src/main.rs

```
fn main() {
    let p = Point { x: 0, y: 7 };

    match p {
        Point { x, y: 0 } => println!("On the x axis at {}", x),
        Point { x: 0, y } => println!("On the y axis at {}", y),
        Point { x, y } => println!("On neither axis: ({}, {})".format(x, y)),
    }
}
```

Листинг 18-14: Деструктуризация и сопоставление латеральных значений в одном шаблоне

Первая ветка будет соответствовать любой точке, которая лежит на оси `x`, указанием того что поле `y` совпадает, если его значение соответствует литералу равному `0`. Шаблон все ещё создаёт переменную `x`, которую мы можем использовать в коде для этой ветки.

Точно так же вторая ветка соответствует любой точке на оси `y`, указанием того что поле `x` совпадает, если его значение равно `0` и создаёт переменную `y` для значения поля `y`. Третья ветка не указывает никаких литералов, поэтому он соответствует любой другой точке `Point` и создаёт переменные для обоих полей `x` и `y`.

В этом примере значение `p` совпадает по второй ветке так как `x` содержит значение 0, поэтому этот код будет печатать `On the y axis at 7`.

Деструктуризация перечислений

Ранее мы деструктурировали перечисления в книге, например, когда мы деструктурировали тип `Option<i32>` в листинге 6-5 главы 6. Одну деталь про которую мы не упомянули в явном виде является то, что шаблон для деструктуризации перечисления должен соответствовать способу объявления данных, хранящихся в перечислении. Например, в листинге 18-15 мы используем перечисление `Message` из листинга 6-2 и пишем `match` с шаблонами, которые будут деструктурировать каждое внутреннее значение.

Файл: src/main.rs

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

fn main() {
    let msg = Message::ChangeColor(0, 160, 255);

    match msg {
        Message::Quit => {
            println!("The Quit variant has no data to destructure.")
        }
        Message::Move { x, y } => {
            println!(
                "Move in the x direction {} and in the y direction {}",
                x, y
            );
        }
        Message::Write(text) => println!("Text message: {}", text),
        Message::ChangeColor(r, g, b) => println!(
            "Change the color to red {}, green {}, and blue {}",
            r, g, b
        ),
    }
}
```

Листинг 18-15: Деструктуризация вариантов перечисления содержащих разные виды значений

Этот код напечатает `Change the color to red 0, green 160, and blue 255`. Попробуйте изменить значение переменной `msg`, чтобы увидеть выполнение кода в других ветках.

Для вариантов перечисления без каких-либо данных, таких как `Message::Quit`, мы не можем деструктурировать значение, которого нет. Мы можем сопоставлять только буквальное значение `Message::Quit` и в этом шаблоне нет переменных.

Для вариантов перечисления похожих на структуры, таких как `Message::Move`, можно использовать шаблон, подобный шаблону, который мы указываем для соответствия структурам. После имени варианта мы помещаем фигурные скобки и затем перечисляем поля именами переменных, чтобы разделить фрагменты, которые будут использоваться в коде для этой ветки. Здесь мы используем сокращённую форму, как в листинге 18-13.

Для вариантов перечисления, подобных кортежу, таких как `Message::Write`,

который содержит кортеж с одним элементом и `Message::ChangeColor`, содержащему кортеж с тремя элементами, шаблон аналогичен тому, который мы указываем для соответствия кортежей. Количество переменных в шаблоне должно соответствовать количеству элементов в варианте, который мы сопоставляем.

Деструктуризация вложенных структур и перечислений

До этого момента все наши примеры соответствовали структурам или перечислениям, которые были с одним уровнем вложенности. Соответствие шаблону может работать и со вложенными элементами!

Например, мы можем изменить код в листинге 18-15 для поддержки цветов RGB и HSV в сообщении `ChangeColor`, как показано в листинге 18-16.

```
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!(
            "Change the color to red {}, green {}, and blue {}",
            r, g, b
        ),
        Message::ChangeColor(Color::Hsv(h, s, v)) => println!(
            "Change the color to hue {}, saturation {}, and value {}",
            h, s, v
        ),
        _ => (),
    }
}
```

Листинг 18-16: Сопоставление со вложенными перечислениями

Шаблон первой ветки в выражении `match` соответствует варианту перечисления

`Message::ChangeColor`, который содержит вариант `Color::Rgb`; затем шаблон привязывается к трём внутренними значениями `i32`. Шаблон второй ветки также соответствует варианту перечисления `Message::ChangeColor`, но внутреннее перечисление соответствует варианту `Color::Hsv`. Мы можем указать эти сложные условия в одном выражении `match`, даже если задействованы два перечисления.

Деструктуризация структур и кортежей

Можно смешивать, сопоставлять и вкладывать шаблоны деструктуризации ещё более сложными способами. В следующем примере показана сложная структура, где мы вкладываем структуры и кортежи внутри кортежа и деструктуризуем все примитивные значения:

```
let ((feet, inches), Point { x, y }) = ((3, 10), Point { x: 3, y: -10 });
```

Этот код позволяет нам разбивать сложные типы на составные части, чтобы мы могли использовать интересующие нас значения по отдельности.

Деструктуризация с помощью шаблонов - это удобный способ использования фрагментов значений, таких как значение из каждого поля в структуре по отдельности друг от друга.

Игнорирование значений в шаблоне

Вы видели, что иногда полезно игнорировать значения в шаблоне как в последней ветке `match`, чтобы получить ветку обрабатывающую любые значение, которая на самом деле ничего не делает, но учитывает все оставшиеся возможные значения. Есть несколько способов игнорировать целые значения или части значений в шаблоне: используя шаблон `_` (который вы видели), используя шаблон `_` в другом шаблоне и используя имя, начинающееся с подчёркивания, или используя `..`, чтобы игнорировать оставшиеся части значения. Давайте рассмотрим, как и почему использовать каждый из этих шаблонов.

Игнорирование всего значения с помощью шаблона `_`

Мы использовали подчёркивание (`_`) в качестве шаблона подстановочного знака, который будет соответствовать любому значению, но не будет привязываться к значению. Хотя шаблон подчёркивания `_` особенно полезен в качестве последнего

элемента в выражении `match`, мы можем использовать его в любом шаблоне, включая параметры функции как показано в листинге 18-17.

Файл: `src/main.rs`

```
fn foo(_: i32, y: i32) {
    println!("This code only uses the y parameter: {}", y);
}

fn main() {
    foo(3, 4);
}
```

Листинг 18-15: Использование `_` в сигнатуре функции

Этот код полностью игнорирует значение, переданное в качестве первого аргумента, `3` и выведет `This code only uses the y parameter: 4`.

В большинстве случаев, когда вам не нужен конкретный параметр функции вы бы изменили сигнатуру так, чтобы она не включала неиспользуемый параметр. Игнорирование параметра функции может быть особенно полезно в некоторых случаях, например, при реализации типажа с нужной объявленной сигнатурой, но тело функции в вашей реализации не нуждается в одном из параметров. В таком случае компилятор не будет предупреждать о неиспользуемых параметрах функции, как это было бы при использовании имени параметра.

Игнорирование частей значения с помощью вложенного `if`

Также можно использовать `_` внутри другого шаблона, чтобы игнорировать только часть значения, например, когда мы хотим проверить только часть значения, но не используем другие части в соответствующем коде, который мы хотим выполнить. В листинге 18-18 показан код, отвечающий за управление значением настройки. Бизнес-требования заключаются в том, что пользователь не должен иметь права перезаписывать существующую настройку параметра, но может сбросить параметр и присвоить ему значение, если он в данный момент не установлен.

```

let mut setting_value = Some(5);
let new_setting_value = Some(10);

match (setting_value, new_setting_value) {
    (Some(_), Some(_)) => {
        println!("Can't overwrite an existing customized value");
    }
    _ => {
        setting_value = new_setting_value;
    }
}

println!("setting is {:?}", setting_value);

```

Листинг 18-18: Использование подчёркивания в шаблонах, соответствующих вариантам `Some`, когда нам не нужно использовать значение внутри `Some`

Этот код будет печатать `Can't overwrite an existing customized value` и затем `setting is Some(5)`. В первой ветке нам не нужно сопоставлять или использовать значения внутри варианта `Some`, но нам нужно проверить случай, когда `setting_value` и `new_setting_value` являются вариантом `Some`. В этом случае мы печатаем причину, почему мы не меняем значение `setting_value` и оно не меняется.

Во всех других случаях (если или `setting_value` или `new_setting_value` являются вариантом `None`), выраженные шаблоном `_` во второй ветке, то мы хотим, чтобы `new_setting_value` стало `setting_value`.

Мы также можем использовать подчёркивание в нескольких местах в одном шаблоне, чтобы игнорировать конкретные значения. Листинг 18-19 показывает пример игнорирования второго и четвёртого значения в кортеже из пяти элементов.

```

let numbers = (2, 4, 8, 16, 32);

match numbers {
    (first, _, third, _, fifth) => {
        println!("Some numbers: {}, {}, {}", first, third, fifth)
    }
}

```

Листинг 18-19: Игнорирование нескольких частей кортежа

Этот код напечатает `Some numbers: 2, 8, 32` и значения 4 и 16 будут

игнорироваться.

Игнорирование неиспользуемой переменной, начинающейся с символа в имени

Если вы создаёте переменную, но нигде её не используете, Rust обычно выдаёт предупреждение, потому что это может быть ошибкой. Но иногда полезно создать переменную, которую вы пока не будете использовать, например, когда вы создаёте прототип или просто запускаете проект. В этой ситуации вы можете сказать Rust не предупреждать вас о неиспользуемой переменной, начав имя переменной с подчёркивания. В листинге 18-20 мы создаём две неиспользуемые переменные, но когда мы запускаем такой код, мы должны получить предупреждение только об одной из них.

Файл: src/main.rs

```
fn main() {
    let _x = 5;
    let y = 10;
}
```

Листинг 18-20: Начинаем имя переменной с подчёркивания, чтобы избежать получение предупреждения о неиспользованных переменных

Здесь мы получаем предупреждение о том, что не используем переменную  , но мы не получаем предупреждения о не использовании переменной, которой предшествует подчёркивание.

Обратите внимание, что есть небольшая разница между использованием только  и использованием имени, начинающегося с подчёркивания. Синтаксис  по-прежнему привязывает значение к переменной, тогда как  совсем не привязывает. Чтобы показать случай где это различие имеет значение,смотрите листинг 18-21 представляющий ошибку.

```
let s = Some(String::from("Hello!"));

if let Some(_s) = s {
    println!("found a string");
}

println!("{:?}", s);
```



Листинг 18-21: Неиспользуемая переменная, начинающаяся с подчёркивания по-прежнему привязывает значение, которое может забирать значение во владение

Мы получим сообщение об ошибке, поскольку значение `s` по-прежнему будет перемещено в переменную `_s`, что не позволяет нам снова использовать `s`. Однако использование только подчёркивания никогда не привязывает к себе значение. Листинг 18-22 будет компилироваться без каких-либо ошибок, потому что `s` не перемещено в `_s`.

```
let s = Some(String::from("Hello!"));

if let Some(_) = s {
    println!("found a string");
}

println!("{}:?", s);
```

Листинг 18-22. Использование подчёркивания не привязывает значение

Этот код работает нормально, потому что мы никогда не привязываем `s` к чему либо; оно не перемещается.

Игнорирование оставшихся частей значения с помощью `..`

С значениями, которые имеют много частей, можно использовать синтаксис `..`, чтобы использовать только некоторые части и игнорировать остальные, избегая необходимости перечислять подчёркивания для каждого игнорируемого значения. Шаблон `..` игнорирует любые части значения, которые мы явно не сопоставили в остальной частью шаблона. В листинге 18-23 мы имеем структуру `Point`, которая содержит координату в трёхмерном пространстве. В выражении `match` мы хотим работать только с координатой `x` и игнорировать значения полей `y` и `z`.

```
struct Point {
    x: i32,
    y: i32,
    z: i32,
}

let origin = Point { x: 0, y: 0, z: 0 };

match origin {
    Point { x, .. } => println!("x is {}", x),
}
```

Листинг 18-21: Игнорирование полей структуры `Point` кроме поля `x` с помощью `..`

Мы перечисляем значение `x` и затем просто включаем шаблон `..`. Это быстрее, чем перечислять `y: _` и `z: _`, особенно когда мы работаем со структурами, которые имеют много полей в ситуациях, когда только одно или два поля актуальны.

Синтаксис `..` раскроется до необходимого количества значений. В листинге 18-24 показано, как использовать `..` с кортежем.

Файл: src/main.rs

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (first, .., last) => {
            println!("Some numbers: {}, {}", first, last);
        }
    }
}
```

Листинг 18-24: Сопоставление только первого и последнего значений в кортеже и игнорирование всех других значений

В этом коде первое и последнее значение соответствуют `first` и `last`. Конструкция `..` будет соответствовать и игнорировать все посередине.

Однако использование `..` должно быть однозначным. Если неясно, какие значения предназначены для сопоставления, а какие следует игнорировать, то Rust выдаст ошибку. В листинге 18-25 показан пример неоднозначного использования `..`, поэтому он не будет компилироваться.

Файл: src/main.rs

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (.., second, ..) => {
            println!("Some numbers: {}", second)
        },
    }
}
```



Листинг 18-25: Попытка использовать `..` неоднозначным способом

При компиляции примера, мы получаем эту ошибку:

```
$ cargo run
Compiling patterns v0.1.0 (file:///projects/patterns)
error: `..` can only be used once per tuple pattern
--> src/main.rs:5:22
|
5 |     (.., second, ..) => {
|         --          ^^^ can only be used once per tuple pattern
|         |
|         previously used here
|
error: could not compile `patterns` due to previous error
```

Rust не может определить, сколько значений в кортеже нужно игнорировать, прежде чем сопоставить значение с `second`, а затем сколько других значений проигнорировать после. Этот код может означать, что мы хотим игнорировать `2`, связать `second` с `4`, а затем игнорировать `8`, `16` и `32`; или что мы хотим игнорировать `2` и `4`, связать `second` с `8`, а затем игнорировать `16` и `32`; и так далее. Имя переменной `second` не означает ничего особенного для Rust, поэтому мы получаем ошибку компилятора, потому что использование `..` в двух местах подобных этому, является неоднозначным.

Дополнительные условия оператора сопоставления (Match Guards)

Конструкция *match guard* является дополнительным условием `if`, указанным после шаблона в ветке `match`, которое также должно выполняться наряду с сопоставлением образца, чтобы ветка была выбрана. Условия сопоставления полезны для выражения более сложных идей, чем позволяет только шаблон.

Условие может использовать переменные, созданные в шаблоне. В листинге 18-26 показан `match`, в котором первая ветка имеет шаблон `Some(x)`, а также имеет условие сопоставления, `if x % 2 == 0` (которое будет истинным, если число чётное).

```
let num = Some(4);

match num {
    Some(x) if x % 2 == 0 => println!("The number {} is even", x),
    Some(x) => println!("The number {} is odd", x),
    None => (),
}
```

Листинг 18-26: Добавление условия сопоставления в шаблон

В этом примере будет напечатано `The number 4 is even`. Когда `num` сравнивается с шаблоном в первой ветке, он совпадает, потому что `Some(4)` соответствует `Some(x)`. Затем условие сопоставления проверяет, равен ли 0 остаток от деления `x` на 2 и если это так, то выбирается первая ветка.

Если бы `num` вместо этого было `Some(5)`, условие в сопоставлении первой ветки было бы ложным, потому что остаток от 5 делённый на 2, равен 1, что не равно 0. Rust тогда перешёл бы ко второй ветке, которое совпадает, потому что вторая ветка не имеет условия сопоставления и, следовательно, соответствует любому варианту `Some`.

Невозможно выразить условие `if x % 2 == 0` внутри шаблона, поэтому условие в сопоставлении даёт нам возможность выразить эту логику. Недостатком этой дополнительной выразительности является то, что компилятор не пытается проверять полноту, когда задействованы выражения с условием в сопоставлении.

В листинге 18-11 мы упомянули, что можно использовать условия в сопоставлении для решения проблемы затенения в шаблоне. Напомним, что внутри шаблона в выражении `match` была создана новая переменная вместо использования внешней к `match` переменной. Эта новая переменная означала, что мы не могли выполнить сравнение помошью значения внешней переменной. В листинге 18-27 показано, как мы можем использовать условие сопоставления для решения этой проблемы.

Файл: src/main.rs

```
fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        Some(50) => println!("Got 50"),
        Some(n) if n == y => println!("Matched, n = {}", n),
        _ => println!("Default case, x = {:?}", x),
    }

    println!("at the end: x = {:?}", x, y);
}
```

Листинг 18-27. Использование условия сопоставления для проверки на равенство со значением внешней переменной

Этот код теперь напечатает `Default case, x = Some(5)`. Шаблон во второй ветке не вводит новую переменную `y`, которая будет затенять внешнюю `y`, это означает, что теперь можно использовать внешнюю переменную `y` в условии сопоставления. Вместо указания шаблона как `Some(y)`, который бы затенял бы внешнюю `y`, мы указываем `Some(n)`. Это создаёт новую переменную `n`, которая ничего не затеняет, потому нет переменной `n` вне конструкции `match`.

Условие сопоставления `if n == y` не является шаблоном и следовательно, не вводит новые переменные. Переменная `y` является внешней `y`, а не новой затенённой `y` и мы можем искать значение, которое имеет то же значение, что и внешняя `y`, сравнивая значение `n` со значением `y`.

Вы также можете использовать оператор или `|` в условии сопоставления, чтобы указать несколько шаблонов; условие сопоставления будет применяться ко всем шаблонам. В листинге 18-28 показан приоритет комбинирования условия сопоставления с шаблоном, который использует `|`. Важной частью этого примера является то, что условие сопоставления `if y` применяется к `4`, `5`, и к `6`, хотя это может выглядеть как будто `if y` относится только к `6`.

```
let x = 4;
let y = false;

match x {
    4 | 5 | 6 if y => println!("yes"),
    _ => println!("no"),
}
```

Листинг 18-28: Комбинирование нескольких шаблонов с условием сопоставления

Условие сопоставления гласит, что ветка совпадает, только если значение `x` равно `4`, `5` или `6`, и если `y` равно `true`. Когда этот код выполняется, шаблон первой ветки совпадает, потому что `x` равно `4`, но условие сопоставления `if y` равно `false`, поэтому первая ветка не выбрана. Код переходит ко второй ветке, которая совпадает и эта программа печатает `no`. Причина в том, что условие `if` применяется ко всему шаблону `4 | 5 | 6`, а не только к последнему значению `6`. Другими словами, приоритет условия сопоставления по отношению к шаблону ведёт себя так:

```
(4 | 5 | 6) if y => ...
```

а не так:

```
4 | 5 | (6 if y) => ...
```

После запуска кода, старшинство в поведении становится очевидным: если условие сопоставления применялось бы только к конечному значению в списке, указанном с помощью оператора `|`, то ветка бы совпала и программа напечатала бы `yes`.

Связывание @

Оператор `at (@)` позволяет создать переменную, которая содержит значение, которое мы одновременно сравниваем со значением, соответствует ли оно шаблону. В листинге 18-29 показан пример, в котором мы хотим проверить, что перечисление `Message::Hello` со значением поля `id` находится в диапазоне `3..=7`. Но мы также хотим привязать такое значение к переменной `id_variable`, чтобы использовать его внутри кода данной ветки. Мы могли бы назвать эту переменную `id`, так же как поле, но для этого примера мы будем использовать другое имя.

```
enum Message {
    Hello { id: i32 },
}

let msg = Message::Hello { id: 5 };

match msg {
    Message::Hello {
        id: id_variable @ 3..=7,
    } => println!("Found an id in range: {}", id_variable),
    Message::Hello { id: 10..=12 } => {
        println!("Found an id in another range")
    }
    Message::Hello { id } => println!("Found some other id: {}", id),
}
}
```

Листинг 18-29: Использование `@` для привязывания значения в шаблоне одновременно делая его проверку

В этом примере будет напечатано `Found an id in range: 5`. Указывая `id_variable @` перед диапазоном `3..=7`, мы захватываем любое значение, попадающее в диапазон, одновременно проверяя, что это значение соответствует диапазону в шаблоне.

Во второй ветке, где указан только диапазон в шаблоне, код данной ветки не имеет переменной, которая содержит фактическое значение поля `id`. Значение поля `id` могло бы быть 10, 11 или 12, но код, соответствующий этому шаблону не знает, чему оно равно. Код шаблона не может использовать значение из поля `id`, потому что мы не сохранили значение `id` в переменной.

В последней ветке, где мы указали переменную без диапазона, у нас есть значение доступное для использования в коде ветки в переменной с именем `id`. Причина в том, что мы использовали синтаксис сокращённых полей структуры. Но мы не применяли никакого сравнения со значением в поле `id` в этой ветке, как мы это делали в первых двух ветках: любое значение будет соответствовать этому шаблону.

Использование `@` позволяет проверять значение и сохранять его в переменной в пределах одного шаблона.

Итоги

Шаблоны Rust очень полезны тем, что помогают различать разные виды данных. При использовании выражений `match` Rust гарантирует, что ваши шаблоны охватывают все возможные значения, потому что иначе ваша программа не будет компилироваться. Шаблоны в операторах `let` и параметрах функций делают такие конструкции более полезными, позволяя разбивать значения на более мелкие части одновременно назначая значения переменным. Мы можем создавать простые или сложные шаблоны в соответствии с нашими потребностями.

Далее, в предпоследней главе книги, мы рассмотрим некоторые продвинутые аспекты различных возможностей Rust.

Расширенные возможности

К настоящему времени вы изучили наиболее часто используемые части языка программирования Rust. Прежде чем мы сделаем ещё один проект в главе 20, мы рассмотрим несколько аспектов языка с которыми вы можете сталкиваться время от времени. Можете использовать эту главу в качестве справки, когда столкнётесь с неизвестными возможностями Rust. Возможности, которые вы научитесь использовать в этой главе, полезны в специальных ситуациях. Хотя возможно вы не часто их встретите, мы хотим быть уверены, что у вас есть понимание всех возможностей, которые может предложить Rust.

В этой главе мы рассмотрим:

- Небезопасный Rust: как отказаться от некоторых гарантий Rust и взять на себя ответственность за их ручное соблюдение
- Продвинутые типажи: ассоциированные типы, параметры типа по умолчанию, полностью квалифицированный синтаксис, супер-типажи и шаблон создания (newtype) по отношению к типажам
- Расширенные типы: больше о шаблоне newtype, псевдонимах типа, тип never и типы динамических размеров
- Расширенные функции и замыкания: указатели функций и возврат замыканий
- Макросы: способы определения кода, который определяет большую часть кода во время компиляции

Это набор возможностей Rust для всех! Давайте погрузимся в него!

Unsafe Rust

Во всех предыдущих главах этой книги мы обсуждали код на Rust, безопасность памяти в котором гарантируется во время компиляции. Однако внутри Rust скрывается другой язык - небезопасный Rust, который не обеспечивает безопасной работы с памятью. Этот язык называется *unsafe Rust* и работает также как и первый, но предоставляет вам дополнительные возможности.

Небезопасный Rust существует, потому что по своей природе статический анализ является консервативным. Когда компилятор пытается определить, поддерживает ли код некоторые гарантии или нет, то лучше отклонить некоторые действительные программы, которые корректны, чем принимать некоторые программы, которые ошибочны. Бывают случаи, когда ваш код может быть правильным, но Rust считает, что это не так. В этих случаях вы можете использовать небезопасный код, чтобы сообщить компилятору: «поверь мне, я знаю, что делаю». Недостатком является то, что вы используете его на свой страх и риск. Если вы используете небезопасный код неправильно, то могут появиться проблемы из-за небезопасной работы с памятью, такие как разыменование нулевого указателя.

Другая причина, по которой у Rust есть небезопасное альтер эго, заключается в том, что по существу аппаратное обеспечение компьютера небезопасно. Если Rust не позволял бы вам выполнять небезопасные операции, вы не могли бы выполнять определённые задачи. Rust должен позволить вам использовать системное, низкоуровневое программирование, такое как прямое взаимодействие с операционной системой, или даже написание вашей собственной операционной системы. Возможность написания низкоуровневого, системного кода является одной из целей языка. Давайте рассмотрим, что и как можно делать с небезопасным Rust.

Небезопасные сверхспособности

Чтобы переключиться на небезопасный Rust, используйте ключевое слово `unsafe` и начните новый блок, содержащий небезопасный код. Вы можете совершить пять действий в небезопасном Rust коде, которые называются *небезопасными сверхспособностями* и которые вы не можете выполнить в безопасном Rust. Эти сверхспособности включают в себя следующие возможности:

- Разыменование сырого указателя

- Вызов небезопасной функции или небезопасного метода
- Доступ или изменение изменяемой статической переменной
- Реализация небезопасного типажа
- Доступ к полям в `union`

Важно понимать, что `unsafe` не отключает проверку заимствования или любые другие проверки безопасности Rust: если вы используете ссылку в небезопасном коде, она всё равно будет проверена. Единственное, что делает ключевое слово `unsafe` - даёт вам доступ к этим четырём возможностям, безопасность работы с памятью в которых не проверяет компилятор. Вы по-прежнему получаете некоторую степень безопасности внутри небезопасного блока.

Кроме того, `unsafe` не означает, что код внутри этого блока является неизбежно опасным или он точно будет иметь проблемы с безопасностью памяти: цель состоит в том, что вы, как программист, гарантируете, что код внутри блока `unsafe` будет обращаться к действительной памяти корректным образом.

Люди подвержены ошибкам и ошибки будут происходить, но требуя размещение этих четырёх небезопасных операций внутри блоков, помеченных как `unsafe`, вы будете знать, что любые ошибки, связанные с безопасностью памяти, будут находиться внутри `unsafe` блоков. Делайте `unsafe` блоки маленькими; вы будете благодарны себе за это позже, при исследовании ошибок с памятью.

Чтобы максимально изолировать небезопасный код, рекомендуется заключить небезопасный код в безопасную абстракцию и предоставить безопасный API, который мы обсудим позже, когда будем обсуждать небезопасные функции и методы. Части стандартной библиотеки реализованы как проверенные, безопасные абстракции над небезопасным кодом. Оборачивание небезопасного кода в безопасную абстракцию предотвращает возможную утечку использования `unsafe` кода во всех местах, где вы или ваши пользователи могли бы захотеть напрямую использовать функциональность, реализованную `unsafe` кодом, потому что использование безопасной абстракции само безопасно.

Давайте поговорим о каждой из четырёх небезопасных сверх способностей, и по ходу дела рассмотрим некоторые абстракции, которые обеспечивают безопасный интерфейс для небезопасного кода.

Разыменование сырых указателей

В главе 4 раздела "Недействительные ссылки" мы упоминали, что компилятор

гарантирует, что ссылки всегда действительны. Небезопасный Rust имеет два новых типа, называемых *сырыми указателями* (raw pointers), которые похожи на ссылки. Как и в случае ссылок, сырье указатели могут быть неизменяемыми или изменяемыми и записываться как `*const T` и `*mut T` соответственно. Звёздочка не является оператором разыменования; это часть имени типа. В контексте сырых указателей *неизменяемый* (immutable) означает, что указателю нельзя напрямую присвоить что-то после того как он разыменован.

В отличие от ссылок и умных указателей, сырье указатели:

- могут игнорировать правила заимствования и иметь неизменяемые и изменяемые указатели, или множество изменяемых указателей на одну и ту же область памяти
- не гарантируют что ссылаются на действительную память
- могут быть `null`
- не реализуют автоматическую очистку памяти

Отказавшись от этих гарантий, вы можете обменять безопасность на большую производительность или возможность взаимодействия с другим языком или оборудованием, где гарантии Rust не применяются.

В листинге 19-1 показано, как создать неизменяемый и изменяемый сырой указатель из ссылок.

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;
```

Листинг 19-1. Создание необработанных указателей из ссылок

Обратите внимание, что мы не используем ключевое слово `unsafe` в этом коде. Можно создавать сырье указатели в безопасном коде; мы просто не можем разыменовывать сырье указатели за пределами небезопасного блока, как вы увидите чуть позже.

Мы создали сырье указатели, используя `as` для приведения неизменяемой и изменяемой ссылки к соответствующим им типам сырых указателей. Поскольку мы создали их непосредственно из ссылок, которые гарантированно являются действительными, мы знаем, что эти конкретные сырье указатели являются действительными, но мы не можем делать такое же предположение о любом сырому указателе.

Далее мы создадим сырой указатель, в достоверности которого мы не можем быть уверены. В листинге 19-2 показано, как создать сырой указатель на произвольный адрес памяти. Результат попытки использовать произвольную память не определён (undefined): по этому адресу могут быть данные или их может не быть, компилятор может оптимизировать код так, что не будет кода доступа к памяти или программа может выдать ошибку сегментации при выполнении. Обычно нет веских причин для написания такого кода, но это возможно.

```
let address = 0x012345usize;
let r = address as *const i32;
```

Листинг 19-2: Создание сырого указателя на произвольный адрес памяти

Напомним, что можно создавать сырые указатели в безопасном коде, но нельзя разыменовывать сырые указатели и читать данные, на которые они указывают. В листинге 19-3 мы используем оператор разыменования `*` для сырого указателя, который требует `unsafe` блока.

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

Листинг 19-3: Разыменование сырых указателей внутри `unsafe` блока

Создание указателей безопасно. Только при попытке доступа к объекту по адресу в указателе мы можем получить недопустимое значение.

Также обратите внимание, что в примерах кода 19-1 и 19-3 мы создали `*const i32` и `*mut i32`, которые ссылаются на одну и ту же область памяти, где хранится `num`. Если мы попытаемся создать неизменяемую и изменяемую ссылку на `num` вместо сырых указателей, такой код не скомпилируется, т.к. будут нарушены правила заимствования, запрещающие наличие изменяемой ссылки одновременно с неизменяемыми ссылками. С помощью сырых указателей мы можем создать изменяемый указатель и неизменяемый указатель на одну и ту же область памяти и изменять данные с помощью изменяемого указателя, потенциально создавая эффект гонки данных. Будьте осторожны!

С учётом всех этих опасностей, зачем тогда использовать сырые указатели? Одним из основных применений является взаимодействие с кодом C, как вы увидите в следующем разделе "Вызов небезопасной функции или метода". Другой случай это создание безопасных абстракций, которые не понимает анализатор заимствований. Мы введём понятие небезопасных функций и затем рассмотрим пример безопасной абстракции, которая использует небезопасный код.

Вызов небезопасной функции или метода

Второй тип операции, который требует небезопасного блока - это вызов небезопасных функций. Небезопасные функции и методы выглядят точно так же, как обычные функции и методы, но они имеют дополнительное указание `unsafe` перед остальной частью определения. Ключевое слово `unsafe` в этом контексте указывает, что у функции есть требования, которые мы должны соблюдать при её вызове, потому что Rust не может гарантировать выполнение этих требований. Вызывая небезопасную функцию в `unsafe` блоке, мы говорим, что прочитали документацию по этой функции и несём ответственность за соблюдение её контрактов.

Вот небезопасная функция с именем `dangerous`, которая ничего не делает в своём теле:

```
unsafe fn dangerous() {}

unsafe {
    dangerous();
}
```

Мы должны вызвать функцию `dangerous` в отдельном `unsafe` блоке. Если мы попробуем вызвать `dangerous` без `unsafe` блока, мы получим ошибку:

```
$ cargo run
Compiling unsafe-example v0.1.0 (file:///projects/unsafe-example)
error[E0133]: call to unsafe function is unsafe and requires unsafe function
or block
--> src/main.rs:4:5
|
4 |     dangerous();
|     ^^^^^^^^^^ call to unsafe function
|
= note: consult the function's documentation for information on how to
avoid undefined behavior

For more information about this error, try `rustc --explain E0133`.
error: could not compile `unsafe-example` due to previous error
```

Вставив `unsafe` блок вокруг нашего вызова `dangerous`, мы утверждаем, что мы прочитали документацию к функции, мы понимаем как её использовать правильно и мы убедились, что выполняем контракт функции.

Тела небезопасных функций являются фактически `unsafe` блоками, поэтому для выполнения других небезопасных операций внутри небезопасной функции не нужно добавлять ещё один `unsafe` блок.

Создание безопасных абстракций вокруг небезопасного кода

Тот факт, что функция содержит небезопасный код, не означает, что мы должны пометить всю функцию как небезопасную. Фактически, упаковка небезопасного кода в безопасную функцию является обычной абстракцией. В качестве примера давайте изучим функцию из стандартной библиотеки `split_at_mut`, для которой требуется небезопасный код и исследуем, как мы могли бы её реализовать. Этот безопасный метод определён для изменяемых срезов: он берёт один срез и делит его на два, разделяя срез по индексу, указанному в качестве аргумента. В листинге 19.4 показано, как использовать `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]);
```

Листинг 19-4: Использование безопасной функции `split_at_mut`

Эту функцию нельзя реализовать, используя только безопасный Rust. Попытка реализации могла бы выглядеть примерно как в листинге 19-5, который не компилируется. Для простоты мы реализуем `split_at_mut` как функцию, а не как метод, и только для значений типа `i32`, а не обобщённого типа `T`.

```
fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = values.len();
    assert!(mid <= len);
    (&mut values[..mid], &mut values[mid..])
}
```



Листинг 19-5: Попытка реализации функции `split_at_mut` используя только безопасный Rust

Эта функция сначала получает общую длину среза. Затем она проверяет(`assert`), что индекс, переданный в качестве параметра, находится в границах среза, сравнивая его с длиной. `Assert` означает, что если мы передадим индекс, который больше, чем длина среза, функция запаникует ещё до попытки использования этого индекса.

Затем мы возвращаем два изменяемых фрагмента в кортеже: один от начала исходного фрагмента до **mid** индекса (не включая сам **mid**), а другой - от **mid** (включая сам **mid**) до конца фрагмента.

При попытке скомпилировать код в листинге 19-5, мы получим ошибку.

```
$ cargo run
   Compiling unsafe-example v0.1.0 (file:///projects/unsafe-example)
error[E0499]: cannot borrow `*values` as mutable more than once at a time
--> src/main.rs:6:31
1 | fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut
[1 | [i32]) {
| |           - let's call the lifetime of this reference ''1
| ...
6 |     (&mut values[..mid], &mut values[mid..])
-----^~~~~~^-----  
| | |  
| | |           second mutable borrow occurs here  
| | |           first mutable borrow occurs here  
| | returning this value requires that `*values` is borrowed for ''1'
```

```
For more information about this error, try `rustc --explain E0499`.  
error: could not compile `unsafe-example` due to previous error
```

Анализатор заимствований Rust не может понять, что мы заимствуем различные

части среза, он понимает лишь, что мы хотим осуществить заимствование частей одного среза дважды. Заимствование различных частей среза в принципе нормально, потому что они не перекрываются, но Rust недостаточно умён, чтобы это понять. Когда мы знаем, что код верный, но Rust этого не понимает, значит пришло время прибегнуть к небезопасному коду.

Листинг 19-6 демонстрирует, как можно использовать `unsafe` блок, сырой указатель и вызовы небезопасных функций чтобы `split_at_mut` заработала:

```
use std::slice;

fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = values.len();
    let ptr = values.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (
            slice::from_raw_parts_mut(ptr, mid),
            slice::from_raw_parts_mut(ptr.add(mid), len - mid),
        )
    }
}
```

Листинг 19-6. Использование небезопасного кода в реализации функции `split_at_mut`

Напомним, из раздела "Тип срез" главы 4, что срезы состоят из указателя на некоторые данные и длины. Мы используем метод `len` для получения длины среза и метод `as_mut_ptr` для доступа к сырому указателю среза. Поскольку у нас есть изменяемый срез на значения типа `i32`, функция `as_mut_ptr` возвращает сырой указатель типа `*mut i32`, который мы сохранили в переменной `ptr`.

Далее проверяем, что индекс `mid` находится в границах среза. Затем мы обращаемся к небезопасному коду: функция `slice::from_raw_parts_mut` принимает сырой указатель, длину и создаёт срез. Мы используем эту функцию для создания среза, начинающегося с `ptr` и имеющего длину в `mid` элементов. Затем мы вызываем метод `add` у `ptr` с `mid` в качестве аргумента, чтобы получить сырой указатель, который начинается с `mid`, и создаём срез, используя этот указатель и оставшееся количество элементов после `mid` в качестве длины.

Функция `slice::from_raw_parts_mut` небезопасна, потому что она принимает сырой указатель и должна верить, что этот указатель действителен. Метод `offset`

для сырых указателях также небезопасен, поскольку он должен доверять, что местоположение после смещения также является допустимым указателем. Поэтому нам пришлось поместить `unsafe` блок вокруг вызовов `slice::from_raw_parts_mut` и `offset`, чтобы мы могли их вызвать. Посмотрев на код и добавив проверку, что `mid` должно быть меньше или равно `len`, мы можем быть уверены, что все сырые указатели, используемые в `unsafe` блоке будут действительными указателями на данные внутри среза. Это приемлемое и правильное использование `unsafe`.

Обратите внимание, что нам не нужно помечать результирующую функцию `split_at_mut` как `unsafe`, и мы можем вызвать эту функцию из безопасного Rust. Мы создали безопасную абстракцию для небезопасного кода с помощью реализации функции, которая использует код `unsafe` блока безопасным образом, поскольку она создаёт только допустимые указатели из данных, к которым эта функция имеет доступ.

Напротив, использование `slice::from_raw_parts_mut` в листинге 19-7 приведёт к вероятному сбою при использовании среза. Этот код использует произвольный адрес памяти и создаёт срез из 10000 элементов.

```
use std::slice;

let address = 0x01234usize;
let r = address as *mut i32;

let values: &[i32] = unsafe { slice::from_raw_parts_mut(r, 10000) };
```

Листинг 19-7: Создание среза из произвольного адреса памяти

Мы не владеем памятью в этом произвольном месте, и нет никаких гарантий что срез, создаваемый этим кодом, содержит допустимые значения `i32`. Попытка использовать переменную `slice` как будто это допустимый срез приводит к неопределенному поведению (UB - undefined behavior).

Использование `extern` функций для вызова внешнего кода

Иногда в вашем Rust коде может появиться необходимость взаимодействия с кодом, написанным на другом языке программирования. Для этой цели существует специальное ключевое слово `extern`, которое облегчает создание и использование интерфейса внешних функций (FFI - Foreign Function Interface). FFI в языке программирования является способом определять функции и давать возможность другому (внешнему) языку программирования вызывать эти функции.

Листинг 19-8 демонстрирует, как настроить интеграцию с функцией `abs` из стандартной библиотеки С. Функции, объявленные внутри блоков `extern`, всегда небезопасны для вызова из кода Rust. Причина в том, что другие языки не обеспечивают соблюдение правил и гарантий Rust, Rust также не может проверить гарантии, поэтому ответственность за безопасность ложится на программиста.

Файл : src/main.rs

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }
}
```

Листинг 19-8: Объявление и вызов `extern` функции, написанной на другом языке программирования

Внутри блока `extern "C"` мы перечисляем имена и сигнатуры внешних функций из другого языка, которые мы хотим вызвать. Часть `"C"` определяет какой *application binary interface* (ABI - бинарный интерфейс приложений) использует внешняя функция. Интерфейс ABI определяет как вызвать функцию на уровне ассемблера. Использование ABI `"C"` является наиболее часто используемым и следует правилам ABI интерфейса языка Си.

Вызов функций Rust из других языков

Также можно использовать `extern` для создания интерфейса, который позволяет другим языкам вызывать функции Rust. Вместо `extern` блока мы добавляем ключевое слово `extern` и указываем ABI для использования непосредственно перед ключевым словом `fn`. Также нужно добавить аннотацию `#[no_mangle]`, чтобы компилятор Rust не изменял название этой функции. *Искажение (Mangling)* - это когда компилятор изменяет имя нашей функции другим именем, которое содержит больше информации для использования другими этапами процесса компиляции, но такие имена являются менее читабельными. Каждый компилятор языка программирования изменяет имена по-своему, поэтому чтобы функция Rust могла быть доступна

из других языков, мы должны отключить искажение имён компилятором Rust.

В следующем примере мы делаем Rust функцию `call_from_c` доступной из кода на C, после того как она скомпилирована в разделяемую (shared) библиотеку и скомпонована из C:

```
#[no_mangle]
pub extern "C" fn call_from_c() {
    println!("Just called a Rust function from C!");
}
```

Использование `extern` не требует `unsafe`

Получение доступа и внесение изменений в изменяемую статическую переменную

До текущего момента мы не говорили о глобальных переменных (global variables), поддерживаемых языком Rust, но использование которых может быть проблематичным из-за правил заимствования. Если два потока получают доступ к одной и той же глобальной переменной, то это может вызвать ситуацию гонки данных.

Глобальные переменные в Rust называют *статическими* (static). Листинг 19-9 демонстрирует пример объявления и использования в качестве значения статической переменной, имеющей тип строкового среза:

Файл : src/main.rs

```
static HELLO_WORLD: &str = "Hello, world!";

fn main() {
    println!("name is: {}", HELLO_WORLD);
}
```

Листинг 19-9: Определение и использование неизменяемой статической переменной

Статические переменные похожи на константы, которые мы обсуждали в разделе “Различия между переменными и константами” главы 3. Имена статических переменных по общему соглашению пишутся в нотации `SCREAMING_SNAKE_CASE`, и мы должны указывать тип переменной, которым в данном случае является `&'static`

`str`. Статические переменные могут хранить только ссылки со временем жизни `'static`, это означает что компилятор Rust может вывести время жизни и нам не нужно прописывать его явно. Доступ к неизменяемой статической переменной является безопасным.

Константы и неизменяемые статические переменные могут казаться похожими друг на друга, но тонкая разница в том, что значения статических переменных имеют фиксированный адрес в памяти. Использование такого значения всегда будет обращаться к одним и тем же данным (по некоторому фиксированному адресу). Константам, с другой стороны, разрешено дублировать свои данные с помощью компилятора при любом их использовании.

Другое отличие констант от статических переменных в том, что последние могут быть изменяемыми. Чтение и изменение статических переменных является *небезопасным*. Листинг 19-10 показывает как объявлять, получать доступ и изменять изменяемую статическую переменную с именем `COUNTER`:

Имя файла: 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);
    }
}
```

Листинг 19-10: Чтение и запись изменяемой статической переменной является небезопасным

Как и с обычными переменными, мы определяем изменяемость с помощью ключевого слова `mut`. Любой код, который читает из или пишет в переменную `COUNTER` должен находиться в `unsafe` блоке. Этот код компилируется и печатает `COUNTER: 3`, как и следовало ожидать, потому что выполняется в одном потоке. Наличие нескольких потоков с доступом к `COUNTER` приведёт к ситуации гонки данных.

Наличие изменяемых данных, которые доступны глобально, делает трудным реализацию гарантии отсутствия гонок данных, поэтому Rust считает изменяемые статические переменные небезопасными. Там, где это возможно, предпочтительно использовать техники многопоточности и умные указатели, ориентированные на многопоточное исполнение, которые мы обсуждали в главе 16. Таким образом, компилятор сможет проверить, что обращение к данным, доступным из разных потоков, выполняется безопасно.

Реализация небезопасных типажей

Ещё один случай, в котором требуется `unsafe`, это реализация небезопасного типажа. Типаж небезопасен, если хотя бы один из его методов имеет некоторый инвариант, который компилятор не может проверить. Мы можем объявить типаж как `unsafe`, добавив ключевое слово `unsafe` перед `trait`, а также пометив реализацию типажа как `unsafe`, как показано в листинге 19-11.

```
unsafe trait Foo {
    // methods go here
}

unsafe impl Foo for i32 {
    // method implementations go here
}

fn main() {}
```

Листинг 19-11: Объявление и реализация небезопасного типажа

Используя `unsafe impl`, мы даём обещание поддерживать инварианты, которые компилятор не может проверить.

Для примера вспомним маркерные типажи `Sync` и `Send`, которые мы обсуждали в разделе "Расширяемый параллелизм с помощью типажей `Sync` и `Send`" главы 16: компилятор реализует эти типажи автоматически, если наши типы полностью состоят из типов `Send` и `Sync`. Если мы создадим тип, который содержит тип, не являющийся `Send` или `Sync`, такой, как сырой указатель, и мы хотим пометить этот тип как `Send` или `Sync`, мы должны использовать `unsafe` блок. Rust не может проверить, что наш тип поддерживает гарантии того, что он может быть безопасно передан между потоками или доступен из нескольких потоков; поэтому нам нужно добавить эти проверки вручную и указать это с помощью `unsafe`.

Доступ к полям объединений (union)

Последнее действие, которое работает только с `unsafe`, это доступ к полям объединений, `union`. `union` похож на `struct`, но только одно объявленное поле используется в конкретном экземпляре в один момент времени. Объединения в основном используются для взаимодействия с объединениями в коде С. Доступ к полям объединения небезопасен, потому что Rust не может гарантировать тип данных, хранящихся в данный момент в экземпляре объединения. Вы можете узнать больше об объединениях в [справочнике](#).

Когда использовать небезопасный код

Использование `unsafe` для выполнения одного из пяти действий (супер способностей), которые только что обсуждались, не является ошибочным или не одобренным. Но получить корректный `unsafe` код сложнее, потому что компилятор не может помочь в обеспечении безопасности памяти. Если у вас есть причина использовать `unsafe` код, вы можете делать это, а наличие явной `unsafe` аннотации облегчает отслеживание источника проблем, если они возникают.

Продвинутые типажи

Сначала мы рассмотрели типажи в разделе "Типажи: Определение общего поведения" главы 10, но как и со временами жизни, мы не обсудили более сложные детали. Сейчас что вы знаете о Rust больше и мы можем двинуться дальше.

Указание заполнителей типов в определениях типажей с ассоциированными типами

Ассоциированные типы (Associated types) связывают заполнитель типа с типажом, таким образом что объявления методов типажа могут использовать эти заполнители типов в своих сигнатурах. Реализация типажа будет указывать конкретный, используемый тип на месте заполнителя типа, при конкретной реализации. Таким образом, мы можем определить типаж пока он не реализован, который использует какие-то типы без необходимости знать, какими точно типами они будут.

Мы описали большинство расширенных возможностей в этой главе, как редко необходимые. Ассоциированные типы находятся где-то посередине: они используются реже чем возможности описанные в остальной части книги, но чаще чем многие другие возможности обсуждаемые в этой главе.

Одним из примеров типажа с ассоциированным типом является типаж **Iterator**, который предоставляет стандартная библиотека. Ассоциированный тип называется **Item** и представляет тип для значений, которые перебирает тип реализующий типаж **Iterator**. В разделе "Типаж **Iterator** и метод **next**" главы 13, мы упоминали определение типажа **Iterator** показанное в листинге 19-12.

```
pub trait Iterator {
    type Item;

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

Листинг 19-12: Определение типажа **Iterator**, который имеет ассоциированный тип **Item**

Тип **Item** является заполнителем и определение метода **next** показывает, что он будет возвращать значения типа **Option<Self::Item>**. Разработчики типажа **Iterator** определит конкретный тип для **Item**, а метод **next** вернёт **Option** содержащий значение этого конкретного типа.

Ассоциированные типы могли бы показаться концепцией похожей на обобщённые типы, в том смысле, что последние позволяют определить функцию, не указывая, какие типы она может обрабатывать. Так зачем использовать ассоциированные типы?

Давайте рассмотрим разницу между этими двумя понятиями на примере из главы 13, которая реализует типаж `Iterator` у структуры `Counter`. В листинге 13-21 мы указали, что тип для `Item` был `u32`:

Файл: `src/lib.rs`

```
{#rustdoc_include ../listings/ch19-advanced-features/listing-13-21-reproduced/src/lib.rs:ch19}
```

Этот синтаксис весьма напоминает обобщённые типы. Так почему же типаж `Iterator` не определён обобщённым типом, как показано в листинге 19-13?

```
pub trait Iterator<T> {
    fn next(&mut self) -> Option<T>;
}
```

Листинг 19-13: Гипотетическое определение типажа `Iterator` используя обобщённые типы

Разница в том, что при использовании обобщений, как показано в листинге 19-13, мы должны аннотировать типы в каждой реализации; потому что мы также можем реализовать `Iterator<String> for Counter` или любого другого типа, мы могли бы иметь несколько реализаций `Iterator` для `Counter`. Другими словами, когда типаж имеет обобщённый параметр, он может быть реализован для типа несколько раз, каждый раз меняя конкретные типы параметров обобщённого типа. Когда мы используем метод `next` у `Counter`, нам пришлось бы предоставить аннотации типа, указывая какую реализацию `Iterator` мы хотим использовать.

С ассоциированными типами не нужно аннотировать типы, потому что мы не можем реализовать типаж у типа несколько раз. В листинге 19-12 с определением, использующим ассоциированные типы можно выбрать только один тип `Item`, потому что может быть только одно объявление `impl Iterator for Counter`. Нам не нужно указывать, что нужен итератор значений типа `u32` везде, где мы вызываем `next` у `Counter`.

Параметры обобщённого типа по умолчанию и перегрузка операторов

Когда мы используем параметры обобщённого типа, мы можем указать конкретный тип по умолчанию для обобщённого типа. Это устраняет необходимость разработчикам указывать конкретный тип, если работает тип по умолчанию. Синтаксис для указания типа по умолчанию в случае обобщённого типа выглядит как `<PlaceholderType=ConcreteType>`.

Отличным примером ситуации, где этот подход полезен, является перегрузка оператора. *Перегрузка оператора* (Operator overloading) реализует пользовательское поведение некоторого оператора (например, `+`) в конкретных ситуациях.

Rust не позволяет создавать собственные операторы или перегружать произвольные операторы. Но можно перегрузить перечисленные операции и соответствующие им типажи из `std::ops` путём реализации типажей, связанных с этими операторами. Например, в листинге 19-14 мы перегружаем оператор `+`, чтобы складывать два экземпляра `Point`. Мы делаем это реализуя типаж `Add` для структуры `Point`:

Файл: src/main.rs

```
use std::ops::Add;

#[derive(Debug, Copy, Clone, 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 }
);
}
```

Листинг 19-14: Реализация типажа `Add` для перезагрузки оператора `+` у структуры `Point`

Метод `add` складывает значения `x` двух экземпляров `Point` и значения `y` у `Point` для создания нового экземпляра `Point`. Типаж `Add` имеет ассоциированный тип с именем `Output`, который определяет тип, возвращаемый из метода `add`.

Обобщённый тип по умолчанию в этом коде находится в типаже `Add`. Вот его определение:

```
trait Add<Rhs = Self> {
    type Output;

    fn add(self, rhs: Rhs) -> Self::Output;
}
```

Этот код должен выглядеть знакомым: типаж с одним методом и ассоциированным типом. Новый синтаксис это `RHS=Self`. Такой синтаксис называется *параметры типа по умолчанию* (default type parameters). Параметр обобщённого типа `RHS` (сокращённо “right hand side”) определяет тип параметра `rhs` в методе `add`. Если

мы не укажем конкретный тип для `RHS` при реализации типажа `Add`, то типом для `RHS` по умолчанию будет `Self`, который будет типом для которого реализуется типаж `Add`.

Когда мы реализовали `Add` для структуры `Point`, мы использовали стандартное значение для `RHS`, потому что хотели сложить два экземпляра `Point`. Давайте посмотрим на пример реализации типажа `Add`, где мы хотим пользовательский тип `RHS` вместо использования типа по умолчанию.

У нас есть две разные структуры `Millimeters` и `Meters`, хранящие значения в разных единицах измерения. Мы хотим добавить значения в миллиметрах к значениям в метрах и хотим иметь реализацию типажа `Add`, которая делает правильное преобразование единиц. Можно реализовать `Add` для `Millimeters` с типом `Meters` в качестве `Rhs`, как показано в листинге 19-15.

Файл: src/lib.rs

```
use std::ops::Add;

struct Millimeters(u32);
struct Meters(u32);

impl Add<Millimeters> for Meters {
    type Output = Millimeters;

    fn add(self, other: Millimeters) -> Millimeters {
        Millimeters(self.0 + (other.0 * 1000))
    }
}
```

Листинг 19-15: Реализация типажа `Add` для структуры `Millimeters`, чтобы складывать `Millimeters` и `Meters`

Чтобы сложить `Millimeters` и `Meters`, мы указываем `impl Add<Millimeters>`, чтобы указать значение параметра типа `RHS` (`Meters`) вместо использования значения по умолчанию `Self` (`Millimeters`).

Параметры типа по умолчанию используются в двух основных случаях:

- Чтобы расширить тип без внесения изменений ломающих существующий код
- Чтобы позволить пользовательское поведение в специальных случаях, которые не нужны большинству пользователей

Типаж `Add` из стандартной библиотеки является примером второй цели: обычно вы складываете два одинаковых типа, но типаж `Add` позволяет сделать больше. Использование параметра типа по умолчанию в объявлении типажа `Add` означает, что не нужно указывать дополнительный параметр большую часть времени. Другими словами, большая часть кода реализации не нужна, что делает использование типажа проще.

Первая цель похожа на вторую, но используется наоборот: если вы хотите добавить параметр типа к существующему типажу, можно дать ему значение по умолчанию, чтобы разрешить расширение функциональности типажа без нарушения кода существующей реализации.

Полностью квалифицированный синтаксис для устранения неоднозначности: вызов методов с одинаковым именем

В Rust ничего не мешает типажу иметь метод с одинаковым именем, таким же как метод другого типажа и Rust не мешает реализовывать оба таких типажа у одного типа. Также возможно реализовать метод с таким же именем непосредственно у типа, такой как и методы у типажей.

При вызове методов с одинаковыми именами в Rust нужно указать, какой из трёх возможных вы хотите использовать. Рассмотрим код в листинге 19-16, где мы определили два типажа: `Pilot` и `Wizard`, у обоих есть метод `fly`. Затем мы реализуем оба типажа у типа `Human` в котором уже реализован метод с именем `fly`. Каждый метод `fly` делает что-то своё.

Файл: `src/main.rs`

```

trait Pilot {
    fn fly(&self);
}

trait Wizard {
    fn fly(&self);
}

struct Human;

impl Pilot for Human {
    fn fly(&self) {
        println!("This is your captain speaking.");
    }
}

impl Wizard for Human {
    fn fly(&self) {
        println!("Up!");
    }
}

impl Human {
    fn fly(&self) {
        println!("*waving arms furiously*");
    }
}

```

Листинг 19-16: Два типажа определены с методом `fly` и реализованы у типа `Human`, а также метод `fly` реализован непосредственно у `Human`

Когда мы вызываем `fly` у экземпляра `Human`, то компилятор по умолчанию вызывает метод, который непосредственно реализован для типа, как показано в листинге 19-17.

Файл: `src/main.rs`

```

fn main() {
    let person = Human;
    person.fly();
}

```

Листинг 19-17: Вызов `fly` у экземпляра `Human`

Запуск этого кода напечатает `*waving arms furiously*`, показывая, что Rust называется метод `fly` реализованный непосредственно у `Human`.

Чтобы вызвать методы `fly` у типажа `Pilot` или типажа `Wizard` нужно использовать более явный синтаксис, указывая какой метод `fly` мы имеем в виду. Листинг 19-18 демонстрирует такой синтаксис.

Файл: `src/main.rs`

```
fn main() {
    let person = Human;
    Pilot::fly(&person);
    Wizard::fly(&person);
    person.fly();
}
```

Листинг 19-18: Указание какой метода `fly` мы хотим вызвать

Указание имени типажа перед именем метода проясняет компилятору Rust, какую именно реализацию `fly` мы хотим вызвать. Мы могли бы также написать `Human::fly(&person)`, что эквивалентно используемому нами `person.fly()` в листинге 19-18, но это писание немного длиннее, когда нужна неоднозначность.

Выполнение этого кода выводит следующее:

```
$ cargo run
Compiling traits-example v0.1.0 (file:///projects/traits-example)
Finished dev [unoptimized + debuginfo] target(s) in 0.46s
    Running `target/debug/traits-example`
This is your captain speaking.
Up!
*waving arms furiously*
```

Поскольку метод `fly` принимает параметр `self`, если у нас было два *типа* оба реализующих один *типаж*, то Rust может понять, какую реализацию типажа использовать в зависимости от типа `self`.

Однако ассоциированные функции являющиеся частью типажей не имеют `self` параметра. Когда два типа в одной области видимости реализуют такой типаж, Rust не может выяснить, какой тип вы имеете в виду если вы не используете *полностью квалифицированный синтаксис* (fully qualified). Например, типаж `Animal` в листинге 19-19 имеет: ассоциированную функцию `baby_name`, реализацию типажа `Animal` для структуры `Dog` и ассоциированную функцию `baby_name`, объявленную напрямую у структуры `Dog`.

Файл: `src/main.rs`

```

trait Animal {
    fn baby_name() -> String;
}

struct Dog;

impl Dog {
    fn baby_name() -> String {
        String::from("Spot")
    }
}

impl Animal for Dog {
    fn baby_name() -> String {
        String::from("puppy")
    }
}

fn main() {
    println!("A baby dog is called a {}", Dog::baby_name());
}

```

Листинг 19-19: Типаж с ассоциированной функцией и тип с ассоциированной функцией с тем же именем, которая тоже реализует типаж

Этот код для приюта для животных, который хочет назвать всех щенков именем Spot, что реализовано в ассоциированной функции `baby_name`, которая определена для `Dog`. Тип `Dog` также реализует типаж `Animal`, который описывает характеристики, которые есть у всех животных. Маленьких собак называют щенками, и это выражается в реализации `Animal` у `Dog` в функции `baby_name` ассоциированной с типажом `Animal`.

В `main` мы вызываем функцию `Dog::baby_name`, которая вызывает ассоциированную функцию определённую напрямую у `Dog`. Этот код печатает следующее:

```

$ cargo run
Compiling traits-example v0.1.0 (file:///projects/traits-example)
Finished dev [unoptimized + debuginfo] target(s) in 0.54s
Running `target/debug/traits-example`
A baby dog is called a Spot

```

Этот вывод является не тем, что мы хотели получить. Мы хотим вызывать функцию `baby_name`, которая является частью типажа `Animal` реализованного у `Dog`, так чтобы код печатал `A baby dog is called a puppy`. Техника указания имени типажа

использованная в листинге 19-18 здесь не помогает; если мы изменим `main` код как в листинге 19-20, мы получим ошибку компиляции.

Файл: src/main.rs

```
fn main() {
    println!("A baby dog is called a {}", Animal::baby_name());
}
```



Листинг 19-20. Попытка вызвать функцию `baby_name` из типажа `Animal`, но Rust не знает какую реализацию использовать

Так как `Animal::baby_name` является ассоциированной функцией не имеющей `self` параметра в сигнатуре, а не методом, то Rust не может понять, какую реализацию `Animal::baby_name` мы хотим вызвать. Мы получим эту ошибку компилятора:

```
$ cargo run
Compiling traits-example v0.1.0 (file:///projects/traits-example)
error[E0283]: type annotations needed
--> src/main.rs:20:43
20 |     println!("A baby dog is called a {}", Animal::baby_name());
   |                                     ^^^^^^^^^^^^^^^^^ cannot infer
type
|
= note: cannot satisfy `_: Animal`

For more information about this error, try `rustc --explain E0283`.
error: could not compile `traits-example` due to previous error
```

Чтобы устранить неоднозначность и сказать Rust, что мы хотим использовать реализацию `Animal` для `Dog`, нужно использовать полный синтаксис. Листинг 19-21 демонстрирует, как использовать полный синтаксис.

Файл: src/main.rs

```
fn main() {
    println!("A baby dog is called a {}", <Dog as Animal>::baby_name());
}
```

Листинг 19-21: Использование полностью квалифицированного синтаксиса для указания, что мы хотим вызвать функцию `baby_name` у типажа `Animal` реализованную в `Dog`

Мы указываем аннотацию типа в угловых скобках, которая указывает на то что мы

хотим вызвать метод `baby_name` из типажа `Animal` реализованный в `Dog`, также указывая что мы хотим рассматривать тип `Dog` в качестве `Animal` для вызова этой функции. Этот код теперь напечатает то, что мы хотим:

```
$ cargo run
Compiling traits-example v0.1.0 (file:///projects/traits-example)
Finished dev [unoptimized + debuginfo] target(s) in 0.48s
    Running `target/debug/traits-example`
A baby dog is called a puppy
```

В общем, полностью квалифицированный синтаксис определяется следующим образом:

```
<Type as Trait>::function(receiver_if_method, next_arg, ...);
```

Для ассоциированных функций при их вызове не будет `receiver` (объекта приёмника), а будет только список аргументов. Вы можете использовать полностью квалифицированный синтаксис везде, где вызываете функции или методы. Тем не менее, разрешается опустить любую часть этого синтаксиса, которую Rust может понять из другой информации в программе. Необходимость использования этого наиболее подробного синтаксиса возникает только в тех случаях, когда есть несколько реализаций, которые используют одинаковое имя и Rust нуждается в помощи для определения, какой вариант реализации вы хотите вызвать.

Использование супер типажей для требования функциональности одного типажа в рамках другого типажа

Иногда вам может понадобиться, чтобы один типаж использовал функциональность другого типажа. В этом случае нужно полагаться на зависимый типаж, который также реализуется. Типаж на который вы полагаетесь, является *супер типажом* типажа, который реализуете вы.

Например, мы хотим создать типаж `OutlinePrint` с методом `outline_print`, который будет печатать значение обрамлённое звёздочками. Мы хотим чтобы структура `Point` реализующая типаж `Display` вывела на печать `(x, y)` при вызове `outline_print` у экземпляра `Point`, который имеет значение `1` для `x` и значение `3` для `y`. Она должна напечатать следующее:

```
*****
*      *
* (1, 3) *
*      *
*****
```

В реализации `outline_print` мы хотим использовать функциональность типажа `Display`. Поэтому нам нужно указать, что типаж `OutlinePrint` будет работать только для типов, которые также реализуют `Display` и предоставляют функциональность, которая нужна в `OutlinePrint`. Мы можем сделать это в объявлении типажа, указав `OutlinePrint: Display`. Этот метод похож на добавление ограничения в типаж. В листинге 19-22 показана реализация типажа `OutlinePrint`.

Файл: 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));
    }
}
```

Листинг 19-22: Реализация типажа `OutlinePrint` которая требует функциональности типажа `Display`

Поскольку мы указали, что типаж `OutlinePrint` требует типажа `Display`, мы можем использовать функцию `to_string`, которая автоматически реализована для любого типа реализующего `Display`. Если бы мы попытались использовать `to_string` не добавляя двоеточие и не указывая типаж `Display` после имени типажа, мы получили бы сообщение о том, что метод с именем `to_string` не был найден у типа `&Self` в текущей области видимости.

Давайте посмотрим что происходит, если мы пытаемся реализовать типаж `OutlinePrint` для типа, который не реализует `Display`, например структура `Point`:

Файл: src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

impl OutlinePrint for Point {}
```



Мы получаем сообщение о том, что требуется реализация `Display`, но её нет:

```
$ cargo run
Compiling traits-example v0.1.0 (file:///projects/traits-example)
error[E0277]: `Point` doesn't implement `std::fmt::Display`
--> src/main.rs:20:6
20 | impl OutlinePrint for Point {}  
     ^^^^^^^^^^ `Point` cannot be formatted with the default formatter
|  
= help: the trait `std::fmt::Display` is not implemented for `Point`
= note: in format strings you may be able to use `{:?}` (or `{:#{?}}` for
pretty-print) instead
note: required by a bound in `OutlinePrint`
--> src/main.rs:3:21
3  | trait OutlinePrint: fmt::Display {  
     ^^^^^^^^^^ required by this bound in  
`OutlinePrint`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `traits-example` due to previous error
```

Чтобы исправить, мы реализуем `Display` у структуры `Point` и выполняем требуемое ограничение `OutlinePrint`, вот так:

Файл: 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` для структуры `Point` будет скомпилирована успешно и мы можем вызвать `outline_print` у экземпляра `Point` для отображения значения обрамлённое звёздочками.

Шаблон Newtype для реализации внешних типажей у внешних типов

В разделе "Реализация типажа у типа" главы 10, мы упоминали "правило сироты" (orphan rule), которое гласит, что разрешается реализовать типаж у типа, если либо типаж, либо тип являются локальными для нашего крейта. Можно обойти это ограничение, используя *шаблон нового типа* (newtype pattern), который включает в себя создание нового типа в кортежной структуре. (Мы рассмотрели кортежные структуры в разделе "Использование структур кортежей без именованных полей для создания различных типов" главы 5.) Структура кортежа будет иметь одно поле и будет тонкой оболочкой для типа которому мы хотим реализовать типаж. Тогда тип оболочки является локальным для нашего крейта и мы можем реализовать типаж для локальной обёртки. *Newtype* это термин, который происходит от языка программирования Haskell. В нем нет ухудшения производительности времени выполнения при использовании этого шаблона и тип оболочки исключается во время компиляции.

В качестве примера, мы хотим реализовать типаж `Display` для типа `Vec<T>`, где "правило сироты" (orphan rule) не позволяет нам этого делать напрямую, потому что типаж `Display` и тип `Vec<T>` объявлены вне нашего крейта. Мы можем сделать структуру `Wrapper`, которая содержит экземпляр `Vec<T>`; тогда мы можем реализовать `Display` у структуры `Wrapper` и использовать значение `Vec<T>` как показано в листинге 19-23.

Файл: 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);
}
```

Листинг 19-23. Создание типа `Wrapper` `Vec<String>` для реализации `Display`

Реализация `Display` использует `self.0` для доступа к внутреннему `Vec<T>`, потому что `Wrapper` это структура кортежа, а `Vec<T>` это элемент с индексом 0 в кортеже. Затем мы можем использовать функциональные возможности типа `Display` у `Wrapper`.

Недостатком использования этой техники является то, что `Wrapper` является новым типом, поэтому он не имеет методов для значения, которое он держит в себе. Мы должны были бы реализовать все методы для `Vec<T>` непосредственно во `Wrapper`, так чтобы эти методы делегировались внутреннему `self.0`, что позволило бы нам обращаться с `Wrapper` точно так же, как с `Vec<T>`. Если бы мы хотели, чтобы новый тип имел каждый метод имеющийся у внутреннего типа, реализуя типаж `Deref` (обсуждается в разделе "Работа с умными указателями как с обычными ссылками с помощью `Deref` типажа" главы 15) у `Wrapper` для возвращения внутреннего типа, то это было бы решением. Если мы не хотим, чтобы тип `Wrapper` имел все методы внутреннего типа, например, для ограничения поведения типа `Wrapper`, то пришлось бы вручную реализовать только те методы, которые нам нужны.

Теперь вы знаете, как используется newtype шаблон по отношению к типажам; это также полезный шаблон, даже когда типажи не используются. Давайте переключимся и посмотрим на некоторые продвинутые способы взаимодействия с системой типов Rust.

Расширенные типы

Система типов Rust имеет некоторые возможности, которые мы упоминали в этой книге, но ещё не обсуждали. Мы начнём с обсуждения новых типов (newtypes) в целом, по мере изучения того, почему новые типы полезны в качестве типов. Затем мы перейдём к псевдонимам, возможности похожей на новые типы (newtypes), но с немного другой семантикой. Мы также обсудим тип `!` и с динамическими типами (dynamically sized type).

Использование Newtype шаблона для безопасности типов и реализации абстракций

Примечание. В следующем разделе предполагается, что вы прочитали предыдущий раздел ["Использование шаблона Newtype для реализации внешних типажей у внешних типов"](#)

Шаблон newtype полезен для задач помимо тех, которые мы обсуждали до сих пор, включая статическое обеспечение того, чтобы значения никогда не путались и указывали единицы значения. Вы видели пример использования newtype для обозначения единиц в листинге 19-15. Вспомним, что структуры `Millimeters` и `Meters` содержат обёрнутые значения `u32` в newtype. Если бы мы написали функцию с параметром типа `Millimeters`, мы не смогли бы скомпилировать программу, которая случайно пыталась вызвать эту функцию со значением типа `Meters` или обычным `u32`.

Другое использование шаблона newtype - абстрагирование от некоторых деталей реализации типа: новый тип может предоставлять открытый API, отличный от API приватного внутреннего типа, если мы напрямую использовали новый тип для ограничения доступного функционала, например.

Варианты шаблона (Newtypes) также могут скрывать внутреннюю реализацию. Например, мы могли бы предоставить тип `People` для оборачивания типа `HashMap<i32, String>`, который хранит идентификатор человека связанного с его именем. Код использующий `People` будет взаимодействовать только с предоставляемым нами открытым API, например метод добавления строки имени в коллекцию `People`; этому коду не понадобилось бы знать, что мы внутри присваиваем ID код типа `i32` именам. Шаблон newtype - это лёгкий способ

добраться инкапсуляции, скрыть детали реализации, которые мы обсуждали в разделе "Инкапсуляция, которая скрывает детали реализации" главы 17.

Создание синонимов типа с помощью псевдонимов типа

Наряду с шаблоном newtype, Rust предоставляет возможность объявить *псевдоним типа* чтобы дать существующему типу другое имя. Для этого мы используем ключевое слово `type`. Например, мы можем создать псевдоним типа `Kilometers` для `i32` следующим образом:

```
type Kilometers = i32;
```

Теперь псевдоним `Kilometers` является *синонимом* для `i32`; в отличие от типов `Millimeters` и `Meters`, которые мы создали в листинге 19-15, `Kilometers` не являются отдельными, новыми типами. Значения с типом `Kilometers` будут обрабатываться так же, как значения типа `i32`:

```
type Kilometers = i32;

let x: i32 = 5;
let y: Kilometers = 5;

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

Поскольку `Kilometers` и `i32` являются одинаковым типом, мы можем сложить значения обоих типов и мы можем передать значения `Kilometers` в функции, которые принимают параметры типа `i32`. Однако, используя этот метод, мы не получаем преимуществ проверки типа, которые доступны в шаблоне newtype, обсуждавшемся ранее.

Синонимы в основном используются для уменьшения повторяемости. Например, у нас есть тип:

```
Box<dyn Fn() + Send + 'static>
```

Запись этого длинного типа в сигнатурах функций и в виде аннотаций типов по всему коду может быть утомительной и приводить к ошибкам. Представьте, что у вас есть проект, полный кодом как в листинге 19-24.

```
let f: Box<dyn Fn() + Send + 'static> = Box::new(|| println!("hi"));

fn takes_long_type(f: Box<dyn Fn() + Send + 'static>) {
    // --snip--
}

fn returns_long_type() -> Box<dyn Fn() + Send + 'static> {
    // --snip--
}
```

Листинг 19-24: Использование длинного типа во многих местах

Псевдоним типа делает этот код более управляемым за счёт сокращения повторений. В листинге 19-25 мы представили псевдоним **Thunk** для "многословного" типа и теперь можем заменить все использование такого типа на более короткий псевдонимом **Thunk**.

```
type Thunk = Box<dyn Fn() + Send + 'static>;

let f: Thunk = Box::new(|| println!("hi"));

fn takes_long_type(f: Thunk) {
    // --snip--
}

fn returns_long_type() -> Thunk {
    // --snip--
}
```

Листинг 19-25: Представление псевдонима **Thunk** для уменьшения количества повторений

Этот код намного легче читать и писать! Выбор значимого имени для псевдонима типа может также помочь сообщить о ваших намерениях (*thunk* является словом для кода, который будет вычисляться позднее, так что это подходящее название для замыкания, которое сохраняется).

Псевдонимы типов также обычно используются с типом **Result<T, E>** для сокращения повторения. Рассмотрим модуль **std::io** в стандартной библиотеке. Операции ввода/вывода часто возвращают тип **Result<T, E>** для обработки ситуаций, когда операция не выполняется из-за ошибки. Эта библиотека имеет структуру **std::io::Error**, которая представляет все возможные ошибки ввода/вывода. Многие функции в библиотеке **std::io** будут возвращать **Result<T, E>**, где **E** - это **std::io::Error**, например, такие как функции в **Write** типаже:

```
use std::fmt;
use std::io::Error;

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>;
}
```

Тип `Result<..., Error>` многократно повторяется. Таким образом, `std::io` имеет этот тип как объявление псевдонима:

```
type Result<T> = std::result::Result<T, std::io::Error>;
```

Поскольку это объявление находится в модуле `std::io`, мы можем использовать полностью квалифицированный псевдоним `std::io::Result<T>`, что является `Result<T, E>` с типом `E` заполненным типом `std::io::Error`. Сигнатуры функций типажа `Write` в конечном итоге выглядят как:

```
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: fmt::Arguments) -> Result<()>;
}
```

Псевдоним типа помогает двумя способами: он облегчает написание кода и даёт нам согласованный интерфейс для всего из `std::io`. Поскольку это псевдоним, то это просто ещё один тип `Result<T, E>`, что означает, что с ним мы можем использовать любые методы, которые работают с `Result<T, E>`, а также специальный синтаксис вроде `?` оператора.

Тип `Never`, который никогда не возвращается

Rust имеет специальный тип с названием `!`, который известен в теории типов как *пустой тип* (empty type), потому что у него нет значений. Мы предпочитаем называть его *тип никогда* (never type), потому что он стоит на месте возвращаемого типа, такая функция никогда не возвращает управление. Вот пример:

```
fn bar() -> ! {
    // --snip--
}
```

Этот код читается как «функция `bar` никогда не возвращается». Функции, которые никогда не возвращаются называются *расходящимися функциями* (diverging functions). Нельзя создавать значения типа `!`, так как `bar` никогда не может вернуться.

Но для чего нужен тип, для которого вы никогда не сможете создать значения? Напомним код из листинга 2-5; мы воспроизвели его часть здесь в листинге 19-26.

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

Листинг 19-26: Сопоставление `match` с веткой, которая заканчивается `continue`

В то время мы опустили некоторые детали в этом коде. В главе 6 раздела «[Оператор управления потоком `match`](#)» мы обсуждали, что все ветви `match` должны возвращать одинаковый тип. Например, следующий код не работает:

```
let guess = match guess.trim().parse() {
    Ok(_) => 5,
    Err(_) => "hello",
};
```



Тип `guess` в этом коде должен быть целым числом и строкой и Rust требует, чтобы `guess` имел только один тип. Так что тогда возвращает код `continue`? Как нам разрешили вернуть `u32` из одной ветки и иметь другую ветку заканчивающуюся на `continue` в листинге 19-26?

Как вы уже возможно догадались, `continue` имеет значение `!`. То есть, когда Rust вычисляет тип `guess`, он смотрит на обе сопоставляемые ветки, первая со значением `u32` и последняя со значением `!`. Так как `!` никогда не может иметь значение, то Rust решает что типом `guess` является тип `u32`.

Формальным способом описания этого поведения является то, что выражения типа `!` могут быть приведены (coerced) к любому другому типу. Нам разрешено закончить сопоставление этой `match` ветки с помощью `continue`, потому что `continue` не возвращает значение; вместо этого она передаёт контроль обратно в

начало цикла, поэтому в случае `Err` мы никогда не присваиваем `guess` значение.

Never тип полезен также с макросом `panic!`. Помните, функцию `unwrap`, которую мы вызываем для значений `Option<T>`, чтобы создать значение или вызвать панику? Вот её определение:

```
impl<T> Option<T> {
    pub fn unwrap(self) -> T {
        match self {
            Some(val) => val,
            None => panic!("called `Option::unwrap()` on a `None` value"),
        }
    }
}
```

В этом коде происходит то же самое, что и в выражении `match` из листинга 19-26: Rust видит, что `val` имеет тип `T` и `panic!` имеет тип `!`, поэтому общим результатом `match` выражения является `T`. Этот код работает, потому что `panic!` не производит значения; он завершает выполнение программы. В случае `None`, мы не будем возвращать значение из `unwrap`, поэтому этот код действительный.

Последнее выражение, которое имеет тип `!` это `loop`:

```
print!("forever ");

loop {
    print!("and ever ");
}
```

Здесь цикл никогда не заканчивается, так что `!` (never type) является значением выражения. Тем не менее, это не будет правдой, если мы добавим в цикл `break`, потому что цикл мог бы завершиться, когда дело дойдёт до `break`.

Динамические типы и `Sized` типаж

В связи с необходимостью Rust знать определённые детали, например, сколько места выделять для значения определённого типа, то существует краеугольный камень его системы типов, который может сбивать с толку. Это концепция *динамических типов* (dynamically sized types). Иногда она упоминается как *DST* или *безразмерные типы* (unsized types), эти типы позволяют писать код, используя значения, чей размер известен только во время выполнения.

Давайте углубимся в детали динамического типа `str`, который мы использовали на протяжении всей книги. Все верно, не типа `&str`, а типа `str` самого по себе, который является DST. Мы не можем знать, какой длины строка до момента времени выполнения, то есть мы не можем создать переменную типа `str` и не можем принять аргумент типа `str`. Рассмотрим следующий код, который не работает:

```
let s1: str = "Hello there!";
let s2: str = "How's it going?";
```



Rust должен знать, сколько памяти выделить для любого значения конкретного типа и все значения типа должны использовать одинаковый объем памяти. Если Rust позволил бы нам написать такой код, то эти два значения `str` должны были бы занимать одинаковое количество памяти. Но они имеют разную длину: `s1` нужно 12 байтов памяти, а для `s2` нужно 15. Вот почему невозможно создать переменную имеющую динамический тип.

Так что же нам делать? В этом случае вы уже знаете ответ: мы делаем типы `s1` и `s2` в виде типа `&str`, а не `str`. Напомним, что в разделе "Строковые срезы" главы 4, мы сказали, что структура данных срез хранит начальную позицию и длину среза.

Таким образом, хотя `&T` является единственным значением, которое хранит адрес памяти где находится тип `T`, тип `&str` является двумя значениями: адресом `str` и его длиной. Таким образом, мы можем знать размер значения `&str` во время компиляции: это двойная длина от типа `usize`. То есть мы всегда знаем размер `&str`, неважно какой длины является строка на которую она ссылается. В общем, это способ которым в Rust используются динамические типы: у них есть дополнительные метаданные в которых хранится размер динамической информации. Золотое правило динамических типов в том, что мы всегда должны ставить значения динамических типов позади некоторого указателя.

Можно комбинировать `str` со всеми видами указателей: например, `Box<str>` или `Rc<str>`. На самом деле, вы видели это раньше, но с другим динамическим типом: типажом. Каждый типаж является динамическим типом к которому можно обратиться используя имя типажа. В разделе "Использование объектов-типажей, которые разрешают использовать разные значения типов" главы 17, мы упоминали, что для использования типажей в качестве объектов-типажей мы должны поместить их за указателем, например `&dyn Trait` или `Box<dyn Trait>` (`Rc<dyn Trait>` тоже будет работать).

Для работы с DST в Rust есть особый типаж, называемый **Sized** для определения, известен ли размер типа во время компиляции. Этот типаж автоматически реализуется для всех типов, чей размер известен во время компиляции. Кроме того, Rust неявно добавляет ограничение **Sized** в каждую обобщённую функцию. То есть определение обобщённой функции написанное как:

```
fn generic<T>(t: T) {  
    // --snip--  
}
```

на самом деле рассматривается как если бы мы написали её в виде:

```
fn generic<T: Sized>(t: T) {  
    // --snip--  
}
```

По умолчанию обобщённые функции будут работать только с типами чей размер известен в время компиляции. Тем не менее, можно использовать следующий специальный синтаксис, чтобы ослабить это ограничение:

```
fn generic<T: ?Sized>(t: &T) {  
    // --snip--  
}
```

Ограничение на типаж **?Sized** означает «**T** может или не может быть **Sized**», и это обозначение имеет приоритет по умолчанию. Общие типы должны иметь известный размер во время компиляции. Синтаксис **?Trait** с таким значением доступен только для **Sized**, но не для любых других типажей.

Также обратите внимание, что мы поменяли тип параметра **t** с **T** на **&T**. Поскольку тип мог бы не быть **Sized**, мы должны использовать его за каким-либо указателем. В этом случае мы выбрали ссылку.

Далее мы поговорим о функциях и замыканиях!

Продвинутые функции и замыкания

Наконец, мы рассмотрим некоторые дополнительные возможности, связанные с функциями и замыканиями, которые включают указатели на функции и возврат замыканий.

Указатели функций

Мы говорили о том, как передавать замыкания в функции; но вы также можете передавать обычные функции в функции! Эта техника полезна, когда вы хотите передать функцию, которую вы уже определили, а не объявлять новое замыкание. Указатель функции позволит использовать функции как аргументы к другим функциям. Функции приводятся (coerce) к типу `fn` (с нижним регистром f), не к путать с типажом замыкания `Fn`. Тип `fn` называется *указателем функции*. Синтаксис для указания того, что параметр является указателем функции, похож на замыкание как показано в листинге 19-27.

Файл: `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!("The answer is: {}", answer);
}
```

Листинг 19-27: Использование типа `fn` для принятия указателя функции в качестве аргумента

Этот код печатает `The answer is: 12`. Мы указываем, что параметр вызова `f` для функции `do_twice` является `fn`, которая принимает один параметр типа `i32` и возвращает тип `i32`. Затем мы можем вызвать `f` в теле функции `do_twice`. В `main` показано как можно передать имя функции `add_one` в качестве первого аргумента для функции `do_twice`.

В отличие от замыканий, `fn` является типом, а не типажом, поэтому мы указываем `fn` как параметр типа напрямую, а не объявляем параметр обобщённого типа с одним из типажей `Fn` в качестве ограничения типажа.

Указатели функций реализуют все три типажа замыканий (`Fn`, `FnMut` и `FnOnce`), поэтому вы всегда можете передать указатель функции в качестве аргумента функции ожидающей замыкание. Лучше всего объявлять функции, используя обобщённый тип и одним из типажей замыкания, так что ваши функции могут принимать либо функции, либо замыкания.

Пример того, где вы хотели бы принимать только тип `fn`, а не замыкания является взаимодействие с внешним кодом, который не имеет замыканий: функции в C могут принимать функции в качестве аргументов, но C не имеет замыканий.

Для примера того, где вы могли бы использовать либо замыкание, определённое как встроенное, либо именованную функцию, давайте посмотрим на использование `map`. Для использования функции `map`, чтобы превратить вектор чисел в вектор строк, мы могли бы использовать замыкание, как здесь:

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> =
    list_of_numbers.iter().map(|i| i.to_string()).collect();
```

Или мы могли бы назвать функцию вместо замыкания в качестве аргумента при вызове `map`, как здесь:

```
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, что мы должны использовать полный синтаксис, о котором мы говорили ранее в разделе "Расширенные типажи", потому что доступно несколько функций с именем `to_string`. Здесь мы используем функцию `to_string` определённую в типаже `ToString`, который реализован в стандартной библиотеке для любого типа реализующего типаж `Display`.

У нас есть ещё один полезный шаблон, который использует детали реализации структур кортежей (tuple structs) и вариантов перечислений структур кортежей (tuple-struct enum). Эти типы используют `()` в качестве синтаксиса инициализатора, который выглядит как вызов функции. Инициализаторы на самом деле реализованы как функции, возвращающие экземпляр, который построен из их

аргументов. Мы можем использовать эти функции инициализаторы как указатели на функции, которые реализуют типажи замыканий, что означает мы можем указать инициализирующие функции в качестве аргументов для методов, которые принимают замыкания, например:

```
enum Status {
    Value(u32),
    Stop,
}

let list_of_statuses: Vec<Status> =
(0u32..20).map(Status::Value).collect();
```

Здесь мы создаём экземпляры `Status::Value`, используя каждое значение `u32` в диапазоне (0..20), с которым вызывается `map` с помощью функции инициализатора `Status::Value`. Некоторые люди предпочитают этот стиль, а некоторые предпочитают использовать замыкания. Оба варианта компилируются в один и тот же код, поэтому используйте любой стиль, который вам понятнее.

Возврат замыканий

Замыкания представлены типажами, что означает невозможность напрямую вернуть замыкания. В большинстве случаев, когда вы возможно хотите вернуть типаж, вы вместо этого используете конкретный тип, который реализует типаж в качестве возвращаемого значения функции. Но вы не можете сделать этого с замыканиями, потому что у них нет конкретного типа, который можно вернуть; не разрешается использовать указатель функции `fn` в качестве возвращаемого типа, например.

Следующий код пытается напрямую вернуть замыкание, но он не компилируется:

```
fn returns_closure() -> dyn Fn(i32) -> i32 {
    |x| x + 1
}
```

Ошибка компилятора выглядит следующим образом:

```
$ cargo build
Compiling functions-example v0.1.0 (file:///projects/functions-example)
error[E0746]: return type cannot have an unboxed trait object
--> src/lib.rs:1:25
|
1 | fn returns_closure() -> dyn Fn(i32) -> i32 {
|                                     ^^^^^^^^^^^^^^ doesn't have a size known at
compile-time
|
= note: for information on `impl Trait`, see <https://doc.rust-lang.org/book/ch10-02-trait.html#returning-types-that-implement-trait>
help: use `impl Fn(i32) -> i32` as the return type, as all return paths are
of type `[closure@src/lib.rs:2:5: 2:14]`, which implements `Fn(i32) -> i32`
|
1 | fn returns_closure() -> impl Fn(i32) -> i32 {
|                                     ~~~~~~
```

For more information about this error, try `rustc --explain E0746`.
error: could not compile `functions-example` due to previous error

Ошибка снова ссылается на типаж **Sized**! Rust не знает, сколько памяти нужно будет выделить для замыкания. Мы видели решение этой проблемы ранее. Мы можем использовать типаж-объект:

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

Этот код просто отлично компилируется. Для получения дополнительной информации об типаж-объектах обратитесь к разделу "[Использование типаж-объектов которые допускают значения разных типов](#)" главы 17.

Далее давайте посмотрим на макросы!

Макросы

Мы использовали макросы, такие как `println!` на протяжении всей этой книги, но мы не изучили полностью, что такое макрос и как он работает. Термин *макрос* относится к семейству возможностей в Rust. Это *декларативные* (declarative) макросы с помощью `macro_rules!` и три вида *процедурных* (procedural) макросов:

- Пользовательские (выводимые) `#[derive]` макросы, которые указывают код добавленный с помощью `derive` атрибута, используемые для структур и перечислений
- Макросы подобные атрибутам (attribute-like), которые определяют настраиваемые атрибуты, используемые для любого элемента языка
- Функционально подобные (function-like) макросы, которые выглядят как вызовы функций, но работают с `TokenStream`

Мы поговорим о каждом из них по очереди, но сначала давайте рассмотрим, зачем вообще нужны макросы, если есть функции.

Разница между макросами и функциями

По сути, макросы являются способом написания кода, который записывает другой код, что известно как *мета программирование*. В приложении С мы обсуждаем атрибут `derive`, который генерирует за вас реализацию различных типажей. Вы также использовали макросы `println!` и `vec!` в книге. Все эти макросы *раскрываются* для генерации большего количества кода, чем исходный код написанный вами вручную.

Мета программирование полезно для уменьшения объёма кода, который вы должны написать и поддерживать, что также является одним из предназначений функций. Однако макросы имеют некоторые дополнительные возможности, которых функции не имеют.

Сигнатура функции должна объявлять некоторое количество и тип этих параметров имеющихся у функции. Макросы, с другой стороны, могут принимать переменное число параметров: мы можем вызвать `println!("hello")` с одним аргументом или `println!("hello {}", name)` с двумя аргументами. Также макросы раскрываются до того как компилятор интерпретирует смысл кода, поэтому макрос может, например, реализовать типаж заданного типа. Функция этого не может, потому что она вызывается во время выполнения и типаж должен быть реализован во время

компиляции.

Обратной стороной реализации макроса вместо функции является то, что определения макросов являются более сложными, чем определения функций, потому что вы создаёте Rust код, который записывает другой Rust код. Из-за этой косвенности, объявления макросов, как правило, труднее читать, понимать и поддерживать, чем объявления функций.

Другое важное различие между макросами и функциями заключается в том, что вы должны объявить макросы или добавить их в область видимости *прежде* чем можете вызывать их в файле, в отличии от функций, которые вы можете объявить где угодно и вызывать из любого места.

Декларативные макросы с `macro_rules!` для общего мета программирования

Наиболее широко используемой формой макросов в Rust являются *декларативные макросы*. Они также иногда упоминаются как "макросы на примере", "`macro_rules!` макрос" или просто "макросы". По своей сути декларативные макросы позволяют писать нечто похожее на выражение `match` в Rust. Как обсуждалось в главе 6, `match` выражения являются управляемыми структурами, которые принимают некоторое выражение, результат значения выражения сопоставляют с шаблонами, а затем запускают код для сопоставляемой ветки. Макросы также сравнивают значение с шаблонами, которые связаны с конкретным кодом: в этой ситуации значение является литералом исходного кода Rust, переданным в макрос. Шаблоны сравниваются со структурами этого исходного кода и при совпадении код, связанный с каждым шаблоном, заменяет код переданный макросу. Все это происходит во время компиляции.

Для определения макроса используется конструкция `macro_rules!`. Давайте рассмотрим, как использовать `macro_rules!` глядя на то, как объявлен макрос `vec!`. В главе 8 рассказано, как можно использовать макрос `vec!` для создания нового вектора с определёнными значениями. Например, следующий макрос создаёт новый вектор, содержащий три целых числа:

```
let v: Vec<u32> = vec![1, 2, 3];
```

Мы также могли использовать макрос `vec!` для создания вектора из двух целых

чисел или вектора из пяти строковых срезов. Мы не смогли бы использовать функцию, чтобы сделать то же самое, потому что мы не знали бы заранее количество или тип значений.

В листинге 19-28 приведено несколько упрощённое определение макроса `vec!`.

Файл: `src/lib.rs`

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Листинг 19-28: Упрощённая версия определения макроса `vec!`

Примечание: фактическое определение макроса `vec!` в стандартной библиотеке включает сначала код для предварительного выделения правильного объёма памяти. Этот код является оптимизацией, которую мы здесь не включаем, чтобы сделать пример проще.

Аннотация `#[macro_export]` указывает, что данный макрос должен быть доступен всякий раз, когда крейт с объявлением макросом, добавлен в область видимости. Без этой аннотации макрос нельзя добавить в область видимости.

Затем мы начинаем объявление макроса с помощью `macro_rules!` и имени макроса, который объявляется без восклицательного знака. Название, в данном случае `vec`, после которого следуют фигурные скобки, указывающие тело определения макроса.

Структура в теле макроса `vec!` похожа на структуру `match` выражения. Здесь у нас есть одна ветвь с шаблоном `($($x:expr),*)`, затем следует ветвь `=>` и блок кода, связанный с этим шаблоном. Если шаблон сопоставлен успешно, то соответствующий блок кода будет сгенерирован. Учитывая, что данный код является единственным шаблоном в этом макросе, существует только один

действительный способ сопоставления, любой другой шаблон приведёт к ошибке. Более сложные макросы будут иметь более чем одна ветвь.

Допустимый синтаксис шаблона в определениях макросов отличается от синтаксиса шаблона рассмотренного в главе 18, потому что шаблоны макроса сопоставляются со структурами кода Rust, а не со значениями. Давайте пройдёмся по тому, какие части шаблона в листинге 19-28 что означают; полный синтаксис макроса см. [в ссылке](#).

Во-первых, набор скобок охватывает весь шаблон. Далее идёт знак доллара (`$`), затем следует набор скобок, который захватывает значения, соответствующие шаблону в скобках для использования в коде замены. Внутри `$()` находится `$x:expr`, который соответствует любому выражению Rust и даёт выражению имя `$x`.

Запятая, следующая за `$()` указывает на то, что буквенный символ-разделитель запятой может дополнительно появиться после кода, который соответствует коду в `$()`. Звёздочка `*` указывает, что шаблон соответствует ноль или больше раз тому, что предшествует `*`.

Когда вызывается этот макрос с помощью `vec![1, 2, 3];` шаблон `$x` соответствует три раза всем трём выражениям `1`, `2` и `3`.

Теперь давайте посмотрим на шаблон в теле кода, связанного с этой ветвью: `temp_vec.push()` внутри `$()*` генерируется для каждой части, которая соответствует символу `$()` в шаблоне ноль или более раз в зависимости от того, сколько раз шаблон сопоставлен. Символ `$x` заменяется на каждое совпадающее выражение. Когда мы вызываем этот макрос с `vec![1, 2, 3];`, сгенерированный код, заменяющий этот вызов макроса будет следующим:

```
{  
    let mut temp_vec = Vec::new();  
    temp_vec.push(1);  
    temp_vec.push(2);  
    temp_vec.push(3);  
    temp_vec  
}
```

Мы определили макрос, который может принимать любое количество аргументов любого типа и может генерировать код для создания вектора, содержащего указанные элементы.

Есть несколько странных краевых случаев у макроса `macro_rules!`. В будущем у Rust будет второй вид декларативного макроса, который будет работать аналогичным образом, но поправит некоторые из этих краевых случаев. После этого обновления `macro_rules!` будет фактически устаревшим. Имея это в виду, а также тот факт, что большинство Rust программистов будут использовать макросы больше, чем сами писать макросы, мы далее не будем обсуждать `macro_rules!`. Чтобы узнать больше о том, как писать макросы, обратитесь к электронной документации или другим ресурсам, таким как ["The Little Book of Rust Macros"](#).

Процедурные макросы для генерации кода из атрибутов

Вторая форма макросов - это *процедурные макросы* (procedural macros), которые действуют как функции (и являются типом процедуры). Процедурные макросы принимают некоторый код в качестве входных данных, работают над этим кодом и создают некоторый код в качестве вывода, а не выполняют сопоставления с шаблонами и замену кода другим кодом, как это делают декларативные макросы.

Все три вида процедурных макросов (пользовательские выводимые, похожие на атрибуты и похожие на функции) все работают аналогично.

При создании процедурных макросов объявления должны находиться в собственном крейте специального типа. Это из-за сложных технических причин, которые мы надеемся будут устранены в будущем. Использование процедурных макросов выглядит как код в листинге 19-29, где `some_attribute` является заполнителем для использования специального макроса.

Файл: src/lib.rs

```
use proc_macro;

#[some_attribute]
pub fn some_name(input: TokenStream) -> TokenStream {
}
```

Листинг 19-29: Пример использования процедурного макроса

Функция, которая определяет процедурный макрос, принимает `TokenStream` в качестве входных данных и создаёт `TokenStream` в качестве вывода. Тип `TokenStream` объявлен крейтом `proc_macro`, включённым в Rust и представляет собой последовательность токенов. Это ядро макроса: исходный код над которым

работает макрос, является входным `TokenStream`, а код создаваемый макросом является выходным `TokenStream`. К функции имеет также прикреплённый атрибут, определяющий какой тип процедурного макроса мы создаём. Можно иметь несколько видов процедурных макросов в одном и том же крейте.

Давайте посмотрим на различные виды процедурных макросов. Начнём с пользовательского, выводимого (`derive`) макроса и затем объясним небольшие различия, делающие другие формы отличающимися.

Как написать пользовательский `derive` макрос

Давайте создадим крейт с именем `hello_macro`, который определяет типаж с именем `HelloMacro` и имеет одну с ним ассоциированную функцию с именем `hello_macro`. Вместо того, чтобы пользователи нашего крейта самостоятельно реализовывали типаж `HelloMacro` для каждого из своих типов, мы предоставим им процедурный макрос, чтобы они могли аннотировать свой тип с помощью атрибута `#[derive(HelloMacro)]` и получили реализацию по умолчанию для функции `hello_macro`. Реализация по умолчанию выведет `Hello, Macro! My name is TypeName!`, где `TypeName` - это имя типа, для которого был определён этот типаж. Другими словами, мы напишем крейт, использование которого позволит другому программисту писать код показанный в листинге 19-30.

Файл: `src/main.rs`

```
use hello_macro::HelloMacro;
use hello_macro_derive::HelloMacro;

#[derive(HelloMacro)]
struct Pancakes;

fn main() {
    Pancakes::hello_macro();
}
```



Листинг 19-30: Код, который сможет писать пользователь нашего крейта при использовании нашего процедурного макроса

Этот код напечатает `Hello, Macro! My name is Pancakes!`, когда мы закончим. Первый шаг - создать новый, библиотечный крейт так:

```
$ cargo new hello_macro --lib
```

Далее, мы определим типаж `HelloMacro` и ассоциированную с ним функцию:

Файл: `src/lib.rs`

```
pub trait HelloMacro {  
    fn hello_macro();  
}
```

У нас есть типаж и его функция. На этом этапе пользователь крейта может реализовать типаж для достижения желаемой функциональности, так:

```
use hello_macro::HelloMacro;  
  
struct Pancakes;  
  
impl HelloMacro for Pancakes {  
    fn hello_macro() {  
        println!("Hello, Macro! My name is Pancakes!");  
    }  
}  
  
fn main() {  
    Pancakes::hello_macro();  
}
```

Тем не менее, ему придётся написать блок реализации для каждого типа, который он хотел использовать вместе с `hello_macro`; а мы хотим избавить их от необходимости делать эту работу.

Кроме того, мы пока не можем предоставить функцию `hello_macro` с реализацией по умолчанию, которая будет печатать имя типа, для которого реализован типаж: Rust не имеет возможностей рефлексии (reflection), поэтому он не может выполнить поиск имени типа во время выполнения кода. Нам нужен макрос для генерации кода во время компиляции.

Следующим шагом является определение процедурного макроса. На момент написания этой статьи процедурные макросы должны быть в собственном крейте. Со временем это ограничение может быть отменено. Соглашение о структурировании крейтов и макросов является следующим: для крейта с именем `foo`, его пользовательский, крейт с выводимым процедурным макросом называется `foo_derive`. Давайте начнём с создания нового крейта с именем `hello_macro_derive` внутри проекта `hello_macro`:

```
$ cargo new hello_macro_derive --lib
```

Наши два крейта тесно связаны, поэтому мы создаём процедурный макрос-крайт в каталоге крейта `hello_macro`. Если мы изменим определение типажа в `hello_macro`, то нам придётся также изменить реализацию процедурного макроса в `hello_macro_derive`. Два крейта нужно будет опубликованы отдельно и программисты, использующие эти крейты, должны будут добавить их как зависимости, а затем добавить их в область видимости. Мы могли вместо этого сделать так, что крейт `hello_macro` использует `hello_macro_derive` как зависимость и реэкспортирует код процедурного макроса. Однако то, как мы структурировали проект, делает возможным программистам использовать `hello_macro` даже если они не хотят `derive` функциональность.

Нам нужно объявить крейт `hello_macro_derive` как процедурный макрос-крайт. Также понадобятся функционал из крейтов `syn` и `quote`, как вы увидите через мгновение, поэтому нам нужно добавить их как зависимости. Добавьте следующее в файл `Cargo.toml` для `hello_macro_derive`:

Файл: `hello_macro_derive/Cargo.toml`

```
[lib]
proc-macro = true

[dependencies]
syn = "1.0"
quote = "1.0"
```

Чтобы начать определение процедурного макроса, поместите код листинга 19-31 в ваш файл `src/lib.rs` крейта `hello_macro_derive`. Обратите внимание, что этот код не скомпилируется пока мы не добавим определение для функции `impl_hello_macro`.

Файл: `hello_macro_derive/src/lib.rs`

```
use proc_macro::TokenStream;
use quote::quote;
use syn;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // Construct a representation of Rust code as a syntax tree
    // that we can manipulate
    let ast = syn::parse(input).unwrap();

    // Build the trait implementation
    impl_hello_macro(&ast)
}
```



Листинг 19-31: Код, который потребуется в большинстве процедурных макро крейтов для обработки Rust кода

Обратите внимание, что мы разделили код на функцию `hello_macro_derive`, которая отвечает за синтаксический анализ `TokenStream` и функцию `impl_hello_macro`, которая отвечает за преобразование синтаксического дерева: это делает написание процедурного макроса удобнее. Код во внешней функции (`hello_macro_derive` в данном случае) будет одинаковым для почти любого процедурного макроса крейта, который вы видите или создаёте. Код, который вы указываете в теле внутренней функции (в данном случае `impl_hello_macro`) будет отличаться в зависимости от цели вашего процедурного макроса.

Мы представили три новых крейта: `proc_macro` `syn` и `quote`. Макрос `proc_macro` поставляется с Rust, поэтому нам не нужно было добавлять его в зависимости внутри `Cargo.toml`. Макрос `proc_macro` - это API компилятора, который позволяет нам читать и манипулировать Rust кодом из нашего кода.

Крейт `syn` разбирает Rust код из строки в структуру данных над которой мы можем выполнять операции. Крейт `quote` превращает структуры данных `syn` обратно в код Rust. Эти крейты упрощают разбор любого вида Rust кода, который мы хотели бы обрабатывать: написание полного синтаксического анализатора для кода Rust не является простой задачей.

Функция `hello_macro_derive` будет вызываться, когда пользователь нашей библиотеки указывает своему типу `#[derive(HelloMacro)]`. Это возможно, потому что мы аннотировали функцию `hello_macro_derive` с помощью `proc_macro_derive` и указали имя `HelloMacro`, которое соответствует имени нашего типажа; это соглашение, которому следуют большинство процедурных макросов.

Функция `hello_macro_derive` сначала преобразует `input` из `TokenStream` в структуру данных, которую мы можем затем интерпретировать и над которой выполнять операции. Здесь крейт `syn` вступает в игру. Функция `parse` в `syn` принимает `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;`:

```
DeriveInput {
    // --snip--

    ident: Ident {
        ident: "Pancakes",
        span: #0 bytes(95..103)
    },
    data: Struct(
        DataStruct {
            struct_token: Struct,
            fields: Unit,
            semi_token: Some(
                Semi
            )
        }
    )
}
```

Листинг 19-32: Экземпляр `DeriveInput` получаемый, когда разбирается код имеющий атрибут макроса из Листинга 19-30

Поля этой структуры показывают, что код Rust, который мы разбрали, является блок структуры с `ident` (идентификатором, означающим имя) для `Pancakes`. Есть больше полей в этой структуре для описания всех видов кода Rust; проверьте [документацию](#) `syn` о структуре `DeriveInput` для получения дополнительной информации.

Вскоре мы определим функцию `impl_hello_macro`, в которой построим новый, дополнительный код Rust. Но прежде чем мы это сделаем, обратите внимание, что выводом для нашего выводимого (derive) макроса также является `TokenStream`. Возвращаемый `TokenStream` добавляется в код, написанный пользователями макроса, поэтому, когда они соберут свой крейт, они получат дополнительную функциональность, которую мы предоставляем в изменённом `TokenStream`.

Возможно, вы заметили, что мы вызываем `unwrap` чтобы выполнить панику в функции `hello_macro_derive`, если вызов функции `syn::parse` потерпит неудачу.

Наш процедурный макрос должен паниковать при ошибках, потому что функции `proc_macro_derive` должны возвращать `TokenStream`, а не тип `Result` для соответствия API процедурного макроса. Мы упростили этот пример с помощью `unwrap`, но в рабочем коде вы должны предоставить более конкретные сообщения об ошибках, если что-то пошло не правильно, используя `panic!` или `expect`.

Теперь, когда у нас есть код для преобразования аннотированного Rust кода из `TokenStream` в экземпляр `DeriveInput`, давайте сгенерируем код реализующий типаж `HelloMacro` у аннотированного типа, как показано в листинге 19-33.

Файл: `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!("Hello, Macro! My name is {}!", stringify!(#name));
            }
        };
    };
    gen.into()
}
```

Листинг 19-33. Реализация типажа `HelloMacro` с использованием проанализированного кода Rust.

Мы получаем экземпляр структуры `Ident` содержащий имя (идентификатор) аннотированного типа с использованием `ast.ident`. Структура в листинге 19-32 показывает, что когда мы запускаем функцию `impl_hello_macro` для кода из листинга 19-30, то получаемый `ident` будет иметь поле `ident` со значением `"Pancakes"`. Таким образом, переменная `name` в листинге 19-33 будет содержать экземпляр структуры `Ident`, что при печати выдаст строку `"Pancakes"`, что является именем структуры в листинге 19-30.

Макрос `quote!` позволяет определить код Rust, который мы хотим вернуть. Компилятор ожидает что-то отличное от прямого результата выполнения макроса `quote!`, поэтому нужно преобразовать его в `TokenStream`. Мы делаем это путём вызова метода `into`, который использует промежуточное представление и возвращает значение требуемого типа `TokenStream`.

Макрос `quote!` также предоставляет очень классную механику шаблонов: мы можем ввести `#name` и `quote!` заменит его значением из переменной `name`. Вы

можете даже сделать некоторое повторение, подобное тому, как работают обычные макросы. Проверьте [документацию крейта quote](#) для подробного введения.

Мы хотим, чтобы наш процедурный макрос генерировал реализацию нашего типажа `HelloMacro` для типа, который аннотировал пользователь, который мы можем получить, используя `#name`. Реализация типажа имеет одну функцию `hello_macro`, тело которой содержит функциональность, которую мы хотим предоставить: напечатать `Hello, Macro! My name is` с именем аннотированного типа.

Макрос `stringify!` используемый здесь, встроен в Rust. Он принимает Rust выражение, такое как `1 + 2` и во время компиляции компилятор превращает выражение в строковый литерал, такой как `"1 + 2"`. Он отличается от макросов `format!` или `println!`, которые вычисляют выражение, а затем превращают результат в виде типа `String`. Существует возможность того, что введённый `#name` может оказаться выражением для печати буквально как есть, поэтому здесь мы используем `stringify!`. Использование `stringify!` также сохраняет выделение путём преобразования `#name` в строковый литерал во время компиляции.

На этом этапе команда `cargo build` должна завершиться успешно для обоих `hello_macro` и `hello_macro_derive`. Давайте подключим эти крейты к коду в листинге 19-30, чтобы увидеть процедурный макрос в действии! Создайте новый бинарный проект в каталоге ваших проектов с использованием команды `cargo new pancakes`. Нам нужно добавить `hello_macro` и `hello_macro_derive` в качестве зависимостей для крейта `pancakes` в файл `Cargo.toml`. Если вы публикуете свои версии `hello_macro` и `hello_macro_derive` на сайт [crates.io](#), они будут обычными зависимостями; если нет, вы можете указать их как `path` зависимости следующим образом:

```
hello_macro = { path = "../hello_macro" }
hello_macro_derive = { path = "../hello_macro/hello_macro_derive" }
```

Поместите код в листинге 19-30 в `src/main.rs` и выполните `cargo run`: он должен вывести `Hello, Macro! My name is Pancakes!`. Реализация типажа `HelloMacro` из процедурного макроса была включена без необходимости его реализации крейтом `pancakes`; `#[derive(HelloMacro)]` добавил реализацию типажа.

Далее давайте рассмотрим, как другие виды процедурных макросов отличаются от пользовательских выводимых макросов.

подобные атрибутам макросы

Подобные атрибутам макросы похожи на пользовательские выводимые макросы, но вместо генерации кода для `derive` атрибута, они позволяют создавать новые атрибуты. Они являются также более гибкими: `derive` работает только для структур и перечислений; атрибут-подобные могут применяться и к другим элементам, таким как функции. Вот пример использования атрибутного макроса: допустим, у вас есть атрибут именованный `route` который аннотирует функции при использовании фреймворка для веб-приложений:

```
#[route(GET, "/")]
fn index() {
```

Данный атрибут `#[route]` будет определён платформой как процедурный макрос. Сигнатура функции определения макроса будет выглядеть так:

```
#[proc_macro_attribute]
pub fn route(attr: TokenStream, item: TokenStream) -> TokenStream {
```

Здесь есть два параметра типа `TokenStream`. Первый для содержимого атрибута: часть `GET, "/"`. Второй это тело элемента, к которому прикреплён атрибут: в данном случае `fn index() {}` и остальная часть тела функции.

Кроме того, атрибутные макросы работают так же как и пользовательские выводимые макросы: вы создаёте крейт с типом `proc-macro` и реализуете функцию, которая генерирует код, который хотите!

Функционально подобные макросы

Функционально подобные макросы выглядят подобно вызову функций. Они аналогично макросам `macro_rules!` и являются более гибкими, чем функции; например, они могут принимать неизвестное количество аргументов. Тем не менее, макросы `macro_rules!` можно объявлять только с использованием синтаксиса подобного сопоставлению, который мы обсуждали ранее в разделе "Декларативные макросы `macro_rules!` для общего мета программирования". Функционально подобные макросы принимают параметр `TokenStream` и их определение манипулирует этим `TokenStream`, используя код Rust, как это делают два других типа процедурных макроса. Примером подобного функционально подобного макроса является макрос `sql! /code6`, который можно вызвать так:

```
let sql = sql!(SELECT * FROM posts WHERE id=1);
```

Этот макрос будет разбирать SQL оператор внутри него и проверять, что он синтаксически правильный, что является гораздо более сложной обработкой, чем то что может сделать макрос `macro_rules!`. Макрос `sql!` мог бы быть определён так:

```
#[proc_macro]
pub fn sql(input: TokenStream) -> TokenStream {
```

Это определение похоже на сигнатуру пользовательского выводимого макроса: мы получаем токены, которые находятся внутри скобок и возвращаем код, который мы хотели сгенерировать.

Итоги

Уф! Теперь у вас есть некоторые возможности Rust, которые вы не будете часто использовать, но вы будете знать, что они доступны в особых обстоятельствах. Мы представили несколько сложных тем, чтобы при появлении сообщения с предложением исправить ошибку или в коде других людей, вы могли бы распознать эти концепции и синтаксис. Используйте эту главу как справочник, который поможет вам в решениях.

Далее мы применим все, что мы обсуждали в книге и сделаем ещё один проект!

Финальный проект: создание многопоточного веб-сервера

Это был долгий путь, но мы дошли до финала книги. В этой главе мы создадим ещё один проект для демонстрации некоторых концепций, которые мы рассмотрели в последних главах, а также резюмировать некоторые предыдущие уроки.

Для нашего финального проекта мы создадим веб-сервер, который говорит "hello" и выглядит как рисунок 20-1 в веб-браузере.



Hello!

Hi from Rust

Рисунок 20-1: Наш последний совместный проект

Вот план по созданию веб-сервера:

1. Узнать немного о протоколах TCP и HTTP.
2. Прослушивать TCP соединения у сокета.
3. Разобрать небольшое количество HTTP-запросов.
4. Создать правильный HTTP ответ.
5. Улучшите пропускную способность нашего сервера с помощью пула потоков.

Но прежде чем мы начнём, мы должны упомянуть одну деталь. Способ который мы будем использовать не является лучшим способом построения веб-сервер в Rust. Несколько готовых к использованию крейтов доступны на [crates.io](#), и способны обеспечить более полную реализацию веб-сервера и пула потоков, чем сделаем мы

сами.

Однако в этой главе мы хотим помочь вам научиться, а не выбирать лёгкий путь. Поскольку Rust является языком системного программирования, мы можем выбрать тот уровень абстракции на котором мы хотим работать и можем перейти на более низкий уровень, чем возможно или практично для использования в других языках. Мы напишем базовый HTTP сервер и пул потоков вручную, чтобы вы могли изучить общие идеи и техники из крейтов, которые вы могли бы использовать в будущем.

Создание однопоточного веб-сервера

Начнём с однопоточного веб-сервера. Перед тем, как начать, давайте рассмотрим краткий обзор протоколов, задействованных в создании веб-серверов. Детальное описание этих протоколов выходит за рамки этой книги, но краткий обзор даст вам необходимую информацию.

Два основных протокола, задействованных в веб-серверах, - это *протокол передачи гипертекста (HTTP)* и *протокол управления передачей (TCP)*. Оба протокола являются протоколами типа *запрос-ответ*, что означает, что *клиент* инициирует запросы, а *сервер* слушает запросы и предоставляет ответ клиенту. Содержание этих запросов и ответов определяется протоколами.

TCP - это протокол нижнего уровня, который описывает детали того, как информация передаётся от одного сервера к другому, но не определяет, что это за информация. HTTP строится поверх TCP, определяя содержимое запросов и ответов. Технически возможно использовать HTTP с другими протоколами, но в подавляющем большинстве случаев HTTP отправляет свои данные поверх TCP. Мы будем работать с необработанными байтами в TCP и запросами и ответами в HTTP.

Прослушивание TCP соединения

Нашему веб-серверу необходимо прослушивать TCP-соединение, так что это первая часть, над которой мы будем работать. Стандартная библиотека предлагает для этого модуль `std::net`. Сделаем новый проект обычным способом:

```
$ cargo new hello
     Created binary (application) `hello` project
$ cd hello
```

Теперь введите код из Листинга 20-1 в `src/main.rs`, чтобы начать. Этот код будет использовать адрес `127.0.0.1:7878` для входящих TCP-потоков. Когда он получит входящее соединение, он напечатает `Connection established!`,

Файл: `src/main.rs`

```
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();

        println!("Connection established!");
    }
}
```

Листинг 20-1: Приём и прослушивание входящих потоков, печать сообщения, когда мы получаем поток

Используя `TcpListener`, можно прослушивать TCP соединения по адресу `127.0.0.1:7878`. В адресе, в его части перед двоеточием, сначала идёт IP-адрес представляя ваш компьютер (он одинаковый на каждом компьютере и не представляет конкретный компьютер автора), а часть `7878` является портом. Мы выбрали этот порт по двум причинам: HTTP обычно может принимать на этом порту, и 7878 - это слово *rust* набранное на телефоне.

Функция `bind` в этом сценарии работает так же, как функция `new`, поскольку она возвращает новый экземпляр `TcpListener`. Причина, по которой функция называется `bind` заключается в том, что в сетевой терминологии подключение к порту для прослушивания называется «привязка к порту».

Функция `bind` возвращает `Result<T, E>`, который указывает, что привязка может завершиться ошибкой. Например, для подключения к порту 80 требуются права администратора (не администраторы могут прослушивать только порты выше 1024), поэтому, если мы попытаемся подключиться к порту 80, не будучи администратором, привязка не сработает. Другой пример: привязка не сработает, если мы запустили два экземпляра нашей программы, и поэтому две программы будут прослушивать один и тот же порт. Поскольку мы пишем базовый сервер только в учебных целях, мы не будем беспокоиться об обработке таких ошибок; вместо этого мы используем `unwrap` чтобы остановить программу в случае возникновения ошибок.

Метод `incoming` в `TcpListener` возвращает итератор, который даёт нам последовательность потоков (конкретнее, потоков типа `TcpStream`). Один *поток* представляет собой открытое соединение между клиентом и сервером. *Соединение* - это полный процесс запроса и ответа, в котором клиент подключается к серверу,

сервер генерирует ответ, и сервер закрывает соединение. Таким образом, `TcpStream` позволяет прочитать из себя, то что отправил клиент, а затем позволяет записать наш ответ в поток. В целом, цикл `for` будет обрабатывать каждое соединение по очереди и создавать серию потоков, которые мы будем обрабатывать.

На данный момент обработка потока состоит из вызова `unwrap` для завершения программы, если поток имеет какие-либо ошибки, а если ошибок нет, то печатается сообщение. Мы добавим больше функциональности для случая успешной работы в следующем листинге кода. Причина, по которой мы можем получить ошибки из метода `incoming` при подключении клиента к серверу, является то, что мы на самом деле не перебираем соединения. Вместо этого мы перебираем *попытки подключения*. Соединение может быть не успешным по ряду причин, многие из них специфичны в операционной системе. Например, многие операционные системы имеют ограничение количества одновременных открытых соединений, которые они поддерживают; попытки создания нового соединения, превышающее это число, будут приводить к ошибкам до тех пор, пока некоторые из ранее открытых соединений не будут закрыты.

Попробуем запустить этот код! Вызовите `cargo run` в терминале, а затем загрузите `127.0.0.1:7878` в веб-браузере. В браузере должно отображаться сообщение об ошибке, например «Connection reset», поскольку сервер в настоящее время не отправляет обратно никаких данных. Но когда вы посмотрите на свой терминал, вы должны увидеть несколько сообщений, которые были напечатаны, когда браузер подключался к серверу!

```
Running `target/debug/hello`  
Connection established!  
Connection established!  
Connection established!
```

Иногда вы видите несколько сообщений, напечатанных для одного запроса браузера; Причина может заключаться в том, что браузер выполняет запрос страницы, а также других ресурсов, таких как значок `favicon.ico`, который отображается на вкладке браузера.

Также может быть, что браузер пытается подключиться к серверу несколько раз, потому что сервер не отвечает. Когда `stream` выходит из области видимости и отбрасывается в конце цикла, соединение закрывается как часть реализации `drop`. Браузеры иногда обрабатывают закрытые соединения, повторяя попытки, потому что проблема может быть временной. Важным фактором является то, что мы

успешно получили дескриптор TCP-соединения!

Не забудьте остановить программу, нажав `ctrl-c`, когда вы закончите запускать определённую версию кода. Затем перезапустите `cargo run` после того, как вы внесли следующий набор изменений, чтобы убедиться, что вы используете самый новый код.

Чтение запросов

Реализуем функционал чтения запроса из браузера! Чтобы разделить части, связанные с получением соединения и последующим действием с ним, мы запустим новую функцию для обработки соединения. В этой новой функции `handle_connection` мы будем читать данные из потока TCP и распечатывать их, чтобы мы могли видеть данные, отправленные из браузера. Измените код, чтобы он выглядел как в листинге 20-2.

Файл: `src/main.rs`

```
use std::{
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
};

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        handle_connection(stream);
    }
}

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    println!("Request: {:#?}", http_request);
}
```

Листинг 20-2: Чтение из потока `TcpStream` и печать данных

Мы добавляем `std::io::prelude` в область видимости, чтобы получить доступ к определённым свойствам, которые позволяют нам читать и писать в поток. В цикле `for` функции `main` вместо вывода сообщения о том, что мы установили соединение, мы теперь вызываем новую функцию `handle_connection` и передаём ей `stream`.

В функции `handle_connection` мы сделали параметр `stream` изменяемым. Причина в том, что экземпляр `TcpStream` отслеживает, какие данные он нам возвращает. Он может прочитать больше данных, чем мы запрашивали, и сохранить их для следующего раза, когда мы запросим данные. Следовательно, он должен быть `mut` поскольку его внутреннее состояние может измениться; Обычно мы думаем, что «чтение» не требует мутации, но в этом случае нам нужно ключевое слово `mut`.

Далее нам нужно фактически прочитать данные из потока. Мы делаем это в два этапа: во-первых, мы объявляем `buffer` в стеке для хранения считываемых данных. Мы сделали буфер размером 1024 байта, что достаточно для хранения данных базового запроса и достаточно для наших целей в этой главе. Если бы мы хотели обрабатывать запросы произвольного размера, управление буфером должно было бы быть более сложным; пока делаем проще. Мы передаём буфер в `stream.read`, который считывает байты из `TcpStream` и помещает их в буфер.

Во-вторых, мы конвертируем байты из буфера в строку и печатаем эту строку. Функция `String::from_utf8_lossy` принимает `&[u8]` и создаёт из неё `String`. Названия «*lossy*» (с потерями) в её имени указывает на поведение этой функции. Когда она видит недопустимую последовательность UTF-8: она заменяет недопустимую последовательность на символ  (U+FFFD), символ замены `REPLACEMENT CHARACTER`. Вы могли видеть заменяющие символы в буфере, который не заполнен данными из запроса.

Попробуем этот код! Запустите программу и снова сделайте запрос в веб-браузере. Обратите внимание, что мы по-прежнему будем получать в браузере страницу с ошибкой, но вывод нашей программы в терминале теперь будет выглядеть примерно так:

В зависимости от вашего браузера результат может немного отличаться. Теперь, когда мы печатаем данные запроса, мы можем понять, почему мы получаем несколько подключений из одного запроса браузера, посмотрев на путь после **Request: GET**. Если все повторяющиеся соединения запрашивают /, мы знаем, что браузер пытается получить / повторно, потому что он не получает ответа от нашей программы.

Давайте разберём эти данные запроса, чтобы понять, что браузер запрашивает у нашей программы.

Пристальный взгляд на HTTP запрос

HTTP - это текстовый протокол и запрос имеет следующий формат:

Method Request-URI HTTP-Version CRLF
headers CRLF
message-body

Первая строка - это строка запроса, содержащая информацию о том, что запрашивает клиент. Первая часть строки запроса указывает используемый метод, например **GET** или **POST**, который описывает, как клиент выполняет этот запрос. Наш клиент использовал запрос **GET**.

Следующая часть строки запроса - это `/`, которая указывает унифицированный идентификатор ресурса (*URI*), который запрашивает клиент: *URI* почти, но не совсем то же самое, что и унифицированный указатель ресурса (*URL*). Разница между *URI* и *URL*-адресами не важна для наших целей в этой главе, но спецификация HTTP

использует термин URI, поэтому мы можем просто мысленно заменить URL-адрес здесь.

Последняя часть - это версия HTTP, которую использует клиент, а затем строка запроса заканчивается *последовательностью CRLF*. (CRLF обозначает *возврат каретки и перевод строки*, что является термином из дней пишущих машинок!). Последовательность CRLF также может быть записана как `\r\n`, где `\r` - возврат каретки, а `\n` - перевод строки. Последовательность CRLF отделяет строку запроса от остальных данных запроса. Обратите внимание, что при печати CRLF мы видим начало новой строки, а не `\r\n`.

Глядя на данные строки запроса, которые мы получили от запуска нашей программы, мы видим, что `GET` - это метод, `/` - это URI запроса, а `HTTP/1.1` - это версия.

После строки запроса оставшиеся строки, начиная с `Host:` далее, являются заголовками. `GET` запросы 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, когда мы знаем, что запрашивает браузер, давайте отправим обратно в ответ 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 - это *строка состояния*, которая содержит версию HTTP, используемую в ответе, числовой код состояния, который суммирует результат запроса, и фразу причины, которая предоставляет текстовое описание кода состояния. После последовательности CRLF идут любые заголовки, другая последовательность CRLF и тело ответа.

Вот пример ответа, который использует HTTP версии 1.1, имеет код состояния 200, фразу причины OK, без заголовков и без тела:

```
HTTP/1.1 200 OK\r\n\r\n
```

Код состояния 200 - это стандартный успешный ответ. Текст представляет собой крошечный успешный HTTP-ответ. Давайте запишем это в поток как наш ответ на успешный запрос! Из функции `handle_connection` удалите `println!`, который печатал данные запроса и заменил их кодом из Листинга 20-3.

Файл: src/main.rs

```
fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    let response = "HTTP/1.1 200 OK\r\n\r\n";

    stream.write_all(response.as_bytes()).unwrap();
}
```

Листинг 20-3: Запись короткого успешного HTTP ответа в поток

Первая новая строка определяет переменную `response`, которая содержит данные сообщения об успешном выполнении. Затем мы вызываем `as_bytes` в нашем `response`, чтобы преобразовать строковые данные в байты. Метод `write` в `stream` принимает `&[u8]` и отправляет эти байты напрямую по соединению.

Поскольку операция `write` может завершиться неудачно, мы, как и раньше, используем `unwrap` для любого результата ошибки. Опять же, в реальном приложении вы бы добавили сюда обработку ошибок. Наконец, `flush` подождёт и предотвратит продолжение программы, пока все байты не будут записаны в соединение; `TcpStream` содержит внутренний буфер для минимизации обращений к базовой операционной системе.

Сделав этим изменения давайте запустим код и сделаем запрос. Мы больше не выводим в терминал любые данные, поэтому мы не увидим ничего, кроме вывода из Cargo. Когда вы загружаете адрес `127.0.0.1:7878` в веб-браузер, вы должны получить пустую страницу вместо ошибки. Вы только что вручную закодировали запрос и ответ HTTP!

Возвращение реального HTML

Реализуем функционал для возврата более пустой страницы. Создайте новый файл *hello.html* в корне каталога вашего проекта, а не в каталоге *src*. Вы можете ввести любой HTML-код; В листинге 20-4 показана одна возможность.

Файл: *hello.html*

```
{#{include ../../listings/ch20-web-server/listing-20-04/hello.html}}
```

Листинг 20-4. Образец HTML-файла для возврата в ответ

Это минимальный документ HTML5 с заголовком и некоторым текстом. Чтобы вернуть это с сервера при получении запроса, мы **handle_connection** как показано в листинге 20-5, чтобы прочитать файл HTML, добавить его в ответ в виде тела и отправить.

Файл: *src/main.rs*

```
use std::{
    fs,
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
};

// --snip--


fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    let status_line = "HTTP/1.1 200 OK";
    let contents = fs::read_to_string("hello.html").unwrap();
    let length = contents.len();

    let response =
        format!("{}Content-Length:{}\r\n\r\n{}", status_line, length, contents);

    stream.write_all(response.as_bytes()).unwrap();
}
```

Листинг 20-5. Отправка содержимого *hello.html* в качестве тела ответа

Мы добавили строку вверху, чтобы включить в область видимости модуль файловой системы стандартной библиотеки. Код для чтения содержимого файла в строку должен выглядеть знакомо; мы использовали его в главе 12, когда читали содержимое файла для нашего проекта ввода-вывода в листинге 12-4.

Далее мы используем `format!` чтобы добавить содержимое файла в качестве тела ответа об успешном завершении. Чтобы гарантировать действительный HTTP-ответ, мы добавляем заголовок `Content-Length` который имеет размер тела нашего ответа, в данном случае размер `hello.html`.

Запустите этот код командой `cargo run` и загрузите `127.0.0.1:7878` в браузере; вы должны увидеть выведенный HTML в браузере!

В настоящее время мы игнорируем данные запроса в `buffer` и просто безоговорочно отправляем обратно содержимое HTML-файла. Это означает, что если вы попытаетесь запросить `127.0.0.1:7878/something-else` в своём браузере, вы все равно получите тот же ответ HTML. Наш сервер очень ограничен, и это не то, что делает большинство веб-серверов. Мы хотим настроить наши ответы в зависимости от запроса и отправлять обратно HTML-файл только для правильно сформированного запроса в `/`.

Проверка запроса и выборочное возвращение ответа

Прямо сейчас наш веб-сервер вернёт HTML-код в файле независимо от того, что запросил клиент. Давайте добавим функциональность, чтобы проверять, запрашивает ли браузер `/` перед возвратом HTML-файла, и возвращать ошибку, если браузер запрашивает что-либо ещё. Для этого нам нужно изменить `handle_connection`, как показано в листинге 20-6. Этот новый код проверяет содержимое полученного запроса на соответствие тому, как мы знаем, что запрос на `/` выглядит как, и добавляет блоки `if` и `else` чтобы обрабатывать запросы по-разному.

Файл: `src/main.rs`

```
// --snip--  
  
fn handle_connection(mut stream: TcpStream) {  
    let buf_reader = BufReader::new(&mut stream);  
    let request_line = buf_reader.lines().next().unwrap().unwrap();  
  
    if request_line == "GET / HTTP/1.1" {  
        let status_line = "HTTP/1.1 200 OK";  
        let contents = fs::read_to_string("hello.html").unwrap();  
        let length = contents.len();  
  
        let response = format!(  
            "{}\r\nContent-Length: {}\r\n\r\n{}",  
            status_line, length, contents  
        );  
  
        stream.write_all(response.as_bytes()).unwrap();  
    } else {  
        // some other request  
    }  
}
```

Листинг 20-6: Сопоставление запроса и обработка запросов для корневого ресурса `/`, отличающимся от запросов других ресурсов

Сначала мы жёстко кодируем данные, соответствующие запросу `/`, в переменную `get`. Поскольку мы читаем необработанные байты в буфер, мы преобразуем `get` в байтовую строку, добавляя синтаксис байтовой строки `b""` в начало данных содержимого. Затем мы проверяем, начинается ли `buffer` с байтов в `get`. Если это так, это означает, что мы получили правильно сформированный запрос к `/`, и это успешный случай, который мы обрабатываем в блоке `if` который возвращает содержимое нашего HTML-файла.

Если `buffer` не начинается с байтов в `get`, это означает, что мы получили другой запрос. Мы добавим код в блок `else` через мгновение, чтобы ответить на все остальные запросы.

Запустите этот код сейчас и запросите `127.0.0.1:7878`; вы должны получить HTML в `hello.html`. Если вы сделаете любой другой запрос, например `127.0.0.1:7878/something-else`, вы получите ошибку соединения, подобную той, которую вы видели при запуске кода из Листинга 20-1 и Листинга 20-2.

Теперь давайте добавим код из листинга 20-7 в блок `else` чтобы вернуть ответ с кодом состояния 404, который сигнализирует о том, что контент для запроса не найден. Мы также вернём HTML-код для страницы, отображаемой в браузере, с

указанием ответа конечному пользователю.

Файл: src/main.rs

```
// --snip--  
} else {  
    let status_line = "HTTP/1.1 404 NOT FOUND";  
    let contents = fs::read_to_string("404.html").unwrap();  
    let length = contents.len();  
  
    let response = format!(  
        "{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}"  
    );  
  
    stream.write_all(response.as_bytes()).unwrap();  
}
```

Листинг 20-7. Ответ с кодом состояния 404 и страницей с ошибкой, если было запрошено что-либо, кроме /

Здесь ответ имеет строку состояния с кодом 404 и фразу причины **NOT FOUND**. Тело ответа будет HTML из файла *404.html*. Вам нужно создать файл *404.html* рядом с *hello.html* для этой страницы ошибки; снова не стесняйтесь использовать любой HTML код или пример HTML кода в листинге 20-8.

Файл: 404.html

```
{#include ..../listings/ch20-web-server/listing-20-08/404.html} }
```

Листинг 20-8. Пример содержимого страницы для отправки с любым ответом 404

С этими изменениями снова запустите сервер. Запрос на `127.0.0.1:7878` должен возвращать содержимое `hello.html`, и любой другой запрос, как `127.0.0.1:7878/foo`, должен возвращать сообщение об ошибке HTML от `404.html`.

Рефакторинг

В настоящий момент блоки `if` и `else` часто повторяются: они читают файлы и записывают содержимое файлов в поток. Единственные различия - это строка состояния и имя файла. Давайте сделаем код более кратким, выделив эти различия в отдельные строки `if` и `else`, которые будут назначать значения строки состояния и имени файла переменным; затем мы можем безоговорочно использовать эти

переменные в коде для чтения файла и записи ответа. В листинге 20-9 показан код, полученный после замены больших блоков `if` и `else`.

Файл: src/main.rs

```
// --snip--  
  
fn handle_connection(mut stream: TcpStream) {  
    // --snip--  
  
    let (status_line, filename) = if request_line == "GET / HTTP/1.1" {  
        ("HTTP/1.1 200 OK", "hello.html")  
    } else {  
        ("HTTP/1.1 404 NOT FOUND", "404.html")  
    };  
  
    let contents = fs::read_to_string(filename).unwrap();  
    let length = contents.len();  
  
    let response =  
        format!("{}Content-Length:{}\r\n\r\n{}", status_line, length, contents);  
  
    stream.write_all(response.as_bytes()).unwrap();  
}
```

Листинг 20-9. Реорганизация блоков `if` и `else` чтобы они содержали только код, который отличается в двух случаях.

Теперь блоки `if` и `else` возвращают только соответствующие значения для строки состояния и имени файла в кортеже; Затем мы используем деструктурирование, чтобы присвоить эти два значения `status_line` и `filename` используя шаблон в операторе `let`, как обсуждалось в главе 18.

Ранее дублированный код теперь находится вне блоков `if` и `else` и использует переменные `status_line` и `filename`. Это позволяет легче увидеть разницу между этими двумя случаями и означает, что у нас есть только одно место для обновления кода, если захотим изменить работу чтения файлов и записи ответов. Поведение кода в листинге 20-9 будет таким же, как и в 20-8.

Потрясающие! Теперь у нас есть простой веб-сервер примерно на 40 строках кода Rust, который отвечает на один запрос страницей с контентом и отвечает на все остальные запросы ответом 404.

В настоящее время наш сервер работает в одном потоке, что означает, что он

может обслуживать только один запрос за раз. Давайте посмотрим, как это может быть проблемой, смоделировав несколько медленных запросов. Затем мы исправим это, чтобы наш сервер мог обрабатывать несколько запросов одновременно.

Превращение однопоточного сервера в многопоточный сервер

Прямо сейчас сервер будет обрабатывать каждый запрос в очереди, что означает, что он не будет обрабатывать второе соединение, пока первое не завершит обработку. Если бы сервер получал все больше и больше запросов, это последовательное выполнение было бы все менее и менее оптимальным. Если сервер получает какой-то запрос, обработка которого занимает слишком много времени, то последующие запросы должны будут ждать завершения обработки длительного запроса, даже если эти новые запросы могут быть обработаны гораздо быстрее. Нам нужно это исправить, но сначала мы рассмотрим проблему в действии.

Имитация медленного запроса в текущей реализации сервера

Мы посмотрим, как запрос с медленной обработкой может повлиять на другие запросы, сделанные к серверу в текущей реализации. В листинге 20-10 реализована обработка запроса к ресурсу `/sleep` с эмуляцией медленного ответа, который заставит сервер не работать в течение 5 секунд перед ответом.

Файл: `src/main.rs`

```
use std::{
    fs,
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
    thread,
    time::Duration,
};

// --snip--


fn handle_connection(mut stream: TcpStream) {
    // --snip--


    let (status_line, filename) = match &request_line[..] {
        "GET / HTTP/1.1" => ("HTTP/1.1 200 OK", "hello.html"),
        "GET /sleep HTTP/1.1" => {
            thread::sleep(Duration::from_secs(5));
            ("HTTP/1.1 200 OK", "hello.html")
        }
        _ => ("HTTP/1.1 404 NOT FOUND", "404.html"),
    };

    // --snip--
}
```

Листинг 20-10: Имитация медленного запроса путём распознавания обращения к `/sleep` и засыпанию на 5 секунд

Этот код немного неряшливый, но он достаточно хорошо подходит для целей имитации. Мы создали второй запрос `sleep`, данные которого распознает сервер. Мы добавили `else if` после блока `if`, чтобы проверить запрос к `/sleep`. Когда этот запрос будет получен, сервер заснёт на 5 секунд, прежде чем отобразить HTML страницу успешного выполнения.

Можно увидеть, насколько примитивен наш сервер: реальные библиотеки будут обрабатывать распознавание нескольких запросов гораздо менее многословно!

Запустите сервер командой `cargo run`. Затем откройте два окна браузера: одно с адресом `http://127.0.0.1:7878/`, другое с `http://127.0.0.1:7878/sleep`. Если вы несколько раз обратитесь к URI `/`, то как и раньше увидите, что сервер быстро ответит. Но если вы введёте URI `/sleep`, затем загрузите URI `/`, то увидите что `/` ждёт, пока `/sleep` не отработает полные 5 секунд перед загрузкой страницы.

Есть несколько способов изменить работу нашего веб-сервера, чтобы избежать медленной обработки большого количества запросов из-за одного медленного; способ который мы реализуем является пулом потоков.

Улучшение пропускной способности с помощью пула потоков

Пул потоков является группой заранее порождённых потоков, ожидающих в пуле и готовых выполнить задачу. Когда программа получает новую задачу, она назначает задачу одному из потоков в пуле и этот поток будет обрабатывать задачу. Остальные потоки в пуле доступны для обработки любых других задач, возникающих во время обработки первого потока. Когда первый поток завершает обработку своей задачи, он возвращается в пул свободных потоков, готовых обработать новую задачу. Пул потоков позволяет обрабатывать соединения одновременно, увеличивая пропускную способность вашего сервера.

Мы ограничим число потоков в пуле небольшим числом, чтобы защитить нас от атак типа «отказ в обслуживании» (DoS - Denial of Service); если бы наша программа создавала новый поток в момент поступления каждого запроса, то кто-то сделавший 10 миллионов запросов к серверу, мог бы создать хаос, использовать все ресурсы нашего сервера и остановить обработку запросов.

Вместо порождения неограниченного количества потоков, у нас будет фиксированное количество потоков, ожидающих в пуле. По мере поступления запросов они будут отправляться в пул для обработки. Пул будет поддерживать очередь входящих запросов. Каждый из потоков в пуле будет извлекать запрос из этой очереди, обрабатывать запрос и затем запрашивать в очереди следующий запрос. При таком дизайне мы можем обрабатывать N запросов одновременно, где N - количество потоков. Если каждый поток отвечает на длительный запрос, последующие запросы могут по-прежнему задержаться в очереди, но мы увеличили число долго играющих запросов, которые можно обработать до достижения этой точки.

Этот подход является лишь одним из многих способов улучшить пропускную способность веб-сервера. Другими вариантами, которые вы могли бы изучить являются модель fork/join и однопоточная модель асинхронного ввода-вывода. Если вам интересна эта тема, вы можете прочитать о других решениях больше и попробовать внедрить их в помощь Rust. С языком низкого уровня как Rust, возможны все эти варианты.

Прежде чем приступить к реализации пула потоков, давайте поговорим о том, как должно выглядеть использование пула. Когда вы пытаетесь проектировать код, сначала необходимо написать клиентский интерфейс. Напишите API кода, чтобы он был структурирован так, как вы хотите его вызывать, затем реализуйте функциональность данной структуры, вместо подхода реализовывать функционал, а затем разрабатывать общедоступный API.

Подобно тому, как мы использовали разработку через тестирование (test-driven) в проекте главы 12, мы будем использовать здесь разработку, управляемую компилятором (compiler-driven). Мы напишем код, который вызывает нужные нам функции, а затем посмотрим на ошибки компилятора, чтобы определить, что мы должны изменить дальше, чтобы заставить код работать.

Структура кода, если мы могли бы создавать поток для каждого запроса

Сначала давайте рассмотрим, как мог бы выглядеть код, если он создавал бы новый поток для каждого соединения. Как упоминалось ранее, это не окончательный план, а это отправная точка из-за проблем с возможным порождением неограниченного количества потоков. В листинге 20-11 показаны изменения, которые нужно внести в `main`, чтобы запускать новый поток для обработки каждого входящего потока соединения в цикле `for`.

Файл: `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);
        });
    }
}
```

Листинг 20-11: Порождение нового потока для каждого потока соединения

Как вы изучили в главе 16, `thread::spawn` создаст новый поток и затем запустит код замыкания в этом новом потоке. Если вы запустите этот код и загрузите `/sleep` в своём браузере, в затем загрузите `/` в двух других вкладках браузера, вы действительно увидите, что запросы к `/` не должны ждать завершения `/sleep`. Но, как мы уже упоминали, это в конечном счёте перегрузит систему, потому что вы будете создавать новые потоки без каких-либо ограничений.

Создание аналогичного интерфейса для конечного числа потоков

Мы хотим, чтобы наш пул потоков работал аналогичным, знакомым образом, чтобы переключение с потоков на пул потоков не требовало больших изменений в коде

используем наш API. В листинге 20-12 показан гипотетический интерфейс для структуры `ThreadPool`, который мы хотим использовать вместо `thread::spawn`.

Файл: `src/main.rs`

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }
}
```



Листинг 20-12: Наш идеальный интерфейс `ThreadPool`

Мы используем `ThreadPool::new`, чтобы создать новый пул потоков с конфигурируемым количеством потоков, в данном случае четыре. Затем в цикле `for` выполняем `pool.execute` имеющий интерфейс, аналогичный интерфейсу `thread::spawn`, в котором выполняется замыкание, которое пул должен выполнить для каждого потока соединения. Нам нужно реализовать `pool.execute`, чтобы он принимал замыкание и передавал его потоку из пула для выполнения. Этот код не компилируется, но мы постараемся, чтобы компилятор в его исправлении.

Создание структуры `ThreadPool` использованием разработки, управляемой компилятором

Внесите изменения листинга 20-12 в файл `src/main.rs`, а затем давайте воспользуемся ошибками компилятора из команды `cargo check` для управления нашей разработкой. Вот первая ошибка, которую мы получаем:

```
$ cargo check
  Checking hello v0.1.0 (file:///projects/hello)
error[E0433]: failed to resolve: use of undeclared type `ThreadPool`
--> src/main.rs:10:16
   |
10 |     let pool = ThreadPool::new(4);
   |           ^^^^^^^^^^ use of undeclared type `ThreadPool`


For more information about this error, try `rustc --explain E0433`.
error: could not compile `hello` due to previous error
```

Замечательно! Ошибка говорит о том, что нам нужен тип или модуль `ThreadPool`, поэтому мы создадим его сейчас. Наша реализация `ThreadPool` будет зависеть от того, какую работу выполняет наш веб-сервер. Итак, давайте переделаем крейт `hello` из бинарного в библиотечный для хранения реализации `ThreadPool`. После того, как поменяем в библиотечный крейт, мы также сможем использовать отдельную библиотеку пула потоков для любой работы, которую мы хотим выполнить с его использованием, а не только для обслуживания веб-запросов.

Создайте файл `src/lib.rs`, который содержит следующее, что является простейшим определением структуры `ThreadPool`, которую мы можем иметь в данный момент:

Файл: `src/lib.rs`

```
pub struct ThreadPool;
```

Затем создайте новый каталог `src/bin` и переместите двоичный крейт с корнем в `src/main.rs` в `src/bin/main.rs`. Это сделает библиотечный крейт основным крейтом в каталоге `hello`; мы все ещё можем запустить двоичный файл из `src/bin/main.rs`, используя `cargo run`. Переместив файл `main.rs`, отредактируйте его, чтобы подключить крейт библиотеки и добавить тип `ThreadPool` в область видимости, добавив следующий код в начало `src/bin/main.rs`:

Файл: `src/bin/main.rs`

```
{[#rustdoc_include ../../listings/ch20-web-server/no-listing-01-define-
threadpool-struct/src/bin/main.rs:here]}
```

Этот код по-прежнему не будет работать, но давайте проверим его ещё раз, чтобы получить следующую ошибку, которую нам нужно устранить:

```
$ cargo check
  Checking hello v0.1.0 (file:///projects/hello)
error[E0599]: no function or associated item named `new` found for struct
`ThreadPool` in the current scope
--> src/bin/main.rs:11:28
11 |     let pool = ThreadPool::new(4);
      |             ^^^ function or associated item not found in
`ThreadPool`

For more information about this error, try `rustc --explain E0599`.
error: could not compile `hello` due to previous error
```

Эта ошибка указывает, что далее нам нужно создать ассоциированную функцию с именем `new` для `ThreadPool`. Мы также знаем, что `new` должен иметь один параметр, который может принимать `4` в качестве аргумента и должен возвращать экземпляр `ThreadPool`. Давайте реализуем простейшую функцию `new`, которая будет иметь эти характеристики:

Файл: src/lib.rs

```
pub struct ThreadPool;

impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        ThreadPool
    }
}
```

Мы выбираем `usize` в качестве типа параметра `size`, потому что мы знаем, что отрицательное число потоков не имеет никакого смысла. Мы также знаем, что мы будем использовать число 4 в качестве количества элементов в коллекции потоков, для чего предназначен тип `usize`, как обсуждалось в разделе "[Целочисленные типы](#)" главы 3.

Давайте проверим код ещё раз:

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
error[E0599]: no method named `execute` found for struct `ThreadPool` in the
current scope
--> src/bin/main.rs:16:14
16 |         pool.execute(|| {
|             ^^^^^^ method not found in `ThreadPool`

For more information about this error, try `rustc --explain E0599`.
error: could not compile `hello` due to previous error
```

Теперь мы получаем предупреждение и ошибку. Игнорируем предупреждение не надолго, ошибка происходит потому что у нас нет метода `execute` в структуре `ThreadPool`. Вспомните раздел "[Создание подобного интерфейса для конечного числа потоков](#)", в котором мы решили, что наш пул потоков должен иметь интерфейс, похожий на `thread::spawn`. Кроме того, мы реализуем функцию `execute`, чтобы она принимала замыкание и передавала его свободному потоку из пула для запуска.

Мы определим метод `execute` у `ThreadPool` для приёма замыкания в качестве параметра. Вспомните раздел "[Хранение замыканий с использованием общих параметров и типажей Fn](#)" главы 13 и о том, что мы можем принимать замыкания в качестве параметров с тремя различными типажами: `Fn`, `FnMut` и `FnOnce`. Нам нужно решить, какой тип замыкания использовать здесь. Мы знаем, что в конечном счёте мы сделаем что-то похожее на реализацию стандартной библиотеки `thread::spawn`, поэтому мы можем посмотреть, какие ограничения накладывает на его параметр в сигнатуре `thread::spawn`. Документация показывает следующее:

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
```

Параметр типа `F` - это тот, который нас интересует; параметр типа `T` относится к возвращаемому значению и нам он не интересен. Можно увидеть, что `spawn` использует `FnOnce` в качестве ограничения типажа у `F`. Это, вероятно то, чего мы хотим, потому что мы в конечном итоге передадим получаемый аргумент в `execute` для `spawn`. Мы также можем быть ещё более уверены, что `FnOnce` - это тот типаж, который мы хотим использовать, поскольку поток для выполнения запроса будет выполнять этот запрос только один раз, что соответствует параметру `Once` в

тираже `FnOnce`.

Параметр типа `F` также имеет ограничение типажа `Send` и ограничение времени жизни `'static`, которые полезны в нашей ситуации: нам нужен `Send` для передачи замыкания из одного потока в другой и `'static`, потому что мы не знаем, сколько времени займёт выполнение потока. Давайте создадим метод `execute` для `ThreadPool`, который будет принимать обобщённый параметр типа `F` со следующими ограничениями:

Файл: src/lib.rs

```
impl ThreadPool {
    // --snip--
    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
    }
}
```

Мы по-прежнему используем `()` после `FnOnce` потому что типаж `FnOnce` представляет замыкание, которое не принимает параметров и возвращает единичный тип `()`. Также как при определении функций, тип возвращаемого значения может быть опущен в сигнатуре, но даже если у нас нет параметров, нам все равно нужны скобки.

Опять же, это самая простая реализация метода `execute`: она ничего не делает, мы только пытаемся сделать код компилируемым. Давайте проверим снова:

```
$ cargo check
  Checking hello v0.1.0 (file:///projects/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.24s
```

Сейчас мы получаем только предупреждения, что означает, что код компилируется! Но обратите внимание, если вы попробуете `cargo run` и сделаете запрос в браузере, вы увидите ошибки в браузере, которые мы видели в начале главы. Наша библиотека на самом деле ещё не вызывает замыкание, переданное в `execute`!

Примечание: вы возможно слышали высказывание о языках со строгими компиляторами, таких как Haskell и Rust, которое звучит так: «Если код компилируется, то он работает». Но это высказывание не всегда верно. Наш проект компилируется, но абсолютно ничего не делает! Если бы мы создавали

реальный, законченный проект, это был бы хороший момент начать писать модульные тесты, чтобы проверять, что код компилируется и имеет желаемое поведение.

Проверка количества потоков в `new`

Мы продолжим получать предупреждения, потому что мы ничего не делаем с параметрами для `new` и `execute`. Давайте реализуем тела этих функций в соответствии с желаемым поведением. Для начала давайте подумаем о `new`. Ранее мы выбирали без знаковый тип для параметра `size`, потому что пул с отрицательным числом потоков не имеет смысла. Тем не менее, пул с нулевым значением для потоков также не имеет смысла, но ноль является совершенно корректным для типа `usize`. Мы добавим код, чтобы проверить, что `size` больше нуля, перед возвращением экземпляра `ThreadPool` и будем паниковать, если программа получит ноль, используя макрос `assert!`, как показано в листинге 20-13.

Файл: src/lib.rs

```
impl ThreadPool {
    /// Create a new ThreadPool.
    ///
    /// # Panics
    ///
    /// The size is the number of threads in the pool.
    /// The `new` function will panic if the size is zero.
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        ThreadPool
    }

    // --snip--
}
```

Листинг 20-13: Реализация `ThreadPool::new` с паникой, если `size` равен нулю

Мы добавили документации в `ThreadPool` с помощью комментариев. Обратите внимание, мы следовали хорошим практикам документирования, добавив раздел, в котором указывается ситуация при которой функция может паниковать как обсуждалось в главе 14. Попробуйте запустить `cargo doc --open` и кликнуть структуру `ThreadPool`, чтобы увидеть как выглядит сгенерированная документация

для `new`.

Вместо добавления макроса `assert!`, как мы здесь сделали, мы могли бы описать у `new` возвращать `Result` как мы делали в `Config::new` проекта ввода/вывода в коде 12-9. Но сейчас мы решили, что попытка создания пула потоков без любого указания количества потоков должно быть не восстанавливаемой ошибкой. Если вы чувствуете себя честолюбивым, попробуйте написать версию `new` со следующей сигнатурой, чтобы сравнить обе версии:

```
pub fn new(size: usize) -> Result<ThreadPool, PoolCreationError> {
```

Создание места для хранения потоков

Теперь у нас есть способ узнать, что задано допустимое число потоков для хранения в пуле и мы можем создать эти потоки и сохранить их в структуре `ThreadPool` перед её возвратом. Но как мы "храним" поток? Давайте ещё раз посмотрим на сигнатуру `thread::spawn`:

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
```

Функция `spawn` возвращает тип `JoinHandle<T>`, где `T` является типом, который возвращает замыкание. Давайте попробуем использовать `JoinHandle` и посмотрим, что произойдёт. В нашем случае замыкания, которые мы передаём пулу потоков, будут обрабатывать соединение и ничего не будут возвращать, поэтому `T` будет единичным (unit) типом `()`.

Листинг 20-14 скомпилируется, но пока не создаёт потоков. Мы изменили объявление `ThreadPool`, чтобы оно содержало вектор экземпляров `thread::JoinHandle<()>`, инициализировали вектор с размером `size`, установили цикл `for`, который будет запускать некоторый код для создания потоков и вернули экземпляр `ThreadPool` содержащий потоки.

Файл: src/lib.rs



```
use std::thread;

pub struct ThreadPool {
    threads: Vec<thread::JoinHandle<()>>,
}

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut threads = Vec::with_capacity(size);

        for _ in 0..size {
            // create some threads and store them in the vector
        }

        ThreadPool { threads }
    }
    // --snip--
}
```

Листинг 20-14: Создание вектора в `ThreadPool` для хранения потоков

Мы добавили `std::thread` в область видимости библиотечного крейта, потому что мы используем `thread::JoinHandle` в качестве типа элементов вектора в `ThreadPool`.

После получения корректного значения `size`, наш `ThreadPool` создаёт новый вектор, который может содержать `size` элементов. В этой книге мы ещё не использовали функцию `with_capacity`, которая выполняет ту же задачу что и `Vec::new`, но с важным отличием: она заранее выделяет указанную память в векторе. Поскольку мы знаем, что нам нужно хранить `size` элементов в векторе, выполнение этого выделения немного более эффективно, чем использование `Vec::new`, который изменяет размеры при вставке элементов.

Когда вы снова запустите `cargo check`, вы получите ещё несколько предупреждений, но все должно завершится успехом.

Структура `Worker` ответственная за отправку кода из `ThreadPool` в поток

Мы оставили комментарий относительно создания потоков в цикле `for` кода 20-14. Здесь мы рассмотрим, как мы на самом деле создаём потоки. Стандартная библиотека предоставляет `thread::spawn` как способ создания потоков, а

`thread::spawn` ожидает получить некоторый код, который поток должен запустить как только поток создан. Однако в нашем случае мы хотим создать потоки и заставить их ждать код, который мы отправим им позже. Реализация потоков в стандартной библиотеке не имеет какого то способа это сделать, мы должны реализовать это вручную.

Мы будем реализовывать это поведение с помощью новой структуры данных между `ThreadPool` и потоками, которая будет управлять этим новым поведением. Мы назовём эту структуру данных `Worker`, что является общим термином в реализации пулов. Подумайте о людях, работающих на кухне в ресторане: рабочие ждут пока не поступят заказы от клиентов, а затем они несут ответственность за принятие этих заказов и их выполнение.

Вместо хранения вектора `JoinHandle<()>` в пуле потоков, мы будем сохранять экземпляры структуры `Worker`. Каждый `Worker` будет хранить один экземпляр `JoinHandle<()>`. Затем мы реализуем метод у `Worker`, который берет код замыкания для запуска и отправляет его в уже запущенный поток для выполнения. Мы также назначим каждому работнику `id`, чтобы мы могли различать разных работников в пуле при ведении журнала или отладке.

Давайте внесём изменения в последовательность действий, которая выполняется при создании `ThreadPool`. Мы реализуем код, который отправляет замыкание в поток после того, как мы настроили `Worker` следующим образом:

1. Определим структуру `Worker` (работник), которая содержит `id` и `JoinHandle<()>`.
2. Изменим `ThreadPool`, чтобы он содержал вектор экземпляров `Worker`.
3. Определим функцию `Worker::new`, которая принимает номер `id` и возвращает экземпляр `Worker`, который содержит `id` и поток, порождённый пустым замыканием.
4. В `ThreadPool::new` используем счётчик цикла `for` для генерации `id`, создаём новый `Worker` с этим `id` и сохраняем экземпляр "рабочника" в вектор.

Если вы готовы принять вызов, попробуйте реализовать эти изменения самостоятельно, прежде чем смотреть код листинге 20-15.

Готовы? Вот листинг 20-15 с одним из способов сделать предыдущие модификации.

Файл: src/lib.rs

```

use std::thread;

pub struct ThreadPool {
    workers: Vec<Worker>,
}

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool { workers }
    }
    // --snip--
}

struct Worker {
    id: usize,
    thread: thread::JoinHandle<()>,
}

impl Worker {
    fn new(id: usize) -> Worker {
        let thread = thread::spawn(|| {});

        Worker { id, thread }
    }
}

```

Листинг 20-15: Изменение `ThreadPool` для хранения экземпляров `Worker` вместо непосредственного хранения потоков

Мы изменили имя поля в `ThreadPool` с `threads` на `workers`, потому что теперь оно содержит экземпляры `Worker` вместо экземпляров `JoinHandle<()>`. Мы используем счётчик в цикле `for` в качестве аргумента для `Worker::new` и сохраняем каждый новый `Worker` в векторе с именем `workers`.

Внешний код (вроде нашего сервера в `src/bin/main.rs`) не должен знать подробности реализации касательно использования структуры `Worker` внутри `ThreadPool`, поэтому мы делаем структуру `Worker` и её новую функцию `new` приватными. Функция `Worker::new` использует заданный нами `id` и сохраняет экземпляр

`JoinHandle<()>`, который создаётся путём порождение нового потока с пустым замыканием.

Этот код скомпилируется и будет хранить количество экземпляров `Worker`, которое мы указали в качестве аргумента функции `ThreadPool::new`. Но мы все *ещё* не обрабатываем замыкание, которое мы получаем в методе `execute`. Давайте взглянем на то, как это сделать.

Отправка запросов в потоки через каналы

Теперь мы рассмотрим проблему, заключающуюся в том, что замыкания переданные в `thread::spawn` абсолютно ничего не делают. Вот мы получаем замыкание, которое хотим выполнить в методе `execute`. Но для запуска нам необходимо передать замыкание в метод `thread::spawn`, где на каждый `ThreadPool` создаётся один `Worker`.

Мы хотим, чтобы только что созданные структуры `Worker` извлекали код для запуска из очереди хранящейся в `ThreadPool` и отправляли этот код в свой поток для выполнения.

В главе 16 вы узнали о *каналах* (channels) - простом способе связи между двумя потоками, который идеально подойдёт для этого сценария. Мы будем использовать канал в качестве очереди заданий, а команда `execute` отправит задание из `ThreadPool` экземплярам `Worker`, который отправит задание в свой поток. Вот план:

1. `ThreadPool` создаст канал и будет удерживать его передающую сторону.
2. Каждый `Worker` будет удерживать принимающую сторону канала.
3. Мы создадим новую структуру `Job` которая будет содержать замыкания, которые мы хотим отправить в канал.
4. Метод `execute` отправит задание, которое он хочет выполнить, в отправляющую сторону канала.
5. В своём потоке `Worker` будет выполнять цикл с принимающей стороной канала и выполнит замыкание любого получаемого задания.

Давайте начнём с создания канала в `ThreadPool::new` и удержания отправляющей стороны в экземпляре `ThreadPool`, как показано в листинге 20-16. В структуре `Job` сейчас ничего не содержится, но это будет тип элемента который мы отправляем в канал.

Файл: src/lib.rs

```
use std::sync::mpsc, thread;

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Job>,
}

struct Job;

impl ThreadPool {
    // --snip--
    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 {
            workers.push(Worker::new(id));
        }

        ThreadPool { workers, sender }
    }
    // --snip--
}
```

Листинг 20-16: Модификация `ThreadPool` для хранения отправляющей части канала, который отправляет экземпляры `Job`

В `ThreadPool::new` мы создаём наш новый канал и пул содержащий отправляющую сторону. Код успешно скомпилируется все ещё с предупреждениями.

Давайте попробуем передавать принимающую сторону канала каждому "работнику" (структуре `woker`), когда пул потоков создаёт канал. Мы знаем, что хотим использовать получающую часть канала в потоке порождаемым "работником", поэтому мы будем ссылаться на параметр `receiver` в замыкании. Код 20-17 пока не компилируется.

Файл: src/lib.rs



```
impl ThreadPool {
    // --snip--
    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 {
            workers.push(Worker::new(id, receiver));
        }

        ThreadPool { workers, sender }
    }
    // --snip--
}

// --snip--

impl Worker {
    fn new(id: usize, receiver: mpsc::Receiver<Job>) -> Worker {
        let thread = thread::spawn(|| {
            receiver;
        });

        Worker { id, thread }
    }
}
```

Листинг 20-17: Передача принимающей части канала "работнику"

Мы внесли несколько небольших и простых изменений: мы передаём принимающую часть канала в `Worker::new`, а затем используем его внутри замыкания.

При попытке проверить код, мы получаем ошибку:

```
$ cargo check
  Checking hello v0.1.0 (file:///projects/hello)
error[E0382]: use of moved value: `receiver`
--> src/lib.rs:27:42
   |
22 |     let (sender, receiver) = mpsc::channel();
   |             ----- move occurs because `receiver` has type
`std::sync::mpsc::Receiver<Job>`, which does not implement the `Copy` trait
...
27 |         workers.push(Worker::new(id, receiver));
   |                     ^^^^^^^^^^ value moved here, in
previous iteration of loop

For more information about this error, try `rustc --explain E0382`.
error: could not compile `hello` due to previous error
```

Код пытается передать `receiver` в несколько экземпляров `Worker`. Это не будет работать, как вы помните из главы 16: реализация канала предоставляемая Rust, является моделью нескольких производителей (multiple producer), один потребитель (single consumer). Это означает, что мы не можем просто клонировать принимающую часть канала для исправления этого кода. Даже если бы мы это могли, это не техника которую мы хотели бы использовать; вместо этого мы хотим распределить задачи среди потоков, разделяя один `receiver` среди всех "работников".

Кроме того, удаление задачи из очереди канала включает изменение `receiver`, поэтому потокам необходим безопасный способ делиться и изменять `receiver`, в противном случае мы можем получить условия гонки (как описано в главе 16).

Вспомните умные указатели, которые обсуждались в главе 16: чтобы делиться владением между несколькими потоками и позволить потокам изменять значение, нам нужно использовать тип `Arc<Mutex<T>>`. Тип `Arc` позволит нескольким "работникам" владеть получателем (`receiver`), а `Mutex` гарантирует что только один "работник" получит задание (`job`) от получателя в один момент времени. Листинг 20-18 показывает изменения, которые мы должны сделать.

Файл: `src/lib.rs`

```

use std::{
    sync::mpsc, Arc, Mutex,
    thread,
};
// --snip--

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        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)));
        }

        ThreadPool { workers, sender }
    }
    // --snip--
}

// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        // --snip--
    }
}

```

Листинг 20-18: Совместное использование принимающей стороны канала среди "работников" используя `Arc` и `Mutex`

В `ThreadPool::new` мы помещаем принимающую сторону канала внутрь `Arc` и `Mutex`. Для каждого нового "работника" мы клонируем `Arc`, чтобы увеличить счётчик ссылок так, что "работники" могут разделять владение принимающей стороны канала.

С этими изменениями код компилируется! Мы подбираемся к цели!

Реализация метода `execute`

Давайте реализуем метод `execute` у структуры `ThreadPool`. Мы также изменим тип `Job` со структуры на псевдоним типа для типаж-объекта, который содержит тип замыкания принимаемый методом `execute`. Как описано в разделе "Создание синонимов типа с помощью псевдонимов типа" главы 19, псевдонимы типов позволяют делать длинные типы короче. Посмотрите в листинг 20-19.

Файл: `src/lib.rs`

```
// --snip--

type Job = Box<dyn FnOnce() + Send + 'static>;

impl ThreadPool {
    // --snip--

    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
        let job = Box::new(f);

        self.sender.send(job).unwrap();
    }
}

// --snip--
```

Листинг 20-19: Создание псевдонима типа `Job` для `Box`, содержащего каждое замыкание и затем отправляющее задание (job) в канал

После создания нового экземпляра `Job` с помощью замыкания, получаемого в метод `execute`, мы отправляем это задание в отправляющую часть канала. Мы вызываем `unwrap` для `send` в случае неудачной отправки. Это может произойти, если например, мы остановим выполнение всех наших потоков, что означает, что принимающая сторона прекратила получение новых сообщений. На данный момент мы не можем остановить выполнение наших потоков: наши потоки продолжают выполняться, пока существует пул. Причина, по которой мы используем `unwrap`, заключается в том, что мы знаем, что сбоя не произойдёт, но компилятор этого не знает.

Но мы ещё не закончили! В "работнике" (worker) наше замыкание, переданное в `thread::spawn` все ещё ссылается только на принимающую сторону канала. Вместо этого нам нужно, чтобы замыкание работало в бесконечном цикле, запрашивая задание у принимающей части канала и выполняя задание, когда оно принято.

Давайте внесём изменения, показанные в листинге 20-20 внутри `Worker::new`.

Файл: `src/lib.rs`

```
// --snip--

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!("Worker {id} got a job; executing.");

            job();
        });

        Worker { id, thread }
    }
}
```

Листинг 20-20: Получение и выполнение задачий в потоке "работника"

Здесь мы сначала вызываем `lock` у `receiver`, чтобы получить мьютекс, а затем вызываем `unwrap` для паники при любых ошибках. Захват блокировки может завершиться неудачей, если мьютекс находится в *отравленном state* (poisoned state), что может произойти если какой-то другой поток запаниковал, удерживая блокировку, вместо снятия блокировки. В этой ситуации правильное действие - вызвать `unwrap` для паники потока. Не стесняйтесь заменить `unwrap` на `expect` с сообщением об ошибке, которое имеет для вас значение.

Если мы получим блокировку мьютекса, мы вызываем `recv` для получения `Job` из канала. Окончательный вызов `unwrap` проходит мимо любых ошибок, которые могут произойти, если поток удерживающий отправляющую сторону канала, завершил работу подобно тому, как метод `send` возвращает `Err`, если принимающая сторона закрывается.

Вызов `recv` блокирующий, поэтому если ещё нет задач (job), то текущий поток будет ждать, пока задача не станет доступной. `Mutex<T>` гарантирует, что только один поток `Worker` пытается запросить задачу за раз.

Наш пул потоков теперь находится в рабочем состоянии! Выполните `cargo run` и сделайте несколько запросов:

```
$ cargo run
Compiling hello v0.1.0 (file:///projects/hello)
warning: field is never read: `workers`
--> src/lib.rs:7:5
|
7 |     workers: Vec<Worker>,
|     ^^^^^^^^^^^^^^^^^^^^^^
|
= note: `#[warn(dead_code)]` on by default

warning: field is never read: `id`
--> src/lib.rs:48:5
|
48 |     id: usize,
|     ^^^^^^^^
|
warning: field is never read: `thread`
--> src/lib.rs:49:5
|
49 |     thread: thread::JoinHandle<()>,
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

warning: 3 warnings emitted

    Finished dev [unoptimized + debuginfo] target(s) in 1.40s
    Running `target/debug/main`
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
```

Успех! Теперь у нас есть пул потоков, который обрабатывает соединения асинхронно. Никогда не создаётся более четырёх потоков, поэтому наша система не будет перегружена, если сервер получает много запросов. Если мы отправим запрос ресурса `/sleep`, сервер сможет обслуживать другие запросы, запустив их в другом потоке.

Примечание: если вы запрашиваете `/sleep` в нескольких окнах браузера одновременно, они могут загружаться по одному с интервалами в 5 секунд. Некоторые веб-браузеры выполняют несколько экземпляров одного и того же

запроса последовательно из-за кэширования. Данное ограничение не вызвано нашим веб-сервером.

После изучения цикла `while let` в главе 18 вы можете удивиться, почему мы не написали код рабочего потока (worker thread), как показано в листинге 20-22.

Файл: src/lib.rs

```
// --snip--  
  
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!("Worker {id} got a job; executing.");  
  
                job();  
            }  
        });  
  
        Worker { id, thread }  
    }  
}
```



Листинг 20-22: Альтернативная реализация `Worker::new` с использованием `while let`

Этот код компилируется и запускается, но не приводит к желаемому поведению: медленный запрос всё равно приведёт к тому, что другие запросы будут ждать обработки. Причина здесь несколько тоньше: `Mutex` не имеет общедоступного `unlock` потому что право собственности на блокировку зависит от времени жизни `MutexGuard<T>`. В `LockResult<MutexGuard<T>>` которое возвращает метод `lock`. Во время компиляции анализатор заимствований может затем применить правило, согласно которому к ресурсу, охраняемому `Mutex` нельзя получить доступ, если мы не удерживаем блокировку. Но эта реализация также может привести к тому, что блокировка будет удерживаться дольше, чем предполагалось, если мы не будем тщательно продумывать время жизни `MutexGuard<T>`.

Код в листинге 20-20, который использует `let job = receiver.lock().unwrap().recv().unwrap();` работает, потому что с `let` любые временные значения, используемые в выражении справа от знака равенства, немедленно удаляются после завершения оператора `let`. Однако `while let` (и `if let` and `match`) не удаляет временные значения до конца связанного блока. В

листе 20-21 блокировка сохраняется на время вызова `job()`, что означает, что другие исполнители не могут получать задания.

Изящное завершение и освобождение ресурсов

Листинг 20-20 асинхронно отвечает на запросы с помощью использования пула потоков, как мы и хотели. Мы получаем некоторые предупреждения про `workers`, `id` и поля `thread`, которые мы не используем напрямую, что напоминает нам о том, что мы не освобождаем все ресурсы. Когда мы используем менее элегантный метод остановки основного потока клавиатурной комбинацией `ctrl-c`, все остальные потоки также немедленно останавливаются, даже если они находятся в середине обработки запроса.

Теперь мы будем реализовывать типаж `Drop` для вызова `join` у каждого потока в пуле, чтобы они могли завершить запросы над которыми они работают до закрытия. Затем мы реализуем способ сообщить потокам, что они должны перестать принимать новые запросы и завершить работу. Чтобы увидеть этот код в действии, мы изменим наш сервер так, чтобы он принимал только два запроса, прежде чем корректно завершить работу его пула потоков.

Реализация типажа `Drop` для `ThreadPool`

Давайте начнём с реализации `Drop` у нашего пула потоков. Когда пул удаляется, все наши потоки должны объединиться (`join`), чтобы убедиться, что они завершают свою работу. В листинге 20-22 показана первая попытка реализации `Drop`, код пока не будет работать.

Файл: `src/lib.rs`

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            worker.thread.join().unwrap();
        }
    }
}
```



Листинг 20-22: Присоединение (Joining) каждого потока, когда пул потоков выходит из области видимости

Во-первых, мы проходим циклом по каждому `workers` из пула потоков. Для этого мы используем `&mut`, потому что `self` является изменяемой ссылкой и нам также

нужно иметь возможность изменять экземпляр `worker`. Для каждого "работника" мы печатаем сообщение о том, что этот конкретный "работник" завершается, затем вызываем `join` у потока этого "работника". Если вызов `join` происходит с ошибкой, мы используем `unwrap`, чтобы вызвать панику в Rust и завершить не совсем красиво.

Ошибка получаемая при компиляции этого кода:

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
error[E0507]: cannot move out of `worker.thread` which is behind a mutable
reference
--> src/lib.rs:52:13
   |
52 |         worker.thread.join().unwrap();
   |         ^^^^^^^^^^ move occurs because `worker.thread` has type
`JoinHandle<()>`, which does not implement the `Copy` trait

For more information about this error, try `rustc --explain E0507`.
error: could not compile `hello` due to previous error
```

Ошибка говорит, что мы не можем вызвать `join`, потому что у нас есть только изменяемое заимствование каждого `worker` и что `join` забирает во владение его аргумент. Чтобы решить эту проблему, нужно переместить поток из экземпляра `Worker`, который владеет `thread`, чтобы `join` мог использовать внутренний поток. Мы сделали это в коде 17-15: если вместо этого `Worker` содержит тип `Option<thread::JoinHandle<()>>`, мы можем вызвать метод `take` у `Option`, чтобы переместить значение из варианта `Some` и оставить вариант `None` на его месте. Другими словами, работающий `Worker` будет содержать вариант `Some` внутри `thread`, и когда мы хотим очистить `Worker`, мы заменяем значение варианта `Some` на вариант `None`, чтобы у `Worker` не было потока для запуска.

Итак, мы хотим обновить объявление `Worker` следующим образом:

Файл: src/lib.rs

```
struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>,
}
```



Теперь давайте опираться на компилятор, чтобы найти другие места, которые нужно изменить. Проверяя код, мы получаем две ошибки:

```
$ cargo check
  Checking hello v0.1.0 (file:///projects/hello)
error[E0599]: no method named `join` found for enum `Option` in the current
scope
--> src/lib.rs:52:27
   |
52 |         worker.thread.join().unwrap();
   |         ^^^^^^ method not found in
`Option<JoinHandle<()>>`  

error[E0308]: mismatched types
--> src/lib.rs:72:22
   |
72 |         Worker { id, thread }
   |         ^^^^^^^^ expected enum `Option`, found struct
`JoinHandle`  

   |
= note: expected enum `Option<JoinHandle<()>>`
        found struct `JoinHandle<_>`  

help: try wrapping the expression in `Some`  

   |
72 |         Worker { id, Some(thread) }
   |         ++++++ +  

Some errors have detailed explanations: E0308, E0599.
For more information about an error, try `rustc --explain E0308`.
error: could not compile `hello` due to 2 previous errors
```

Давайте обратимся ко второй ошибке, которая указывает на код в конце

`Worker::new`; нам нужно обернуть значение `thread` в вариант `Some` при создании нового `Worker`. Внесите следующие изменения, чтобы исправить эту ошибку:

Файл: `src/lib.rs`

```
impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        // --snip--

        Worker {
            id,
            thread: Some(thread),
        }
    }
}
```

Первая ошибка находится в нашей реализации `Drop`. Ранее мы упоминали, что намеревались вызвать `take` для параметра `Option`, чтобы забрать `thread` из

процесса `worker`. Следующие изменения делают это:

Файл: src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}
```

Как уже говорилось в главе 17, метод `take` у типа `Option` забирает значение из варианта `Some` и оставляет вариант `None` в этом месте. Мы используем `if let`, чтобы деструктурировать `Some` и получить поток; затем вызываем `join` у потока. Если поток "работника" уже `None`, мы знаем, что этот "работник" уже очистил свой поток, поэтому в этом случае ничего не происходит.

Сигнализация потокам прекратить прослушивание получения задач

Теперь со всеми внесёнными нами изменениями код компилируется без каких-либо предупреждений. Но плохая новость в том, что этот код ещё не работает так, как мы этого хотим. Ключом является логика в замыканиях, запускаемых потоками экземпляров `Worker`. В данный момент мы вызываем `join`, но это не завершит потоки, потому что они работают в цикле `loop` бесконечно в поиске новой задачи (job). Если мы попытаемся удалить `ThreadPool` в текущей реализации `drop`, то основной поток навсегда заблокируется в ожидании завершения первого потока из пула.

Для решения этой проблемы, мы изменим потоки так, чтобы они прослушивали либо задачи `Job` для её выполнения, либо сигнал, что они должны прекратить прослушивание и выйти из бесконечного цикла. Вместо отправки экземпляров задач `Job`, наш канал отправит один из этих двух вариантов перечисления.

Файл: src/lib.rs

```
 {{#rustdoc_include ../listings/ch20-web-server/no-listing-07-define-message-  
 enum/src/lib.rs:here}}
```

Это перечисление `Message` будет либо вариантом `NewJob`, который внутри держит `Job` с потоком для выполнения, или это будет вариант `Terminate`, который сделает так, чтобы поток вышел из цикла и остановился.

Нам нужно настроить канал для использования значений типа `Message`, а не типа `Job` как показано в листинге 20-23.

Файл: `src/lib.rs`

```

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: Option<mpsc::Sender<Job>>,
}
// --snip--
impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        // --snip--

        ThreadPool {
            workers,
            sender: Some(sender),
        }
    }

    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
        let job = Box::new(f);

        self.sender.as_ref().unwrap().send(job).unwrap();
    }
}

impl Drop for ThreadPool {
    fn drop(&mut self) {
        drop(self.sender.take());

        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}

```

Листинг 20-23: Отправка и получение значений перечисления `Message` И выход из цикла, если `Worker` получает перечисление `Message::Terminate`

Чтобы встроить в код перечисление `Message`, нужно изменить `Job` на `Message` в двух местах: в объявлении `ThreadPool` и сигнатуре `Worker::new`. Метод `execute` у `ThreadPool` должен отправлять задания, завёрнутые в вариант `Message::NewJob`. Затем в `Worker::new`, где `Message` получен из канала, задание (`job`) будет обработано, если получен вариант `NewJob` и поток выйдет из цикла, если получен

вариант `Terminate`.

С такими изменениями код компилируется и продолжит функционировать так же, как и после кода 20-20. Но мы получим предупреждение, потому что мы не создаём сообщения типа `Terminate`. Давайте исправим это предупреждение, изменив нашу реализацию `Drop` как в листинге 20-25.

Файл: src/lib.rs

```
impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || loop {
            match receiver.lock().unwrap().recv() {
                Ok(job) => {
                    println!("Worker {} got a job; executing.");
                    job();
                }
                Err(_) => {
                    println!("Worker {} disconnected; shutting down.");
                    break;
                }
            }
        });
        Worker {
            id,
            thread: Some(thread),
        }
    }
}
```

Листинг 20-24: Отправка `Message::Terminate` рабочим потокам перед вызовом `join` в каждом потоке

Теперь мы дважды проходим по потокам "работников": один раз для отправки одного сообщения `Terminate` каждому потоку и один раз для вызова `join` у каждого рабочего потока. Если бы мы попытались отправить сообщение и сразу же выполнить `join` в этом же цикле, мы не смогли бы гарантировать, что рабочий поток в текущей итерации получит сообщение из канала.

Чтобы лучше понять, почему нужны два отдельных цикла, представьте сценарий с двумя работниками. Если бы мы использовали один цикл для перебора каждого работника, на первой итерации сообщение о прекращении работы было бы отправлено в канал и метод `join` вызывался бы у первого рабочего потока. Если

этот первый поток занят обработкой запроса в данный момент, второй рабочий поток получит сообщение о завершении из канала и завершит свою работу. В главном потоке мы бы остались в ожидании завершения работы первого рабочего потока, но этого не произошло, потому что второй поток получил сообщение о завершении. Это ситуация взаимной блокировки (deadlock)!

Чтобы предотвратить такой сценарий, мы сначала помещаем все сообщения `Terminate` в канал в первом цикле; затем мы объединяем (`join`) для завершения все потоки во втором цикле. Каждый рабочий поток прекратит получать запросы из канала, как только получит сообщение о завершении. Таким образом, мы можем быть уверены, что если мы отправим количество завершающих сообщений равное количеству рабочих потоков то, каждый получит сообщение о завершении до вызова `join` в его потоке.

Чтобы увидеть этот код в действии, давайте изменим `main`, чтобы принимать только два запроса, прежде чем корректно завершить работу сервера как показано в листинге 20-25.

Файл: `src/bin/main.rs`

```
{{{#rustdoc_include ../listings/ch20-web-server/listing-20-25/src/bin/main.rs:here}}}
```

Код 20-25. Выключение сервера после обслуживания двух запросов с помощью выхода из цикла

Вы бы не хотели, чтобы реальный веб-сервер отключался после обслуживания только двух запросов. Этот код всего лишь демонстрирует, что корректное завершение работы и освобождение ресурсов находятся в рабочем состоянии.

Метод `take` определён в типаже `Iterator` и ограничивает итерацию максимум первыми двумя элементами. `ThreadPool` выйдет из области видимости в конце `main` и будет запущена его реализация `drop`.

Запустите сервер с `cargo run` и сделайте три запроса. Третий запрос должен выдать ошибку и в терминале вы должны увидеть вывод, подобный следующему:

```
$ cargo run
Compiling hello v0.1.0 (file:///projects/hello)
  Finished dev [unoptimized + debuginfo] target(s) in 1.0s
    Running `target/debug/main`
Worker 0 got a job; executing.
Worker 3 got a job; executing.
Shutting down.
Sending terminate message to all workers.
Shutting down all workers.
Shutting down worker 0
Worker 1 was told to terminate.
Worker 2 was told to terminate.
Worker 0 was told to terminate.
Worker 3 was told to terminate.
Shutting down worker 1
Shutting down worker 2
Shutting down worker 3
```

Вы возможно увидите другой порядок рабочих потоков и напечатанных сообщений. Мы можем увидеть, как этот код работает по сообщениям: "работники" номер 0 и 3 получили первые два запроса, затем на третьем запросе сервер прекратил принимать соединения. Когда `ThreadPool` выходит из области видимости в конце `main`, то срабатывает его реализация типажа `Drop` и пул сообщает всем рабочим потокам прекратить выполнение. Каждый рабочий поток распечатывает сообщение, когда видит сообщение о завершении, а затем пул потоков вызывает `join`, чтобы завершить работу каждого рабочего потока.

Обратите внимание на один интересный аспект этого конкретного выполнения: `ThreadPool` отправил сообщения о завершении в канал и прежде чем какой-либо рабочий получил сообщение, мы пытались присоединить (`join`) "рабочника" с номером 0. Рабочий поток 0 ещё не получил сообщение о прекращении, поэтому основной поток заблокировал ожидание потока 0 для завершения. Тем временем каждый из рабочих потоков получил сообщения об завершении. Когда рабочий поток 0 завершился, основной поток ждал окончания завершения выполнения остальных рабочих потоков. В этот момент все они получили сообщение о завершении и смогли завершиться.

Поздравления! Теперь мы завершили проект; у нас есть базовый веб-сервер, который использует пул потоков для асинхронных ответов. Мы можем выполнить корректное завершение работы сервера, который очищает все потоки в пуле.

Вот полный код для справки:

Файл: `src/bin/main.rs`

```
{{{#rustdoc_include ../listings/ch20-web-server/no-listing-08-final-
code/src/bin/main.rs}}}
```

Файл: src/lib.rs

```
{{{#rustdoc_include ../listings/ch20-web-server/no-listing-08-final-
code/src/lib.rs}}}
```

В коде можно сделать больше! Если вы хотите продолжить совершенствование этого проекта, вот несколько идей:

- Добавьте больше документации в `ThreadPool` и его публичным методам.
- Добавьте тесты функционалу из библиотеки.
- Заменить вызовы `unwrap` на более правильную обработку ошибок.
- Используйте `ThreadPool` для выполнения некоторых других задач, помимо обслуживания веб-запросов.
- Найдите крейт для пула потоков на [crates.io](#) и реализуйте аналогичный веб-сервер, используя такой крейт. Затем сравните его API и надёжность с пулом потоков, который мы реализовали.

Итоги

Отличная работа! Вы сделали это к концу книги! Мы хотим поблагодарить вас за то, что присоединились к нам в этом путешествии по языку Rust. Теперь вы готовы реализовать свои собственные проекты на Rust и помочь с проектами другим людям. Имейте в виду, что существует приветливое сообщество других Rust разработчиков, которые хотели бы помочь вам с любыми сложными задачами с которыми вы столкнётесь в своём Rust путешествии.

Дополнительная информация

Следующие разделы содержат справочные материалы, которые могут оказаться полезными в вашем путешествии по Rust.

Дополнение A: Ключевые слова

Следующий список содержит ключевые слова, зарезервированные для текущего или будущего использования в Rust. Таким образом, данные слова нельзя использовать для идентификаторов (кроме сырых идентификаторов, которые мы обсудили в разделе "Сырые идентификаторы"), включая имена функций, переменных, параметров, полей структур, модулей, пакетов, констант, макросов, статических переменных, атрибутов, типов, типажей или времён жизни.

Ключевые слова, использующиеся в Rust в настоящее время

Следующие ключевые слова в настоящее время имеют описанную функциональность.

- **as** - простое приведение типа, устранение неоднозначности реализованного для элемента трейта, или переименование элементов в **use** и **extern crate**
- **async** - возврат объекта **Future** вместо блокировки выполнения текущего потока
- **await** - приостановка выполнения до тех пор, пока результат выполнения **Future** не станет готов
- **break** - немедленное прекращение цикла
- **const** - объявляет константу или константный сырой указатель
- **continue** - перейти к следующей итерации цикла
- **crate** - подключение внешнего крейта или макро-переменной, представляющей внешний пакет, в котором она объявлена
- **dyn** - динамическая диспетчеризация для трейт-объектов
- **else** - альтернатива для **if** и **if let**
- **enum** - определение перечисления
- **extern** - определение использования внешнего пакета, функции или переменной
- **false** - логический литерал `false`
- **fn** - определение функции или типа-указателя на функцию
- **for** - цикл по элементам итератора, реализация трейта или указание времени жизни высокого уровня
- **if** - условный оператор ветвления
- **impl** - наследование или реализация трейта
- **in** - часть синтаксической конструкции цикла **for**
- **let** - определение, привязывание переменной

- **loop** - бесконечный цикл
- **match** - оператор сопоставления значения с образцом
- **mod** - оператор определения модуля
- **move** - позволяет замыканию брать во владение всё, что "захватывает" замыкание
- **mut** - обозначение изменяемых переменных, ссылок, сырых указателей или привязок к шаблону
- **pub** - обозначение публичного доступа к полям структуры, **impl** блокам или модулям
- **ref** - ссылочное связывание
- **return** - оператор возврата из функции
- **Self** - псевдоним типа, для которого объявляется или реализуется типаж
- **self** - предмет метода или текущий модуль
- **static** - глобальная переменная или время жизни, продолжающееся всё время работы программы
- **struct** - определение структуры
- **super** - родительский модуль относительно текущего
- **trait** - обозначение трейта
- **true** - логический литерал true
- **type** - объявление псевдонима типа или ассоциированного типа
- **union** - определение **объединения** что и является единственным ключевым словом при использовании в объединении
- **unsafe** - определение небезопасного кода, функции, трейта или реализаций
- **use** - оператор импорта символов в текущую область видимости
- **where** - оператор условия-ограничения для типа
- **while** - условный цикл, основанный на результате вычисления выражения

Ключевые слова, зарезервированные для будущего использования

Следующие ключевые слова не имеют функциональности, но они зарезервированы в Rust для возможного использования в будущем.

- **abstract**
- **become**
- **box**
- **do**

- `final`
- `macro`
- `override`
- `priv`
- `try`
- `typeof`
- `unsized`
- `virtual`
- `yield`

Сырые идентификаторы

Сырые идентификаторы - это синтаксис, позволяющий вам использовать ключевые слова там, где обычно они не могут быть. Для создания и использования сырого идентификатора к ключевому слову добавляется префикс `r#`.

Например, ключевое слово `match`. Если вы попытаетесь скомпилировать следующую функцию, использующую в качестве имени `match`:

Файл: `src/main.rs`

```
fn match(needle: &str, haystack: &str) -> bool {
    haystack.contains(needle)
}
```

вы получите 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 говорит о том, что вы не можете использовать ключевое слово `match` в качестве идентификатора функции. Чтобы получить возможность использования слова `match` в качестве имени функции, нужно использовать синтаксис "сырых идентификаторов", например так:

Файл: `src/main.rs`

```
fn r#match(needle: &str, haystack: &str) -> bool {
    haystack.contains(needle)
}

fn main() {
    assert!(r#match("foo", "foobar"));
}
```

Этот код скомпилируется без ошибок. Обратите внимание, что префикс `r#` в определении имени функции, указан также, как он указан в месте её вызова в `main`.

Сырые идентификаторы позволяют использовать любые слова в качестве идентификатора, даже если это зарезервированное слово. Дополнительно сырье идентификаторы позволяют использовать библиотеки, написанные на отличной от используемой вами редакции Rust.

Например `try` является ключевым словом в 2018 редакции, но не в 2015. Если вы зависите от библиотеки, написанной с использованием 2015 редакции и имеющей функцию `try`, то для вызова такой функции из кода 2018 редакции, вам необходимо использовать синтаксис сырьих идентификаторов. В данном случае `r#try`. Больше подробностей про редакции можно найти в [Приложении E](#)

Дополнение Б: Операторы и обозначения

Это дополнение содержит глоссарий синтаксиса Rust, включая операторы и другие обозначения, которые появляются сами по себе или в контексте путей, обобщений, типажей, макросов, атрибутов, комментариев, кортежей и скобок.

Операторы

Таблица Б-1 содержит операторы языка Rust, пример появления оператора, краткое объяснение, возможность перегрузки оператора. Если оператор можно перегрузить, то показан типаж, с помощью которого его можно перегрузить.

Таблица Б-1: Операторы

Оператор	Пример	Объяснение	Перегружаемость
!	<code>ident!(...)</code> , <code>ident!{...}</code> , <code>ident![...]</code>	Вызов макроса	
!	<code>!expr</code>	Побитовое или логическое отрицание	<code>Not</code>
<code>!=</code>	<code>var != expr</code>	Сравнение "не равно"	<code>PartialEq</code>
<code>%</code>	<code>expr % expr</code>	Остаток от деления	<code>Rem</code>
<code>%=</code>	<code>var %= expr</code>	Остаток от деления и присваивание	<code>RemAssign</code>
<code>&</code>	<code>&expr</code> , <code>&mut expr</code>	Займствование	
<code>&</code>	<code>&type</code> , <code>&mut type</code> , <code>'a</code> , <code>'a mut</code>	Указывает что данный тип заимствуется	
<code>&</code>	<code>expr & expr</code>	Побитовое И	<code>BitAnd</code>
<code>&=</code>	<code>var &= expr</code>	Побитовое И и присваивание	<code>BitAndAssign</code>
<code>&&</code>	<code>expr && expr</code>	Логическое И	

<code>*</code>	<code>expr * expr</code>	Арифметическое умножение	<code>Mul</code>
<code>*=</code>	<code>var *= expr</code>	Арифметическое умножение и присваивание	<code>MulAssign</code>
<code>*</code>	<code>*expr</code>	Разыменование ссылки	
<code>*</code>	<code>*const type</code> , <code>*mut type</code>	Указывает, что данный тип является сырым указателем	
<code>+</code>	<code>trait + trait, 'a + trait</code>	Соединение ограничений типа	
<code>+</code>	<code>expr + expr</code>	Арифметическое сложение	<code>Add</code>
<code>+=</code>	<code>var += expr</code>	Арифметическое сложение и присваивание	<code>AddAssign</code>
<code>,</code>	<code>expr, expr</code>	Аргумент и разделитель элементов	
<code>-</code>	<code>- expr</code>	Арифметическое отрицание	<code>Neg</code>
<code>-</code>	<code>expr - expr</code>	Арифметическое вычитание	<code>Sub</code>
<code>-</code>	<code>var -= expr</code>	Арифметическое вычитание и присваивание	<code>SubAssign</code>
<code>-></code>	<code>fn(...) -> type</code> ,	...	
<code>.</code>	<code>expr.ident</code>	Доступ к элементу	
<code>..</code>	<code>.., expr..,</code> <code>..expr,</code> <code>expr..expr</code>	Указывает на диапазон чисел, исключая правый	
<code>...=</code>	<code>..=expr</code> , <code>expr..=expr</code>	Указывает на диапазон чисел,	

		включая правый	
..	..expr	Синтаксис обновления структуры	
..	variant(x, ..), struct_type { x, .. }	Привязка «И все остальное»	
...	expr...expr	В шаблоне: шаблон диапазона включая правый элемент	
/	expr / expr	Арифметическое деление	Div
/=	var /= expr	Арифметическое деление и присваивание	DivAssign
:	pat: type, ident: type	Ограничения типов	
:	ident: expr	Инициализация поля структуры	
:	'a: loop {...}	Метка цикла	
;	expr;	Оператор, указывающий на конец выражения	
;	[...; len]	Часть синтаксиса массива фиксированного размера	
<<	expr << expr	Битовый сдвиг влево	Shl
<<=	var <<= expr	Битовый сдвиг влево и присваивание	ShlAssign
<	expr < expr	Сравнение	PartialOrd

		"меньше чем"	
<code><=</code>	<code>expr <= expr</code>	Сравнение "меньше или равно"	<code>PartialOrd</code>
<code>=</code>	<code>var = expr</code> , <code>ident = type</code>	Присваивание/эквивалентность	
<code>==</code>	<code>expr == expr</code>	Сравнение "равно"	<code>PartialEq</code>
<code>=></code>	<code>pat => expr</code>	Часть синтаксиса конструкции <code>match</code>	
<code>></code>	<code>expr > expr</code>	Сравнение "больше чем"	<code>PartialOrd</code>
<code>>=</code>	<code>expr >= expr</code>	Сравнение "больше или равно"	<code>PartialOrd</code>
<code>>></code>	<code>expr >> expr</code>	Битовый сдвиг вправо	<code>Shr</code>
<code>>>=</code>	<code>var >>= expr</code>	Битовый сдвиг вправо и присваивание	<code>ShrAssign</code>
<code>@</code>	<code>ident @ pat</code>	Pattern binding	
<code>^</code>	<code>expr ^ expr</code>	Побитовое исключающее ИЛИ	<code>BitXor</code>
<code>^=</code>	<code>var ^= expr</code>	Побитовое исключающее ИЛИ и присваивание	<code>BitXorAssign</code>
<code> </code>	<code>pat</code>		<code>pat</code>
<code> </code>	<code>expr</code>		<code>expr</code>
<code>=</code>	<code>var</code>		<code>= expr</code>
<code>?</code>	<code>expr?</code>	Возврат ошибки	<code>expr</code>

Обозначения не-операторы

Следующий список содержит все не-литералы, которые не являются операторами. То есть они не ведут себя как вызов функции или метода.

Таблица Б-2 показывает символы, которые появляются сами по себе и допустимы в различных местах.

Таблица Б-2: Автономный синтаксис

Обозначение	Объяснение
' ident '	Именованное время жизни или метка цикла
... u8 , ... i32 , ... f64 , ... usize , etc.	Числовой литерал определённого типа
"..."	Строковый литерал
r"..." , r#"..."# , r##"..."## , etc.	Необработанный строковый литерал, в котором не обрабатываются escape-символы
b"..."	Строковый байтовый литерал; создаёт [u8] вместо строки
br"..." , br#"..."# , br##"..."## , etc.	Необработанный строковый байтовый литерал, комбинация необработанного и байтового литерала
'...'!	Символьный литерал
b'...'!	ASCII байтовый литерал
—	...
!	Всегда пустой тип для расходящихся функций
—	«Игнорируемое» связывание шаблонов; также используется для читабельности целочисленных литералов

Таблица Б-3 показывает обозначения которые появляются в контексте путей иерархии модулей

Таблица Б-3. Синтаксис, связанный с путями

Обозначение	Объяснение
ident :: ident	Путь к пространству имён
:: path	Путь относительно корня крейта (т. е. явный абсолютный путь)
	Путь относительно текущего модуля (т. е. явный

<code>self::path</code>	относительный путь).
<code>super::path</code>	Путь относительно родительского модуля текущего модуля
<code>type::ident</code> , <code><type as trait>::ident</code>	Ассоциированные константы, функции и типы
<code><type>::...</code>	Ассоциированный элемент для типа, который не может быть назван прямо (например <code><&T>::...</code> , <code><[T]>::...</code> , etc.)
<code>trait::method(...)</code>	Устранение неоднозначности вызова метода путём именования типажа, который определяет его
<code>type::method(...)</code>	Устранение неоднозначности путём вызова метода через имя типа, для которого он определён
<code><type as trait>::method(...)</code>	Устранение неоднозначности вызова метода путём именования типажа и типа

Таблица Б-4 показывает обозначения которые появляются в контексте использования обобщённых типов параметров

Таблица Б-4: Обобщения

Обозначение	Объяснение
<code>path<...></code>	Определяет параметры для обобщённых параметров в типе (e.g., <code>Vec<u8></code>)
<code>path::<...></code> , <code>method::<...></code>	Определяет параметры для обобщённых параметров, функций, или методов в выражении. Часто называют <i>turbofish</i> (например <code>"42".parse::<i32>()</code>)
<code>fn ident<...></code> ...	Определение обобщённой функции
<code>struct</code> <code>ident<...> ...</code>	Определение обобщённой структуры
<code>enum ident<...></code> ...	Обявление обобщённого перечисления
<code>impl<...> ...</code>	Определение обобщённой реализации
<code>for<...> type</code>	Высокоуровневое связывание времени жизни
<code>type<ident=>type</code>	Обобщённый тип где один или более ассоциированных типов имеют определённое присваивание (например

Iterator<Item=T>()

Таблица Б-5 показывает обозначения которые появляются в контексте использования обобщённых типов параметров с ограничениями типов

Таблица Б-5: Ограничения типов

Обозначение	Объяснение
<code>T: U</code>	Обобщённый параметр <code>T</code> ограничивается до типов которые реализуют типаж <code>U</code>
<code>T: 'a</code>	Обобщённый тип <code>T</code> должен существовать не меньше чем <code>'a</code> (то есть тип не может иметь ссылки с временем жизни меньше чем <code>'a</code>)
<code>T : 'static</code>	Обобщённый тип <code>T</code> не имеет заимствованных ссылок кроме имеющих время жизни <code>'static</code>
<code>'b: 'a</code>	Обобщённое время жизни <code>'b</code> должно быть не меньше чем <code>'a</code>
<code>T: ?Sized</code>	Позволяет обобщённым типам параметра иметь динамический размер
<code>'a + trait</code> , <code>trait +</code> <code>trait</code>	Соединение ограничений типов

Таблица Б-6 показывает обозначения, которые появляются в контексте вызова или определения макросов и указания атрибутов элемента.

Таблица Б-6: Макросы и атрибуты

Обозначение	Объяснение
<code># [meta]</code>	Внешний атрибут
<code>#! [meta]</code>	Внутренний атрибут
<code>\$ident</code>	Подстановка в макросе
<code>\$ident:kind</code>	Захват макроса
<code>\$(...)</code>	Повторение макроса
<code>ident!(...)</code> , <code>ident!{...}</code> , <code>ident![...]</code>	Вызов макроса

Таблица Б-7 показывает обозначения, которые создают комментарии.

Таблица Б-7: Комментарии

Обозначение	Объяснение
//	Однострочный комментарий
//!	Внутренний однострочный комментарий документации
///	Внешний однострочный комментарий документации
/*...*/	Многострочный комментарий
/*!...*/	Внутренний многострочный комментарий документации
/**...*/	Внешний многострочный комментарий документации

Таблица Б-8 показывает обозначения, которые появляются в контексте использования кортежей.

Таблица Б-8: Кортежи

Обозначение	Объяснение
()	Пустой кортеж, он же пустой тип. И литерал и тип.
(expr)	Выражение в скобках
(expr,)	Кортеж с одним элементом выражения
(type,)	Кортеж с одним элементом типа
(expr, ...)	Выражение кортежа
(type, ...)	Тип кортежа
(type, ...)	Выражение вызова функции; также используется для инициализации структур-кортежей и вариантов-кортежей перечисления
expr.0, expr.1, etc.	Взятие элемента по индексу в кортеже

Таблица Б-9 показывает контексты, в которых используются фигурные скобки.

Таблица Б-9: Фигурные скобки

Контекст	Объяснение
{...}	Выражение блока
Type {...}	<code>struct</code> литерал

Таблица Б-10 показывает контексты, в которых используются квадратные скобки.

Таблица Б-10: Квадратные скобки

Контекст	Объяснение
[...]	Литерал массива
[expr; len]	Литерал массива, содержащий <code>len</code> копий <code>expr</code>
[type; len]	Массив, содержащий <code>len</code> экземпляров типа <code>type</code>
expr[expr]	Взятие по индексу в коллекции. Возможна перегрузка (<code>Index</code> , <code>IndexMut</code>)
expr[..], expr[a..], expr[..b], expr[a..b]	Взятие среза коллекции по индексу, используется <code>Range</code> , <code>RangeFrom</code> , <code>RangeTo</code> , или <code>RangeFull</code> как "индекс"

Дополнение В: Выводимые типажи

Во многих частях книги мы обсуждали атрибут `derive`, которые Вы могли применить к объявлению структуры или перечисления. Атрибут `derive` генерирует код по умолчанию для реализации типажа, который вы указали в `derive`.

В этом дополнении, мы расскажем про все типажи, которые вы можете использовать в атрибуте `derive`. Каждая секция содержит:

- Операции и методы, добавляемые типажом
- Как представлена реализация типажа через `derive`
- Что реализация типажа рассказывает про тип
- Условия, в которых разрешено или запрещено реализовывать типаж
- Примеры ситуаций, которые требуют наличие типажа

Если Вам понадобилось поведение отличное от поведения при реализации через `derive`, обратитесь к [документации по стандартной библиотеке](#) чтобы узнать как вручную реализовать типаж.

Остальные типажи, которые объявлены в стандартной библиотеке, не могут быть реализованы через `derive`. Эти типажи не имеют разумного поведения по умолчанию, поэтому Вам нужно будет реализовать их самим.

Пример типажа, который нельзя реализовать через `derive` - `Display`, который обрабатывает форматирование для конечных пользователей. Вы всегда должны сами рассмотреть лучший способ для отображения типа конечному пользователю. Какие части типа должны быть разрешены для просмотра конечному пользователю? Какие части они найдут подходящими? Какой формат вывода для них будет самым подходящим? Компилятор Rust не знает ответы на эти вопросы, поэтому он не может подобрать подходящее стандартное поведение.

Список типов, реализуемых через `derive`, в этом дополнении не является исчерпывающим: библиотеки могут реализовывать `derive` для их собственных типажей, составляя свои списки типажей, которые Вы можете использовать с помощью `derive`. Реализация `derive` включает в себя использование процедурных макросов, которые были рассмотрены в разделе "[Макросы](#)" главы 19.

Debug для отладочного вывода

Типаж `Debug` включает отладочное форматирование в форматируемых строках, которые вы можете указать с помощью `:?` внутри `{}` фигурных скобок.

Типаж `Debug` позволяет Вам напечатать объекты типа с целью отладки, поэтому Вы и другие программисты, использующие Ваш тип, смогут проверить объект в определённой точке выполнения программы.

Типаж `Debug` обязателен для некоторых случаях. Например, при использовании макроса `assert_eq!`. Этот макрос печатает значения входных аргументов если они не совпадают. Это позволяет программистам увидеть, почему эти объекты не равны.

PartialEq и Eq для сравнения равенства

Типаж `PartialEq` позволяет Вам сравнить объекты одного типа на эквивалентность, и включает для них использование операторов `==` и `!=`.

Использование `PartialEq` реализует метод `eq`. Когда `PartialEq` используют для структуры, два объекта равны если равны все поля объектов, и объекты не равны, если хотя бы одно поле отлично. Когда используется для перечислений, каждый вариант равен себе, и не равен другим вариантам.

Типаж `PartialEq` обязателен в некоторых случаях. Например для макроса `assert_eq!`, где необходимо сравнивать два объекта одного типа на эквивалентность.

Типаж `Eq` не имеет методов. Он сигнализирует что каждое значение аннотированного типа равно самому себе. Типаж `Eq` может быть применён только для типов реализующих типаж `PartialEq`, хотя не все типы, которые реализуют `PartialEq` могут реализовывать `Eq`. Примером являются числа с плавающей запятой: реализация чисел с плавающей запятой говорит, что два экземпляра со значениями не-число (`NaN`) не равны друг другу.

Типаж `Eq` необходим в некоторых случаях. Например, для ключей в `HashMap<K, V>`. Поэтому `HashMap<K, V>` может сказать, что два ключа являются одним и тем же.

PartialOrd и Ord для сравнения порядка

Типаж **PartialOrd** позволяет Вам сравнить объекты одного типа с помощью сортировки. Тип, реализующий **PartialOrd** может использоваться с операторами `<`, `>`, `<=`, и `>=`. Вы можете реализовать типаж **PartialOrd** только для типов, реализующих **PartialEq**.

Использование **PartialOrd** реализует метод `partial_cmp`, который возвращает `Option<Ordering>` который является `None` когда значения не выстраивают порядок. Примером значения, которое не может быть упорядочено, не являются числом (`NaN`) значение с плавающей запятой. Вызов `partial_cmp` с любым числом с плавающей запятой и значением `NaN` вернёт `None`.

Когда используется для структур, **PartialOrd** сравнивает два объекта путём сравнения значений каждого поля в порядке, в котором поля объявлены в структуре. Когда используется для перечислений, то варианты перечисления объявленные ранее будут меньше чем варианты объявленные позже.

Например, типаж **PartialOrd** может потребоваться для метода `gen_range` из `rand` крейта который генерирует случайные значения в заданном диапазоне (который определён выражением диапазона).

Типаж **Ord** позволяет знать, для двух значений аннотированного типа всегда будет существовать валидный порядок. Типаж **Ord** реализовывает метод `cmp`, который возвращает `Ordering` а не `Option<Ordering>` потому что валидный порядок всегда будет существовать. Вы можете применить типаж **Ord** только для типов, реализовывающих типаж **PartialOrd** и **Eq** (**Eq** также требует **PartialEq**). При использовании на структурах или перечислениях, `cmp` имеет такое же поведение, как и `partial_cmp` в **PartialOrd**.

Типаж **Ord** необходим в некоторых случаях. Например, сохранение значений в `BTreeSet<T>`, типе данных, который хранит информацию на основе порядка отсортированных данных.

Clone и Copy для дублирования значений

Типаж **Clone** позволяет вам явно создать глубокую копию значения, а также процесс дублирования может вызывать специальный код и копировать данные с кучи. Более детально про **Clone** смотрите в секции "[Способы взаимодействия](#)

переменных и данных: клонирование" в разделе 4.

Использование `Clone` реализует метод `clone`, который в случае реализации на всем типе, вызывает `clone` для каждой части данных типа. Это подразумевает, что все поля или значения в типе также должны реализовывать `Clone` для использования `Clone`.

Типаж `Clone` необходим в некоторых случаях. Например, для вызова метода `to_vec` для среза. Срез не владеет данными, содержащимися в нем, но вектор значений, возвращённый из `to_vec` должен владеть этими объектами, поэтому `to_vec` вызывает `clone` для всех данных. Таким образом, тип хранящийся в срезе, должен реализовывать `Clone`.

Типаж `Copy` позволяет дублировать значения копируя только данные, которые хранятся на стеке, произвольный код не требуется. Смотрите секцию "Стековые данные: Копирование" в разделе 4 для большей информации о `Copy`.

Типаж `Copy` не содержит методов для предотвращения перегрузки этих методов программистами, иначе бы это нарушило соглашение, что никакой произвольный код не запускается. Таким образом все программисты могут предполагать, что копирование значений будет происходить быстро.

Вы можете вывести `Copy` для любого типа все части которого реализуют `Copy`. Тип, который реализует `Copy` должен также реализовывать `Clone`, потому что тип реализующий `Copy` имеет тривиальную реализацию `Clone` который выполняет ту же задачу, что и `Copy`.

Типаж `Copy` нужен очень редко; типы, реализовывающие `Copy` имеют небольшую оптимизацию, то есть для него не нужно вызывать метод `clone`, который делает код более кратким.

Все, что вы делаете с `Copy` можно также делать и с `Clone`, но код может быть медленнее и требовать вызов метода `clone` в некоторых местах.

Hash для превращения значения в значение фиксированного размера

Типаж `Hash` позволяет превратить значение произвольного размера в значение фиксированного размера с использованием хеш-функции. Использование `Hash`

реализует метод `hash`. При реализации через derive, метод `hash` комбинирует результаты вызова `hash` на каждой части данных типа, то есть все поля или значения должны реализовывать `Hash` для использования `Hash` с помощью derive.

Типаж `Hash` необходим в некоторых случаях. Например, для хранения ключей в `HashMap<K, V>`, для их более эффективного хранения.

Default для значений по умолчанию

Типаж `Default` позволяет создавать значение по умолчанию для типа.

Использование `Default` реализует функцию `default`. Стандартная реализация метода `default` вызовет функцию `default` на каждой части данных типа, то есть для использования `Default` через derive, все поля и значения типа данных должны также реализовывать `Default`.

Функция `Default::default` часто используется в комбинации с синтаксисом обновления структуры, который мы обсуждали в секции "["Создание экземпляра структуры из экземпляра другой структуры с помощью синтаксиса обновления структуры"](#)" главы 5. Вы можете настроить несколько полей для структуры, а для остальных полей установить значения с помощью `..Default::default()`.

Типаж `Default` необходим в некоторых случаях. Например, для метода `unwrap_or_default` у типа `Option<T>`. Если значение `Option<T>` будет `None`, метод `unwrap_or_default` вернёт результат вызова функции `Default::default` для типа `T`, хранящегося в `Option<T>`.

Дополнение Г - Средства разработки

В этом дополнении мы расскажем про часто используемые средства разработки, предоставляемые Rust. Мы рассмотрим автоматическое форматирование, быстрый путь исправления предупреждений, линтер, и интеграцию с IDE.

Автоматическое форматирование с `rustfmt`

Инструмент `rustfmt` отформатирует Ваш код в соответствии с принятым в сообществе стилем. Многие совместные проекты используют `rustfmt` для предотвращения споров об используемом стиле для написания кода на Rust: каждый форматирует свой код с помощью этой утилиты.

Для установки `rustfmt`, введите следующее:

```
$ rustup component add rustfmt
```

Эта команда установит `rustfmt` и `cargo-fmt`, также как Rust даёт Вам одновременно `rustc` и `cargo`. Для форматирования проекта, использующего Cargo, введите следующее:

```
$ cargo fmt
```

Эта команда отформатирует весь код на языке Rust в текущем крейте. Будет изменён только стиль кода, семантика останется прежней. Для большей информации о `rustfmt`, смотрите [документацию](#).

Исправление кода с `rustfix`

Инструмент `rustfix` включён в Rust, и вам не нужно дополнительно его устанавливать. Эта утилита умеет автоматически исправлять некоторые предупреждения компиляции. Если Вы пишите код на Rust, Вы наверное встречали предупреждения компилятора. Например, рассмотрим следующий код:

Файл: `src/main.rs`

```
fn do_something() {}

fn main() {
    for i in 0..100 {
        do_something();
    }
}
```

Мы вызываем функцию `do_something` 100 раз, но никогда не используем переменную `i` в теле цикла `for`. Rust предупреждает нас об этом:

```
$ 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 предлагает нам использовать `_i` как имя переменной: нижнее подчёркивание в начале идентификатора предполагает, что мы его не используем. Мы можем автоматически применить это предположение с помощью `rustfix`, запустив 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 посмотрим в `src/main.rs` снова, мы увидим что `cargo fix` изменил наш код:

Файл: `src/main.rs`

```
fn do_something() {}

fn main() {
    for _i in 0..100 {
        do_something();
    }
}
```

Переменная цикла `for` теперь носит имя `_i`, и предупреждение больше не появляется.

Также Вы можете использовать команду `cargo fix` для перемещения вашего кода между различными редакциями Rust. Редакции будут рассмотрены в дополнении Д.

Больше проверок с Clippy

Инструмент Clippy является коллекцией проверок (lints) для анализа Вашего кода, поэтому Вы можете найти простые ошибки и улучшить ваш Rust код.

Для установки Clippy, введите следующее:

```
$ rustup component add clippy
```

Для запуска проверок Clippy's для проекта Cargo, введите следующее:

```
$ cargo clippy
```

Например, скажем что Вы хотите написать программу, в которой будет использоваться приближенная математическая константа, такая как число Пи, как в следующей программе:

Файл: src/main.rs

```
fn main() {
    let x = 3.1415;
    let r = 8.0;
    println!("the area of the circle is {}", x * r * r);
}
```

Запуск `cargo clippy` для этого проекта вызовет следующую ошибку:

```
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
```

Эта ошибка сообщает Вам, что эта константа уже определена более точно в Rust, и используя её Ваша программа будет работать более правильно. Затем Вы бы изменили Ваш код, добавив константу **PI**. Следующий код не вызовет каких-либо ошибок или предупреждений от Clippy:

Файл: src/main.rs

```
fn main() {  
    let x = std::f64::consts::PI;  
    let r = 8.0;  
    println!("the area of the circle is {}", x * r * r);  
}
```

Для большей информации о Clippy смотрите [документацию](#).

Интеграция с IDE используя Rust Language Server

Для помощи в интеграции с IDE, Rust распространяет *Rust Language Server* (**rls**). Этот инструмент говорит на [Language Server Protocol](#), который является спецификацией для общения между IDE и языками программирования. Различные клиенты могут использовать **rls**, например [плагин Rust для Visual Studio Code](#).

Для установки **rls**, введите следующее:

```
$ rustup component add rls
```

Затем установите поддержку языкового сервера в Вашей IDE. Вы получите такие вещи, как автодополнение, просмотр определения и подсветку ошибок.

Для большей информации про **rls**, смотрите [документацию](#).

Приложение E - Редакции языка

В главе 1, можно увидеть, что команда `cargo new` добавляет некоторые метаданные о редакции языка в файл *Cargo.toml*. Данное приложение рассказывает, что они означают.

Язык Rust и его компилятор имеют шестинедельный цикл выпуска, означающий, что пользователи постоянно получают новые функции. В других языках обычно выпускают большие обновления, но редко. Команда Rust выпускает меньшие обновления, но более часто. Через некоторое время все эти небольшие изменения накапливаются. Между релизами обычно сложно оглянуться назад и сказать "Ого, язык сильно изменился между версиями Rust 1.10 и Rust 1.31!"

Каждые два или три года, команда Rust выпускает новую редакцию языка (*Rust edition*). Каждая редакция объединяет все новые особенности, которые попали в язык с новыми пакетами, с полной, обновлённой документацией и инструментарием. Новые редакции поставляются как часть шестинедельного процесса релизов.

Для разных людей редакции служат разным целям:

- Для активных пользователей новая редакция приносит все инкрементальные изменения в удобный и понятный пакет.
- Для тех, кто языком не пользуется, новая реакция является сигналом, что некоторые важные улучшения, на которые возможно надо взглянуть ещё раз, попали в язык.
- Для тех кто разрабатывает на Rust, новая редакция даёт некоторую точку отсчёта для проекта в целом.

На момент написания доступны две редакции Rust: Rust 2015 и Rust 2018. Данная книга написана с использованием идиом редакции Rust 2018.

Ключ `edition` в конфигурационном файле *Cargo.toml* отображает, какую редакцию компилятор должен использовать для вашего кода. Если ключа нет, то для обратной совместимости компилятор Rust использует редакцию `2015`.

Любой проект может выбрать редакцию отличную от редакции по умолчанию, которая равна 2015. Редакции могут содержать несовместимые изменения, включая новые ключевые слова, которые могут конфликтовать с идентификаторами в коде. Однако, пока вы не переключитесь на новую редакцию, ваш код будет продолжать компилироваться даже после обновления используемой

версии компилятора.

Все версии компилятора Rust поддерживают любую редакцию, которая предшествовала выпуску текущей, и они могут линковать пакеты любой поддерживаемой редакции. Изменения редакций действуют только на способ начального разбора компилятором исходного кода. Поэтому, если вы используете 2015 редакцию, а одна из ваших зависимостей использует 2018, ваш проект будет скомпилирован и сможет пользоваться этой зависимостью. Обратная ситуация, когда ваш проект использует Rust 2018, а зависимость использует Rust 2015, работает таким же образом.

Внесём ясность: большая часть возможностей будет доступна во всех редакциях. Разработчики, использующие любую редакцию Rust, будут продолжать получать улучшения по мере выпуска новых релизов. Однако в некоторых случаях, в основном, когда добавляются новые ключевые слова, некоторые новые возможности могут быть доступны только в последних редакциях. Нужно переключить редакцию, чтобы воспользоваться новыми возможностями.

Для получения больше деталей, есть полная книга *Edition Guide* про редакции, в которой перечисляются различия между редакциями и объясняется, как автоматически обновить свой код на новую редакцию с помощью команды `cargo fix`.

Приложение Е: Переводы книги

Для ресурсов на языках, отличных от английского. Большинство из них все ещё в разработке; см. [ярлык «Переводы»](#), чтобы помочь или сообщить нам о новом переводе!

- [Português \(BR\)](#)
- [Português \(PT\)](#)
- [简体中文](#)
- [正體中文](#)
- [Українська](#)
- [Español, alternate](#)
- [Italiano](#)
- [Русский](#)
- [한국어](#)
- [日本語](#)
- [Français](#)
- [Polski](#)
- [Cebuano](#)
- [Tagalog](#)
- [Esperanto](#)
- [ελληνική](#)
- [Svenska](#)
- [Farsi](#)
- [Deutsch](#)
- [Turkish, online](#)
- [हिंदी](#)
- [ไทย](#)

Дополнение Ё - Как создаётся Rust и “Nightly Rust”

Это дополнение рассказывает как создаётся Rust, и как это влияет на Вас как на разработчика.

Стабильность без стагнации

Как язык, Rust много заботиться о стабильности Вашего кода. Мы хотим чтобы Rust был прочным фундаментом, вашей опорой, и если бы все постоянно менялось, это было бы невозможно. В то же время, если мы не можем экспериментировать с различными возможностями, мы не можем обнаружить важные проблемы до релиза, когда мы не можем их изменить.

Нашим решением проблемы является “стабильность без стагнации”, и наш руководящий принцип: Вы никогда не должны бояться перехода на новую стабильную версию Rust. Каждое обновление должно быть безболезненным, но также должно добавлять новые функции, меньше дефектов и более быструю скорость компиляции.

Ту-ту! Каналы выпуска и поездка на поезде

Разработка языка Rust работает по принципу *расписания поездов*. То есть, вся разработка совершается в ветке **master** Rust репозитория. Выпуски следуют модели последовательного выпуска продукта (software release train), которая была использована Cisco IOS и другими программными продуктами. Есть три канала выпуска Rust:

- Ночной (Nightly)
- Бета (Beta)
- Стабильный (Stable)

Большинство Rust разработчиков используют стабильную версию, но те кто хотят попробовать экспериментальные новые функции, должны использовать Nightly или Beta.

Приведём пример, как работает процесс разработки и выпуска новых версий. Давайте предположим, что команда Rust работает над версией Rust 1.5. Его релиз состоялся в декабре 2015 года, но это даст реалистичность номера версии. Была

добавлена новая функциональность в Rust: новые коммиты в ветку **master**. Каждую ночь выпускается новая новая версия Rust. Каждый день является днём выпуска новой версии и эти выпуски создаются нашей структурой автоматически. По мере того как идёт время, наши выпуски выглядят так:

```
nightly: * - - * - - *
```

Каждые шесть недель наступает время подготовки новой Beta версии! Ветка **beta** Rust репозитория ответвляется от ветки **master**, используемой версией Nightly. Теперь мы имеем два выпуска:

```
nightly: * - - * - - *
          |
beta:      *
```

Многие пользователи Rust не используют активно бета-версию, но тестируют бета-версию в их системе CI для помощи Rust обнаружить проблемы обратной совместимости. В это время каждую ночь выпускается новая версия Nightly:

```
nightly: * - - * - - * - - * - - *
          |
beta:      *
```

Предположим, что была найдена регрессия. Хорошо, что мы можем протестировать бета-версию перед тем как регрессия попала в стабильную версию! Исправление отправляется в ветку **master**, поэтому версия nightly исправлена и затем исправление также направляется в ветку **beta**, и происходит новый выпуск бета-версии:

```
nightly: * - - * - - * - - * - - * - - *
          |
beta:      * - - - - - - - - *
```

Через шесть недель после выпуска бета-версии, наступает время для выпуска стабильной версии! Ветка **stable** создаётся из ветки **beta**:

```
nightly: * - - * - - * - - * - - * - - * - -
          |
beta:      * - - - - - - - - *
          |
stable:   *
```

Ура! Rust 1.5 выпущена! Но мы также забыли про одну вещь: так как прошло шесть недель, мы должны выпустить бета-версию следующей версии Rust 1.6. Поэтому после ответвления ветки **stable** из ветки **beta**, следующая версия **beta** ответвляется снова от **nightly**:

```
nightly: * - - * - - * - - * - - * - * - *
          |           |
beta:      * - - - - - - - *           *
          |
stable:   *
```

Это называется “модель поезда” (train model), потому что каждые шесть недель выпуск “покидает станцию”, но ему все ещё нужно пройти канал beta, чтобы попасть в стабильную версию.

Rust выпускается каждые шесть недель, как часы. Если вы знаете дату одного выпуска Rust, вы знаете дату выпуска следующего: это шесть недель позднее. Хорошим аспектом выпуска версий каждые шесть недель является то, что следующий поезд прибывает скоро. Если какая-то функция не попадает в релиз, не надо волноваться: ещё один выпуск произойдёт очень скоро! Это помогает снизить давление в случае если функция возможно не отполирована к дате выпуска.

Благодаря этому процессу, вы всегда можете посмотреть следующую версию Rust и убедиться, что на неё легко будет перейти: если бета-выпуск будет работать не так как ожидалось, вы можете сообщить об этом разработчикам и он будет исправлен перед выпуском стабильной версии! Поломки в бета-версии случаются относительно редко, но **rustc** все ещё является частью программного обеспечения, поэтому дефекты все ещё существуют.

Нестабильные функции

У этой модели выпуска есть ещё один плюс: нестабильные функции. Rust использует технику называемую “флаги функционала” (feature flags) для определения функций, которые были включены в выпуске. Если новая функция находится в активной разработке, она попадает в ветку **master**, и поэтому попадает в ночную версию, но с *флагом функции* (feature flag). Если как пользователь, вы хотите попробовать работу такой функции, находящейся в разработке, вы должны использовать ночную версию Rust и указать в вашем исходном коде определённый флаг.

Если вы используете бета или стабильную версию Rust, Вы не можете использовать флаги функций. Этот ключевой момент позволяет использовать на практике новые возможности перед их стабилизацией. Это может использоваться желающими идти в ногу со временем, а другие могут использовать стабильную версию и быть уверенными что их код не сломается. Стабильность без стагнации.

Эта книга содержит информацию только о стабильных возможностях, так как разрабатываемые возможности продолжают меняться в процессе и несомненно они будут отличаться в зависимости от того, когда эта книга написана и когда эти возможности будут включены в стабильные сборки. Вы можете найти информацию о возможностях ночной версии в интернете.

Rustup и роль ночной версии Rust

Rustup делает лёгким изменение между различными каналами Rust, на глобальном или локальном для проекта уровне. По умолчанию устанавливается стабильная версия Rust. Для установки ночной версии выполните команду:

```
$ rustup toolchain install nightly
```

Вы можете также увидеть все установленные *инструменты разработчика* (*toolchains*) (версии Rust и ассоциированные компоненты) с помощью **rustup**. Это пример вывода у одного из авторов Rust с компьютером на Windows:

```
> rustup toolchain list
stable-x86_64-pc-windows-msvc (default)
beta-x86_64-pc-windows-msvc
nightly-x86_64-pc-windows-msvc
```

Как видите, стабильный набор инструментов (*toolchain*) используется по умолчанию. Большинство пользователей Rust используют стабильные версии большую часть времени. Возможно, вы захотите использовать стабильную большую часть времени, но использовать каждую ночную версию в конкретном проекте, потому что заботитесь о передовых возможностях. Для этого вы можете использовать команду **rustup override** в каталоге этого проекта, чтобы установить ночной набор инструментов, должна использоваться команда **rustup**, когда вы находитесь в этом каталоге:

```
$ cd ~/projects/needs-nightly
$ rustup override set nightly
```

Теперь каждый раз, когда вы вызываете `rustc` или `cargo` внутри `~/projects/needs-nightly`, `rustup` будет следить за тем, чтобы вы используете ночную версию Rust, а не стабильную по умолчанию. Это очень удобно, когда у вас есть множество Rust проектов!

Процесс RFC и команды

Итак, как вы узнаете об этих новых возможностях? Модель разработки Rust следует процессу запроса комментариев (*RFC - Request For Comments*). Если хотите улучшить Rust, вы можете написать предложение, которое называется RFC.

Любой может написать RFC для улучшения Rust, предложения рассматриваются и обсуждаются командой Rust, которая состоит из множества тематических подгрупп. На [веб-сайте Rust](#) есть полный список команд, который включает команды для каждой области проекта: дизайн языка, реализация компилятора, инфраструктура, документация и многое другое. Соответствующая команда читает предложение и комментарии, пишет некоторые собственные комментарии и в конечном итоге, приходит к согласию принять или отклонить эту возможность.

Если новая возможность принята и кто-то может реализовать её, то задача открывается в репозитории Rust. Человек реализующий её, вполне может не быть тем, кто предложил эту возможность! Когда реализация готова, она попадает в `master` ветвь с флагом функции, как мы обсуждали в разделе "["Нестабильных функциях"](#)".

Через некоторое время, разработчики Rust использующиеочные выпуски, смогут опробовать новую возможность, члены команды обсудят её, как она работает вочной версии и решат, должна ли она попасть в стабильную версию Rust или нет. Если принимается решение двигать её вперёд, ограничение функции с помощью флага убирается и функция теперь считается стабильной! Она едет в новую стабильную версию Rust.