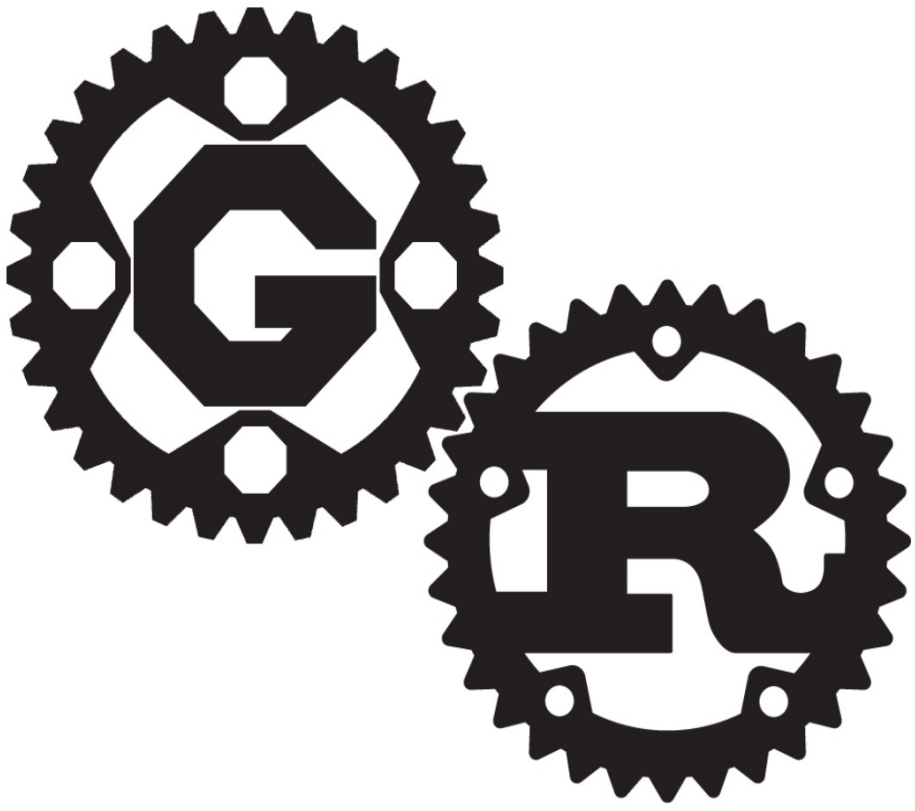
Язык программирования Rust

Перевод на русский язык
"The Rust Programming Language"



Введение к русскоязычному переводу

Эта книга представляет собой перевод "The Rust Programming Language". Оригинал книги расположен здесь.

ВНИМАНИЕ! Перевод не окончен, поэтому могут встречаться ошибки.

- Читать книгу
- Скачать в PDF
- Скачать в EPUB
- Скачать в MOBI

Помощь в переводе приветствуется. –^

Соавторам

Ход выполнения перевода.

Правила перевода.

Если вы беретесь за работу над каким-нибудь файлом, просьба отписаться здесь.

Благодарности

Выражаем благодарность всем, кто принимал участие в создании этой книги.

От @kgv: "Хочу поблагодарить моих родителей: **Таню** и **Володю**. Без них не было бы этой книги".

Ошибки

Если вы встретили ошибку или неточность, пожалуйста, напишите о ней сюда.

Ресурсы

- rustbook расположен здесь
- gitbook расположен здесь
- github репозиторий расположен здесь

Язык программирования Rust

Добро пожаловать! Эта книга обучает основным принципам работы с языком программирования Rust. Rust - это системный язык программирования, внимание которого сосредоточено на трёх задачах: безопасность, скорость и параллелизм. Он решает эти задачи без сборщика мусора, что делает его полезным в ряде случаев, когда использование других языков было бы не целесообразно: при встраивании в другие языки, при написании программ с особенными пространственными и временными требованиями, при написании низкоуровневого кода, такого как драйверы устройств и операционные системы. Во время компиляции Rust делает ряд проверок безопасности. За счёт этого не появляется накладных расходов во время выполнения приложения и устраняются все гонки данных. Это даёт Rust'y преимущество над другими языками программирования, имеющими аналогичную направленность. Rust также направлен на достижение ‘абстракции с нулевой стоимостью‘, хотя некоторые из этих абстракций ведут себя как в языках высокого уровня. Даже тогда Rust по-прежнему обеспечивает точный контроль, как делал бы язык низкого уровня.

Книга "Язык программирования Rust" делится на семь разделов. Это введение является первым из них. Затем идут:

- С чего начать - Настройка компьютера для разработки на Rust.
- Изучение Rust - Обучение программированию на Rust на примере небольших проектов.
- Эффективное использование Rust - Понятия более высокого уровня для написания качественного код на Rust.
- Синтаксис и семантика - Каждая концепция Rust разбивается на небольшие кусочки.
- Нестабильные фичи Rust - Передовые фичи, которые пока не добавлены в стабильную сборку.
- Глоссарий - Ссылки на термины, используемые в книге.

После прочтения этого введения, в зависимости от ваших предпочтений, вы можете продолжить дальнейшее изучение либо в направлении ’Изучение Rust’, либо в направлении ’Синтаксис и семантика’. Если вы предпочитаете изучить язык на примере реального проекта, лучшим выбором будет раздел ’Изучение Rust’. Раздел ’Синтаксис и семантика’ подойдёт тем, кто предпочитает тщательно изучить каждую концепцию языка отдельно, перед тем как двигаться дальше. Большое количество перекрёстных ссылок соединяет эти части воедино.

Краткое введение в Rust

Чем же Rust может заинтересовать вас? Давайте рассмотрим несколько небольших примеров кода, чтобы продемонстрировать некоторые из его сильных сторон.

Основная концепция, которая делает Rust уникальным, называется ‘владение’. Рассмотрим следующий небольшой пример:

```
fn main() {
    let mut x = vec!["Hello", "world"];
}
```

Эта программа создаёт привязку переменной с именем `x`. Значением этого связывания является `Vec<T>`, ‘вектор‘, который мы создаём с помощью макроса, определённого в стандартной библиотеке. Этот макрос называется `vec`, и при его вызове используется символ `!`. Это следует из общего принципа Rust: делать вещи явными. Макрос может делать значительно более сложные вещи, чем вызовов функций, и поэтому они визуально отличаются. Символ `!` также помогает при парсинге, что облегчает написание инструментов, а это тоже важно.

Мы использовали `mut`, чтобы сделать `x` изменяемой: в Rust по умолчанию привязки являются неизменяемыми. Дальше в примере мы будем изменять этот вектор.

Стоит также отметить, что здесь нам не нужно указывать тип: несмотря на то, что Rust является статически типизированным, нам не нужно явно указывать тип. Rust может выводить типы, что балансирует мощь статической типизации с многословностью указания типов.

Rust предпочитает выделять память в стеке, а не в куче: `x` находится непосредственно в стеке. Однако тип `Vec<T>` выделяет пространство для элементов вектора в куче. Если вы не знакомы с различиями этих двух видов выделения памяти, вы пока что можете просто проигнорировать эту информацию или же ознакомиться с разделом ‘Стек и Куча‘. Как системный язык программирования, Rust даёт вам возможность контролировать выделение памяти, но не будем забегать вперёд, мы только начинаем изучение языка.

Ранее мы упоминали, что ‘владение‘ является основной новой концепцией в Rust. В терминологии Rust, `x` ‘владеет‘ вектором. Это означает, что как только `x` выходит из области видимости, выделенная для вектора память будет освобождена. Когда это будет происходить, определяется средствами компилятора Rust, а не через механизмы, такие как сборщик мусора. Другими словами, в Rust вы не вызываете функции вроде `malloc` и `free` самостоятельно: компилятор статически определяет, когда нужно выделить или освободить память, и вставляет эти вызовы самостоятельно. Человек может совершить ошибку при использовании этих вызовов, а компилятор - никогда.

Давайте добавим ещё одну строку в наш пример:

```
fn main() {
    let mut x = vec!["Hello", "world"];

    let y = &x[0];
}
```

Мы создаём ещё одну привязку, `y`. В этом случае, `y` является ‘ссылкой‘ на первый элемент вектора. Ссылки в Rust похожи на указатели в других языках, но с дополнительными проверками безопасности на этапе компиляции. Ссылки взаимодействуют с системой прав владения при помощи ‘заимствования’. Ссылки заимствуют то, на что они указывают, а не

получают права владения им. Разница в том, что при заимствовании, ссылка не освобождает основную память, когда выходит за пределы области видимости. Если бы это было не так, то память освобождалась два раза, что плохо!

Давайте добавим третью строку. На первый взгляд в коде нет ничего такого, но он вызывает ошибку компиляции:

```
fn main() {
    let mut x = vec!["Hello", "world"];

    let y = &x[0];

    x.push("foo");
}
```

`push` - это метод, который добавляет ещё один элемент в конец вектора. Когда мы пытаемся скомпилировать эту программу, то получаем ошибку:

```
error: cannot borrow `x` as mutable because it is also borrowed as immutable
    x.push(4);
    ^
note: previous borrow of `x` occurs here; the immutable borrow prevents
subsequent moves or mutable borrows of `x` until the borrow ends
    let y = &x[0];
             ^
note: previous borrow ends here
fn main() {

}
^
```

Вот так! Компилятор Rust в некоторых случаях выдаёт достаточно подробные ошибки, и это как раз один из таких случаев. Как объясняется в ошибке, мы не можем изменить связывание (не можем вызвать метод `push`). Это потому, что у нас уже есть ссылка на элемент вектора, `y`. Изменять вектор, пока существует другая ссылка на него, опасно, потому что можно сделать ссылку недействительной. В данном конкретном случае, когда мы создаём вектор, у нас есть выделенное пространство памяти только для трёх элементов. Добавление четвёртого элемента будет означать выделение нового блока памяти для всех этих элементов, копирование старых значений и обновление внутреннего указателя на эту память. Всё это работает просто отлично. Проблема заключается в том, что `y` не будет обновлена, из-за чего мы получим ‘зависший указатель‘. И это плохо. В этом случае любое использование `y` будет означать ошибку, и поэтому компилятор поймал её для нас.

Так как же нам решить эту проблему? Есть два подхода, которые мы можем использовать. Первый заключается в создании копии вместо ссылки:

```
fn main() {
    let mut x = vec!["Hello", "world"];

    let y = x[0].clone();

    x.push("foo");
}
```

По умолчанию, Rust использует семантику перемещения, поэтому, если мы хотим сделать копию некоторых данных, мы должны вызывать метод `clone()`. В этом примере `y` больше не является ссылкой на вектор, хранящийся в `x`, но является копией его первого элемента, `"Hello"`. Теперь, когда у нас больше нет ссылки, метод `push()` работает просто отлично.

Если мы все же хотим ссылку, то следует использовать другой вариант: убедиться, что наша ссылка выходит из области видимости прежде чем мы попытаемся сделать изменения. Это выглядит примерно так:

```
fn main() {
    let mut x = vec!["Hello", "world"];

    {
        let y = &x[0];
    }

    x.push("foo");
}
```

Мы создали внутреннюю область видимости с помощью дополнительных фигурных скобок. `y` выйдет за пределы этой области видимости до вызова метода `push()`, и поэтому все будет хорошо.

Концепция права владения хороша не только для предотвращения повисших указателей, но также всей совокупности связанных с этим проблем, таких как: недействительность итератора, параллелизм и многое другое.

С чего начать

Этот первый раздел книги, который позволит вам начать работать с Rust и его инструментами. Сначала - мы установим Rust. Затем напишем классическую программу ‘Hello World’. И наконец - поговорим о Cargo, который представляет собой систему сборки и менеджер пакетов в Rust.

Установка Rust

Первым шагом к использованию Rust является его установка! Есть несколько способов установить Rust, но самый простой из них - использовать скрипт `rustup`. Если вы используете Linux или Mac, то всё, что вам нужно сделать - это ввести следующую команду в консоль (вам не нужно вводить символ `$`, он лишь показывает приглашение командной строки к вводу новой команды):

```
$ curl -sf -L https://static.rust-lang.org/rustup.sh | sh
```

Если вы беспокоитесь о потенциальной безопасности использования команды `curl | sh`, то продолжайте читать далее. Вы также можете использовать двухступенчатый вариант установки и изучить наш установочный скрипт:

```
$ curl -f -L https://static.rust-lang.org/rustup.sh -O
$ sh rustup.sh
```

Если же вы используете Windows, то, пожалуйста, скачайте один из установочных пакетов: 32-битный или 64-битный и запустите его.

Удаление

Если вы решили, что Rust вам больше не нужен, то мы будем чуть-чуть огорчены, но это нормально. Не каждый язык программирования отлично подходит для всех. Просто запустите скрипт деинсталляции:

```
$ sudo /usr/local/lib/rustlib/uninstall.sh
```

Если вы использовали установщик Windows, то просто повторно запустите `.msi`, который предложит вам возможность удаления.

Некоторые люди, причём не безосновательно, насторожились, когда мы сказали использовать `curl | sh`. Когда вы делаете так, вы должны доверять тем хорошим людям, которые поддерживают Rust, и не бояться, что они попытаются взломать ваш компьютер и сделать какие-либо плохие вещи. Озабоченность своей безопасностью - это очень хорошо. Если вы один из таких людей, пожалуйста посмотрите в документации как собрать Rust из исходных кодов или скачайте уже скомпилированный Rust. Мы обещаем, что данный способ не будет использоваться для установки Rust'а всегда: скрипт был сделан для быстрого обновления пока Rust находится в стадии alpha.

Мы так же должны упомянуть официально поддерживаемые платформы:

- Windows (7, 8, Server 2008 R2)
- Linux (2.6.18 и более новые, разные дистрибутивы), x86 и x86-64
- OSX 10.7 (Lion) и более новые, x86 и x86-64

Rust активно тестируется на всех этих платформах, а также на некоторых других, например на Android. Но мы указали те, на которых Rust точно должен работать, ибо для этих платформ он тестируется больше всего.

Напоследок, замечание о Windows. Rust считает, что Windows - это первоклассная платформа для релиза, но если быть честными, то опыт разработки для Windows не на столько хорош, как для Linux/OS X. Мы работаем над этим! Если что-то не работает, то это ошибка. Пожалуйста, дайте нам знать, если такое произойдёт. Каждый коммит тестируется на Windows, впрочем так же, как и на любой другой платформе.

Если вы уже установили Rust, то откройте терминал и введите это:

```
$ rustc --version
```

Вы должны увидеть версию, хэш коммита, дату коммита и дату сборки:

```
rustc 1.0.0 (a59de37e9 2015-05-13) (built 2015-05-14)
```

Итак, теперь у вас есть установленный Rust! Поздравляем!

Установщик также устанавливает документацию, которая доступна без подключения к сети. На UNIX системах она располагается в каталоге `/usr/local/share/doc/rust`. В Windows - в директории `share/doc`, относительно того куда вы установили Rust.

Также есть ещё ряд мест, где можно получить помощь. Канал #rust на irc.mozilla.org, к которому вы можете подключиться через Mibbit. Нажмите на эту ссылку, и вы будете общаться в чате с другими Rustaceans (это дурашливое прозвище, которым мы себя называем), и мы поможем вам. Другие полезные ресурсы, посвящённые Rust: форум пользователей, /r/rust subreddit, stack overflow. Русскоязычные ресурсы: канал #rust-ru на irc.mozilla.org, google groups.

Hello, world!

Теперь, когда вы установили Rust, давайте напишем первую программу на Rust. Традиционно, в любом новом изучаемом языке программирования, первая программа только выводит на экран текст "Hello, World!". Хорошо начинать с такой простой программы, т.к. вы можете убедиться, что ваш компилятор не только установлен, но и работает правильно. Вывод информации на экран будет замечательным способом проверить это.

Первое, с чего мы должны начать, это создать файл для нашего кода. Мне нравится размещать каталог `projects` в домашней директории и хранить там все мои проекты. Для Rust не имеет значения где располагается ваш код.

На самом деле это приводит к ещё одной проблеме о которой мы должны предупредить: данное руководство предполагает, что у вас есть базовые навыки работы в командной строке. У Rust нет специфичных требований к вашей среде разработки или тому, где вы храните свой код. Если вы больше предпочитаете использовать IDE, можно посмотреть на проект SolidOak, или на плагины к вашей любимой IDE. Существует множество расширений разного качества, созданные сообществом. Команда разработчиков Rust'a так же предоставила плагины для различных редакторов. Настройка вашего редактора или IDE выходит за пределы данного руководства. Посмотрите руководство по использованию выбранного вами плагина.

С учётом вышесказанного, давайте сделаем каталог для нашей программы в директории с проектами.

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

Если вы используете Windows и не используете PowerShell, `~` может не работать. Обратитесь к документации вашей оболочки для уточнения деталей.

Теперь создадим новый файл для текста программы. Назовём наш файл `main.rs`. Файлы с исходными текстами на Rust'e всегда имеют расширение `.rs`. Если вы хотите использовать в имени вашего файла больше одного слова, разделяйте их подчёркиванием: `hello_world.rs`, а не `helloworld.rs`.

Теперь когда файл открыт, добавьте в него следующий код:

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

Сохраните файл, а затем введите эти команды в ваше окно терминала:

```
$ rustc main.rs
$ ./main # или main.exe в Windows
Hello, world!
```

Работает! Разберём подробнее что же произошло.

```
fn main() {

}
```

Эти строки определяют ‘функцию‘ в Rust'е. Функция `main` особенна: это начало каждой программы на Rust. Первая строка говорит: "Я объявляю функцию именуемую `main`, которая не получает параметров и ничего не возвращает". Если бы мы хотели передать в функцию параметры, то указали бы их в скобках (`(` и `)`). Так как нам не надо ничего возвращать из этой функции, мы можем опустить указание типа возвращаемого значения. Мы вернёмся к этому позже.

Вы должны были заметить, что функция обёрнута в фигурные скобки (`{` и `}`). Rust требует их вокруг тел всех функций. Так же хорошим стилем считается ставить открывающую фигурную скобку на той же строке, что и объявление функции, отделённую от него одним пробелом.

Теперь эта строка:

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

Эта строка делает всю работу в нашей маленькой программе. Тут есть несколько нюансов, которые имеют существенное значение. Во-первых, отступ в четыре пробела, а не табуляция. Пожалуйста, настройте выбранный вами редактор так, чтобы вставлять четыре пробела при помощи клавиши табуляции. Мы предоставляем некоторые примеры настроек для различных редакторов.

Теперь разберёмся с `println!()`. Это вызов макроса, которыми представлено метапрограммирование в Rust'е. Если бы вместо макроса была функция, это бы выглядело следующим образом: `println()`. Для достижения нашей цели, нас не должна волновать эта разница. Просто знайте, что иногда вы будете видеть `!`, по которому можно понять, что вы вызываете макрос вместо обычной функции. Rust реализует `println!` как макрос вместо функции по веским причинам, но это достаточно глубокая тема о которой мы поговорим позже. И последнее, что нужно отметить: макросы Rust'а значительно отличаются от макросов C, если вы их использовали. Не бойтесь использовать макросы. В конце концов мы вернёмся к деталям, а сейчас просто доверьтесь нам.

Идём дальше. `"Hello, world!"` - это ‘строка‘. Строки удивительно сложная тема в системном языке программирования. Это ‘статически расположенная в памяти‘ строка. Если вы хотите больше узнать про расположении в памяти, рекомендуем посмотреть про стек и кучу, но если вы не хотите этого, вы можете этого и не делать. Мы передаём строку в качестве аргумента в `println!`, который выводит строки на экран. Это достаточно просто!

В завершение, строка заканчивается точкой с запятой (`;`). Rust `выражение-ориентированный язык`, а это означает, что в нём большая часть вещей является выражением. `;` используется для указания, что выражение закончилось и начинается следующее. Большинство строк кода на Rust заканчивается на `;`.

На самом деле, завершением будет сборка и запуск нашей программы. Соберём программу компилятором `rustc`, передав ему в качестве аргумента название нашего файла с кодом:

```
$ rustc main.rs
```

Это похоже на `gcc` или `clang`, если вы программировали раньше на C или C++. Rust создаст двоичный исполняемый файл. Вы можете убедиться в этом с помощью `ls`:

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

Или в Windows:

```
$ dir
main.exe  main.rs
```

У нас есть два файла: наш исходный код, с расширением `.rs` и исполняемый файл (`main.exe` в Windows, `main` в остальных случаях)

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

Мы получили выведенный в окне терминала текст `"Hello, world!"`.

Если вы перешли из динамически-типизированных языков программирования вроде Ruby, Python или JavaScript, вы не можете использовать эти два шага отдельно. Rust *компилируемый перед исполнением* язык, это означает, что вы можете собрать программу, дать её кому-то ещё, и ему не нужно устанавливать Rust. Если вы передадите кому-нибудь `.rb`, `.py` или `.js` файл, им понадобится Ruby/Python/JavaScript, чтобы скомпилировать и запустить вашу программу, но запустить его они смогу одной командой. Все это взаимоисключаемо в дизайне языков программирования, и Rust сделал свой выбор.

Поздравляем! Вы официально написали программу на Rust. Это делает вас Rust-программистом! Добро пожаловать!

Дальше мы познакомимся с новым инструментом `Cargo`, который используется для написания настоящих программ в Rust. Использовать `rustc` удобно лишь для небольших программ, но по мере роста проекта, потребуется инструмент, помогающий управлять настройками проекта, а также облегчит обмен кода с другими людьми и проектами.

Hello, Cargo!

Cargo - это инструмент, который используют разработчики для управления своими Rust проектами. Работа над Cargo пока ещё не закончена. Сейчас он находится в состоянии pre-1.0. Тем не менее, он уже достаточно хорош для использования во многих Rust проектах, и поэтому предполагается, что проекты на Rust будут использовать Cargo с самого начала.

Cargo делает три вещи: собирает ваш код, скачивает нужные вашему коду зависимости и собирает их. Поначалу, вашей программе не понадобится никаких зависимостей, поэтому будем использовать только первую часть его функционала. Со временем нам понадобится добавить несколько зависимостей, и нам не составит труда сделать это, поскольку мы начали использовать Cargo.

Если вы использовали официальный установщик, то Cargo установился вместе с Rust'ом. Если же вы установили Rust каким-либо другим образом, вы можете посмотреть инструкции по установке Cargo.

Переходим на Cargo

Давайте начнём использовать Cargo для сборки кода нашей программы "Hello World".

Чтобы Cargo-фицировать ваш проект, вы должны сделать две вещи: создать конфигурационный файл `Cargo.toml` и поместить файл с исходным кодом в правильное место. Давайте сделаем это:

```
$ mkdir src
$ mv main.rs src/main.rs
```

Отметим, что поскольку мы создаём исполняемый файл, то мы использовали `main.rs`. Если же вместо этого мы хотим сделать библиотеку, то мы должны использовать `lib.rs`. Специальное расположение файла для точки входа может быть задано с помощью ключа `[[lib]]` или `[[bin]]` в файле TOML, который описывается ниже.

Cargo ожидает что ваши файлы с исходным кодом находятся в директории `src`. Это оставляет верхний уровень для других вещей вроде README, файлов с текстом лицензии и других не относящихся к вашему коду. Cargo помогает нам сохранять наши проекты красивыми и аккуратными. Всему своё место и всё на своём месте.

Дальше, создадим конфигурационный файл для Cargo:

```
$ editor Cargo.toml
```

Убедитесь, что имя правильное: вам нужна заглавная `C`!

Вставьте эту конфигурацию в свой `Cargo.toml`:

```
[package]

name = "hello_world"
version = "0.0.1"
authors = [ "Ваше имя <you@example.ru>" ]
```

Этот файл в формате TOML. Позволим ему самому рассказать о себе:

> TOML стремится быть минималистичным форматом для конфигурационных файлов, который легко читается благодаря понятной семантике. TOML спроектирован для однозначного отображения в хэш-таблицу. TOML должен легко преобразовываться в структуры данных широкого спектра языков программирования.

TOML очень похож на INI, но с некоторыми дополнительными возможностями.

Итак, мы с этим закончили и готовы к сборке! Попробуйте собрать:

```
$ cargo build
   Compiling hello_world v0.0.1 (file:///home/yourname/projects/hello_world)
$ ./target/debug/hello_world
Hello, world!
```

Та-да! Мы собрали наш проект вызвав `cargo build` и запустили его с помощью `./target/debug/hello_world`. Мы можем сделать это в один шаг используя `cargo run`:

```
$ cargo run
     Running `target/debug/hello_world`
Hello, world!
```

Заметьте, что сейчас мы не пересобирали наш проект. Cargo понял, что мы не изменили файл с исходным кодом и только лишь запустил исполняемый файл. Если бы мы изменили файл, мы бы увидели оба шага:

```
$ cargo run
   Compiling hello_world v0.0.1 (file:///home/yourname/projects/hello_world)
     Running `target/debug/hello_world`
Hello, world!
```

На первый взгляд это кажется сложнее, по сравнению с более простым использованием `rustc`, но подумаем о будущем: если бы в нашем проекте было больше одного файла, мы бы должны были вызвать `rustc` для каждого и передать ему кучу параметров, что бы собрать их все вместе. С Cargo, когда наш проект вырастет, нам понадобится вызвать только команду `cargo build` и она всё сделает за нас.

Когда вы закончите работать над проектом, и он окончательно будет готов к релизу, то можете использовать команду `cargo build --release` для компиляции ваших контейнеров (crates) с оптимизацией.

Так же вы должны были заметить, что Cargo создал новый файл: `Cargo.lock`.

```
[root]
name = "hello_world"
version = "0.0.1"
```

Этот файл используется Cargo для отслеживания зависимостей в вашем приложении. Прямо сейчас у нас нет ни одной, поэтому этот файл немного пустоват. Вам не нужно редактировать этот файл самостоятельно, Cargo сам с ним разберётся.

Так! Мы успешно собрали `hello_world` с помощью Cargo. Несмотря на то, что наша программа проста, мы использовали большую часть реальных инструментов, которые вы будете использовать в своём дальнейшем пути Rust программиста. Вы можете использовать их во всех Rust проектах:

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

Новый проект

Вам не нужно повторять вышеприведённые шаги каждый раз, когда вы хотите создать новый проект! Cargo может создать директорию проекта, в котором вы сразу сможете приступить к разработке.

Чтобы создать новый проект с помощью Cargo, нужно ввести команду `cargo new`:

```
$ cargo new hello_world --bin
```

Мы указываем аргумент `--bin`, т.к. хотим создать исполняемую программу. Если мы не укажем этот аргумент, то Cargo создаст проект для библиотеки.

Давайте теперь посмотрим на то, что Cargo создал нам:

```
$ cd hello_world
$ tree .
.
├── Cargo.toml
└── src
    └── main.rs

1 директория, 2 файла
```

Если у вас нет команды `tree`, то скорее всего эта программа не установлена в вашей системе. Попробуйте установить её через менеджер пакетов вашего дистрибутива. Это не обязательно, но данная утилита очень полезна.

Все файлы и директории уже на месте. Теперь можем начинать. Для начала проверим файл `Cargo.toml`:

```
[package]

name = "hello_world"
version = "0.0.1"
authors = ["Ваше Имя <you@example.ru>"]
```

Cargo наполнил этот файл значениями по умолчанию на основании переданных аргументов и глобальной конфигурации `git`. Обратите внимание, что Cargo уже в директории `hello_world` создал репозиторий для `git`.

Также заглянем в `src/main.rs`:

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

Cargo создал "Hello World!" для нас и вы уже можете приступить к программированию! У Cargo есть собственное руководство в котором про него рассказано более полно.

Теперь давайте отложим инструментарий и узнаем больше о самом языке. Это основы, которые вы будете часто использовать на протяжении всего вашего взаимодействия с Rust.

У вас есть два пути: погрузиться в изучение реального проекта, раздел ‘Изучение Rust’, или начать с самого низа и постепенно продвигаться наверх, раздел ‘Синтаксис и семантика’. Программисты, имеющие опыт работы с системными языками, вероятно, предпочтут ‘Изучение Rust‘, в то время как программисты, имеющие опыт работы с динамическими языками, вполне возможно, пойдут по второму пути. Разные люди учатся по-разному! Выберите то, что подходит именно вам.

Изучение Rust

Добро пожаловать! Этот раздел книги содержит несколько глав, которые научат вас создавать проекты на Rust. Вы также получите поверхностное представление о языке. Мы не будем углубляться в детали.

Для получения полного и объемного понимания языка читайте раздел "Синтаксис и семантика".

Угадайка

В качестве нашего первого проекта, мы решим классическую для начинающих программистов задачу: игра-угадайка. Немного о том, как игра должна работать: наша программа генерирует случайное целое число из промежутка от 1 до 100. Затем она просит ввести число, которое она "загадала". Для каждого введённого нами числа, она говорит, больше ли оно, чем "загаданное", или меньше. Игра заканчивается когда мы отгадываем число. Звучит не плохо, не так ли?

Создание нового проекта

Давайте создадим новый проект. Перейдите в вашу директорию с проектами. Помните, как мы создавали структуру директорий и `Cargo.toml` для `hello_world`? Cargo может сделать это за нас. Давайте воспользуемся этим:

```
$ cd ~/projects
$ cargo new guessing_game --bin
$ cd guessing_game
```

Мы сказали Cargo, что хотим создать новый проект с именем `guessing_game`.При помощи флага `--bin`, мы указали что хотим создать исполняемый файл, а не библиотеку.

Давайте посмотрим сгенерированный `Cargo.toml`:

```
[package]

name = "guessing_game"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
```

Cargo взял эту информацию из вашего рабочего окружения. Если информация не корректна, исправьте её.

Наконец, Cargo создал программу `Hello, world!`. Посмотрите файл `src/main.rs`:

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

Давайте попробуем скомпилировать созданный Cargo проект:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
```

Замечательно! Снова откройте `src/main.rs`. Мы будем писать весь наш код в этом файле.

Прежде, чем мы начнём работу, давайте рассмотрим ещё одну команду Cargo: `run`. `cargo run` похожа на `cargo build`, но после завершения компиляции, она запускает получившийся исполняемый файл:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/debug/guessing_game`
Hello, world!
```

Великолепно! Команда `run` помогает, когда надо быстро пересобирать проект. Наша игра как раз и есть такой проект: нам надо быстро тестировать каждое изменение, прежде чем мы приступим к следующей части программы.

Обработка предположения

Давайте начнём! Первая вещь, которую мы должны сделать для нашей игры - это позволить игроку вводить предположения. Поместите следующий код в ваш `src/main.rs`:

```
use std::io;

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

    println!("Пожалуйста, введите предположение.");

    let mut guess = String::new();

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

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

Здесь много чего! Давайте разберём этот участок по частям.

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

Нам надо получить то, что ввёл пользователь, а затем вывести результат на экран. Значит нам понадобится библиотека `io` из стандартной библиотеки. Изначально, во вступлении (prelude), Rust импортирует в нашу программу лишь самые необходимые вещи. Если чего-то нет по вступлении, мы должны указать при помощи `use`, что хотим это использовать.

```
fn main() {
```

Как вы уже видели до этого, функция `main()` - это точка входа в нашу программу. `fn` объявляет новую функцию. Пустые круглые скобки `()` показывают, что она не принимает аргументов. Открывающая фигурная скобка `{` начинает тело нашей функции. Из-за того, что мы не указали тип возвращаемого значения, предполагается, что будет возвращаться `()` - пустой кортеж.

```
    println!("Угадайте число!");

    println!("Пожалуйста, введите предположение.");
```

Мы уже изучили, что `println!()` - это макрос, который выводит строки на экран.

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

Теперь интереснее! Как же много всего происходит в этой строке! Первая вещь, на которую следует обратить внимание - выражение let, которое используется для `создания связи`. Оно выглядит так:

```
let foo = bar;
```

Это создаёт новую связь с именем `foo` и привязывает ей значение `bar`. Во многих языках это называется `переменная`, но в Rust связывание переменных имеет несколько трюков в рукаве.

Например, по умолчанию, связи неизменяемы. По этой причине наш пример использует `mut`: этот модификатор разрешает менять связь. С левой стороны у `let` может быть не просто имя связи, а образец. Мы будем использовать их дальше. Их достаточно просто использовать:

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

Ах да, `//` начинает комментарий, который заканчивается в конце строки. Rust игнорирует всё, что находится в комментариях.

Теперь мы знаем, что `let mut guess` объявляет изменяемую связь с именем `guess`, а по другую сторону от `=` находится то, что будет привязано: `String::new()`.

`String` - это строковый тип, предоставляемый нам стандартной библиотекой. `String` - это текст в кодировке UTF-8 переменной длины.

Синтаксис `::new()` использует `::`, так как это привязанная к определённому типу функция. То есть, она привязана к самому типу `String`, а не к определённой переменной типа `String`. Некоторые языки называют это "статическим методом".

Имя этой функции - `new()`, так как она создаёт новый, пустой `String`. Вы можете найти эту функцию у многих типов, потому что это общее имя для создания нового значения определённого типа.

Давайте посмотрим дальше:

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

Это уже побольше! Давайте это всё разберём. В первой строке есть две части. Это первая:

```
io::stdin()
```

Помните, как мы импортировали (`use`) `std::io` в самом начале нашей программы? Сейчас мы вызвали ассоциированную с ним функцию. Если бы мы не сделали `use std::io`, нам бы пришлось здесь написать `std::io::stdin()`.

Эта функция возвращает обработчик стандартного ввода нашего терминала. Более подробно об это можно почитать в std::io::Stdin.

Следующая часть использует этот обработчик для получения всего, что введёт пользователь:

```
.read_line(&mut guess)
```

Здесь мы вызвали метод `read_line()` обработчика. Методы похожи на привязанные функции, но доступны только у определённого экземпляра типа, а не самого типа. Мы указали один аргумент функции `read_line()`: `&mut guess`.

Помните, как мы выше привязали `guess`? Мы сказали, что она изменяема. Однако, `read_line` не получает в качестве аргумента `String`: она получает `&mut String`. В Rust есть такая особенность, называемая "ссылки", которая позволяет нам иметь несколько ссылок на одни и так же данные, что позволяет избежать излишнего их копирования. Ссылки - достаточно сложная особенность, и одним из основных подкупающих достоинств Rust является то, как он решает вопрос безопасности и простоты их использования. Пока что мы не должны знать об этих деталях, чтобы завершить нашу программу. Сейчас, всё, что нам нужно - это знать что ссылки, как и связывание при помощи `let`, неизменяемо по умолчанию. Следовательно, мы должны написать `&mut guess`, а не `&guess`.

Почему `read_line()` получает изменяемую ссылку на строку? Его работа - это взять то, что пользователь написал в стандартный ввод, и положить это в строку. Итак, функция получает строку в качестве аргумента, и для того, чтобы добавить в эту строку что-то, она должна быть изменяемой.

Но мы пока что ещё не закончили с этой строкой кода. Пока это одна строка текста, это только первая часть одной логической строки кода:

```
        .ok()
        .expect("Не удалось прочитать строку");
```

Когда мы вызываем метод, используя синтаксис `.foo()`, мы можем перенести вызов в новую строку и сделать для него отступ. Это помогает работать с длинными строками. Мы могли бы сделать и так:

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

Но это достаточно трудно читать. Поэтому мы разделили строку: по строке на каждый вызов метода. Мы уже поговорили о `read_line()`, но ещё ничего не сказали про `ok()` и `expect()`. Мы узнали, что `read_line()` передаёт всё, что пользователь ввёл в `&mut String`, которую мы ему передали. Но этот метод так же и возвращает значение: в данном случае - `io::Result`. В стандартной библиотеке Rust есть несколько типов с именем `Result`: общая версия `Result` и несколько отдельных версий в подбиблиотеках, например `io::Result`.

Целью типов `Result` является преобразование информации об ошибках, полученных от обработчика. У значений типа `Result`, как и любого другого типа, есть определённые для него методы. В данном случае, у `io::Result` имеется метод `ok()`, который говорит, что "мы хотим получить это значение, если всё прошло хорошо. Если это не так, выбрось сообщение об ошибке". Но зачем выбрасывать? Для небольших программ, мы можем захотеть только вывести сообщение об ошибке и прекратить выполнение программы. Метод `ok()` возвращает значение, у которого объявлен другой метод: `expect()`. Метод `expect()` берёт значение, для которого он вызван, и если оно не удачное, выполняет `panic!` со строкой, заданной методу в качестве аргумента. `panic!` остановит нашу программу и выведет сообщение об ошибке.

Если мы выйдем за пределы этих двух методов, наша программа скомпилируется, но мы получим следующее предупреждение:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
src/main.rs:10:5: 10:39 warning: unused result which must be used,
#[warn(unused_must_use)] on by default
src/main.rs:10     io::stdin().read_line(&mut guess);
                   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Rust предупреждает, что мы не используем значение `Result`. Это предупреждение пришло из специальной аннотации, которая указана в `io::Result`. Rust пытается сказать нам, что мы не обрабатываем ошибки, которые могут возникнуть. Наиболее правильным решением предотвращения ошибки будет её обработка. К счастью, если мы только хотим обрушить приложение, если есть проблема, мы можем использовать эти два небольших метода. Если мы можем восстановить что-либо из ошибки, мы должны сделать что-либо другое, но мы сохраним это для будущего проекта.

Там всего одна строка из первого примера:

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

Здесь выводится на экран строка, которая была получена с нашего ввода. `{}` - это указатель места заполнения. В качестве второго аргумента макроса `println!` мы указали `guess`. Если нам надо вывести несколько привязок, в самом простом случае, мы должны поставить несколько указателей, по одному на каждую привязку:

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

println!("x и y: {} и {}", x, y);
```

Просто.

Мы можем запустить то, что у нас есть при помощи `cargo run`:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/debug/guessing_game`
Угадайте число!
Пожалуйста, введите предположение.
6
Вы загадали: 6
```

Всё правильно! Наша первая часть завершена: мы можем получать данные с клавиатуры и потом печатать их на экран.

Генерация секретного числа

Далее, нам надо сгенерировать секретное число. В стандартной библиотеке Rust нет ничего, что могло бы нам предоставить функционал для генерации случайных чисел. Однако, разработчики Rust для этого предоставили контейнер `rand`. "Контейнер" - это пакет с кодом Rust. Наш проект - "бинарный контейнер", из которого в итоге получится исполняемый файл. `rand` - "библиотечный контейнер", который содержит код, предназначенный для использования с другими программами.

Прежде, чем мы начнём писать код с использованием `rand`, мы должны модифицировать наш `Cargo.toml`. Откроем его и добавим в конец следующие строчки:

```
[dependencies]

rand="0.3.0"
```

Секция `[dependencies]` похожа на секцию `[package]`: всё, что расположено после объявления секции и до начала следующей, является частью этой секции. Cargo использует секцию с зависимостями чтобы знать о том, какие сторонние контейнеры потребуются, а так же какие их версии необходимы. В данном случае, мы используем версию `0.3.0`. Cargo понимает семантическое версионирование, которое является стандартом нумерации версий. Если мы хотим использовать последнюю версию контейнера, мы можем использовать `*`. Так же мы можем указать необходимый промежуток версий. В документации Cargo есть больше информации.

Теперь, без каких-либо изменений в нашем коде, давайте соберём наш проект:

```
$ cargo build
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading rand v0.3.8
 Downloading libc v0.1.6
   Compiling libc v0.1.6
   Compiling rand v0.3.8
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
```

(Конечно же, вы можете видеть другие версии.)

Lots of new output! Now that we have an external dependency, Cargo fetches the latest versions of everything from the registry, which is a copy of data from Crates.io. Crates.io is where people in the Rust ecosystem post their open source Rust projects for others to use.

After updating the registry, Cargo checks our `[dependencies]` and downloads any we don't have yet. In this case, while we only said we wanted to depend on `rand`, we've also grabbed a copy of `libc`. This is because `rand` depends on `libc` to work. After downloading them, it compiles them, and then compiles our project.

If we run `cargo build` again, we'll get different output:

```
$ cargo build
```

That's right, no output! Cargo knows that our project has been built, and that all of its dependencies are built, and so there's no reason to do all that stuff. With nothing to do, it simply exits. If we open up `src/main.rs` again, make a trivial change, and then save it again, we'll just see one line:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
```

So, we told Cargo we wanted any `0.3.x` version of `rand`, and so it fetched the latest version at the time this was written, `v0.3.8`. But what happens when next week, version `v0.3.9` comes out, with an important bugfix? While getting bugfixes is important, what if `0.3.9` contains a regression that breaks our code?

The answer to this problem is the `Cargo.lock` file you'll now find in your project directory. When you build your project for the first time, Cargo figures out all of the versions that fit your criteria, and then writes them to the `Cargo.lock` file. When you build your project in the future, Cargo will see that the `Cargo.lock` file exists, and then use that specific version rather than do all the work of figuring out versions again. This lets you have a repeatable build automatically. In other words, we'll stay at `0.3.8` until we explicitly upgrade, and so will anyone who we share our code with, thanks to the lock file.

What about when we *do* want to use `v0.3.9`? Cargo has another command, `update`, which says 'ignore the lock, figure out all the latest versions that fit what we've specified. If that works, write those versions out to the lock file'. But, by default, Cargo will only look for versions larger than `0.3.0` and smaller than `0.4.0`. If we want to move to `0.4.x`, we'd have to update the `Cargo.toml` directly. When we do, the next time we `cargo build`, Cargo will update the index and re-evaluate our `rand` requirements.

There's a lot more to say about Cargo and its ecosystem, but for now, that's all we need to know. Cargo makes it really easy to re-use libraries, and so Rustaceans tend to write smaller projects which are assembled out of a number of sub-packages.

Let's get on to actually *using* `rand`. Here's our next step:

```
extern crate rand;

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

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

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

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

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

    let mut guess = String::new();

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

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

The first thing we've done is change the first line. It now says `extern crate rand`. Because we declared `rand` in our `[dependencies]`, we can use `extern crate` to let Rust know we'll be making use of it. This also does the equivalent of a `use rand;` as well, so we can make use of anything in the `rand` crate by prefixing it with `rand::`.

Next, we added another `use` line: `use rand::Rng`. We're going to use a method in a moment, and it requires that `Rng` be in scope to work. The basic idea is this: methods are defined on something called 'traits', and for the method to work, it needs the trait to be in scope. For more about the details, read the traits section.

There are two other lines we added, in the middle:

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

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

We use the `rand::thread_rng()` function to get a copy of the random number generator, which is local to the particular thread of execution we're in. Because we `use rand::Rng`'d above, it has a `gen_range()` method available. This method takes two arguments, and generates a number between them. It's inclusive on the lower bound, but exclusive on the upper bound, so we need `1` and `101` to get a number between one and a hundred.

The second line just prints out the secret number. This is useful while we're developing our program, so we can easily test it out. But we'll be deleting it for the final version. It's not much of a game if it prints out the answer when you start it up!

Try running our new program a few times:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 7
Please input your guess.
4
You guessed: 4
$ cargo run
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
```

Great! Next up: let's compare our guess to the secret guess.

Comparing guesses

Now that we've got user input, let's compare our guess to the random guess. Here's our next step, though it doesn't quite work yet:

```
extern crate rand;

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

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

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

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

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

    let mut guess = String::new();

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

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

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

A few new bits here. The first is another `use`. We bring a type called `std::cmp::Ordering` into scope. Then, five new lines at the bottom that use it:

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

The `cmp()` method can be called on anything that can be compared, and it takes a reference to the thing you want to compare it to. It returns the `Ordering` type we `used` earlier. We use a `match` statement to determine exactly what kind of `Ordering` it is. `Ordering` is an `enum`, short for 'enumeration', which looks like this:

```
enum Foo {
    Bar,
    Baz,
}
```

With this definition, anything of type `Foo` can be either a `Foo::Bar` or a `Foo::Baz`. We use the `::` to indicate the namespace for a particular `enum` variant.

The `Ordering` enum has three possible variants: `Less`, `Equal`, and `Greater`. The `match` statement takes a value of a type, and lets you create an 'arm' for each possible value. Since we have three types of `Ordering`, we have three arms:

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

If it's `Less`, we print `Too small!`, if it's `Greater`, `Too big!`, and if `Equal`, `You win!`. `match` is really useful, and is used often in Rust.

I did mention that this won't quite work yet, though. Let's try it:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
src/main.rs:28:21: 28:35 error: mismatched types:
 expected `&collections::string::String`,
    found `&_`
(expected struct `collections::string::String`,
    found integral variable) [E0308]
src/main.rs:28     match guess.cmp(&secret_number) {
                                   ^~~~~~~~~~~~~~
error: aborting due to previous error
Could not compile `guessing_game`.
```

Whew! This is a big error. The core of it is that we have 'mismatched types'. Rust has a strong, static type system. However, it also has type inference. When we wrote `let guess = String::new()`, Rust was able to infer that `guess` should be a `String`, and so it doesn't make us write out the type. And with our `secret_number`, there are a number of types which can have a

value between one and a hundred: `i32`, a thirty-two-bit number, or `u32`, an unsigned thirty-two-bit number, or `i64`, a sixty-four-bit number. Or others. So far, that hasn't mattered, and so Rust defaults to an `i32`. However, here, Rust doesn't know how to compare the `guess` and the `secret_number`. They need to be the same type. Ultimately, we want to convert the `String` we read as input into a real number type, for comparison. We can do that with three more lines. Here's our new program:

```
extern crate rand;

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

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

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

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

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

    let mut guess = String::new();

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

    let guess: u32 = guess.trim().parse()
        .ok()
        .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!"),
    }
}
```

The new three lines:

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

Wait a minute, I thought we already had a `guess`? We do, but Rust allows us to 'shadow' the previous `guess` with a new one. This is often used in this exact situation, where `guess` starts as a `String`, but we want to convert it to an `u32`. Shadowing lets us re-use the `guess` name, rather than forcing us to come up with two unique names like `guess_str` and `guess`, or something else.

We bind `guess` to an expression that looks like something we wrote earlier:

```
guess.trim().parse()
```

Followed by an `ok().expect()` invocation. Here, `guess` refers to the old `guess`, the one that was a `String` with our input in it. The `trim()` method on `String`s will eliminate any white space at the beginning and end of our string. This is important, as we had to press the 'return' key to satisfy `read_line()`. This means that if we type `5` and hit return, `guess` looks like this: `5\n`. The `\n` represents 'newline', the enter key. `trim()` gets rid of this, leaving our string with just the `5`. The `parse()` method on strings parses a string into some kind of number. Since it can parse a variety of numbers, we need to give Rust a hint as to the exact type of number we want. Hence, `let guess: u32`. The colon (`:`) after `guess` tells Rust we're going to annotate its type. `u32` is an unsigned, thirty-two bit integer. Rust has a number of built-in number types, but we've chosen `u32`. It's a good default choice for a small positive number.

Just like `read_line()`, our call to `parse()` could cause an error. What if our string contained `A%`? There'd be no way to convert that to a number. As such, we'll do the same thing we did with `read_line()`: use the `ok()` and `expect()` methods to crash if there's an error.

Let's try our program out!

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
  76
You guessed: 76
Too big!
```

Nice! You can see I even added spaces before my guess, and it still figured out that I guessed 76. Run the program a few times, and verify that guessing the number works, as well as guessing a number too small.

Now we've got most of the game working, but we can only make one guess. Let's change that by adding loops!

Looping

The `loop` keyword gives us an infinite loop. Let's add that in:

```
extern crate rand;

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

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

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

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

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

        let mut guess = String::new();

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

        let guess: u32 = guess.trim().parse()
            .ok()
            .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!"),
        }
    }
}
```

And try it out. But wait, didn't we just add an infinite loop? Yup. Remember our discussion about `parse()`? If we give a non-number answer, we'll `return` and quit. Observe:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/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!'
```

Ha! `quit` actually quits. As does any other non-number input. Well, this is suboptimal to say the least. First, let’s actually quit when you win the game:

```
extern crate rand;

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

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

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

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

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

        let mut guess = String::new();

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

        let guess: u32 = guess.trim().parse()
            .ok()
            .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!");
                break;
            }
        }
    }
}
```

By adding the `break` line after the `You win!`, we'll exit the loop when we win. Exiting the loop also means exiting the program, since it's the last thing in `main()`. We have just one more tweak to make: when someone inputs a non-number, we don't want to quit, we just want to ignore it. We can do that like this:

```
extern crate rand;

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

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

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

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

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

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .ok()
            .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;
            }
        }
    }
}
```

These are the lines that changed:

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

This is how you generally move from ‘crash on error’ to ‘actually handle the error’, by switching from `ok().expect()` to a `match` statement. The `Result` returned by `parse()` is an enum just like `Ordering`, but in this case, each variant has some data associated with it: `Ok` is a success, and `Err` is a failure. Each contains more information: the successful parsed integer, or an error type. In this case, we `match` on `Ok(num)`, which sets the inner value of the `Ok` to the name

`num`, and then we just return it on the right-hand side. In the `Err` case, we don't care what kind of error it is, so we just use `_` instead of a name. This ignores the error, and `continue` causes us to go to the next iteration of the `loop`.

Now we should be good! Let's try:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/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!
```

Awesome! With one tiny last tweak, we have finished the guessing game. Can you think of what it is? That's right, we don't want to print out the secret number. It was good for testing, but it kind of ruins the game. Here's our final source:

```
extern crate rand;

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

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

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

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

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .ok()
            .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;
            }
        }
    }
}
```

Complete!

At this point, you have successfully built the Guessing Game! Congratulations!

This first project showed you a lot: `let`, `match`, methods, associated functions, using external crates, and more. Our next project will show off even more.

Обедающие философы

Для нашего второго проекта мы выбрали классическую задачу с параллелизмом. Она называется "Обедающие философы". Задача была сформулирована в 1965 году Эдсгером Дейкстрой, но мы будем использовать версию задачи, адаптированную в 1985 году Ричардом Хоаром.

> В древние времена, богатые филантропы пригласили погостить пятерых выдающихся философов. Им выделили каждому по комнате, в которой они могли заниматься своей профессиональной деятельностью - мышлением. Также была общая столовая, где стоял большой круглый стол, а вокруг него пять стульев. Каждый стул имел табличку с именем философа, который должен был сидеть на нем. Слева от каждого философа лежала золотая вилка и в центре стола стояла большая миска со спагетти, которая постоянно пополняется. Как подобает философам, они большую часть своего времени проводят в раздумьях. Но однажды они почувствовали голод и отправились в столовую. Каждый сел на свой стул, взял по вилке и воткнул её в миску со спагетти. Но из-за запутанной сущности спагетти потребовалась вторая вилка, которая бы отправляла спагетти в рот. Для этого требовалась бы вилка справа от каждого философа. Философы положили свои вилки и встали из-за стола, продолжая думать. Вилка может быть использована только одним философом одновременно. Если другой философ захочет её взять, то ему придется ждать когда она освободится.

Эта классическая задача показывает различные элементы параллелизма. Сложность реализации задачи состоит в том, что простая реализация может зайти в безвыходное состояние. Давайте рассмотрим простой пример решения этой проблемы:

1. Философ берет вилку в свою левую руку.
2. Затем берет вилку в свою правую руку.
3. Он ест.
4. Кладет вилки на место.

Теперь представим это как последовательность действий философов:

1. Философ 1 начинает выполнять алгоритм, берет вилку в левую руку.
2. Философ 2 начинает выполнять алгоритм, берет вилку в левую руку.
3. Философ 3 начинает выполнять алгоритм, берет вилку в левую руку.
4. Философ 4 начинает выполнять алгоритм, берет вилку в левую руку.
5. Философ 5 начинает выполнять алгоритм, берет вилку в левую руку.
6. ...? Все вилки заняты и никто не может начать есть! Безвыходное состояние.

Есть различные пути решения этой задачи. Мы в этом руководстве покажем свое решение. Сначала давайте начнем с моделирования задачи. Начнем с философов:

```
struct Philosopher {
    name: String,
}

impl Philosopher {
    fn new(name: &str) -> Philosopher {
        Philosopher {
            name: name.to_string(),
        }
    }
}

fn main() {
    let p1 = Philosopher::new("Джудит Батлер");
    let p2 = Philosopher::new("Рая Дунаевская");
    let p3 = Philosopher::new("Зарубина Наталья");
    let p4 = Philosopher::new("Эмма Гольдман");
    let p5 = Philosopher::new("Шмидт Анна");
}
```

Здесь мы создаем структуру представляющую философа. На данный момент, нам нужно всего лишь имя. Мы выбрали тип `String` для хранения имени вместо `&str`. Обычно проще работать с типом, владеющим данными, чем с типом, использующим ссылки.

Посмотрим на это:

```
impl Philosopher {
    fn new(name: &str) -> Philosopher {
        Philosopher {
            name: name.to_string(),
        }
    }
}
```

Этот блок `impl` позволяет объявлять что-нибудь для структуры `Philosopher`. В нашем случае мы объявляем "статическую функцию" `new`. Первая строка этой функции выглядит так:

```
fn new(name: &str) -> Philosopher {
```

Она принимает один аргумент `name` типа `&str`. Это ссылка на другую строку. Возвращает новый экземпляр нашей структуры `Philosopher`.

```
Philosopher {
    name: name.to_string(),
}
```

Это создаст новый экземпляр `Philosopher` и присвоит полю `name` значение переданного аргумента `name`. Но не сам аргумент, а результат его вызова `.to_string()`. Это создаст копию строки нашего указателя `&str` и даст новый экземпляр `String`, который будет присвоен полю `name` структуры `Philosopher`.

Почему бы сразу не передавать строку типа `String` напрямую? Так легче ее вызывать. Если мы принимали бы тип `String`, а тот кто вызывает функцию имеет ссылку на строку `&str`, то должен был бы приводить ее к типу `String` перед каждым вызовом. Это уменьшит гибкость кода и придется *каждый раз* делать копию строки. Для этой небольшой программы это не очень важно, т.к. мы знаем что будем использовать только короткие строки.

И последнее на что вы обратите внимание: мы просто объявляем структуру `Philosopher` и кажется, что ничего больше не делаем. Rust это expression ориентированный язык программирования, означающее, что каждое выражение возвращает значение. Это применяется для функций у которых автоматически возвращается последнее выражение. В нашем случае мы создаем структуру `Philosopher` в последнем выражении функции, которое возвращается функцией.

Имя функции `new()` не является особенным в Rust, но негласным соглашением принято так называть функции, которые возвращают новые экземпляры структур. Давайте посмотрим на функцию `main()`:

```
fn main() {
    let p1 = Philosopher::new("Джудит Батлер");
    let p2 = Philosopher::new("Рая Дунаевская");
    let p3 = Philosopher::new("Зарубина Наталья");
    let p4 = Philosopher::new("Эмма Гольдман");
    let p5 = Philosopher::new("Шмидт Анна");
}
```

Здесь, мы связываем пять переменных с пятью новыми философами. Здесь указаны имена некоторых известных философов, но вы можете указать любые другие. Если бы мы *не объявили* свою реализацию функции `new()`, то наш код выглядел бы так:

```
fn main() {
    let p1 = Philosopher { name: "Джудит Батлер".to_string() };
    let p2 = Philosopher { name: "Рая Дунаевская".to_string() };
    let p3 = Philosopher { name: "Зарубина Наталья".to_string() };
    let p4 = Philosopher { name: "Эмма Гольдман".to_string() };
    let p5 = Philosopher { name: "Шмидт Анна".to_string() };
}
```

Это было бы очень не очень изящно и трудно читаемо. Использование статической функции `new` имеет много плюсов, в нашем простом случае даст более элегантный код.

Сейчас у нас уже есть каркас программы и теперь можно заняться решением задачи с обедающими философами. Начнем с конца: сделаем так, чтобы философ сообщал нам когда он закончит есть. Для этого потребуется метод, сообщающий нам об окончании обеда, и цикл, запускающий этот метод для каждого философа.

```
struct Philosopher {
    name: String,
}

impl Philosopher {
    fn new(name: &str) -> Philosopher {
        Philosopher {
            name: name.to_string(),
        }
    }

    fn eat(&self) {
        println!("{} закончила обедать.", self.name);
    }
}

fn main() {
    let philosophers = vec![
        Philosopher::new("Джудит Батлер"),
        Philosopher::new("Рая Дунаевская"),
        Philosopher::new("Зарубина Наталья"),
        Philosopher::new("Эмма Гольдман"),
        Philosopher::new("Шмидт Анна"),
    ];

    for p in &philosophers {
        p.eat();
    }
}
```

Давайте сначала рассмотрим на `main()`. Вместо того чтобы каждого философа привязывать к отдельной переменной, мы создаем `Vec<T>` с ними. `Vec<T>` называют "вектор" и он является расширенной версией массива. Затем в цикле `for` мы перебираем вектор, получая ссылку на каждого философа за каждую итерацию.

В теле цикла мы вызываем `p.eat()`, который объявлен выше:

```
fn eat(&self) {
    println!("{} закончила есть.", self.name);
}
```

В Rust методы явно получают параметр self. Вот почему `eat()` является методом, а `new` статической функцией: `new()` не получает параметр `self`. Для нашей первой версии `eat()`, мы только выводим имя философа и сообщение о том, что он закончил есть. Запустив эту программу вы получите:

```
Джудит Батлер закончила есть.
Рая Дунаевская закончила есть.
Зарубина Наталья закончила есть.
Эмма Гольдман закончила есть.
Шмидт Анна закончила есть.
```

Это было не сложно! Осталось чуть-чуть и приступим к самой задаче.

Дальше нам надо сделать так, чтобы философы не только заканчивали есть, но также и

начинали. Это новая версия программы:

```
use std::thread;

struct Philosopher {
    name: String,
}

impl Philosopher {
    fn new(name: &str) -> Philosopher {
        Philosopher {
            name: name.to_string(),
        }
    }

    fn eat(&self) {
        println!("{} начала есть.", self.name);

        thread::sleep_ms(1000);

        println!("{} закончила есть.", self.name);
    }
}

fn main() {
    let philosophers = vec![
        Philosopher::new("Джудит Батлер"),
        Philosopher::new("Рая Дунаевская"),
        Philosopher::new("Зарубина Наталья"),
        Philosopher::new("Эмма Гольдман"),
        Philosopher::new("Шмидт Анна"),
    ];

    for p in &philosophers {
        p.eat();
    }
}
```

Появилось немного изменений. Давайте посмотрим что изменилось:

```
use std::thread;
```

Конструкция `use` предоставляет доступ к области видимости модуля `thread` из стандартной библиотеки, т.к. мы хотим использовать этот модуль далее в коде.

```
    fn eat(&self) {
        println!("{} начала есть.", self.name);

        thread::sleep_ms(1000);

        println!("{} закончила есть.", self.name);
    }
```

Здесь мы выводим на экран два сообщения и используем функцию `sleep_ms` между выводом. Эта функция останавливает рабочий поток на 1000 миллисекунд и мы её используем для симуляции процесса обеда философа.

Если вы запустите теперь программу, то увидите что теперь каждый философ по очереди начинает есть и затем заканчивает:

```
Джудит Батлер начала есть.
Джудит Батлер закончила есть.
Рая Дунаевская начала есть.
Рая Дунаевская закончила есть.
Зарубина Наталья начала есть.
Зарубина Наталья закончила есть.
Эмма Гольдман начала есть.
Эмма Гольдман закончила есть.
Шмидт Анна начала есть.
Шмидт Анна закончила есть.
```

Превосходно! Теперь у нас осталась проблема: наши философы едят по очереди, а не одновременно, т.е. мы пока не решили задачу параллелизма.

Для того, чтобы наши философы начали есть одновременно, нам надо внести немного изменений в код:

```
use std::thread;

struct Philosopher {
    name: String,
}

impl Philosopher {
    fn new(name: &str) -> Philosopher {
        Philosopher {
            name: name.to_string(),
        }
    }

    fn eat(&self) {
        println!("{} начала есть.", self.name);

        thread::sleep_ms(1000);

        println!("{} закончила есть.", self.name);
    }
}

fn main() {
    let philosophers = vec![
        Philosopher::new("Джудит Батлер"),
        Philosopher::new("Рая Дунаевская"),
        Philosopher::new("Зарубина Наталья"),
        Philosopher::new("Эмма Гольдман"),
        Philosopher::new("Шмидт Анна"),
    ];

    let handles: Vec<_> = philosophers.into_iter().map(|p| {
        thread::spawn(move || {
            p.eat();
        })
    }).collect();

    for h in handles {
        h.join().unwrap();
    }
}
```

Мы добавили еще один цикл в функцию `main()`. Теперь она выглядит так:

```
let handles: Vec<_> = philosophers.into_iter().map(|p| {
    thread::spawn(move || {
        p.eat();
    })
}).collect();
```

Тут добавились трудные к пониманию пять строк кода. Давайте разбираться.

```
let handles: Vec<_> =
```

Объявляем новое связывание с именем `handles`. Мы дали ему такое имя, т.к. собираемся создать несколько потоков и для контролирования их работы нужно получить дескрипторы каждого из них. Нам надо явно указать здесь тип, а для чего это надо мы скажем чуть позже. `_` это заполнитель типа. Мы говорим компилятору "`handles` это вектор тип которого ты можешь самостоятельно вычислить".

```
philosophers.into_iter().map(|p| {
```

Мы берем наш список философов и вызываем `into_iter()`. Это создаст итератор, который заберёт право владения (ownership) для каждого философа. Нам надо сделать это для передачи в поток. Мы берем этот итератор и вызываем `map`, который принимает замыкание как аргумент и вызывает это замыкание для каждого элемента итерации.

```
    thread::spawn(move || {
        p.eat();
    })
```

Вот здесь происходит сам параллелизм. Функция `thread::spawn` берет замыкание в качестве аргумента и исполняет это замыкание в новом потоке. Это замыкание нуждается дополнительном указании ключевого слова `move`, которое сообщает что это замыкание получает право владения (ownership) значениями которое оно захватывает. В данном случае переменную `p` функции `map`.

Внутри потока мы всего лишь вызываем функцию `eat()` у `p`.

```
}).collect();
```

И в завершении мы получаем результат этих вызовов `map` и собираем полученный результат в в коллекцию с помощью функции `collect()`. Для того чтобы Rust понял какой тип коллекции хотим получить, мы указали для `handle` тип принимаемого значения `Vec<T>`. Элементы коллекции будут возвращаемые значения типа `thread::spawn`, которые являются ручками их потоков. Вот так!

```
for h in handles {
    h.join().unwrap();
}
```

В конце функции `main()` мы в цикле перебираем каждую ручку и вызываем функцию `join()` для каждой ручки, которая блокирует дальнейшее исполнение основного потока, пока не завершится дочерний поток. Это позволяет нам быть уверенными, что потоки завершат работу до того как произойдет выход их программы.

Если вы запустите эту программу, то вы увидите, что философы едят не дожидаясь своей очереди! У нас многопоточность!

```
Рая Дунаевская начала есть.
Рая Дунаевская закончила есть.
Эмма Гольдман начала есть.
Эмма Гольдман закончила есть.
Шмидт Анна начала есть.
Джудит Батлер начала есть.
Джудит Батлер закончила есть.
Зарубина Наталья начала есть.
Зарубина Наталья закончила есть.
Шмидт Анна закончила есть.
```

Но где же вилки? Они пока ещё не смоделированы у нас.

Давайте же начнем. Сначала сделаем новую структуру:

```
use std::sync::Mutex;

struct Table {
    forks: Vec<Mutex<()>>,
}
```

Структура `Table` имеет вектор мьютексов (`Mutex`). Мьютекс позволяет управлять одновременно выполняющимися потоками, давая доступ к содержимому только одному потоку. Это свойство нужно для реализации наших вилок. Мы в коде используем пустой кортеж `()` внутри мьютекса, т.к. мы не собираемся использовать значение, а мьютекс использовать только для приостановки потока.

Давайте изменим программу используя структуру `Table`:

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

struct Philosopher {
    name: String,
    left: usize,
    right: usize,
}

impl Philosopher {
    fn new(name: &str, left: usize, right: usize) -> Philosopher {
        Philosopher {
            name: name.to_string(),
            left: left,
            right: right,
        }
    }

    fn eat(&self, table: &Table) {
        let _left = table.forks[self.left].lock().unwrap();
        let _right = table.forks[self.right].lock().unwrap();

        println!("{} начала есть.", self.name);

        thread::sleep_ms(1000);
```

```
        println!("{} закончила есть.", self.name);
    }
}

struct Table {
    forks: Vec<Mutex<()>>,
}

fn main() {
    let table = Arc::new(Table { forks: vec![
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
    ]});

    let philosophers = vec![
        Philosopher::new("Джудит Батлер", 0, 1),
        Philosopher::new("Рая Дунаевская", 1, 2),
        Philosopher::new("Зарубина Наталья", 2, 3),
        Philosopher::new("Эмма Гольдман", 3, 4),
        Philosopher::new("Шмидт Анна", 0, 4),
    ];

    let handles: Vec<_> = philosophers.into_iter().map(|p| {
        let table = table.clone();

        thread::spawn(move || {
            p.eat(&table);
        })
    }).collect();

    for h in handles {
        h.join().unwrap();
    }
}
```

Много изменений! Однако эта версия является корректно работающей версией. Приступим к рассмотрению:

```
use std::sync::{Mutex, Arc};
```

Нам далее понадобится структура `Arc<T>` из модуля стандартной библиотеки `std::sync`. Мы поговорим о ней чуть позже.

```
struct Philosopher {
    name: String,
    left: usize,
    right: usize,
}
```

Нам понадобилось добавить еще два поля в нашу структуру `Philosopher`. Каждый философ должен иметь две вилки: одну для левой руки, другую для правой руки. Мы используем тип `usize` для идентификации каждой вилки. Мы используем его при при

создании философа, передавая идентификаторы двух вилок. Эти два значения будут использоваться полем `forks` структуры `Table`.

```
fn new(name: &str, left: usize, right: usize) -> Philosopher {
    Philosopher {
        name: name.to_string(),
        left: left,
        right: right,
    }
}
```

Используем функцию `new()` для задания значений `left` и `right`.

```
fn eat(&self, table: &Table) {
    let _left = table.forks[self.left].lock().unwrap();
    let _right = table.forks[self.right].lock().unwrap();

    println!("{} начала есть.", self.name);

    thread::sleep_ms(1000);

    println!("{} закончила есть.", self.name);
}
```

Здесь появились две новые строки, а также добавили один аргумент `table`. У нас есть доступ к вилкам через структуру `Table`, передавая в нее идентификаторы вилок `self.left` и `self.right`. Она хранит в себе `Mutex` для каждой вилки и вызываем `lock()`, блокируя к ней доступ.

Вызов `lock()` может быть неудачным и если это случится, то нужно будет аварийно завершить работу программы. Это может произойти, если вдруг поток аварийно завершит работу, а мьютекс останется заблокирован. Такой мьютекс называется "отравленным". Но в нашем случае это не может произойти, то мы просто используем `unwarp()`.

В этих двух строках мы результат сохраняем в `_left` и `_right`. Зачем мы используем знаки подчеркивания в начале идентификаторов имен? Это для того чтобы сказать компилятору, что хотим получить значения, которые далее *не планируем использовать*. Таким образом Rust не будет выводить предупреждение о неиспользуемых идентификаторов имен.

Когда же освободится мьютекс? Это произойдет когда `_left` и `_right` выйдут из области видимости, т.е. по окончанию работы функции.

```
    let table = Arc::new(Table { forks: vec![
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
    ]});
```

Далее в `main()` мы создаем `Table` и оборачиваем его в `Arc<T>`. Это "атомарный счетчик ссылок" и нам он нужен для использования нашей структуры `Table` получения доступа в нескольких потоках. Когда мы его будем передавать в поток, то будет счетчик увеличиваться, а если поток завершит работу, то счетчик уменьшится.

```
let philosophers = vec![
    Philosopher::new("Джудит Батлер", 0, 1),
    Philosopher::new("Рая Дунаевская", 1, 2),
    Philosopher::new("Зарубина Наталья", 2, 3),
    Philosopher::new("Эмма Гольдман", 3, 4),
    Philosopher::new("Шмидт Анна", 0, 4),
];
```

Здесь мы добавили наши значения `left` и `right` при создании структуры `Philosopher`. Здесь есть *очень важная* деталь на которую следует обратить внимание. Посмотрите на последнюю строку создания `Philosopher`. У Анны Шмидт должны бы иметь значения `4` и `0` в качестве аргументов, но у нас имеют `0` и `4`. Это помешает нашей программе попасть в безвыходное состояние, если все одновременно возьмут вилки одновременно. Так что давайте представим, что один из философов у нас левша!

```
let handles: Vec<_> = philosophers.into_iter().map(|p| {
    let table = table.clone();

    thread::spawn(move || {
        p.eat(&table);
    })
}).collect();
```

Внутри нашего цикла `map()`/`collect()` мы вызываем `table.clone()`. Метод `clone()` структуры `Arc<T>` инкрементирует счетчик и когда покинет область видимости декрементирует. Вы можете заметить, что мы делаем новое связывание с `table` здесь, затеняя оригинальную `table`. Это позволяет нам не вводить новый уникальный идентификатор.

Теперь наша программа работает! Только два философа могут обедать одновременно и после запуска программы вы можете получить такой результат.

```
Рая Дунаевская начала есть.
Эмма Гольдман начала есть.
Эмма Гольдман закончила есть.
Рая Дунаевская закончила есть.
Джудит Батлер начала есть.
Зарубина Наталья начала есть.
Джудит Батлер закончила есть.
Шмидт Анна начала есть.
Зарубина Наталья закончила есть.
Шмидт Анна закончила есть.
```

Поздравляем! Вы реализовали классическую задачу параллелизма на языке Rust.

Вызов кода на Rust из других языков

Для нашего третьего проекта мы собираемся выбрать что-то, что подчеркнёт одну из самых сильных сторон в Rust: фактическое отсутствие среды исполнения.

По мере роста организации, программисты все больше полагаются на множество языков программирования. У каждого языка программирования есть свои сильные и слабые стороны, а знание нескольких языков позволяет позволяет использовать определенный язык там, где проявляется его сильные стороны, и использовать другой язык там, где он не очень хорош.

Существует несколько областей, где многие языки программирования слабы в плане производительности выполнения программ. Часто компромисс заключается в том, чтобы использовать более медленный язык, который взамен способствует повышению производительности программиста. Чтобы решить эту проблему, часть кода системы можно написать на C, а затем вызвать этот код, написанный на C, как если бы он был написан на языке высокого уровня. Это называется "интерфейс внешних функций" (foreign function interface), часто сокращается до "FFI".

Rust включает поддержку FFI в обоих направлениях: он легко может вызвать C код, и он так же легко, как и C код, может быть вызван *извне*. Rust сочетает в себе отсутствие сборщика мусора и низкие требования к среде исполнения, что делает Rust отличным кандидатом на роль вызываемого из других языков, когда нужны некоторые дополнительные возможности.

В этой книге есть целая глава, посвящённая FFI и его специфике, а в этой главе мы рассмотрим именно конкретный частный случай FFI, с тремя примерами, на Ruby, Python и JavaScript.

Проблема

Есть много различных проектов, которые мы могли бы выбрать, но мы хотим подобрать такой пример, который продемонстрирует явное преимущество Rust над многими другими языками: сложные вычисления и многопоточность.

Во многих языках числа размещаются в куче, а не в стеке. Это обеспечивает целостность поведения языка при работе с числами и с другими объектами. Особенно в языках, которые сосредотачиваются на объектно-ориентированном программировании и использовании сборщика мусора, по умолчанию память выделяется из кучи. Иногда, при оптимизации, для конкретных чисел память может выделяться в стеке, но вместо того, чтобы полагаться на работу оптимизации, мы можем захотеть убедиться в том, что мы используем примитивные типы чисел, а не какой-либо тип объекта.

Во-вторых, многие языки имеют "глобальную блокировку интерпретатора" ("global interpreter lock"), которая ограничивает параллелизм во многих ситуациях. Это делается во имя безопасности, что оказывает положительный эффект, но это также и ограничивает объем работ,

который может быть выполнен одновременно, что, в свою очередь, оказывает большой отрицательный эффект.

Чтобы подчеркнуть эти два аспекта, мы собираемся создать небольшой проект, который в значительной степени их использует. Поскольку внимание в этом примере сфокусировано на встраивание Rust в другие языки, а не самой проблеме, мы будем использовать игрушечный пример:

> Запустить десять потоков. Внутри каждого потока считать от одного до пяти миллионов. После того как все десять потоков завершатся, напечатать "сделано!".

Я выбрал пять миллионов, основываясь на моем конкретном компьютере. Вот пример этого кода на Ruby:

```
threads = []

10.times do
  threads << Thread.new do
    count = 0

    5_000_000.times do
      count += 1
    end
  end
end

threads.each { |t| t.join }
puts "сделано!"
```

Попробуйте запустить этот пример, и подберите число, которое обеспечит работу в течение нескольких секунд. В зависимости от аппаратного обеспечения компьютера, возможно, придется увеличить или уменьшить это число.

На моей системе работа этой программы занимает `2.156` секунд. И если я воспользуюсь какой-нибудь утилитой для мониторинга процессов, например `top`, то увижу, что она использует только одно ядро на моей машине. Это GIL делает свое дело.

Хотя это и игрушечная программа, на ее примере можно продемонстрировать много проблем, аналогичных этой, характерных для реального мира. Для наших целей, долго крутящиеся занятые потоки представляют собой параллельные, требующие больших затрат, вычисления.

Библиотека на Rust

Давайте перепишем эту задачу на Rust. Во-первых, давайте сделаем новый проект с помощью Cargo:

```
$ cargo new embed
$ cd embed
```

Эта программа переписывается на Rust довольно легко:

```
use std::thread;

fn process() {
    let handles: Vec<_> = (0..10).map(|_| {
        thread::spawn(|| {
            let mut _x = 0;
            for _ in (0..5_000_000) {
                _x += 1
            }
        })
    }).collect();

    for h in handles {
        h.join().ok().expect("Could not join a thread!");
    }
}
```

Часть из этого уже должно выглядеть знакомым из предыдущих примеров. Мы создаем десять потоков, собирая их в вектор `handles`. Внутри каждого потока мы создаем цикл из пяти миллионов повторений, и прибавляем к `_x` единицу каждый раз. Почему здесь подчеркивание? Ну, если мы удалим его и скомпилируем:

```
$ cargo build
   Compiling embed v0.1.0 (file:///home/steve/src/embed)
src/lib.rs:3:1: 16:2 warning: function is never used: `process`, #[warn(dead_code)] on by d
efault
src/lib.rs:3 fn process() {
src/lib.rs:4     let handles: Vec<_> = (0..10).map(|_| {
src/lib.rs:5         thread::spawn(|| {
src/lib.rs:6             let mut x = 0;
src/lib.rs:7             for _ in (0..5_000_000) {
src/lib.rs:8                 x += 1
             ...
src/lib.rs:6:17: 6:22 warning: variable `x` is assigned to, but never used, #[warn(unused_v
ariables)] on by default
src/lib.rs:6             let mut x = 0;
                             ^~~~~
```

Первое предупреждение происходит, потому что мы собираем библиотеку. Если бы у нас был тест для этой функции, то предупреждение бы пропало. Но сейчас, она никогда не вызывается.

Второе предупреждение связано с использованием `x` вместо `_x`. Так как мы не выполняем каких-либо *действий* с `x`, то получаем предупреждение об этом. В нашем случае это вполне нормально, так как мы просто пытаемся тратить циклы процессора. Префикс подчеркивание перед `x` удаляет это предупреждение.

В конце мы соединяем все потоки обратно

Сейчас, однако, это просто библиотека Rust, которая не включает все необходимое для успешного вызова из другого языка. Если мы попытаемся подключить её к другому языку в том виде, в котором она сейчас, то это не будет работать. Нам нужно сделать два небольших изменения, чтобы исправить это. Первое, что мы должны сделать, это изменить начало нашего кода:

```
#[no_mangle]
pub extern fn process() {
```

Мы добавили новый атрибут, `no_mangle`. В процессе создания библиотеки Rust, в выходном скомпилированном файле происходит изменение имени функции. Причины этого выходят за рамки данного руководства, но для того, чтобы и другие языки знали, как вызвать функцию, мы должны не делать этого. Указанный атрибут выключает такое поведение.

Другим изменением, которое мы добавили, является `pub extern`. `pub` означает, что эта функция может быть вызвана за пределами этого модуля, а `extern` говорит, что её возможно вызвать из C. Вот и все! Не так и много изменений.

Второе, что мы должны сделать, это изменить настройки в `Cargo.toml`. Добавьте это в конец файла:

```
[lib]
name = "embed"
crate-type = ["dylib"]
```

Это говорит Rust, что мы хотим скомпилировать нашу библиотеку в виде стандартной динамической библиотеки. По умолчанию, Rust компилирует в "rlib", Rust- специфичный формат.

Давайте теперь соберем проект:

```
$ cargo build --release
   Compiling embed v0.1.0 (file:///home/steve/src/embed)
```

Мы ввели команду `cargo build --release`, которая выполняет сборку с включенной оптимизацией. Мы хотим, чтобы код был как можно более быстрым! Вы можете найти собранную библиотеку в `target/release`:

```
$ ls target/release/
build  deps  examples  libembed.so  native
```

Файл `libembed.so` и есть наша библиотека "совместно используемых объектов" ("shared object"). Мы можем использовать этот файл также как и любую другую динамическую библиотеку, написанную на C! Попутно следует отметить, это может быть `embed.dll` или `libembed.dylib`, в зависимости от платформы.

Теперь, когда мы получили нашу собранную библиотеку Rust, давайте используем её из нашего кода на Ruby.

Ruby

Откройте файл `embed.rb` внутри нашего проекта, и сделайте следующее:

```
require 'ffi'

module Hello
  extend FFI::Library
  ffi_lib 'target/release/libembed.so'
  attach_function :process, [], :void
end

Hello.process

puts 'сделано!'
```

Прежде чем мы сможем запустить этот код, нам нужно установить пакет `ffi`:

```
$ gem install ffi # this may need sudo
Fetching: ffi-1.9.8.gem (100%)
Building native extensions.  This could take a while...
Successfully installed ffi-1.9.8
Parsing documentation for ffi-1.9.8
Installing ri documentation for ffi-1.9.8
Done installing documentation for ffi after 0 seconds
1 gem installed
```

И, наконец, мы можем попробовать запустить его:

```
$ ruby embed.rb
сделано!
$
```

Ничего себе, это было быстро! На моей системе это заняло `0.086` секунд, а не две секунды как это было на чистом Ruby. Давайте разберем этот Ruby код:

```
require 'ffi'
```

Первый делом, нам надо объявить пакет `ffi`. Он предоставляет нам интерфейс для использования нашей библиотеки на Rust, как библиотеку на C.

```
module Hello
  extend FFI::Library
  ffi_lib 'target/release/libembed.so'
```

Автор пакета `ffi` рекомендует использовать модуль, чтобы ограничить область действия функции, которую мы импортировали из разделяемой библиотеки. Внутри мы указали `extend`, чтобы воспользоваться необходимым модулем `FFI::Library`, а затем вызвали `ffi_lib`, чтобы подгрузить нашу библиотеку. Мы просто передаем путь к библиотеке, который мы уже видели раньше, это `target/release/libembed.so`.

```
attach_function :process, [], :void
```

Метод `attach_function` предоставляется пакетом `FFI`. Здесь соединяются наша функция `process()`, написанная на Rust, и одноименная функция на Ruby. Так как `process()` не принимает аргументов, второй параметр является пустым массивом, и

поскольку функция ничего не возвращает, мы передаем `:void` в качестве завершающего аргумента.

```
Hello.process
```

Здесь мы совершаем вызов нашей Rust функции. Сочетание нашего `module` и вызова к `attach_function` завершает подготовку. Это выглядит как функция Ruby, но на самом деле это Rust!

```
puts 'сделано!'
```

Наконец, в соответствие с нашими требованиями к проекту, мы пишем `сделано!` по окончанию работы программы.

Вот и все! Как мы увидели, совместить два языка очень просто, и взамен мы получили большую производительность.

Теперь давайте попробуем на Python!

Python

Создайте файл `embed.py` в этой директории и поместите в него следующее:

```
from ctypes import cdll

lib = cdll.LoadLibrary("target/release/libembed.so")

lib.process()

print("сделано!")
```

Довольно просто! Мы импортируем `cdll` из модуля `ctypes`. Затем вызваем `LoadLibrary`. И теперь мы можем вызвать `process()`.

На моей системе это заняло `0.017` секунд. Быстро!

Node.js

Node не является языком, но, в настоящее время, это доминирующая реализация для выполнения JavaScript на стороне сервера.

Для того, чтобы сделать FFI в Node, нам сначала надо установить библиотеку:

```
$ npm install ffi
```

После установки, мы можем ей воспользоваться:

```
var ffi = require('ffi');

var lib = ffi.Library('target/release/libembed', {
  'process': ['void', []]
});

lib.process();

console.log("сделано!");
```

Пример больше похож на Ruby, чем на Python. Мы используем модуль `ffi`, чтобы получить доступ к `ffi.Library()`, который загружает нашу библиотеку. Нам нужно указать тип возвращаемого значения и типы аргументов функции: ‘void’ для возвращаемого значения и пустой массив для указания отсутствия аргументов. После этого мы просто вызываем функцию и печатаем результат.

На моей системе это заняло `0.092` секунды.

Заключение

Как вы можете видеть, основы, рассмотренные здесь, являются *очень* простыми. Конечно, мы могли бы сделать куда больше того, что мы здесь показали. Посмотрите главу FFI для более подробной информации.

Эффективное использование Rust

Итак, вы узнали, как писать код на Rust. Но есть разница между писать *любой* код на Rust и писать *хороший* код на Rust.

Этот раздел состоит из относительно самостоятельных туториалов, которые показывают вам, как повысить уровень вашего кода на Rust. В нем представлены общие паттерны и стандартные функции библиотеки. Подразделы в этом разделе могут быть прочитаны в любом порядке по вашему выбору.

Стек и Куча

Как любой системный язык программирования, Rust работает на низком уровне. Если вы пришли из языка высокого уровня, то вам могут быть незнакомы некоторые аспекты системного программирования. Наиболее важными из них являются те, которые касаются работы с памятью в стеке и в куче. Если вы уже знакомы с тем, как в C-подобных языках используется выделение памяти в стеке, то эта глава освежит ваши знания. Если же вы еще не знакомы с этим, то в общих чертах узнаете об этой концепции, но с акцентом на Rust.

Управление памятью

Эти два термина касаются управления памятью. Стек и куча - это абстракции, которые помогают вам определить, когда требуется выделение и освобождение памяти.

Вот высокоуровневое сравнение:

Стек работает очень быстро, в Rust память выделяется в стеке по умолчанию. Выделение памяти в стеке является локальным по отношению к вызову функции, и имеет ограниченный размер. Куча, с другой стороны, работает медленнее, а выделение памяти в куче осуществляется в программе явно. Но такая память имеет теоретически неограниченный размер, и доступна глобально.

Стек

Давайте поговорим о следующей программе на Rust:

```
fn main() {
    let x = 42;
}
```

Эта программа имеет одно связывание переменной, `x`. Память для него необходимо где-то выделить. Rust по умолчанию "выделяет память в стеке", что означает, что переменные "помещаются в стеке". Что это значит?

Хорошо, когда функция вызывается, то выделяется некоторый объем памяти для всех её локальных переменных и некоторой дополнительной информации. Это называется "стековый кадр", в этом руководстве мы будем игнорировать эту дополнительную информацию, и будем рассматривать лишь локальные переменные, которые мы выделяем. Таким образом, в этом случае, когда выполняется `main()`, мы выделяем одно 32-битное целое число в нашем кадре стека. Как вы можете видеть, это происходит автоматически, мы не должны писать какой-либо специальный код на Rust или что-нибудь ещё для этого.

Когда функция завершается, её стековый кадр освобождается. Это происходит автоматически, мы не должны делать что-либо специальное для этого.

Вот и все, что касается этой простой программы. Главное, что здесь нужно понять, это что выделение в стеке очень, очень быстро. Поскольку нам известны все локальные переменные заранее, то мы можем выделить для них сразу всю память единовременно. И так как они, как правило, одновременно выходят из области своего определения, то мы можем освободить выделенную память также очень быстро.

Недостатком является то, что мы не можем хранить необходимые значения дольше, чем в рамках одной функции. Мы ещё не говорили о том, что же означает название "стек". Для этого мы должны привести немного более сложный пример:

```
fn foo() {
    let y = 5;
    let z = 100;
}

fn main() {
    let x = 42;

    foo();
}
```

Эта программа имеет в общей сложности три переменные: две в `foo()`, одну в `main()`. Так же как и раньше, когда вызывается `main()`, в её стековом кадре выделяется одно целое число. Но, прежде чем мы сможем показать, что происходит, когда вызывается `foo()`, мы должны визуализировать то, что происходит с памятью. Ваша операционная система представляет отображение памяти для вашей программы, это довольно просто: огромный список адресов, от 0 до большого числа, представляющего количество оперативной памяти у вашего компьютера. Например, если у вас есть гигабайт оперативной памяти, то ваши адреса будут от `0` до `1,073,741,824`. Это число получается из `2<sup>30</sup>`, число байтов в гигабайте.

Эта память вроде гигантского массива: адреса начинаются с нуля и продолжаются до конечного числа. Так вот схема нашего первого кадра стека:

Address	Name	Value
0	x	42

У нас есть переменная `x`, расположенная по адресу `0`, имеющая значение `42`.

Когда вызывается `foo()`, выделяется новый стековый кадр:

Address	Name	Value
2	z	100
1	y	5
0	x	42

Поскольку `0` было задействовано в первом кадре, для кадра `foo()` используются `1` и `2`. Стек растет вверх, для дальнейших функций, которые мы вызываем.

Здесь необходимо принять к сведению некоторые важные замечания. Адреса 0, 1 и 2 приведены исключительно в иллюстративных целях, и не имеют никакого отношения к фактическим адресам, которые компьютер будет использовать. В частности, набор адресов в действительности включает разделители, состоящие из некоторого числа байтов, которые отделяют каждый из адресов, и размер этого разделителя может даже превышать размер хранящегося значения.

После того, как `foo()` завершается, её кадр будет освобожден:

Address	Name	Value
0	x	42

А потом, после `main()`, даже это последнее значение уходит. Легко!

Это называется "стек" (по-русски, стопка), потому что он работает как стопка тарелок: первая тарелка, которую вы положили, будет последней тарелкой, которую вы возьмете обратно. По этой причине, стек иногда называют очередью "последним пришел, первым вышел". Последнее значение, которое вы положили в стек, будет первым, которое вы получите из него.

Давайте попробуем трех-уровневый пример:

```
fn bar() {
    let i = 6;
}

fn foo() {
    let a = 5;
    let b = 100;
    let c = 1;

    bar();
}

fn main() {
    let x = 42;

    foo();
}
```

Хорошо, сначала вызывается `main()`:

Address	Name	Value
0	x	42

Затем из `main()` вызывается `foo()`:

Address	Name	Value
3	c	1
2	b	100
1	a	5

0	x	42

И затем из `foo()` вызывается `bar()`:

Address	Name	Value
4	i	6
3	c	1
2	b	100
1	a	5
0	x	42

Уф! Наш стек растет вверх.

После того, как `bar()` завершается, её кадр будет освобожден, оставляя только `foo()` и `main()`:

Address	Name	Value
3	c	1
2	b	100
1	a	5
0	x	42

И затем завершается `foo()`, оставляя только `main()`

Address	Name	Value
0	x	42

И вот мы закончили. Уловили суть? Это как стопка тарелок: вы кладете наверх, и вы берете сверху.

Куча

Такой способ выделения памяти работает очень хорошо, но он не всегда может быть использован. Иногда вам необходимо передать некоторую память между различными функциями или сохранить её валидность после окончания выполнения функции. Для этого, мы можем использовать кучу.

В Rust, вы можете выделить память в куче с помощью типа `Box<T>`. Вот пример:

```
fn main() {
    let x = Box::new(5);
    let y = 42;
}
```

Вот что происходит с памятью, когда вызывается `main()`:

Address	Name	Value

1	y	42
0	x	??????

Мы выделяем место для двух переменных в стеке. `y` представляет собой `42`, тут всё как обычно, но что насчёт `x`? Хорошо, `x` представляет собой `Box<i32>`, а упаковки выделяют память в куче. Фактическое значение упаковки - структура, которая имеет указатель на "кучу". Когда начинает выполняться функция, осуществляется вызов `Box::new()`, который выделяет некоторый объем памяти в куче, и кладет туда `5`. Память в настоящее время выглядит следующим образом:

Address	Name	Value
2^{30}		5
...	...	...
1	y	42
0	x	2^{30}

У нас есть 2^{30} в нашем гипотетическом компьютере с 1Гб оперативной памяти. А так как наш стек растет от нуля, то проще всего выделить память с другого конца. Таким образом, наше первое значение находится на самом высоком месте в памяти. А значением структуры в `x` является сырой указатель на адрес, который мы выделили в куче, так что значение `x` равно 2^{30}, это то самое местоположение в памяти.

Мы не слишком много говорили о том, что на самом деле означает выделить и освободить память в этих контекстах. Чрезмерное углубление в детали по этому вопросу выходит за рамки данного руководства, но важно отметить, что куча - это не просто стек, который растет с противоположного конца. Как мы увидим в примерах в этой книге дальше, память из кучи может быть выделена и освобождена в любом порядке, что в конечном итоге может привести к "дыркам". Вот схема размещения памяти программы, проработавшей в течение некоторого времени:

Address	Name	Value
2^{30}		5
$(2^{30}) - 1$		
$(2^{30}) - 2$		
$(2^{30}) - 3$		42
...	...	...
3	y	$(2^{30}) - 3$
2	y	42
1	y	42
0	x	2^{30}

В этом примере мы выделили четыре элемента в куче, но освободили лишь два из них. Отсюда разрыв между 2^{30} и (2^{30}) - 3, который в настоящее время не используется. Конкретные детали того, как и почему это происходит, зависят от того, какую стратегию вы используете для управления кучей. Различные программы могут использовать различные "распределители памяти", которые представляют собой библиотеки, которые управляют памятью за вас. Rust программы используют jemalloc для этой цели.

Короче, вернемся к нашему примеру. Так как эта память расположена в куче, то она может оставаться в живых (валидной) дольше, чем функция, которая выделяет упаковку. В данном случае, однако, это не так.[moving] Когда функция завершается, мы должны освободить кадр стека для `main()`. Хотя у `Box<T>` для этого есть свой трюк: Drop. Реализация `Drop` для `Box` освобождает память, которая была выделена при создании. Отлично! Поэтому, когда `x` уходит, сначала освобождается память, выделенная в куче:

Address	Name	Value
1	y	42
0	x	??????

[moving]: Мы можем сделать время жизни памяти более долгим путем передачи права собственности, что иногда называют "перемещение из упаковки" ("moving out of the box"). Более сложные примеры будут рассмотрены позже.

А потом кадр стека уходит, освобождая всю нашу память.

Аргументы и заимствование

У нас есть некоторые простые примеры со стеком и кучей, но что насчёт аргументов функции и заимствования? Вот небольшая программа на Rust:

```
fn foo(i: &i32) {
    let z = 42;
}

fn main() {
    let x = 5;
    let y = &x;

    foo(y);
}
```

Когда мы входим в `main()`, память выглядит следующим образом:

Address	Name	Value
1	y	0
0	x	5

Значением `x` является `5`, а `y` представляет собой ссылку на `x`. То есть, ее значением является адрес памяти, в котором расположен `x`, который в данном случае является `0`.

А что насчёт случая, когда мы вызываем `foo()`, передавая `y` в качестве аргумента?

Address	Name	Value
3	z	42
2	i	0
1	y	0
0	x	5

Фреймы стека используются не только для локальных привязок, но также и для аргументов. Таким образом, в этом случае, наш кадр должен содержать как `i`, наш аргумент, так и `z`, нашу привязку локальной переменной. `i` - это копия аргумента `y`. Соответственно, значением `i`, как и значением `y`, является `0`.

Это одна из причин, почему заимствование переменной не освобождает какую-либо память: значением ссылки является просто указатель на область памяти. Если мы освободим нижележащую по этому указателю память, то это может привести к ошибкам в дальнейшей работе.

Сложный пример

Хорошо, давайте рассмотрим следующую комплексную программу шаг за шагом:

```
fn foo(x: &i32) {
    let y = 10;
    let z = &y;

    baz(z);
    bar(x, z);
}

fn bar(a: &i32, b: &i32) {
    let c = 5;
    let d = Box::new(5);
    let e = &d;

    baz(e);
}

fn baz(f: &i32) {
    let g = 100;
}

fn main() {
    let h = 3;
    let i = Box::new(20);
    let j = &h;

    foo(j);
}
```

Сначала, мы вызываем `main()`:

Address	Name	Value
2^{30}		20
...	...	...
2	j	0
1	i	2^{30}
0	h	3

Мы выделяем память для `j`, `i`, и `h`. `i` выделена в куче и поэтому содержит указатель на значение в куче.

Далее, в конце вызова `main()`, вызывается `foo()`:

Address	Name	Value
2^{30}		20
...	...	...
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

Пространство выделяется для `x`, `y` и `z`. Аргумент `x` имеет такое же значение, как и `j`, так как мы передали в качестве аргумента именно его. Это указатель на адрес `0`, так как `j` указывает на `h`.

Далее, `foo()` вызывает `baz()`, передавая `z`:

Address	Name	Value
2^{30}		20
...	...	...
7	g	100
6	f	4
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

Мы выделили память для `f` и `g`. `baz()` очень короткая, и когда она завершается, мы избавляемся от её кадра стека:

Address	Name	Value
2^{30}		20
...	...	...
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

Далее `foo()` вызывает `bar()` с аргументами `x` и `z`:

Address	Name	Value
2^{30}		20
$(2^{30}) - 1$		5
...	...	...
10	e	9
9	d	$(2^{30}) - 1$
8	c	5
7	b	4
6	a	0
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

Тут мы выделяем другое значение в куче, и поэтому мы вычитаем единицу из 2^{30}. Это выражение написать легче, чем `1,073,741,823`. В любом случае, переменные создаются, как обычно.

В конце `bar()` вызывает `baz()`:

Address	Name	Value
2^{30}		20

$(2^{30}) - 1$		5
...	...	...
12	g	100
11	f	4
10	e	9
9	d	$(2^{30}) - 1$
8	c	5
7	b	4
6	a	0
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

Сейчас мы находимся в самой глубокой точке! Уф! Поздравляю с достижением данной точки.

После завершения `baz()`, мы избавляемся от `f` и `g`:

Address	**Name**	**Value**
2^{30}		20
$(2^{30}) - 1$		5
...	...	...
10	e	9
9	d	$(2^{30}) - 1$
8	c	5
7	b	4
6	a	0
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

Далее мы выполняем возврат из `bar()`. В этом случае `d` представляет собой `Box<T>`, поэтому он также освобождает и то, на что он указывает: (2^{30}) - 1.

Address	Name	Value
2^{30}		20
...	...	...
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

И после этого происходит возврат из `foo()`:

Address	Name	Value
2^{30}		20
...	...	...
2	j	0
1	i	2^{30}
0	h	3

И вот, наконец, `main()`, которая очищает все остальное. Когда `i` дропается (`Drop`), будет также очищен и конец кучи.

А что делают другие языки?

Большинство языков с сборщиком мусора по умолчанию выделяет память из кучи. Это означает, что каждое значение будет упаковано. Есть ряд причин, почему делается именно так, но они выходят за рамки данного руководства. Есть несколько возможных оптимизаций, которые правда также не достигают своей цели во всех 100% случаях. Вместо того чтобы полагаться на стек и `Drop` в вопросах очистки памяти, сборщик мусора работает с кучей.

Что использовать?

Но, если стек быстрее и проще в управлении, зачем тогда нужна куча? Весомая причина заключается в том, что память в стеке может выделяться только по принципу "первым пришёл - последним вышел". Таким образом, место из-под кадра стека предыдущего вызова функции будет переиспользовано под следующий вызов. Выделение в куче - более общая техника. Она позволяет выделение и освобождение памяти в любом порядке. Однако, это достигается ценой увеличения сложности реализации механизма выделения памяти.

В общем случае, следует предпочитать выделение в стеке, и поэтому, Rust использует выделение в стеке по умолчанию. LIFO модель стека проще, на фундаментальном уровне. Это значит, что программа быстрее исполняется, и проще по смыслу.

Эффективность времени выполнения

Управление памятью для стека тривиально: машина просто увеличивает или уменьшает одно значение, так называемый "указатель стека". Управление памятью для кучи нетривиально: память, выделенная в куче, освобождается в произвольные точки, а каждый блок выделенной в куче памяти может быть произвольного размера. Менеджеру памяти, как правило, требуется приложить гораздо больше усилий для определения памяти, которую можно использовать заново.

Если вы хотите изучить эту тему более подробно, то эта статья будет отличным введением.

Простота программы

Выделение памяти в стеке воздействует как на сам язык Rust, так и на модель мышления разработчиков. Стековая семантика - ключевое понятие Rust. Мы получаем автоматическое управление памятью без усложнения среды исполнения. Именно этот механизм позволяет освободить память в куче, как только её владелец вышел из области видимости - по сути, как только схлопнулся стек кадра, на котором он жил. К сожалению, в некоторых ситуациях стека недостаточно. Если нужна большая гибкость во владении памятью, можно воспользоваться стётчиками ссылок `Rc<T>` и `Arc<T>`.

Желание более удобно пользоваться памятью в куче может доходить до крайности. С одной стороны, можно реализовать сборщик мусора - но это сильно увеличивает сложность среды исполнения. С другой стороны, полностью ручное управление памятью с явным вызовом процедуры освобождения часто приводит к ошибкам, предотвратить которые компилятор Rust не в силах.

Тестирование

> Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.
>
> Edsger W. Dijkstra, "The Humble Programmer" (1972)

Давайте поговорим о том, как тестировать Rust код. Мы не будем рассказывать о том, какой подход к тестированию Rust кода является верным. Есть много подходов, каждый из которых имеет свое представление о правильном написании тестов. Но все эти подходы используют одни и те же основные инструменты, и мы покажем вам синтаксис их использования.

Тесты с атрибутом `test`

В самом простом случае, тест в Rust - это функция, аннотированная с помощью атрибута `test`. Давайте создадим новый проект при помощи Cargo, который будет называться `adder`:

```
$ cargo new adder
$ cd adder
```

При создании нового проекта, Cargo автоматически сгенерирует простой тест. Ниже представлено содержимое `src/lib.rs`:

```
#[test]
fn it_works() {
}
```

Обратите внимание на `#[test]`. Этот атрибут указывает, что это тестовая функция. В этом примере она не имеет тела. Но такого вида функции достаточно, чтобы удачно выполнить тест. Запуск тестов осуществляется командой `cargo test`.

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Cargo скомпилировал и запустил наши тесты. В результате мы получили выходные данные, поделенные на два раздела: один содержит информацию о тесте, который мы написали, а другой - информацию о тестах из документации. Но об этом позже. А сейчас посмотрим на эту строку:

```
test it_works ... ok
```

Обратите внимание на `it_works`. Это название нашей функции:

```
fn it_works() {
```

Мы также получили итоговую строку:

```
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

Так почему же наш ничего не делающий тест был выполнен удачно? Любой тест, который не вызывает `panic!`, выполняется удачно, и любой тест, который вызывает `panic!`, выполняется неудачно. Давайте сделаем тест, который выполнится неудачно:

```
#[test]
fn it_works() {
    assert!(false);
}
```

`assert!` является макросом, определенным в Rust, который принимает один аргумент: если аргумент имеет значение `true`, то ничего не происходит; если аргумент является `false`, то вызывается `panic!`. Давайте запустим наши тесты снова:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... FAILED

failures:

---- it_works stdout ----
        thread 'it_works' panicked at 'assertion failed: false', /home/steve/tmp/adder/src/
lib.rs:3



failures:
    it_works

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured

thread '<main>' panicked at 'Some tests failed', /home/steve/src/rust/src/libtest/lib.rs:24
7
```

Rust сообщает, что наш тест выполнен неудачно:

```
test it_works ... FAILED
```

Это же отражается в итоговой строке:

```
test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured
```

Мы также получаем ненулевой код состояния:

```
$ echo $?
101
```

Это бывает полезно, если вы хотите интегрировать `cargo test` в сторонний инструмент.

Мы можем инвертировать ожидаемый результат теста с помощью атрибута: `should_panic`:

```
#[test]
#[should_panic]
fn it_works() {
    assert!(false);
}
```

Теперь этот тест будет выполнен удачно, если вызывается `panic!`, и неудачно, если `panic!` не вызывается. Давайте попробуем:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Rust предоставляет и другой макрос, `assert_eq!`, который проверяет эквивалентность двух артументов:

```
#[test]
#[should_panic]
fn it_works() {
    assert_eq!("Hello", "world");
}
```

А теперь этот тест будет выполнен удачно или неудачно? Из-за атрибута `should_panic` он завершится удачно:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

`should_panic` тесты могут быть хрупкими, потому как трудно гарантировать, что тест не вызовет панику по неожиданной причине. Чтобы помочь в этом аспекте, к атрибуту `should_panic` может быть добавлен необязательный параметр `expected`. Тогда тест также будет проверять, что сообщение об ошибке содержит ожидаемый текст. Ниже представлен более безопасный вариант приведенного выше примера:

```
#[test]
#[should_panic(expected = "assertion failed")]
fn it_works() {
    assert_eq!("Hello", "world");
}
```

Вот и все, что касается основ! Давайте напишем один 'реальный' тест:

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[test]
fn it_works() {
    assert_eq!(4, add_two(2));
}
```

Это распространенное использование макроса `assert_eq!`: вызывать некоторую функцию с известными аргументами и сравнить результат ее вызова с ожидаемыми результатом.

Тесты в модуле `test`

Есть один нюанс, из-за которого наш пример нельзя назвать идиоматическим: отсутствует модуль тестирования. Идиоматический вариант написания нашего примера будет выглядить примерно так:

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

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

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

Здесь есть несколько изменений. Первое - это введение `mod test` с атрибутом `cfg`. Модуль позволяет сгруппировать все наши тесты вместе, а также определить вспомогательные функции, если это необходимо, которые будут отделены от остальной части контейнера. Атрибут `cfg` указывает на то, что тест будет скомпилирован, только когда мы попытаемся запустить тесты. Это может сэкономить время компиляции, а также гарантирует, что наши тесты полностью исключены из обычной сборки.

Второе изменение заключается в объявлении `use`. Так как мы находимся во внутреннем модуле, то мы должны объявить использование тестируемой функции в его области видимости. Это может раздражать, если у вас большой модуль, и поэтому обычно используют фичу `glob`. Давайте, в соответствии с этим, изменим `src/lib.rs`:

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

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

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

Обратите внимание на различие в строке с `use`. Теперь запустим наши тесты:

```
$ cargo test
    Updating registry `https://github.com/rust-lang/crates.io-index`
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test test::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Работает!

Данный подход представляет собой использование модуля `test` содержащего тесты в "юнит стиле". Любой код, задачей которого является только лишь тестирование небольшего кусочка функциональности, имеет смысл перенести в этот модуль. Но что если мы хотим использовать "интеграционный стиль" для создания тестов? Для этого следует использовать директорию `tests`

Тесты в директории `tests`

Чтобы написать интеграционный тест, давайте создадим директорию `tests`, и положим в нее файл `tests/lib.rs` со следующим содержимым:

```
extern crate adder;

#[test]
fn it_works() {
    assert_eq!(4, adder::add_two(2));
}
```

Выглядит примерно так же, как и наши предыдущие тесты, но есть некоторые отличия. Теперь сверху у нас расположено `extern crate adder`. Это потому, что тесты в директории `tests` - это отдельный контейнер, и, следовательно, мы должны импортировать нашу библиотеку. Это также объясняет, почему директория `tests` наиболее подходящее место для написания интеграционных тестов: они используют библиотеку, как это делал бы любой другой потребитель.

Давайте запустим их:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test test::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

     Running target/lib-c18e7d3494509e74

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Теперь у нас появилось три раздела: запускается предыдущий тест, а также запускается наш новый тест.

Это все, что касается директории `tests`. Модуль `test` здесь не нужен, так как все здесь ориентировано на тесты.

Давайте, наконец, перейдем к третьей части: тесты из документации.

Тесты в документации

Нет ничего лучше, чем документация с примерами. Нет ничего хуже, чем примеры, которые на самом деле не работают, потому что код изменился с тех пор, как документация была написана. Для этого, Rust поддерживает автоматический запуск примеров в документации. Вот дополненный `src/lib.rs` с примерами:

```
//! The `adder` crate provides functions that add numbers to other numbers.
//!
//! # Examples
//!
//! ```
//! assert_eq!(4, adder::add_two(2));
//! ```

/// This function adds two to its argument.
///
/// # Examples
///
/// ```
/// use adder::add_two;
///
/// assert_eq!(4, add_two(2));
/// ```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

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

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

Обратите внимание на документацию уровня модуля, начинающуюся с `//!` и на документацию уровня функции, начинающуюся с `///`. Документация Rust поддерживает Markdown в комментариях, поэтому блоки кода помечают тройными символами `. В комментарии документации обычно включают раздел `# Examples`, содержащий примеры, такие как этот.

Давайте запустим тесты снова:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/steve/tmp/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test test::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

     Running target/lib-c18e7d3494509e74

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 2 tests
test add_two_0 ... ok
test _0 ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured
```

Теперь у нас запускаются все три вида тестов! Обратите внимание на имена тестов из документации: `_0` генерируется для модульных тестов, и `add_two_0` - для функциональных тестов. Цифры на конце будут автоматически инкрементироваться, например `add_two_1`, если вы добавите еще примеров.

Условная компиляция

В Rust есть специальный атрибут, `#[cfg]`, который позволяет компилировать код в зависимости от флагов переданых компилятору. Он имеет две формы:

```
#[cfg(foo)]

#[cfg(bar = "baz")]
```

Он также имеет несколько помощников:

```
#[cfg(any(unix, windows))]

#[cfg(all(unix, target_pointer_width = "32"))]

#[cfg(not(foo))]
```

Которые могут быть как угодно вложены:

```
#[cfg(any(not(unix), all(target_os="macos", target_arch = "powerpc")))]
```

Что же касается того, как включить или отключить эти флаги: если вы используете Cargo, то они устанавливаются в разделе `[features]` вашего `Cargo.toml`:

```
[features]
# no features by default
default = []

# The “secure-password” feature depends on the bcrypt package.
secure-password = ["bcrypt"]
```

Если вы сделаете это, то Cargo передаст флаг в `rustc`:

```
--cfg feature="${feature_name}"
```

Совокупность этих `cfg` флагов будет определять, какие из них будут активны, и, следовательно, какой код будет скомпилирован. Давайте рассмотрим такой код:

```
#[cfg(feature = "foo")]
mod foo {
}
```

Если скомпилировать его с помощью `cargo build --features "foo"`, то будет передан флаг `--cfg feature="foo"` в `rustc` и выход будет содержать модуль `mod foo`. Если скомпилировать его с помощью обычной команды `cargo build`, то никаких дополнительных флагов передано не будет, и поэтому, модуль `mod foo` не будет существовать.

cfg_attr

Вы также можете установить другой атрибут в зависимости от переменной `cfg` с помощью атрибута `cfg_attr`:

```
#[cfg_attr(a, b)]
```

Этот код будет равносилен атрибуту `#[b]`, если в атрибуте `cfg` установлен флаг `a`, или "без атрибута" в противном случае.

cfg!

Расширение синтаксиса `cfg!` позволяет использовать данные виды флагов и в другом месте в коде:

```
if cfg!(target_os = "macos") || cfg!(target_os = "ios") {
    println!("Think Different!");
}
```

Они будут заменены на `true` или `false` во время компиляции, в зависимости от настройки конфигурации.

Документация

Документация является важной частью любого программного проекта, и в Rust она первоклассна. Давайте поговорим об инструментах Rust, предназначенных для создания документации к проекту.

О `rustdoc`

Дистрибутив Rust включает в себя инструмент, `rustdoc`, который генерирует документацию. `rustdoc` также используется Cargo через `cargo doc`.

Документация может быть сгенерирована двумя методами: из исходного кода, и из автономных Markdown файлов.

Документирование исходного кода

Основной способ документирования проекта на Rust заключается в комментировании исходного кода. Для этой цели вы можете использовать комментарии документации:

```
/// Constructs a new `Rc<T>`.
///
/// # Examples
///
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
pub fn new(value: T) -> Rc<T> {
    // implementation goes here
}
```

Этот код генерирует документацию, которая выглядит так. В приведенном коде реализация метода была заменена на обычный комментарий. Первое, на что следует обратить внимание в этом примере: используется `///`, а не `//`. Символы `///` указывают, что это комментарий документации.

Комментарии документации написаны на Markdown.

Rust отслеживает такие комментарии, и использует их при генерировании документации. Что важно при документировании таких вещей, как перечисления:

```
/// The `Option` type. See [the module level documentation](http://doc.rust-lang.org/) for
more.
enum Option<T> {
    /// No value
    None,
    /// Some value `T`
    Some(T),
}
```

Код, приведенный выше работает, а ниже - не работает:

```
/// The `Option` type. See [the module level documentation](http://doc.rust-lang.org/) for
more.
enum Option<T> {
    None, /// No value
    Some(T), /// Some value `T`
}
```

Вы получите ошибку:

```
hello.rs:4:1: 4:2 error: expected ident, found `}`
hello.rs:4 }
           ^
```

Эта досадная ошибка заключается в следующем: комментарии документации распространяются на элементы, расположенные за ними, а в данном примере нет элемента, расположенного после последнего комментария.

Написание комментариев документации

Давайте рассмотрим каждую часть приведенного комментария в деталях:

```
/// Constructs a new `Rc<T>`.
```

Первая строка комментария документации должна представлять из себя краткую информацию о функциональности. Одно предложение. Только самое основное. Высокоуровневое.

```
///
/// Other details about constructing `Rc<T>`s, maybe describing complicated
/// semantics, maybe additional options, all kinds of stuff.
///
```

Наш исходный пример включал только строку с краткой информацией, но если бы у нас было больше информации, о которой следует сказать, мы могли бы добавить эту информацию в новом параграфе.

Специальные разделы

```
/// # Examples
```

Далее идут специальные разделы. Они обозначены с заголовком, который начинается с `#`. Существуют три вида заголовков, которые обычно используются. Они не являются каким-либо специальным синтаксисом, на данный момент это просто соглашение.

```
/// # Panics
```

Неустранимые ошибки при неправильном вызове функции (так называемые ошибки программирования) в Rust как правило вызывают панику, которая, в крайнем случае, убивает весь текущий поток (thread). Если ваша функция имеет подобное нетривиальное поведение, другими словами, обнаруживает/вызывает панику, то очень важно задокументировать это.

```
/// # Failures
```

Если ваша функция или метод возвращает `Result<T, E>`, то было бы правильно описать условия, при которых она возвращает `Err(E)`. Это чуть менее важно, чем описание `Panics`, потому как неудача кодируется в системе типа, но это не значит что стоит пренебрегать этим.

```
/// # Safety
```

Если ваша функция является `unsafe`, необходимо пояснить, какие инварианты вызова должны поддерживаться.

```
/// # Examples
///
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
```

Раздел `Examples`. Включите в этот раздел один или несколько примеров использования функции или метода, и ваши пользователи будут вам благодарны за это. Примеры должны размещаться внутри блоков кода, о которых мы сейчас поговорим. Этот раздел может иметь более одного подраздела:

```
/// # Examples
///
/// Simple `&str` patterns:
///
/// ```
/// let v: Vec<&str> = "Mary had a little lamb".split(' ').collect();
/// assert_eq!(v, vec!["Mary", "had", "a", "little", "lamb"]);
/// ```
///
/// More complex patterns with a lambda:
///
/// ```
/// let v: Vec<&str> = "abc1def2ghi".split(|c: char| c.is_numeric()).collect();
/// assert_eq!(v, vec!["abc", "def", "ghi"]);
/// ```
```

Давайте детально обсудим блоки кода.

Блок кода

Чтобы написать код на Rust в комментарии, используйте символы ```:

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

Если вы хотите написать код на любом другом языке (не на Rust), вы можете добавить аннотацию:

```
/// ```c
/// printf("Hello, world\n");
/// ```
```

Это позволит использовать подсветку синтаксиса, соответствующую тому языку, который был указан в аннотации. Если же это простой текст, то в аннотации указывается `text`.

Важно выбрать правильную аннотацию, потому что `rustdoc` использует ее интересным способом: Rust может выполнять проверку работоспособности примеров на момент запуска, чтобы они не устаревали. Предположим у вас есть код на C. Если вы опустите аннотацию, указывающую, что это код на C, то `rustdoc` будет думать, что это код на Rust, поэтому `rustdoc` будет жаловаться при попытке создания документации.

Тесты в документации

Давайте обсудим наш пример документации:

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

Заметьте, что здесь нет нужды в `fn main()` или чем-нибудь подобном. `rustdoc` автоматически добавит main() обертку вокруг вашего кода в нужном месте. Например:

```
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
```

В конечном итоге это будет тест:

```
fn main() {
    use std::rc::Rc;
    let five = Rc::new(5);
}
```

Вот полный алгоритм, который использует `rustdoc` для постобработки примеров:

1. Любые ведущие (leading) атрибуты `#![foo]` остаются без изменений в качестве атрибутов контейнера.

2. Будут вставлены некоторые общие `allow` атрибуты, в том числе: `unused_variables`, `unused_assignments`, `unused_mut`, `unused_attributes`, `dead_code`. Небольшие примеры часто вызывают эти строки.
3. Если пример не содержит `extern crate`, то будет вставлено `extern crate <mycrate>;`.
4. Наконец, если пример не содержит `fn main`, то оставшаяся часть текста будет обернута в `fn main() { your_code }`

Хотя иногда этого не достаточно. Например, все эти примеры кода с `///`, о которых мы говорим? Представленный ниже блок кода до обработки `rustdoc`, в виде простого текста:

```
/// Some documentation.
# fn foo() {}
```

выглядит иначе, чем после обработки `rustdoc`, в виде сгенерированного вывода:

```
/// Some documentation.
```

Да, именно так: вы можете добавлять строки, которые начинаются с `#`, и они будут скрыты в выводе, но при этом будут использоваться во время компиляции кода. Вы можете использовать это в своих интересах. Если в комментарии документации необходимо обратиться к какой-то функции, то ниже нужно будет добавить определение этой функции. В то же время, это делается только для того, чтобы удовлетворить компилятор, поэтому сокрытие ненужных строк в выводе делает пример более ясным. Вы можете использовать эту технику, чтобы детально объяснять длинные примеры, сохраняя при этом тестируемость документации. Например, этот код:

```
let x = 5;
let y = 6;
println!("{}", x + y);
```

Вот объяснение, которое будет сгенерировано:

Сперва мы устанавливаем `x` равным пяти:

```
let x = 5;
```

Затем мы устанавливаем `y` равным шести:

```
let y = 6;
```

В конце мы печатаем сумму `x` и `y`:

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

Вот то же самое объяснение, но в виде простого текста:

> Сперва мы устанавливаем `x` равным пяти:

```
let x = 5;
# let y = 6;
# println!("{}", x + y);
```

Затем мы устанавливаем `y` равным шести:

```
# let x = 5;
let y = 6;
# println!("{}", x + y);
```

В конце мы печатаем сумму `x` и `y`:

```
# let x = 5;
# let y = 6;
println!("{}", x + y);
```

Повторяя все части примера, вы можете быть уверены, что ваш пример компилируется, а не просто отображает кусоки кода, которые имеют отношение к той или иной части вашего объяснения.

Документирование макросов

Вот пример документирования макроса:

```
/// Panic with a given message unless an expression evaluates to true.
///
/// # Examples
///
/// ```
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(1 + 1 == 2, “Math is broken.”);
/// # }
/// ```
///
/// ```should_panic
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(true == false, “I’m broken.”);
/// # }
/// ```
#[macro_export]
macro_rules! panic_unless {
    ($condition:expr, $($rest:expr),+) => ({ if ! $condition { panic!($($rest),+); } });
}
```

В нем вы можете заметить, три вещи. Во-первых, мы должны собственноручно добавить строку с `extern crate`, для того, чтобы мы могли указать атрибут `#[macro_use]`. Во-вторых, мы также собственноручно должны добавить `main()`. И наконец, разумно будет использовать `#`, чтобы закомментировать все, что мы добавили в первых двух пунктах, что бы оно не отображалось в генерируемом вводе.

Запуск тестов в документации

Для запуска тестов можно использовать одну из двух комманд

```
$ rustdoc --test path/to/my/crate/root.rs
# or
$ cargo test
```

Все верно, `cargo test` также выполняет тесты, встроенные в документацию. Тем не менее, `cargo test` не будет тестировать исполняемые контейнеры, только библиотечные. Это связано с тем, как работает `rustdoc`: он компонует библиотеку в зависимости от результата тестирования, но в случае с исполняемым файлом, его не с чем компоновать.

Есть еще несколько полезных аннотаций, которые помогают `rustdoc` работать правильно (корректно) при тестировании кода:

```
/// ```ignore
/// fn foo() {
/// ```
```

Аннотация `ignore` указывает Rust, что код должен быть проигнорирован. Почти во всех случаях это не то, что вам нужно, так как эта директива носит очень общий характер. Вместо нее лучше использовать аннотацию `text`, если это не код, или `#`, чтобы получить рабочий пример, отображающий только ту часть, которая вам нужна.

```
/// ```should_panic
/// assert!(false);
/// ```
```

Аннотация `should_panic` указывает `rustdoc`, что код должен компилироваться, но выполнение теста должно завершиться ошибкой.

```
/// ```no_run
/// loop {
///     println!("Hello, world");
/// }
/// ```
```

Аннотация `no_run` указывает, что что код должен компилироваться, но запускать его на выполнение не требуется. Это важно для таких примеров, которые должны успешно компилироваться, но которые выполняются в бесконечном цикле! Например: "Вот как запустить сетевой сервис".

Документирование модулей

Rust предоставляет еще один вид комментариев документации, `//!`. Этот комментарий относится не к следующему за ним элементу, а к элементу, который его включает. Другими словами:

```
mod foo {
    //! This is documentation for the `foo` module.
    //!
    //! # Examples

    // ...
}
```

Приведенный пример демонстрирует наиболее распространенное использование `//!`: документирование модуля. Если же модуль расположен в файле `foo.rs`, то вы, открывая его код, часто будете видеть следующее:

```
//! A module for using `foo`s.
//!
//! The `foo` module contains a lot of useful functionality blah blah blah
```

Стиль комментариев документации

Изучите RFC 505 для получения полных сведений о соглашениях по стилю и формату документации.

Другая документация

Все эти правила поведения также применимы и в отношении исходных файлов не на Rust. Так как комментарии пишутся на Markdown, то часто эти файлы имеют расширение `.md`.

Когда вы пишете документацию в Markdown файлах, вам не нужно добавлять префикс комментария документации, `///`. Например:

```
/// # Examples
///
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
```

преобразуется в

```
# Examples

```
use std::rc::Rc;

let five = Rc::new(5);
```
```

когда он находится в Markdown файле. Однако есть один недостаток: Markdown файлы должны иметь заголовок наподобие этого:

```
% The title

This is the example documentation.
```

Строка, начинающаяся с `%`, должна быть самой первой строкой файла.

Атрибуты `doc`

На более глубоком уровне, комментарии документации - это синтаксический сахар для атрибутов документации:

```
/// this

#[doc="this"]
```

представленные выше комментарии идентичны, также как и ниже:

```
//! this

#![doc="/// this"]
```

Вы не часто будете видеть этот атрибут, используемый для написания документации, но он может быть полезен, когда изменения некоторых настроек, или при написании макроса.

Ре-экспорт

`rustdoc` будет показывать документацию для public ре-экспорта в двух местах:

```
extern crate foo;

pub use foo::bar;
```

Это создаст документацию для `bar` как в документации для контейнера `foo`, так и в документации к вашему контейнеру. То есть будет использована одна и та же документация в обоих местах.

Такое поведение может быть подавлено с помощью `no_inline`:

```
extern crate foo;

#[doc(no_inline)]
pub use foo::bar;
```

Управление HTML

Вы можете управлять некоторыми аспектами HTML, который генерируется `rustdoc`, через атрибут `#![doc]`:

```
#![doc(html_logo_url = "http://www.rust-lang.org/logos/rust-logo-128x128-blk-v2.png",
       html_favicon_url = "http://www.rust-lang.org/favicon.ico",
       html_root_url = "http://doc.rust-lang.org/")];
```

В этом примере устанавливается несколько различных опций: логотип, иконка и корневой URL.

Опции генерации

`rustdoc` также содержит несколько опций командной строки, для дальнейшей кастомизации:

- `--html-in-header FILE`: включить содержание FILE в конец `<head>...</head>` раздела.
- `--html-before-content FILE`: включить содержание FILE сразу после `<body>`, перед отображаемым содержимым (в том числе строки поиска).
- `--html-after-content FILE`: включить содержание FILE после всего отображаемого содержимого.

Замечание по безопасности

Комментарии в Markdown документации помещаются в конечную веб-страницу без обработки. Будьте осторожны с HTML литералами:

```
/// <script>alert(document.cookie)</script>
```

Итераторы

Давайте поговорим о циклах.

Помните цикл `for` в Rust? Вот пример:

```
for x in 0..10 {
    println!("{}", x);
}
```

Теперь, когда вы знаете о Rust немного больше, мы можем детально обсудить, как же это работает. Диапазоны (`0..10`) являются 'итераторами'. Итератор - это сущность, для которой мы можем неоднократно вызвать метод `.next()`, в результате чего мы получим последовательность элементов.

Как представлено ниже:

```
let mut range = 0..10;

loop {
    match range.next() {
        Some(x) => {
            println!("{}", x);
        },
        None => { break }
    }
}
```

Мы связываем с диапазоном изменяемую переменную, которая и является нашим итератором. Затем мы используем цикл `loop` с внутренней конструкцией `match`. Здесь `match` применяется к результату `range.next()`, который выдает нам ссылку на следующее значение итератора. В данном случае `next` возвращает `Option<i32>`, который представляет собой: `Some(i32)` - когда у нас есть значение и `None` - когда значение отсутствует. Если мы получаем `Some(i32)`, то печатаем его, а если `None`, то прекращаем выполнение цикла оператором `break`.

Этот пример, по большому счету, такой же, как и пример с циклом `for`. Цикл `for` - просто удобный способ записи конструкции `loop`/`match`/`break`.

Однако, цикл `for` не является единственной конструкцией, которая использует итераторы. Написание своего собственного итератора заключаться в реализации типажа `Iterator`. Хотя эта тема и выходит за рамки данного руководства, Rust предоставляет ряд полезных итераторов для выполнения различных задач. Прежде чем мы поговорим о них, мы должны рассказать о Rust анти-паттерне, связанном с использованием диапазонов, который продемонстрирован в примере ниже.

Да, мы только что говорили о том, какие диапазоны крутые. Но диапазоны также и очень примитивны. Например, если вам нужно проитерировать содержимое вектора, у вас может возникнуть желание написать так:

```
let nums = vec![1, 2, 3];

for i in 0..nums.len() {
    println!("{}", nums[i]);
}
```

Это намного хуже, чем если бы мы использовали итератор непосредственно. Вы можете итерировать по элементам векторов напрямую, как показано ниже:

```
let nums = vec![1, 2, 3];

for num in &nums {
    println!("{}", num);
}
```

Есть две причины для этого. Во-первых, это более ясно выражает то, что мы имеем в виду. Мы итерируем по элементам вектора, а не по индексам с последующей индексацией вектора. Во-вторых, эта версия является более эффективной: первая версия будет выполнять дополнительные проверки границ, потому что используется индексация, `nums[i]`. Во втором примере нет никаких проверок границ, поскольку мы получаем ссылки на каждый элемент вектора, одну за одной, по мере итерирования. Это очень распространенный прием работы с итераторами: мы можем игнорировать ненужные проверки границ, но все еще быть уверенными, что мы в безопасности.

Остается неясной еще одна деталь, как работает `println!`. На самом деле `num` имеет тип `&i32`. То есть, это ссылка на `i32`, не сам `i32`. `println!` выполняет разыменование переменной за нас, поэтому мы его и не наблюдаем. Этот код также прекрасно работает:

```
let nums = vec![1, 2, 3];

for num in &nums {
    println!("{}", *num);
}
```

Здесь мы явно разыменовываем `num`. Почему `&nums` выдает нам ссылки? Во-первых, потому что мы явно попросили его об этом с помощью `&`. Во-вторых, если он будет выдавать нам сами данные, то мы должны быть их владелецем, что подразумевает создание копии данных и выдачу этой копии нам. Со ссылками же мы просто заимствуем ссылку на данные, и поэтому будет выдана просто ссылка, без необходимости перемещать данные.

Теперь, когда мы установили, что зачастую диапазоны - это не то, что вы хотите, давайте поговорим о том, что же можно хотеть вместо диапазонов.

Есть три основных класса объектов, которые имеют отношение к данному вопросу: *итераторы*, *адаптеры итераторов* и *потребители*. Вот некоторые определения:

- *итераторы* выдают последовательность значений.
- *адаптеры итераторов* применяются к итератору, выдают новый итератор с другой выходной последовательностью.
- *потребители* применяются к итератору, выдающему некоторый конечный набор

значений.

Давайте сначала поговорим о потребителях, так как итераторы вы уже видели - диапазоны.

Потребители

Потребитель применяется к итератору, возвращая какое-то значение или значения. Наиболее распространенным потребителем является `collect()`. Этот код не компилируется, но он демонстрирует концепцию:

```
let one_to_one_hundred = (1..101).collect();
```

Как вы можете видеть, мы вызываем `collect()` для нашего итератора. `collect()` принимает столько значений, сколько выдаст итератор, и возвращает коллекцию результатов. Так почему же этот код не компилируется? Rust не может определить, какой тип у элементов, которые вы хотите собрать, и поэтому его необходимо указать явно. Вот версия, которая компилируется:

```
let one_to_one_hundred = (1..101).collect::<Vec<i32>>();
```

Если вы помните, то синтаксис `::<>` позволяет задать подсказку типа. Поэтому в приведенном примере мы указали, что хотим вектор целых чисел. Хотя не всегда бывает нужно задавать весь тип целиком. Использование символа `_` позволит вам задать частичную подсказку типа:

```
let one_to_one_hundred = (1..101).collect::<Vec<_>>();
```

Эта запись говорит компилятору Rust: "Пожалуйста, собери элементы в `Vec<T>`, а логический вывод о типе `T` сделай самостоятельно." По этой причине символ `_` иногда называют "заполнителем типа".

`collect()` является наиболее распространенным из потребителей, но есть и другие. Например `find()`:

```
let greater_than_forty_two = (0..100)
                             .find(|x| *x > 42);

match greater_than_forty_two {
    Some(_) => println!("We got some numbers!"),
    None => println!("No numbers found :("),
}
```

`find` принимает замыкание, которое обрабатывает ссылку на каждый элемент итератора. Замыкание возвращает `true`, если элемент является искомым элементом, и `false` в противном случае. Так как нам не всегда удается найти соответствующий элемент, то `find` возвращает `Option`, а не сам элемент.

Еще один важный потребитель - `fold`. Вот как он выглядит:

```
let sum = (1..4).fold(0, |sum, x| sum + x);
```

`fold()` - это потребитель, который схематически можно представить в виде: `fold(base, |accumulator, element| ...)`. Он принимает два аргумента: первый - это элемент, называемый *база*; второй - это замыкание, которое, в свою очередь, само принимает два аргумента: первый называется *аккумулятор*, а второй - *элемент*. На каждой итерации вызывается замыкание, результат выполнения которого становится значением аккумулятора на следующей итерации. На первой итерации значение аккумулятора равно базе.

Это немного сбивает с толку. Давайте рассмотрим значения всех элементов итератора:

base	accumulator	element	closure result
0	0	1	1
0	1	2	3
0	3	3	6

Мы вызвали `fold()` с этими аргументами:

```
.fold(0, |sum, x| sum + x);
```

Таким образом, `0` - это база, `sum` - это аккумулятор, а `x` - это элемент. На первой итерации мы устанавливаем `sum` равной `0`, а `x` становится первым элементом `nums`, `1`. Затем мы прибавляем `x` к `sum`, что дает нам `0 + 1 = 1`. На второй итерации это значение становится значением аккумулятора, `sum`, а элемент становится вторым элементом массива, `2`. `1 + 2 = 3`, результат этого выражения становится значением аккумулятора на последней итерации. На этой итерации, `x` становится последним элементом, `3`, а значение выражения `3 + 3 = 6` является конечным результатом для нашей суммы. `1 + 2 + 3 = 6` - это результат, который мы получили.

Вот так. `fold` может показаться немного странным, если вы используете его впервые, но когда вы освоите его, то будете использовать его повсеместно. `fold` подходит для случаев, когда у вас есть список элементов, а вам нужно получить один единственный результат.

Потребители имеют очень важное значение в связи с одним свойством итераторов, о котором мы еще не говорили: лень. Давайте поговорим немного об итераторах, и вы поймете, почему потребители имеют такое важное значение.

Итераторы

Как мы уже говорили ранее, итератор являются сущностью, для которой мы можем неоднократно вызвать метод `.next()`, в результате чего мы получим последовательность элементов. Для получения каждого следующего элемента нужно вызвать метод, а это означает, что итераторы *ленивы*, то есть им не требуется создавать все значения заранее. Этот код, например, на самом деле не генерирует номера `1-99`, а просто создает значение, представляющее эту последовательность:

```
let nums = 1..100;
```

В этом примере мы никак не использовали диапазон, поэтому он и не генерировал последовательность. Давайте добавим потребителя:

```
let nums = (1..100).collect::<Vec<i32>>();
```

Теперь `collect()` будет требовать, чтобы диапазон выдавал ему какие-нибудь цифры, поэтому он будет генерировать последовательность.

Диапазоны являются одним из двух основных типов итераторов, которые вы увидите. Другим типом является `iter()`. `iter()` может преобразовать вектор в простой итератор, который выдает вам каждый элемент по очереди:

```
let nums = vec![1, 2, 3];

for num in nums.iter() {
   println!("{}", num);
}
```

Эти два основных итератора должны служить вам хорошо. Есть и более продвинутые итераторы, в том числе и те, которые генерируют бесконечную последовательность.

Это все, что касается итераторов. Адаптеры итераторов - последнее понятие имеющее отношение к итераторам, о котором мы хотели бы рассказать. Давайте перейдем к нему!

Адаптеры итераторов

Адаптеры итераторов получают итератор и изменяют его каким-то образом, выдавая новый итератор. Самый простейший из них называется `map`:

```
(1..100).map(|x| x + 1);
```

`map` вызывается для итератора, и создает новый итератор, каждый элемент которого получается в результате вызова замыкания, в качестве аргумента которому передается ссылка на исходный элемент. Так что этот код выдаст нам числа `2-100`. Ну, почти! Если вы скомпилируете пример, этот код выдаст предупреждение:

```
warning: unused result which must be used: iterator adaptors are lazy and
         do nothing unless consumed, #[warn(unused_must_use)] on by default
(1..100).map(|x| x + 1);
 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Лень является причиной этого! Следующее замыкание никогда не будет выполнено. Пример ниже не напечатает ни одного значения:

```
(1..100).map(|x| println!("{}", x));
```

Если вы пытаетесь выполнить замыкание с целью манипулирования значениями итератора, то вместо этого просто используйте `for`.

Есть масса интересных адаптеров итераторов. `take(n)` вернет итератор, представляющий следующие `n` элементов исходного итератора. Обратите внимание, что это не оказывает никакого влияния на оригинальный итератор. Давайте попробуем применить его для

бесконечных итераторов, о которых мы упоминали прежде:

```
for i in (1..).step_by(5).take(5) {
    println!("{}", i);
}
```

Этот код напечатает

```
1
6
11
16
21
```

`filter()` представляет собой адаптер, который принимает замыкание в качестве аргумента. Это замыкание возвращает `true` или `false`. Новый итератор, полученный применением `filter()`, будет выдавать только те элементы, для которых замыкание возвращает `true`:

```
for i in (1..100).filter(|&x| x % 2 == 0) {
    println!("{}", i);
}
```

Этот пример будет печатать все четные числа от одного до ста. (Обратите внимание, что так как `filter` не потребляет элементы, которые выдаются во время итерации, то передается ссылка на каждый элемент, поэтому фильтрующий предикат использует шаблон `&x`, чтобы извлечь само целое число.)

Вы можете соединить все три понятия вместе: начать с итератора, адаптировать его несколько раз, а затем потребить результат. Например:

```
(1..)
    .filter(|&x| x % 2 == 0)
    .filter(|&x| x % 3 == 0)
    .take(5)
    .collect::<Vec<i32>>();
```

Этот код выдаст вектор, содержащий `6`, `12`, `18`, `24`, `30`.

Это просто небольшой обзор того, как итераторы, адаптеры итераторов и потребители могут помочь вам. Уже написано множество действительно полезных итераторов, а также вы можете написать свой собственный итератор. Итераторы обеспечивают безопасный и эффективный способ манипулирования всеми видами списков. Они немного непривычны сперва, но чем больше вы с ними работаете, тем больше они вас цепляют. Для получения полного списка различных итераторов, адаптеров и потребителей проверьте документацию модуля iter.

Многозадачность

Многозадачность и параллелизм являются невероятно важными проблемами в области компьютерной науки. А также это актуальная тема для современной индустрии. Компьютеры приобретают все больше и больше ядер, но многие программисты не готовы в полной мере использовать это.

Средства Rust для безопасной работы с памятью в полной мере применимы и при работе в многозадачной среде. Поэтому многозадачные программы на Rust должны безопасно работать с памятью, и не создавать состояния гонки данных. Система типов Rust способна справиться с этими задачами еще на этапе компиляции, благодаря мощным средствам, которые она предоставляет.

Прежде чем мы поговорим об особенностях многозадачности, которые идут с Rust, важно понять вот что: Rust является достаточно низкоуровневым, поэтому все это предусмотрено в стандартной библиотеке, а не в самом языке. Это означает, что если вам не нравится какой-то аспект в способе обработки многозадачности, который использует Rust, вы всегда можете реализовать альтернативный способ. mio представляет реальный пример этого принципа в действии.

Справочная информация: `Send` и `Sync`

О многозадачности рассуждать довольно трудно. В Rust, у нас есть система строгой, статической типизации, чтобы помочь нам делать выводы о нашем коде. В связи с этим Rust дает нам два типажа, помогающих нам разбираться в коде, который, по всей вероятности, является многозадачным.

`Send`

Первый типаж, о котором мы будем говорить, называется `Send`. Когда тип `T` реализует `Send`, это указывает компилятору, что право владения переменными этого типа можно безопасно перемещать между потоками.

Это важно для соблюдения некоторых ограничений. Например, если у нас есть канал, соединяющий два потока, и мы хотели бы иметь возможность отправлять некоторые данные по каналу из одного потока в другой. Следовательно, мы должны гарантировать, что для отправляемого типа данных реализован типаж `Send`.

И наоборот, если бы у нас была библиотека, упакованная с помощью FFI, который не является потокобезопасным, то нам не следовало бы реализовывать типаж `Send`, благодаря чему компилятор поможет нам добиться невозможности покинуть текущий поток.

`Sync`

Второй из этих типажей называется `Sync`. Когда тип `T` реализует `Sync`, это указывает компилятору, что переменные этого типа не имеют возможности использовать небезопасную память, когда они используются из нескольких потоков одновременно.

Например, совместное использование неизменяемых данных с помощью атомарного счетчика ссылок является потокобезопасным. Rust обеспечивает такой тип, `Arc<T>`, и он реализует `Sync`, так что при помощи этого типа можно безопасно обмениваться данными между потоками.

Эти два типажа позволяют использовать систему типов, чтобы обеспечить надежные гарантии о свойствах вашего кода в условиях многозадачности. Прежде чем мы продемонстрируем как, сначала мы должны узнать, как создать многозадачную программу в Rust!

Потоки

Стандартная библиотека Rust предоставляет библиотеку для потоков, которая позволяет запускать Rust код параллельно. Вот простой пример использования `std::thread`:

```
use std::thread;

fn main() {
    thread::spawn(|| {
        println!("Hello from a thread!");
    });
}
```

Метод `thread::spawn()` в качестве единственного аргумента принимает замыкание, которое выполняется в новом потоке. Он возвращает дескриптор потока, который может быть использован для ожидания завершения этого потока и извлечения его результата:

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        "Hello from a thread!"
    });

    println!("{}", handle.join().unwrap());
}
```

Многие языки имеют возможность выполнять потоки, но это дико опасно. Есть целые книги о том, как избежать ошибок, которые происходят от совместного использования изменяемого состояния. В Rust снова помогает система типов, которая предотвращает гонки данных на этапе компиляции. Давайте поговорим о том, как же на самом деле обеспечивается совместное использование чего-либо в условиях нескольких потоков.

Безопасное совместное использование изменяемого состояния

Благодаря системе типов Rust, у нас есть понятие, которое звучит как ложь: "безопасное совместное использование изменяемого состояния." Многие программисты считают, что совместное использование изменяемого состояния - это очень, очень плохо.

Кто-то однажды сказал это:

> Совместно используемое изменяемое состояние является корнем всех зол. Большинство языков пытаются решить эту проблему через 'изменяемое' часть, но Rust решает ее через 'совместно используемое' часть.

Та же самая система владения, которая помогает предотвратить неправильное использование указателей, также помогает исключить гонки данных, один из худших видов ошибок многозадачности.

В качестве примера приведем программу на Rust, которая входила бы в состояние гонки данных на многих языках. На Rust она не будет компилироваться:

```
use std::thread;

fn main() {
    let mut data = vec![1u32, 2, 3];

    for i in 0..3 {
        thread::spawn(move || {
            data[i] += 1;
        });
    }

    thread::sleep_ms(50);
}
```

Она выдает ошибку:

```
8:17 error: capture of moved value: `data`
        data[i] += 1;
        ^~~~
```

В данном случае мы знаем, что наш код *должен* быть безопасным, но Rust в этом не уверен. И, на самом деле, он не является безопасным: так как у нас есть ссылка на `data` в каждом потоке, а поток становится владельцем ссылки, то у нас есть три владельца! Это плохо. Мы можем исправить это с помощью типа `Arc<T>`, который является атомарным указателем со счетчиком ссылок. 'атомарный' означает, что им безопасно можно обмениваться между потоками.

`Arc<T>` предполагает наличие еще одного свойства у своего содержимого, чтобы гарантировать, что его можно безопасно использовать из нескольких потоков: он предполагает, что его содержимое реализует типаж `Sync`. В нашем случае мы также хотим, чтобы была возможность изменять значение содержимого. Нам нужен тип, который может обеспечить

возможность изменения своего содержимого лишь одним пользователем единовременно. Для этого мы можем использовать тип `Mutex<T>`. Вот вторая версия нашего кода. Она по-прежнему не работает, но по другой причине:

```
use std::thread;
use std::sync::Mutex;

fn main() {
    let mut data = Mutex::new(vec![1u32, 2, 3]);

    for i in 0..3 {
        let data = data.lock().unwrap();
        thread::spawn(move || {
            data[i] += 1;
        });
    }

    thread::sleep_ms(50);
}
```

Вот ошибка:

```
<anon>:9:9: 9:22 error: the trait `core::marker::Send` is not implemented for the type `std
::sync::mutex::MutexGuard<'_, collections::vec::Vec<u32>>` [E0277]
<anon>:11         thread::spawn(move || {
                  ^~~~~~~~~~~~~
<anon>:9:9: 9:22 note: `std::sync::mutex::MutexGuard<'_, collections::vec::Vec<u32>>` canno
t be sent between threads safely
<anon>:11         thread::spawn(move || {
                  ^~~~~~~~~~~~~
```

Вы можете видеть, что `Mutex` содержит метод `lock`, который имеет следующую сигнатуру:

```
fn lock(&self) -> LockResult<MutexGuard<T>>
```

Так как типаж `Send` не был реализован для `MutexGuard<T>`, мы не можем перемещать гварда через границы потоков, что и сказано в сообщении об ошибке.

Мы можем использовать `Arc<T>`, чтобы исправить это. Вот рабочая версия:

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

fn main() {
    let data = Arc::new(Mutex::new(vec![1u32, 2, 3]));

    for i in 0..3 {
        let data = data.clone();
        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            data[i] += 1;
        });
    }

    thread::sleep_ms(50);
}
```

Теперь мы вызываем `clone()` для нашего `Arc`, что увеличивает внутренний счетчик. Затем эта ручка перемещается в новый поток. Давайте более подробно рассмотрим тело потока:

```
thread::spawn(move || {
    let mut data = data.lock().unwrap();
    data[i] += 1;
});
```

Во-первых, мы вызываем метод `lock()`, который захватывает блокировку мьютекса. Так как вызов данного метода может потерпеть неудачу, то он возвращает `Result<T, E>`, но, поскольку это просто пример, мы используем `unwrap()`, чтобы получить ссылку на данные. Реальный код должен иметь более надежную обработку ошибок в такой ситуации. После этого мы свободно изменяем данные, так как у нас есть блокировка.

Под конец мы запускаем короткий таймер, ожидающий некоторое время, отведенное для выполнения потоков. Но такой вариант не является идеальным: возможно, мы выбрали разумное время ожидания но, скорее всего, мы будем ждать либо больше чем нужно, либо меньше чем необходимо, в зависимости от того, сколько на самом деле времени потребуется потокам, чтобы закончить вычисления.

Более точной альтернативой использованию таймера было бы использование одного из механизмов, предусмотренных стандартной библиотекой Rust для синхронизации потоков друг с другом. Давайте поговорим об одном из них: каналах.

Каналы

Вот версия нашего кода, которая использует каналы для синхронизации, вместо того чтобы ждать в течение определенного времени:

```
use std::sync::{Arc, Mutex};
use std::thread;
use std::sync::mpsc;

fn main() {
    let data = Arc::new(Mutex::new(0u32));

    let (tx, rx) = mpsc::channel();

    for _ in 0..10 {
        let (data, tx) = (data.clone(), tx.clone());

        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            *data += 1;

            tx.send(());
        });
    }

    for _ in 0..10 {
        rx.recv();
    }
}
```

Мы используем метод `mpsc::channel()`, чтобы построить новый канал. В этом примере мы в каждом из десяти потоков вызываем метод `send`, который передает по каналу простое значение `()`, а затем в главном потоке ждем, пока не будут приняты все десять значений.

В то время как по этому каналу послается просто общий сигнал, в общем случае мы можем отправить по каналу любые данные, которые реализуют типаж `Send`!

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    for _ in 0..10 {
        let tx = tx.clone();

        thread::spawn(move || {
            let answer = 42u32;

            tx.send(answer);
        });
    }

   rx.recv().ok().expect("Could not receive answer");
}
```

`u32` реализует `Send` потому что мы можем сделать копию. Итак, создается поток, в котором вычисляется ответ, а затем этот ответ с помощью метода `send()` передается обратно по каналу.

Паника

`panic!` аварийно завершает выполняемый в данный момент поток. Вы можете использовать потоки Rust, как простой механизм изоляции:

```
use std::thread;

let result = thread::spawn(move || {
    panic!("oops!");
}).join();

assert!(result.is_err());
```

Представленный в коде выше `Thread` возвращает `Result`, что позволяет нам проверить, произошло ли завершение потока в результате паники или нет.

Обработка ошибок

> Лучшие планы мышей и людей Часто идут вкривь и вкось
>
> "К полевой мыши", Роберт Бернс

Иногда что-то может пойти не так. Очень важно иметь план действий на тот случай, когда это неизбежно произойдет. Rust имеет богатую поддержку для обработки ошибок, которые могут (давайте будем честны: будут) происходить в ваших программах.

Есть два основных типа ошибок, которые могут возникнуть в ваших программах: неудачи (failures), и паники (panics). Давайте поговорим о разнице между ними, а затем обсудим, как справиться с каждой из них. Затем мы обсудим преобразование неудачи в панику.

Неудача и Паника

Rust использует два понятия, чтобы различать два типа ошибок: неудачи и паники. *Неудача* - это ошибка, после которой, в некоторых случаях, может быть восстановлена работоспособность программы. *Паника* - это ошибка, после которой работоспособность программы не может быть восстановлена.

Что мы подразумеваем под понятием "восстановить"? В большинстве случаев, возможность возникновения ошибок ожидаема. Например, рассмотрим функцию `parse`:

```
"5".parse();
```

Этот метод преобразует строку в другой тип. Но, так как это строка, вы не можете быть уверены, что преобразование пройдет успешно. Например, во что должно быть преобразовано следующее?

```
"hello5world".parse();
```

Это не будет работать. Итак, мы знаем, что эта функция будет работать только при некоторых входных данных. Это ожидаемое поведение. Мы называем такой тип ошибок *неудача*.

С другой стороны, иногда есть ошибки, которые являются неожиданными, или после которых мы не можем восстановить работоспособность программы. Классическим примером является `assert!`:

```
assert!(x == 5);
```

Мы используем `assert!`, чтобы заявить, что что-то не верно. Если это не верно, то что-то очень неправильно. Настолько неправильно, что мы не можем продолжать работу программы в текущем состоянии. Другой пример - использование макроса `unreachable!`

`()`:

```
enum Event {
    NewRelease,
}

fn probability(_: &Event) -> f64 {
    // real implementation would be more complex, of course
    0.95
}

fn descriptive_probability(event: Event) -> &'static str {
    match probability(&event) {
        1.00 => "certain",
        0.00 => "impossible",
        0.00 ... 0.25 => "very unlikely",
        0.25 ... 0.50 => "unlikely",
        0.50 ... 0.75 => "likely",
        0.75 ... 1.00 => "very likely",
    }
}

fn main() {
    std::io::println(descriptive_probability(NewRelease));
}
```

Этот пример выведет следующую ошибку:

```
error: non-exhaustive patterns: `_` not covered [E0004]
```

Хотя мы и знаем, что рассмотрели все возможные случаи, но Rust этого не знает. Он не знает, что вероятность находится между 0,0 и 1,0. Поэтому мы добавляем еще один случай:

```
use Event::NewRelease;

enum Event {
    NewRelease,
}

fn probability(_: &Event) -> f64 {
    // real implementation would be more complex, of course
    0.95
}

fn descriptive_probability(event: Event) -> &'static str {
    match probability(&event) {
        1.00 => "certain",
        0.00 => "impossible",
        0.00 ... 0.25 => "very unlikely",
        0.25 ... 0.50 => "unlikely",
        0.50 ... 0.75 => "likely",
        0.75 ... 1.00 => "very likely",
        _ => unreachable!()
    }
}

fn main() {
    println!("{}", descriptive_probability(NewRelease));
}
```

Мы никогда не должны попадать в случай `_`, поэтому мы используем макрос `unreachable!()`, чтобы сообщить об этом. `unreachable!()` выдает другой тип ошибки, нежели `Result`. Rust называет такой тип ошибок *паника*.

Обработка ошибок с помощью `Option` и `Result`

Простейший способ указать, что функция может потерпеть неудачу, это использовать тип `Option<T>`. Например, метод `find` для строк пытается найти шаблон в строке и возвращает `Option`:

```
let s = "foo";

assert_eq!(s.find('f'), Some(0));
assert_eq!(s.find('z'), None);
```

Такой вариант подходит для простейших случаев, однако он не дает нам достаточно информации в случае неудачи. Что же делать, если мы хотим знать причину, *почему* выполнение функции закончилось неудачей? Для этого мы можем использовать `Result<T, E>`. Выглядит это следующим образом:

```
enum Result<T, E> {
   Ok(T),
   Err(E)
}
```

Это перечисление определено непосредственно в Rust, так что вам не нужно определять его самостоятельно, чтобы использовать в своем коде. `Ok(T)` вариант представляет собой успех, а `Err(E)` вариант представляет собой неудачу. Рекомендуется возвращать `Result` вместо `Option` во всех случаях, кроме самых тривиальных.

Ниже приведен пример использования `Result`:

```
#[derive(Debug)]
enum Version { Version1, Version2 }

#[derive(Debug)]
enum ParseError { InvalidHeaderLength, InvalidVersion }

fn parse_version(header: &[u8]) -> Result<Version, ParseError> {
    if header.len() < 1 {
        return Err(ParseError::InvalidHeaderLength);
    }
    match header[0] {
        1 => Ok(Version::Version1),
        2 => Ok(Version::Version2),
        _ => Err(ParseError::InvalidVersion)
    }
}

let version = parse_version(&[1, 2, 3, 4]);
match version {
    Ok(v) => {
        println!("working with version: {:?}", v);
    }
    Err(e) => {
        println!("error parsing header: {:?}", e);
    }
}
```

В примере показана функция, которая использует перечисление `ParseError`, чтобы определить ошибки, которые могут возникнуть.

Использование `panic!`

В случае ошибки, которая является неожиданной, и после которой не может быть восстановлена работоспособность программы, макрос `panic!` будет вызывать панику. Это приведет к аварийному завершению текущего потока, и выдаст сообщение об ошибке:

```
panic!("boom");
```

выводит

```
thread '<main>' panicked at 'boom', hello.rs:2
```

когда вы запускаете пример.

Подобные ситуации встречаются довольно редко, поэтому используйте паники осмотрительно.

Преобразование неудачи в панику

В некоторых случаях, когда функция может потерпеть неудачу, нам может понадобиться рассматривать эту неудачу как панику. Например, `io::stdin().read_line(&mut buffer)` возвращает `Result<usize>`, если произошла ошибка при чтении строки. Это позволяет нам обработать такого рода ошибку и, возможно, восстановить работоспособность программы.

Если же нам не нужно обрабатывать ошибку, а нужно просто прервать программу, то мы можем использовать метод `unwrap()`:

```
io::stdin().read_line(&mut buffer).unwrap();
```

`unwrap()` вызывает `panic!`, если `Option` будет `None`. Такой подход просто говорит: "Дай мне значение, но если что-то пойдет не так, то просто аварийно заверши программу". Такой подход менее надежен, чем распаковка ошибки с последующей пыткой восстановления работоспособности программы, но он значительно короче. Хотя в некоторых ситуациях аварийное завершение программы более целесообразно.

Вот еще один способ распаковки ошибок, который немного лучше, чем `unwrap()`:

```
let mut buffer = String::new();
let input = io::stdin().read_line(&mut buffer)
                       .ok()
                       .expect("Failed to read line");
```

`ok()` преобразует `Result` в `Option`, а `expect()` делает то же самое, что и `unwrap()`, но принимает сообщение. Это сообщение передается в нижележащий вызов `panic!`, обеспечивая лучшее сообщение об ошибке, если ошибок в коде.

Использование `try!`

При написании кода, который вызывает множество функций, возвращающих тип `Result`, обработка ошибок может быть утомительной. Макрос `try!` скрывает некоторые шаблонные действия, передавая ошибки выше по стеку вызовов.

Он заменяет этот код:

```
use std::fs::File;
use std::io;
use std::io::prelude::*;

struct Info {
    name: String,
    age: i32,
    rating: i32,
}

fn write_info(info: &Info) -> io::Result<()> {
    let mut file = File::create("my_best_friends.txt").unwrap();

    if let Err(e) = writeln!(&mut file, "name: {}", info.name) {
        return Err(e)
    }
    if let Err(e) = writeln!(&mut file, "age: {}", info.age) {
        return Err(e)
    }
    if let Err(e) = writeln!(&mut file, "rating: {}", info.rating) {
        return Err(e)
    }

    return Ok(());
}
```

на этот код:

```
use std::fs::File;
use std::io;
use std::io::prelude::*;

struct Info {
    name: String,
    age: i32,
    rating: i32,
}

fn write_info(info: &Info) -> io::Result<()> {
    let mut file = try!(File::create("my_best_friends.txt"));

    try!(writeln!(&mut file, "name: {}", info.name));
    try!(writeln!(&mut file, "age: {}", info.age));
    try!(writeln!(&mut file, "rating: {}", info.rating));

    return Ok(());
}
```

Упаковка выражения в `try!` приведет к распакованному успешному (`Ok`) значению, если результат не `Err`, и в этом случае `Err` возвращается раньше от функции включения.

Стоит отметить, что вы можете использовать `try!` только из функции, которая возвращает `Result`, что означает, что вы не можете использовать `try!` внутри функции `main()`, потому что `main()` ничего не возвращают.

`try!` использует `From<Error>`, чтобы определять, что следует вернуть в случае ошибки.

Интерфейс внешних функций (Foreign Function Interface)

Введение

В данном руководстве мы будем использовать snappy, библиотеку для упаковки/распаковки данных, в качестве примера для написания привязок к внешнему коду. Rust в настоящее время не в состоянии делать вызовы напрямую из библиотеки C++, но snappy включает в себя интерфейс C (документирован в `snappy-c.h`).

Ниже приведен минимальный пример вызова внешней функции, который будет скомпилирован при условии, что библиотека snappy установлена:

```
extern crate libc;
use libc::size_t;

#[link(name = "snappy")]
extern {
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
}

fn main() {
    let x = unsafe { snappy_max_compressed_length(100) };
    println!("max compressed length of a 100 byte buffer: {}", x);
}
```

Блок `extern` содержит список сигнатур функций из внешней библиотеки, в данном случае для платформы C ABI. Чтобы указать, что программу нужно компоновать с библиотекой snappy, используется атрибут `#[link(...)]`. Благодаря этому, символы будут успешно разрешены.

Предполагается, что внешние функции могут быть небезопасными, поэтому их вызовы должны быть обёрнуты в блок `unsafe {}` как обещание компилятору, что все содержимое внутри этого блока в действительности безопасно. C библиотеки часто предоставляют интерфейсы, которые не являются поточно-безопасным. И почти любая функция, которая принимает в качестве аргумента указатель, не может быть валидной для всех возможных входных значений, поскольку указатель может быть висячим, и сырые указатели выходят за пределы безопасной модели памяти в Rust.

При объявлении типов аргументов для внешней функции, компилятор Rust не может проверить, является ли данное объявление корректным. Поэтому важно правильно указать тип привязываемой функции - иначе ошибка обнаружится только во время исполнения.

Блок `extern` может быть распространён на весь snappy API:

```
extern crate libc;
use libc::{c_int, size_t};

#[link(name = "snappy")]
extern {
    fn snappy_compress(input: *const u8,
                       input_length: size_t,
                       compressed: *mut u8,
                       compressed_length: *mut size_t) -> c_int;
    fn snappy_uncompress(compressed: *const u8,
                         compressed_length: size_t,
                         uncompressed: *mut u8,
                         uncompressed_length: *mut size_t) -> c_int;
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
    fn snappy_uncompressed_length(compressed: *const u8,
                                  compressed_length: size_t,
                                  result: *mut size_t) -> c_int;
    fn snappy_validate_compressed_buffer(compressed: *const u8,
                                         compressed_length: size_t) -> c_int;
}
```

Создание безопасного интерфейса

Сырой C API необходимо обернуть, чтобы обеспечить безопасность памяти, чтобы была возможность использовать концепции более высокого уровня, такие как векторы. Библиотека может выборочно открывать только безопасный, высокоуровневый интерфейс и скрывать небезопасные внутренние детали.

Обёртывание функций, которые принимают в качестве входных параметров буферы, включает в себя использование модуля `slice::raw` для управления векторами Rust как указателями на память. Векторы Rust представляют собой гарантированно непрерывный блок памяти. Длина - это количество элементов, которое в настоящее время содержится в векторе, а мощность - общее количество выделенной памяти в элементах. Длина меньше или равна мощности.

```
pub fn validate_compressed_buffer(src: &[u8]) -> bool {
    unsafe {
        snappy_validate_compressed_buffer(src.as_ptr(), src.len() as size_t) == 0
    }
}
```

Обертка `validate_compressed_buffer` использует блок `unsafe`, но это гарантирует, что ее вызов будет безопасен для всех входных данных, вследствие удаления модификатора `unsafe` из сигнатуры функции.

Функции `snappy_compress` и `snappy_uncompress` являются более сложными, так как должен быть выделен буфер для хранения выходных данных.

Функция `snappy_max_compressed_length` может быть использована для выделения вектора максимальной мощности, требуемой для хранения упакованных выходных данных. Затем этот вектор может быть передан в функцию `snappy_compress` в качестве

выходного параметра. Выходной параметр передается также, чтобы получить истинную длину после сжатия, для установки длины.

```
pub fn compress(src: &[u8]) -> Vec<u8> {
    unsafe {
        let srclen = src.len() as size_t;
        let psrc = src.as_ptr();

        let mut dstlen = snappy_max_compressed_length(srclen);
        let mut dst = Vec::with_capacity(dstlen as usize);
        let pdst = dst.as_mut_ptr();

        snappy_compress(psrc, srclen, pdst, &mut dstlen);
        dst.set_len(dstlen as usize);
        dst
    }
}
```

Распаковка аналогична, потому что snappy хранит размер неупакованных данных как часть формата сжатия, и `snappy_uncompressed_length` будет возвращать точный размер необходимого буфера.

```
pub fn uncompress(src: &[u8]) -> Option<Vec<u8>> {
    unsafe {
        let srclen = src.len() as size_t;
        let psrc = src.as_ptr();

        let mut dstlen: size_t = 0;
        snappy_uncompressed_length(psrc, srclen, &mut dstlen);

        let mut dst = Vec::with_capacity(dstlen as usize);
        let pdst = dst.as_mut_ptr();

        if snappy_uncompress(psrc, srclen, pdst, &mut dstlen) == 0 {
            dst.set_len(dstlen as usize);
            Some(dst)
        } else {
            None // SNAPPY_INVALID_INPUT
        }
    }
}
```

Для справки, примеры, используемые здесь, также доступны в библиотеке на GitHub.

Деструкторы

Внешние библиотеки часто передают право собственности на ресурсы в вызовающий код. Когда это происходит, мы должны использовать деструкторы Rust, чтобы обеспечить безопасность и гарантировать освобождение этих ресурсов (особенно в случае паники).

Чтобы получить более подробную информацию о деструкторах, смотрите Drop trait.

Обратные вызовы Rust функций из C кода

Некоторые внешние библиотеки требуют использование обратных вызовов для передачи вызывающей стороне отчета о своем текущем состоянии или промежуточных данных. Во внешнюю библиотеку можно передавать функции, которые были определены в Rust. При создании функции обратного вызова, которую можно вызывать из C кода, необходимо указать для нее спецификатор `extern`, за которым следует правильное соглашение о вызове.

Затем функция обратного вызова может быть передана в библиотеку C через регистрационный вызов, и уже затем может быть вызывана оттуда.

Простой пример:

Rust код:

```
extern fn callback(a: i32) {
    println!("I'm called from C with value {0}", a);
}

#[link(name = "extlib")]
extern {
   fn register_callback(cb: extern fn(i32)) -> i32;
   fn trigger_callback();
}

fn main() {
    unsafe {
        register_callback(callback);
        trigger_callback(); // Triggers the callback
    }
}
```

C код:

```
typedef void (*rust_callback)(int32_t);
rust_callback cb;

int32_t register_callback(rust_callback callback) {
    cb = callback;
    return 1;
}

void trigger_callback() {
  cb(7); // Will call callback(7) in Rust
}
```

В этом примере функция `main()` в Rust вызовет функцию `trigger_callback()` в C, которая, в свою очередь, выполнит обратный вызов функции `callback()` в Rust.

Обратные вызовы, адресованные объектам Rust (Targeting callbacks to Rust

objects)

Предыдущий пример показал, как глобальная функция может быть вызвана из C кода. Однако зачастую желательно, чтобы обратный вызов был адресован конкретному объекту в Rust. Это может быть объект, который представляет собой обертку для соответствующего объекта C.

Такое поведение может быть достигнуто путем передачи небезопасного указателя на объект в библиотеку C. После чего библиотека C сможет включать указатель на объект Rust при обратном вызове. Это позволит получить небезопасный доступ к ссылке на объект Rust в обратном вызове.

Rust код:

```
#[repr(C)]
struct RustObject {
    a: i32,
    // other members
}

extern "C" fn callback(target: *mut RustObject, a: i32) {
    println!("I'm called from C with value {0}", a);
    unsafe {
        // Update the value in RustObject with the value received from the callback
        (*target).a = a;
    }
}

#[link(name = "extlib")]
extern {
   fn register_callback(target: *mut RustObject,
                        cb: extern fn(*mut RustObject, i32)) -> i32;
   fn trigger_callback();
}

fn main() {
    // Create the object that will be referenced in the callback
    let mut rust_object = Box::new(RustObject { a: 5 });

    unsafe {
        register_callback(&mut *rust_object, callback);
        trigger_callback();
    }
}
```

C код:

```
typedef void (*rust_callback)(void*, int32_t);
void* cb_target;
rust_callback cb;

int32_t register_callback(void* callback_target, rust_callback callback) {
    cb_target = callback_target;
    cb = callback;
    return 1;
}

void trigger_callback() {
  cb(cb_target, 7); // Will call callback(&rustObject, 7) in Rust
}
```

Асинхронные обратные вызовы

В ранее приведенных примерах обратные вызовы выполняются как непосредственная реакция на вызов функции внешней C библиотеки. Для выполнения обратного вызова контроль над текущим потоком переключался из Rust в C, а затем снова в Rust, но, в конце концов, обратный вызов выполнялся в том же потоке, из которого была вызвана функция, инициировавшая обратный вызов.

Все становится более сложным, когда внешняя библиотека порождает свои собственные потоки и осуществляет обратные вызовы из них. В этих случаях доступ к структурам данных Rust внутри обратных вызовов особенно небезопасен, и поэтому должны быть использованы соответствующие механизмы синхронизации. Помимо классических механизмов синхронизации, таких как мьютексы, в Rust есть еще одна возможность: использовать каналы (в `std::comm` (`std::sync::mpsc::channel`)), чтобы направить данные из потока C, который выполнял обратный вызов, в поток Rust.

Если асинхронный обратный вызов адресован конкретному объекту в адресном пространстве Rust, то необходимо потребовать, чтобы обратные вызовы больше не выполнялись библиотекой C после удаления этого Rust объекта. Это может быть достигнуто путем отмены регистрации обратного вызова в деструкторе объекта и проектирования библиотеки таким образом, чтобы гарантировалось, что после отмены регистрации обратного вызова он больше не будет выполняться.

Компоновка

Атрибут `link` для блоков `extern` предоставляет основные инструкции rustc относительно того, как он будет компоновать нативные библиотеки. На данный момент есть две общепринятых формы записи атрибута `link`:

- `#[link(name = "foo")]`
- `#[link(name = "foo", kind = "bar")]`

В обоих этих случаях `foo` - это имя нативной библиотеки, с которой мы компонуем. Во втором случае `bar` - это тип нативной библиотеки, с которой компонует компилятор. В настоящее время известны три типа нативных библиотек:

- Динамические - `#[link(name = "readline")]`
- Статические - `#[link(name = "my_build_dependency", kind = "static")]`
- Фреймворки - `#[link(name = "CoreFoundation", kind = "framework")]`

Обратите внимание, что фреймворки доступны только для OSX.

Различные значения `kind` нужны, чтобы определить, как компоновать нативную библиотеку. С точки зрения компоновки, компилятор Rust создает две разновидности артефактов: промежуточный (rlib/staticlib) и конечный (dylib/binary). Зависимости от нативных динамических библиотек и фреймворков распространяются дальше, пока не дойдут до конечного артефакта, а от статических библиотек - нет.

Вот несколько примеров того, как эта модель может быть использована:

- Нативная зависимость при сборке. Иногда написанный на Rust код необходимо состыковать с некоторым кодом на C/C++, но распранение C/C++ кода в формате библиотеки вызывает дополнительные трудности. В этом случае, код будут упакован в `libfoo.a`, а затем контейнер Rust должен будет объявить зависимость с помощью `#[link(name = "foo", kind = "static")]`.

  Независимо от типа выхода (промежуточный или конечный) для контейнера, нативная статическая библиотека будет включена на выходе, что означает, что нет необходимости в распранении этой нативной статической библиотеки отдельно.

- Нормальная динамическая зависимость. Общие системные библиотеки (такие, как `readline`) доступны на большом количестве систем, и часто можно найти статическую копию этих библиотек. Когда такая зависимость включена в контейнер Rust, промежуточные артефакты (например, rlibs) не будут компоноваться с библиотекой, но когда rlib включается в состав конечного артефакта (например, исполняемый файл), нативная библиотека будет прикомпонована.

На OSX, фреймворки ведут себя с той же семантикой, что и динамические библиотеки.

Небезопасные блоки

Некоторые операции, такие как разыменование небезопасных указателей или вызов функций, которые были отмечены как небезопасные, разрешено использовать только внутри небезопасных блоков. Небезопасных блоки изолируют опасносные ситуации и дают гарантии компилятору, что опасности не вытекут за пределы блока.

Небезопасные функции же, наоборот, делают сильный акцент на этом. Небезопасная функция записывается в виде:

```
unsafe fn kaboom(ptr: *const i32) -> i32 { *ptr }
```

Эта функция может быть вызвана только из блока `unsafe` или из другой `unsafe` функции.

Доступ к внешним глобальным переменным

Внешние API довольно часто экспортируют глобальные переменные, которые могут быть использованы, например, для отслеживание глобального состояния. Для того, чтобы получить доступ к этим переменным, нужно объявить их в блоке `extern`, используя ключевое слово `static`:

```
extern crate libc;

#[link(name = "readline")]
extern {
    static rl_readline_version: libc::c_int;
}

fn main() {
    println!("You have readline version {} installed.",
             rl_readline_version as i32);
}
```

Кроме того, возможно вам потребуется изменить глобальное состояние, предоставленное внешним интерфейсом. Для этого при объявлении статических переменных может быть добавлен модификатор `mut`, чтобы была возможность изменять их.

```
extern crate libc;

use std::ffi::CString;
use std::ptr;

#[link(name = "readline")]
extern {
    static mut rl_prompt: *const libc::c_char;
}

fn main() {
    let prompt = CString::new("[my-awesome-shell] $").unwrap();
    unsafe {
        rl_prompt = prompt.as_ptr();

        println!("{:?}", rl_prompt);

        rl_prompt = ptr::null();
    }
}
```

Обратите внимание, что любое взаимодействие с `static mut` небезопасно, как чтение, так и запись. Работа с изменяемым глобальным состоянием требует значительно большей осторожности.

Соглашение о вызове внешних функций

Большинство внешнего кода предоставляет C ABI (Двоичный Интерфейс Приложений). И Rust при вызове внешних функций по умолчанию использует C соглашение о вызове для данной платформы. Но некоторые внешние функции, в первую очередь Windows API, используют другое соглашение о вызове. Rust обеспечивает способ указать компилятору, какое именно соглашение использовать:

```
extern crate libc;

#[cfg(all(target_os = "win32", target_arch = "x86"))]
#[link(name = "kernel32")]
#[allow(non_snake_case)]
extern "stdcall" {
    fn SetEnvironmentVariableA(n: *const u8, v: *const u8) -> libc::c_int;
}
```

Это указание относится ко всему блоку `extern`. Список поддерживаемых ABI ограничивается следующими:

- `stdcall`
- `aapcs`
- `cdecl`
- `fastcall`
- `Rust`
- `rust-intrinsic`
- `system`
- `C`
- `win64`

Большинство ABI в этом списке не требуют пояснений, но ABI `system` может показаться немного странным. Он выбирает такое ABI, которое подходит (уместно) для взаимодействия с целевыми библиотеками. Например, на платформе win32 с архитектурой x86, это означает, что будет использован `stdcall` ABI. Однако, на windows x86_64 используется `C` соглашение о вызовах, поэтому в этом случае будет использован `C` ABI. Это означает, что в нашем предыдущем примере мы могли бы использовать `extern "system" { ... }`, чтобы определить блок для всех windows систем, а не только для x86.

Взаимодействие с внешним кодом

Rust гарантирует, что схема размещения элементов для `struct` совместима с представлением платформы в C только в том случае, если к ней применяется атрибут `#[repr(C)]`. Атрибут `#[repr(C, packed)]` может быть использован для схемы размещения элементов структуры без выравнивания. Атрибут `#[repr(C)]` также может быть применен и к перечислениям.

Боксы с правом владения в Rust (`Box<T>`) используют ненулевые (non-nullable) указатели, которые указывают на содержащийся в них объект, как ручки. Тем не менее, они не должны быть созданы вручную, так как они находятся под управлением внутренних средств выделения памяти. Ссылки можно без риска считать ненулевыми указателями непосредственно на тип. Однако нарушение правил проверки заимствования или изменяемости не гарантирует безопасность, поэтому предпочитают использовать сырые указатели (`*`), если это необходимо, так как компилятор не может сделать так много предположений о них.

Векторы и строки совместно используют одну и ту же базовую схему размещения памяти и утилиты, доступные в модулях `vec` и `str`, для работы с C API. Тем не менее, строки не завершаются нулевым байтом, `\0`. Если вам нужна строка, завершающаяся нулевым байтом для совместимости с C, вы должны использовать тип `CString` из модуля `std::ffi`.

Стандартная библиотека включает в себя псевдонимы типов и определения функций для стандартной библиотеки C в модуле `libc`, и Rust компонует `libc` и `libm` по умолчанию.

Оптимизация указателей, допускающих нулевое значение

(The "nullable pointer optimization")

Некоторые типы по определению не могут быть `null`. Это ссылки (`&T`, `&mut T`), упаковки (`Box<T>`), указатели на функции (`extern "abi" fn()`). При взаимодействии же с C часто используются указатели, которые могут быть `null`. Как особый случай - обобщенный `enum`, который содержит ровно два варианта, один из которых не содержит данных, а другой содержит одно поле. Такое использование перечисления имеет право на "оптимизацию указателя, допускающего нулевое значение". Когда создан экземпляр такого перечисления с одним из не-обнуляемых типов, то он представляет собой ненулевой указатель для варианта, содержащего данные, и нулевой - для варианта без данных. Таким образом, `Option<extern "C" fn(c_int) -> c_int>` - это представление указателя, на функцию, допускающего нулевое значение, и совместимого с C ABI.

Вызов Rust кода из C кода

Вы можете скомпилировать Rust код таким образом, чтобы он мог быть вызван из C кода. Это довольно легко, но требует нескольких вещей:

```
#[no_mangle]
pub extern fn hello_rust() -> *const u8 {
    "Hello, world!\0".as_ptr()
}
```

`extern` указывает, что эта функцию придерживается C соглашения о вызове, как описано выше в разделе "Соглашение о вызове внешних функций". Атрибут `no_mangle` выключает модификацию имени, применяемую в Rust, для того чтобы было легче компоновать.

Каналы сборок

Проект Rust использует концепцию под названием "каналы сборок" для управления сборками. Важно понять этот процесс, чтобы выбрать, какую версию Rust ваш проект должен использовать.

Обзор

Есть три канала сборок Rust:

- Ночной (Nightly)
- Бета (Beta)
- Стабильный (Stable)

Новые ночные сборки создаются раз в день. Каждые шесть недель последняя ночная сборка переводится в канал "Бета". С этого момента она будет получать только патчи для исправления серьёзных ошибок. Шесть недель спустя бета сборка переводится в канал "Стабильный" и становится очередной стабильной сборкой `1.x`.

Этот процесс происходит параллельно. Так, каждые шесть недель, в один и тот же день, ночная сборка превращается в бета сборку, а бета сборка превращается в стабильную сборку. То есть, одновременно: стабильная сборка получит версию `1.x`, бета сборка получит версию `1.(x + 1)-beta`, а ночная сборка станет первой версией `1.(x + 2)-nightly`.

Выбор версии

Вообще говоря, если у вас нет особых причин, вы должны использовать канал стабильных сборок. Эти сборки предназначены для широкой аудитории.

Однако, в зависимости от ваших интересов к Rust, вы можете вместо этого выбрать ночную сборку. Основной компромисс заключается в следующем: при выборе канала ночных сборок, вы можете использовать неустойчивые, новые возможности Rust. Тем не менее, нестабильные возможности могут быть изменены, и поэтому любая новая ночная сборка может сломать ваш код. Если же вы выберете стабильную сборку, то не сможете использовать экспериментальные возможности, но следующий релиз Rust не вызовет существенных проблем с критическими изменениями.

Помощь экосистеме с помощью непрерывной интеграции

Что же касается бета канала? Мы призываем всех пользователей Rust, которые используют канал стабильных сборок, также протестировать работу с использованием бета канала в их системах непрерывной интеграции. Это поможет предупредить команду в случае

возникновения неожиданных регрессий.

Кроме того, тестирование работы с использованием ночного канала может выявить регрессии даже раньше, а поэтому, если вас не затруднит создание трех сборок, мы будем признательны тестированию работы с использованием всех трех каналов.

Синтаксис и семантика

Этот раздел разбит на небольшие главы, каждая из которых раскрывает определённую концепцию Rust.

Если вы хотите от и до изучить Rust, то чтение этого раздела по порядку будет верным путём сделать это.

Эти главы также сформированы как ссылки на каждое из понятий, так что если при чтении другого туториала вам будет что-то непонятно, вы всегда сможете найти пояснения об этом здесь.

Связывание переменных

Любая реальная не ’Hello World’ программа на Rust использует *связывание переменных*. Это выглядит так:

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

Все операции производимые ниже будут происходить в функции `main()`, так как каждый раз вставлять в примеры `fn main() {` немного утомляет. Убедитесь, что примеры, приведённые в этом разделе, вы вводите в функцию `main()`, иначе можете получить ошибку при компиляции.

Во многих языках программирования это называется *переменная*. Но у связывания переменных в Rust есть пара трюков в рукаве. В левой стороне выражения `let` располагается не просто имя переменной, а ‘шаблон’. Это значит, что мы можем делать вещи вроде этой:

```
let (x, y) = (1, 2);
```

После завершения этого выражения `x` будет единицей, а `y` - двойкой. Шаблоны очень мощны, и для них выделена отдельная глава. Но на данный момент нам не нужны эти фичи, так что мы просто будем помнить о них и пойдём дальше.

Rust - статически типизированный язык программирования, и значит мы должны указывать типы и они будут проверяться во время сборки. Так почему же наш первый пример собрался? В Rust'е есть такая вещь, как *вывод типов*. Если Rust самостоятельно может понять какой тип у переменной, то он не требует указывать его.

Тем не менее, мы можем указать желаемый тип. Он следует после двоеточия (`:`):

```
let x: i32 = 5;
```

Если бы я попросил вас прочитать это вслух, то вы бы сказали "`x` связан с типом `int` и имеет значение `пять`".

В этом случае мы указали, что `x` у нас будет 32-битным целым числом со знаком. В Rust есть и другие целочисленные типы. Они начинаются с `i` для целых чисел со знаком и с `u` для целых чисел без знака. Размер целых чисел может составлять 8, 16, 32 и 64 бита.

В дальнейших примерах мы будем указывать тип в комментариях. Это будет выглядеть вот так:

```
fn main() {
    let x = 5; // x: i32
}
```

Обратите внимание на сходство между этим комментарием и синтаксисом, который вы используете с `let`. Включение такого типа комментариев не является идеоматическим для Rust, но иногда мы будем включать их для того, чтобы помочь вам понять, какие типы будут выведены Rust.

По умолчанию, связывание *неизменяемо*. Этот код не скомпилируется:

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

И вы получите ошибку:

```
error: re-assignment of immutable variable `x`
     x = 10;
     ^~~~~~
```

Если вы хотите чтобы связывание было изменяемым, вы можете использовать модификатор `mut`:

```
let mut x = 5; // mut x: i32
x = 10;
```

Может показаться, что нет ни одной причины делать связывание неизменяемым по умолчанию, но вспомните на чём в первую очередь сфокусирован Rust: на безопасности. Если вы случайно забыли указать `mut` и изменили связывание, компилятор заметит это, и сообщит вам, что вы изменили то, что возможно не собирались менять. Если бы по умолчанию связывание было изменяемым, то в приведённой выше ситуации компилятор не сможет вам помочь. Если вы намерены изменить значение переменной, то просто добавьте `mut`.

Есть и другие весомые аргументы в пользу правила: по возможности, избегать изменяемых состояний, но это выходит за рамки данного руководства. В общем, зачастую вы можете избежать явных изменений, и это предпочтительнее в Rust. Тем не менее, иногда без изменения значения просто не обойтись, так что это не запрещено.

Вернёмся к связыванию. Связывании переменных в Rust имеет ещё одно отличие от других языков: оно требует инициализации перед использованием.

Давайте приступим к рассмотрению вышесказаного. Измените ваш файл `src/main.rs` так, что бы он выглядел следующим образом:

```
fn main() {
    let x: i32;

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

Используйте команду `cargo build` в командной строке чтобы собрать проект. Вы должны получить предупреждение, но программа будет работать и будет выводить строку "Hello, world!":

```
   Compiling hello_world v0.0.1 (file:///home/you/projects/hello_world)
src/main.rs:2:9: 2:10 warning: unused variable: `x`, #[warn(unused_variable)]
   on by default
src/main.rs:2     let x: i32;
                      ^
```

Rust предупредит нас о том, что мы никогда не используем связанную переменную, но от того, что мы её не используем, не будет никакого вреда, поэтому нарушения в этом нет. Однако, всё изменится, если мы попробуем использовать `x`. Сделаем это. Измените вашу программу так, что бы она выглядела следующим образом:

```
fn main() {
    let x: i32;

    println!("x имеет значение {}", x);
}
```

И попробуйте собрать проект. Вы получите ошибку:

```
$ cargo build
   Compiling hello_world v0.0.1 (file:///home/you/projects/hello_world)
src/main.rs:4:39: 4:40 error: use of possibly uninitialized variable: `x`
src/main.rs:4     println!("x имеет значение {}", x);
                                                   ^
note: in expansion of format_args!
<std macros>:2:23: 2:77 note: expansion site
<std macros>:1:1: 3:2 note: in expansion of println!
src/main.rs:4:5: 4:42 note: expansion site
error: aborting due to previous error
Could not compile `hello_world`.
```

Rust не позволит использовать неинициализированную переменную. Далее, поговорим о `{}`, которые мы добавили в `println!`.

Если вы добавите две фигурные скобки (`{}`, иногда называемые "усами"...) в вашу печатаемую строку, Rust истолкует это как просьбу вставки некоторого значения. *Строковая интерполяция* - это термин в информатике, который обозначает "вставить посреди строки". Мы добавили запятую, и затем `x`, что бы указать, что мы хотим вставить `x` в строку. Запятая используется для разделения параметров, если в функцию или макрос передаётся больше одного параметра.

При вставке переменной в строку, Rust проверит её тип и попытается отобразить осмысленное значение. Если вы хотите указать формат более детально, то можете ознакомиться с доступными опциями форматирования строк (англ.). На данный момент мы будем вставлять как есть: целые числа не очень сложны для печати.

Функции

Каждая программа на Rust имеет по крайней мере одну функцию - `main`:

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

Это простейшее объявление функции. Как мы упоминали ранее, ключевое слово `fn` объявляет функцию. За ним следует её имя, пустые круглые скобки (поскольку эта функция не принимает аргументов), а затем тело функции, заключённое в фигурные скобки. Вот функция `foo`:

```
fn foo() {
}
```

Итак, что насчёт аргументов, принимаемых функцией? Вот функция, печатающая число:

```
fn print_number(x: i32) {
    println!("x равен: {}", x);
}
```

Вот полная программа, использующая функцию `print_number`:

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

fn print_number(x: i32) {
    println!("x равен: {}", x);
}
```

Как видите, аргументы функций похожи на операторы `let`: вы можете объявить тип аргумента после двоеточия.

Вот полная программа, которая складывает два числа и печатает их:

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

fn print_sum(x: i32, y: i32) {
    println!("сумма чисел: {}", x + y);
}
```

Аргументы разделяются запятой - и при вызове функции, и при её объявлении.

В отличие от `let`, вы *должны* объявлять типы аргументов функции. Этот код не скомпилируется:

```
fn print_sum(x, y) {
    println!("сумма чисел: {}", x + y);
}
```

Вы увидите такую ошибку:

```
expected one of `!`, `:`, or `@`, found `)`
fn print_number(x, y) {
```

Это осознанное решение при проектировании языка. Бесспорно, вывод типов во всей программе возможен. Однако даже в Haskell считается хорошим стилем явно документировать типы функций, хотя в этом языке и возможен полный вывод типов. Мы считаем, что принудительное объявление типов функций при сохранении локального вывода типов - это хороший компромисс.

Как насчёт возвращаемого значения? Вот функция, которая прибавляет один к целому:

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

Функции в Rust возвращают ровно одно значение, тип которого объявляется после "стрелки". "Стрелка" представляет собой дефис (`-`), за которым следует знак "больше" (`>`). Заметьте, что в функции выше нет точки с запятой. Если бы мы добавили её:

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

Мы бы получили ошибку:

```
error: not all control paths return a value
fn add_one(x: i32) -> i32 {
     x + 1;
}

help: consider removing this semicolon:
     x + 1;
          ^
```

Здесь показаны две интересные особенности Rust. Во-первых, это выражение-ориентированный язык, и во-вторых, смысл точки с запятой отличается от смысла аналогичного символа в других ’фигурные скобки и точка с запятой’-основанных языках. Эти две особенности связаны.

Выражения и операторы

Rust - в первую очередь выражение-ориентированный язык. Есть только два типа операторов, а всё остальное является выражением.

А в чём же разница? Выражение возвращает значение, в то время как оператор - нет. Вот почему мы получаем здесь ‘not all control paths return a value‘: оператор `x + 1;` не возвращает значение. Есть два типа операторов в Rust: ‘операторы объявления‘ и ‘операторов выражение‘. Все остальное - выражения. Давайте поговорим об операторах объявления в первую очередь.

Первым типом операторов в Rust является *оператор объявления* - связывание. В некоторых языках связывание переменных может быть записано как выражение, а не только как оператор. Например, в Ruby:

```
x = y = 5
```

Однако, в Rust использование `let` для связывания *не является* выражением. Следующий код вызовет ошибку компиляции:

```
let x = (let y = 5); // expected identifier, found keyword `let`
```

Здесь компилятор сообщил нам, что ожидал увидеть выражение, но `let` является оператором, а не выражением.

Обратите внимание, что присвоение уже связанной переменной (например: `y = 5`) является выражением, но его значение не особенно полезно. В отличие от других языков, где присвоение вычисляется в присваиваемое значение (например: `5` из предыдущего примера), в Rust значением присвоения является пустой кортеж `()`.

```
let mut y = 5;

let x = (y = 6);  // x будет присвоено значение `()`, а не `6`
```

Вторым типом операторов в Rust является *оператор выражения*. Его цель - превратить любое выражение в оператор. В практическом плане, грамматика Rust ожидает что за операторами будут идти другие операторы. Это означает, что вы используете точку с запятой для разделения выражений друг от друга. Rust выглядит как многие другие языки, которые требуют использовать точку с запятой в конце каждой строки и вы увидите её в конце почти каждой строки кода на Rust.

Из-за чего мы говорим "почти"? Вы это уже видели в этом примере:

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

Наша функция объявлена, как возвращающая `i32`. Но если в конце есть точка с запятой, то вместо этого функция вернёт `()`. Компилятор Rust обрабатывает эту ситуацию и предлагает удалить точку с запятой.

Досрочный возврат из функции

А что насчёт досрочного возврата из функции? У нас есть для этого ключевое слово `return`:

```
fn foo(x: i32) -> i32 {
    return x;

    // следующий код никогда не запустится!
    x + 1
}
```

`return` можно написать в последней строке тела функции, но это считается плохим стилем:

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

Если вы никогда не работали с языком, в котором операторы являются выражениями, предыдущее определение без `return` может показаться вам странным. Но со временем вы просто перестанете замечать это.

Расходящиеся функции

Для функций, которые не возвращают управление ("расходящихся"), в Rust есть специальный синтаксис:

```
fn diverges() -> ! {
    panic!("Эта функция не возвращает управление!");
}
```

`panic!` - это макрос, как и `println!()`, который мы встречали ранее. В отличие от `println!()`, `panic!()` вызывает остановку текущего потока исполнения с заданным сообщением.

Поскольку эта функция вызывает остановку исполнения, она никогда не вернёт управление. Поэтому тип её возвращаемого значения обозначается знаком `!` и читается как "расходится". Значение расходящейся функции может быть использовано как значение любого типа:

```
let x: i32 = diverges();
let x: String = diverges();
```

Простые типы

Язык Rust имеет несколько типов, которые считаются ‘простыми‘ (‘примитивными‘). Это означает, что они встроены в язык. Rust структурирован таким образом, что стандартная библиотека также предоставляет ряд полезных типов, построенных на базе этих простых типов, но это самые простые.

Логический тип (`bool`)

Rust имеет встроенный логический тип, называемый `bool`. Он может принимать два значения, `true` и `false`:

```
let x = true;

let y: bool = false;
```

Логические типы часто используются в конструкции `if`.

Вы можете найти больше информации о логических типах (`bool`) в документации к стандартной библиотеке (англ.).

Символы (`char`)

Тип `char` представляет собой одиночное скалярное значение Unicode. Вы можете создать `char` с помощью одинарных кавычек: (`'`)

```
let x = 'x';
let two_hearts = '💕';
```

Это означает, что в отличие от некоторых других языков, `char` в Rust представлен не одним байтом, а четырьмя.

Вы можете найти больше информации о символах (`char`) в документации к стандартной библиотеке (англ.).

Числовые типы

Rust имеет целый ряд числовых типов, разделённых на несколько категорий: знаковые и беззнаковые, фиксированного и переменного размера, числа с плавающей точкой и целые числа.

Эти типы состоят из двух частей: категория и размер. Например, `u16` представляет собой тип без знака с размером в шестнадцать бит. Чем большим количеством бит представлен тип, тем большее число мы можем задать.

Если для числового литерала не указан тип, то он будет выведен по умолчанию:

```
let x = 42; // x имеет тип i32

let y = 1.0; // y имеет тип f64
```

Ниже представлен список различных числовых типов, со ссылками на их документацию в стандартной библиотеке:

- i8
- i16
- i32
- i64
- u8
- u16
- u32
- u64
- isize
- usize
- f32
- f64

Давайте пройдёмся по их категориям:

Знаковые и беззнаковые

Целые типы бывают двух видов: знаковые и беззнаковые. Чтобы понять разницу, давайте рассмотрим число с размером в четыре бита. Знаковые четырёхбитные числа, позволяют хранить значения от `-8` до `+7`. Знаковые цифры используют представление ’дополнение до двух’. Беззнаковые четырёхбитные числа, ввиду того что не нужно хранить отрицательные значения, позволяют хранить значения от `0` до `+15`.

Беззнаковые типы используют `u` для своей категории, а знаковые типы используют `i`. `i` означает ‘integer’. Так `u8` представляет собой число без знака с размером восемь бит, а `i8` представляет собой число со знаком с размером восемь бит.

Типы фиксированного размера

Типы с фиксированным размером соответственно имеют фиксированное количество бит в своём представлении. Допустимыми размерами являются `8`, `16`, `32`, `64`. Таким образом, `u32` представляет собой целое число без знака с размером 32 бита, а `i64` - целое число со знаком с размером 64 бита.

Типы переменного размера

Rust также предоставляет типы, размер которых зависит от размера указателя на целевой машине. Эти типы имеют ‘size’ в названии в качестве признака размера, и могут быть знаковыми или беззнаковыми. Таким образом, существует два типа: `isize` и `usize`.

С плавающей точкой

В Rust также есть два типа с плавающей точкой: `f32` и `f64`. Они соответствуют IEEE-754 числам с плавающей точкой одинарной и двойной точности соответственно.

Массивы

В Rust, как и во многих других языках программирования, есть типы-списки для представления последовательностей неких вещей. Самый простой из них - это *массив*, то есть список элементов одного и того же типа, имеющий фиксированный размер. Массивы неизменяемы по умолчанию.

```
let a = [1, 2, 3]; // a: [i32; 3]
let mut m = [1, 2, 3]; // m: [i32; 3]
```

Массивы имеют тип `[T; N]`. О значении `T` мы поговорим позже, когда будем рассматривать дженерики. `N` - это константа времени компиляции, представляющая собой длину массива.

Для инициализации всех элементов массива одним и тем же значением есть специальный синтаксис. В этом примере каждый элемент `a` будет инициализирован значением `0`:

```
let a = [0; 20]; // a: [i32; 20]
```

Вы можете получить число элементов массива `a` с помощью метода `a.len()`:

```
let a = [1, 2, 3];

println!("Число элементов в a: {}", a.len());
```

Вы можете получить определённый элемент массива с помощью *индекса*:

```
let names = ["Graydon", "Brian", "Niko"]; // names: [&str; 3]

println!("Второе имя: {}", names[1]);
```

Индексы нумеруются с нуля, как и в большинстве языков программирования, поэтому мы получаем первое имя с помощью `names[0]`, а второе - с помощью `names[1]`. Пример выше печатает `Второе имя: Brian`. Если вы попытаетесь использовать индекс, который не входит в массив, вы получите ошибку: при доступе к массивам происходит проверка границ во время исполнения программы. Такая ошибочная попытка доступа - источник многих проблем в других языках системного программирования.

Вы можете найти больше информации о массивах (`array`) в документации к стандартной библиотеке (англ.).

Срезы

Срез - это ссылка на (или "проекция" в) другую структуру данных. Они полезны, когда нужно обеспечить безопасный, эффективный доступ к части массива без копирования. Например, возможно вам нужно сослаться на единственную строку файла, считанного в память. Из-за своей ссылочной природы, срезы создаются не напрямую, а из существующих переменных. У срезов есть длина, они могут быть изменяемы или нет, и во многих случаях они ведут себя как массивы:

```
let a = [0, 1, 2, 3, 4];
let middle = &a[1..4]; // Срез `a`: только элементы 1, 2, и 3
let complete = &a[..]; // Срез, содержащий все элементы массива `a`
```

Срезы имеют тип `&[T]`. О значении `T` мы поговорим позже, когда будем рассматривать дженерики.

Вы можете найти больше информации о срезах (`slice`) в документации к стандартной библиотеке (англ.).

str

Тип `str` в Rust является наиболее простым типом строк. Это безразмерный тип, поэтому сам по себе он не очень полезен, но он становится полезным при использовании ссылки, `&str`. Таким образом, мы просто остановимся на этом.

Вы можете найти больше информации о строках (`str`) в документации к стандартной библиотеке (англ.).

Кортежи

Кортеж - это упорядоченный список фиксированного размера. Вроде такого:

```
let x = (1, "привет");
```

Этот кортеж из двух элементов создан с помощью скобок и запятой между элементами. Вот тот же код, но с аннотациями типов:

```
let x: (i32, &str) = (1, "привет");
```

Как вы можете видеть, тип кортежа выглядит как сам кортеж, но места элементов занимают типы. Внимательные читатели также отметят, что кортежи гетерогенны: в этом кортеже одновременно хранятся значения типов `i32` и `&str`. В языках системного программирования строки немного более сложны, чем в других языках. Пока вы можете читать `&str` как *срез строки*. Мы вскоре узнаем об этом больше.

Можно присваивать один кортеж другому, если они содержат значения одинаковых типов и имеют одинаковую арность. Арность кортежей одинакова, когда их длина совпадает.

```
let mut x = (1, 2); // x: (i32, i32)
let y = (2, 3); // y: (i32, i32)

x = y;
```

Доступ к полям кортежа можно получить с помощью *деконструирующего let*. Вот пример:

```
let (x, y, z) = (1, 2, 3);

println!("x это {}", x);
```

Помните, я говорил, что левая часть оператора `let` более полезна, чем просто присваивание имени? О том, что мы здесь увидели, я и говорил. Мы можем написать слева от `let` образец, и, если он совпадает со значением справа, произойдёт присваивание имён сразу нескольким значениям. В данном случае, `let` "деконструирует" или "разбивает" кортеж, и присваивает его части трём именам.

Это очень удобный шаблон программирования, и мы ещё не раз увидим его.

Вы можете устранить неоднозначность трактовки для кортежа, состоящего из одного элемента, и значения в скобках с помощью запятой:

```
(0,); // одноэлементный кортеж
(0); // ноль в круглых скобках
```

Индексация кортежей

Вы также можете получить доступ к полям кортежа с помощью индексации:

```
let tuple = (1, 2, 3);

let x = tuple.0;
let y = tuple.1;
let z = tuple.2;

println!("x is {}", x);
```

Как и в случае индексации массивов, индексы начинаются с нуля, но здесь, в отличие от массивов, используется `.`, а не `[]`.

Вы можете найти больше информации о кортежах (`tuple`) в документации к стандартной библиотеке (англ.).

Функции

Функции тоже имеют тип! Это выглядит следующим образом:

```
fn foo(x: i32) -> i32 { x }

let x: fn(i32) -> i32 = foo;
```

В данном примере `x` - это ‘указатель на функцию‘, которая принимает в качестве аргумента `i32` и возвращает `i32`.

Комментарии

Теперь, когда у нас есть несколько функций, неплохо бы узнать о комментариях. Комментарии - это заметки, которые вы оставляете для других программистов, чтобы помочь объяснить некоторые вещи в вашем коде. Компилятор в основном игнорирует их.

В Rust есть два вида комментариев: *строчные комментарии* и *doc-комментарии*.

```
// Строчные комментарии - это всё что угодно после '//' и до конца строки.

let x = 5; // это тоже строчный комментарий.

// Если у вас длинное объяснение для чего-либо, вы можете расположить строчные
// комментарии один за другим. Поместите пробел между '//' и вашим комментарием,
// так как это более читаемо.
```

Другое применение комментария - это doc-комментарий. Doc-комментарий использует `///` вместо `//`, и поддерживает Markdown-разметку внутри:

```
/// Прибавляем единицу к заданному числу.
///
/// # Примеры
///
/// ```
/// let five = 5;
///
/// assert_eq!(6, add_one(5));
/// ```
fn add_one(x: i32) -> i32 {
    x + 1
}
```

При написании doc-комментария, очень полезно добавлять разделы для любых аргументов, возвращаемых значений и приводить некоторые примеры использования. Вы заметили, что здесь мы использовали новый макрос: `assert_eq!`. Он сравнивает два значения и вызывает `panic!`, если они не равны. Для документации такие примеры очень полезны. Так же есть и другой макрос, `assert!`, который вызывает `panic!` когда значение равно `false`.

Вы можете использовать `rustdoc` для генерации HTML- документации из этих doc-комментариев, а так же запуска кода из примеров как тестов.

Конструкция `if`

`if` в Rust'е не сильно сложен и больше похож на `if` в динамически типизированных языках, чем на более традиционный из системных. Давайте поговорим о нём, чтобы вы поняли некоторые его нюансы.

`if` является одной из форм более общего понятия, именуемого *ветвлением*. Это название произошло от ветвей деревьев: конечный результат зависит от того, какой из нескольких вариантов будет выбран.

`if` содержит одно условие, в зависимости от которого будет выполняться одна из двух ветвей:

```
let x = 5;

if x == 5 {
    println!("x равняется пяти!");
}
```

При изменении значения `x` на какое-либо другое, эта строчка не будет выведена на экран. Если подробнее, то когда условие будет иметь значение `true`, следующий после него блок кода, выполнится. В противном случае - нет.

Бывает нужно что-то выполнить, если условие не выполнится (выражение будет иметь значение false). В таком случае можно использовать `else`:

```
let x = 5;

if x == 5 {
    println!("x равняется пяти!");
} else {
    println!("x это не пять :(");
}
```

Когда необходимо больше одного выбора, то можно использовать `else if`:

```
let x = 5;

if x == 5 {
    println!("x равняется пяти!");
} else if x == 6 {
    println!("x это шесть!");
} else {
    println!("x это ни пять, ни шесть :(");
}
```

Всё это довольно прозаично. Однако, вы так же можете сделать такую штуку:

```
let x = 5;

let y = if x == 5 {
    10
} else {
    15
}; // y: i32
```

Которую мы можем (и должны) записать примерно следующим образом:

```
let x = 5;

let y = if x == 5 { 10 } else { 15 }; // y: i32
```

Это работает, потому что `if` является выражением. Его значением является значение последнего выражения из выбранной ветви. `if` без `else` всегда возвращает `()` в качестве значения.

Циклы `for`

Цикл `for` нужен для повторения блока кода определённого количества раз. Циклы `for` в Rust работают немного иначе, чем в других языках программирования. Например в Си-подобном языке цикл `for` выглядит так:

```
for (x = 0; x < 10; x++) {
    printf( "%d\n", x );
}
```

Однако, этот код в Rust будет выглядеть следующим образом:

```
for x in 0..10 {
    println!("{}", x); // x: i32
}
```

Можно представить цикл более абстрактно:

```
for переменная in выражение {
    тело_цикла
}
```

Выражение - это итератор. Их мы будем рассматривать позже в этом руководстве. Итератор возвращает серию элементов, где каждый элемент будет являться одной итерацией цикла. Значение этого элемента затем присваивается `переменной`, которая будет доступна в теле цикла. После окончания тела цикла, берётся следующее значение итератора и снова выполняется тело цикла. Когда в итераторе закончатся значения, то цикл `for` завершается.

В нашем примере, `0..10` - это выражение, которое задаёт начальное и конечное значение, и возвращает итератор. Обратите внимание, что конечное значение не включается в него. В нашем примере будут напечатаны числа от `0` до `9`, но не будет напечатано `10`.

В Rust нет цикла `for` как в Си-подобном синтаксисе, т.к. управлять каждым элементом цикла вручную сложно и это может привести к ошибкам даже у опытных программистов на Си.

Циклы `while`

Цикл `while` это ещё один вид конструкции цикла в Rust. Выглядит он так:

```
let mut x = 5; // mut x: u32
let mut done = false; // mut done: bool

while !done {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 {
        done = true;
    }
}
```

Он применяется, если неизвестно сколько раз нужно выполнить тело цикла, чтобы получить результат. При каждой итерации цикла проверяется условие, если оно истинно, то запускается следующая итерация, иначе цикл `while` завершается.

Если вам нужен бесконечный цикл, то можете сделать условие всегда истинным:

```
while true {
```

Однако, для такого случая в Rust имеется ключевое слово `loop`:

```
loop {
```

В Rust анализатор потока управления обрабатывает конструкцию `loop` иначе, чем `while true`, хотя для нас это одно и тоже. На данном этапе изучения Rust нам не важно знать в чем именно различие между этими конструкциями, но если вы хотите сделать бесконечный цикл, то используйте конструкцию `loop`, т.к. компилятор сможет транслировать ваш код в более эффективный и безопасный машинный код.

Управление итерацией цикла

Давайте посмотрим на цикл `while` снова:

```
let mut x = 5;
let mut done = false;

while !done {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 {
        done = true;
    }
}
```

В этом примере имеется изменяемая переменная `done` типа boolean, которая используется в условии для выхода из цикла. В Rust имеются два ключевых слова, которые помогут нам работать с итерациями цикла: `break` и `continue`.

Мы можем переписать цикл более хорошим способом с помощью `break`, чтобы избавиться от переменной `done`:

```
let mut x = 5;

loop {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 { break; }
}
```

Теперь мы используем бесконечный цикл `loop` и `break` для выхода из цикла.

`continue` похож на `break`, но вместо выхода из цикла переходит к следующей итерации. Следующий пример отобразит только нечётные числа:

```
for x in 0..10 {
    if x % 2 == 0 { continue; }

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

Использовать `continue` и `break` можно как в циклах `while`, так и в циклах `for`.

Владение

Эта глава является одной из трёх, описывающих систему владения ресурсами Rust. Эта система представляет собой наиболее уникальную и привлекательную особенность Rust, о которой разработчики должны иметь полное представление. Владение - это то, как Rust достигает своей главной цели - безопасности памяти. Система владения включает в себя несколько различных концепций, каждая из которых рассматривается в своей собственной главе:

- владение, её вы сейчас читаете
- заимствование, и ассоциированная с ним фича "ссылки"
- время жизни, расширение концепции заимствования

Эти три главы взаимосвязаны, и их порядок важен. Вы должны будете освоить все три главы, чтобы полностью понять систему владения.

Мета

Прежде чем перейти к деталям, отметим два важных нюанса о системе владения.

Rust сфокусирован на безопасности и скорости. Это достигается за счёт "абстракций с нулевой стоимостью" ("zero-cost abstractions"), а значит, в Rust стоимость абстракций должна быть настолько минимальной, насколько это возможно, без ущерба для работоспособности. Система владения ресурсами - это яркий пример абстракции с нулевой стоимостью. Весь анализ, о котором мы будем говорить в этом руководстве, выполняется *во время компиляции*. Вы не платите хоть сколько-нибудь времени исполнения за какую-либо из фич.

Тем не менее, эта система всё же имеет определённую стоимость: кривая обучения. Многие новые пользователи Rust испытывает то, что мы называем "борьба с проверкой заимствования", когда компилятор Rust отказывается компилировать программу, которая по мнению автора является абсолютно правильной. Это часто происходит потому, что мысленное представление программиста о том, как должно работать владение, не совпадает с реальными правилами, которыми оперирует Rust. Вы, наверное, поначалу также будете испытывать подобные трудности. Однако существует и хорошая новость: более опытные разработчики на Rust сообщают, что чем больше они работают с правилами системы владения, тем меньше они борются с проверкой заимствования.

Имея это в виду, давайте перейдём к изучению системы владения.

Владение

Связанные имена имеют одну особенность в Rust: они "обладают правом собственности" на то, с чем они связаны. Это означает, что, когда связывание выходит за пределы области видимости, ресурс, с которым оно связано, будут освобождён. Например:

```
fn foo() {
    let v = vec![1, 2, 3];
}
```

Когда `v` входит в область видимости, создаётся новый `Vec<T>`. В данном случае вектор также выделяет из кучи пространство для трёх элементов. Когда `v` выходит из области видимости, в конце `foo()`, Rust очищает все, связанное с вектором, даже динамически выделенную память. Это происходит детерминировано, в конце области видимости.

Семантика перемещения

Хотя тут есть некоторые тонкости: Rust гарантирует, что существует *ровно одна* привязка к какому-либо ресурсу. Например, если у нас есть вектор, то мы можем присвоить этот вектор другой привязке: Важным аспектом [владения][ownership] является "семантика перемещения". Семантика перемещения контролирует, как и когда право собственности перемещается между привязками (связываниями).

```
let v = vec![1, 2, 3];

let v2 = v;
```

Но, если после этого мы попытаемся использовать `v`, то получим ошибку:

```
let v = vec![1, 2, 3];

let v2 = v;

println!("v[0] = {}", v[0]);
```

Ошибка выглядит следующим образом:

```
error: use of moved value: `v`
println!("v[0] = {}", v[0]);
                      ^
```

То же самое произойдёт, если мы определим функцию, которая принимает право владения, и попробуем использовать что-то после того как мы передали это что-то в качестве аргумента в эту функцию:

```
fn take(v: Vec<i32>) {
    // что будет здесь не очень важно
}

let v = vec![1, 2, 3];

take(v);

println!("v[0] = {}", v[0]);
```

Та же самая ошибка: "use of moved value" ("используется перемещённое значение"). Когда мы передаём право владения куда-то ещё, мы как бы говорим, что мы "перемещаем" то, на что ссылаемся. При этом не нужно указывать какую-либо специальную аннотацию, Rust делает это по умолчанию.

Детали

Причина, по которой мы не можем использовать привязку после того как мы её переместили, трудно заметная, но очень важная. Когда мы пишем код вроде этого:

```
let v = vec![1, 2, 3];

let v2 = v;
```

Первая строка создаёт некоторые данные для вектора в стеке, `v`. Данные самого вектора, однако, сохраняются в куче, и поэтому стековые данные содержат указатель на данные в куче. Когда мы перемещаем `v` в `v2`, то создаётся копия стековых данных, для `v2`. Что будет означать, что два указателя ссылаются на расположенный в куче вектор. Такое поведение могло бы быть проблемой: оно нарушало бы гарантии безопасности Rust, привнося гонки данных. Поэтому Rust запрещает использование `v` после того как мы выполнили его перемещение.

Важно также отметить, что оптимизация может удалить фактическую (точную) копию байтов, в зависимости от обстоятельств. Так что это может быть не так уж неэффективно, как выглядит на первый взгляд.

Типы, реализующие типаж `Copy`

Мы установили, что, как только право собственности передаётся другой привязке, то вы больше не можете использовать оригинальную привязку. Тем не менее, существует типаж, который изменяет такое поведение, и он называется `Copy`. Мы ещё не обсуждали типажи, но пока вы можете думать о них как об аннотациях к конкретному типу, которые придают дополнительное поведение. Например:

```
let v = 1;

let v2 = v;

println!("v = {}", v);
```

В этом примере `v` связан с типом `i32`. Этот тип реализует типаж `Copy`. Это означает, что когда мы присваиваем привязку `v` привязке `v2`, будет создана копия данных, как и при перемещении. Но, в отличие от перемещения, мы можем использовать `v` в дальнейшем. Это происходит потому, что в `i32` нет указателей на данные в каком-либо другом месте. При таком копировании создаётся полная копия.

Мы будем обсуждать, как сделать свои собственные типы, реализующие типаж `Copy` в разделе Типажи.

Больше, чем владение

Конечно, если бы нам нужно было вернуть право владения обратно из функции, то мы бы написали:

```
fn foo(v: Vec<i32>) -> Vec<i32> {
    // делаем что-либо с v

    // возвращаем право владения
    v
}
```

Это сильно утомляет. Функция становится тем хуже, чем больше прав владения она хочет забрать себе:

```
fn foo(v1: Vec<i32>, v2: Vec<i32>) -> (Vec<i32>, Vec<i32>, i32) {
    // делаем что-нибудь с v1 и v2

    // возвращаем право владения и результат нашей функции
    (v1, v2, 42)
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let (v1, v2, answer) = foo(v1, v2);
```

Брр! Возвращаемый тип, возвращающая строка, и вызов функции получается намного более сложным.

К счастью, Rust предлагает такую фичу, как заимствование, которая помогает нам решить эту проблему. Это тема следующего раздела!

Ссылки и Заимствование

Эта глава является одной из трех, описывающих систему владения ресурсами Rust. Эта система представляет собой наиболее уникальную и привлекательную фичу Rust, о которой разработчики на Rust должны иметь полное представление. Владение - это то, как Rust достигает своей главной цели - безопасности памяти. Система владения включает в себя несколько различных концепций, каждая из которых рассматривается в своей собственной главе:

- владение, ключевая концепция
- заимствование, ее вы сейчас читаете
- время жизни, расширение концепции заимствования

Эти три главы взаимосвязаны, и их порядок важен. Вы должны будете освоить все три главы, чтобы полностью понять систему владения.

Мета

Прежде чем перейти к деталям, отметитм два важных нюанса о системе владения.

Rust сфокусирован на безопасности и скорости. Это достигается за счет ‘абстракций с нулевой стоимостью‘ (‘zero-cost abstractions‘), что означает, что в Rust стоимость абстракций должна быть настолько минимальной, насколько это возможно, без ущерба для работоспособности. Система владения ресурсами - это яркий пример абстракции с нулевой стоимостью. Весь анализ, о котором мы будем говорить в этом руководстве, выполняется *во время компиляции*. Вы не платите хоть сколько-нибудь времени рантайма за какую-либо из фич.

Тем не менее, эта система все же имеет определенную стоимость: кривая обучения. Многие новые пользователи Rust испытыват то, что мы называем ‘борьба с проверкой заимствования‘, когда компилятор Rust отказывается компилировать программу, которая по мнению автора является абсолютно правильной. Это часто происходит потому, что мысленное представление программиста о том, как должно работать владение, не совпадает с реальными правилами, которыми оперирует Rust. Вы, наверное, также будете испытывать подобные трудности поначалу. Однако существует и хорошая новость: более опытные разработчики на Rust сообщают, что чем больше они работают с правилами системы владения, тем меньше они борются с проверкой заимствования.

Имея это в виду, давайте перейдем к изучению систему владения.

Заимствование

В конце главы Владение, у нас была скверная функция, которая выглядела так:

```
fn foo(v1: Vec<i32>, v2: Vec<i32>) -> (Vec<i32>, Vec<i32>, i32) {
    // do stuff with v1 and v2

    // hand back ownership, and the result of our function
    (v1, v2, 42)
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let (v1, v2, answer) = foo(v1, v2);
```

Однако, этот код не является идиоматическим с точки зрения Rust, так как он не использует заимствование. Вот первый шаг:

```
fn foo(v1: &Vec<i32>, v2: &Vec<i32>) -> i32 {
    // do stuff with v1 and v2

    // return the answer
    42
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let answer = foo(&v1, &v2);

// we can use v1 and v2 here!
```

Вместо того, чтобы принимать `Vec<i32>` в качестве аргументов, мы будем принимать ссылки: `&Vec<i32>`. И вместо передачи `v1` и `v2` напрямую, мы будем передавать `&v1` и `&v2`. Мы называем тип `&T` ‘ссылка‘, и вместо того, чтобы забирать право владения ресурсом, она его заимствует. Привязки, которые заимствуют что-то, не освобождают ресурс, когда они выходят из области видимости. Это означает, что, после вызова `foo()`, мы снова можем использовать наши оригинальные привязки.

Ссылки являются неизменяемыми, как и привязки. Это означает, что внутри `foo()`, векторы не могут быть изменены:

```
fn foo(v: &Vec<i32>) {
     v.push(5);
}

let v = vec![];

foo(&v);
```

выдает ошибку:

```
error: cannot borrow immutable borrowed content `*v` as mutable
v.push(5);
^
```

Добавление значения изменяет вектор, и поэтому компилятор не позволил нам это сделать.

&mut ссылки

Вот второй вид ссылок: `&mut T`. Это ‘изменяемая ссылка‘, которая позволяет изменять ресурс, который вы заимствуете. Например:

```
let mut x = 5;
{
    let y = &mut x;
    *y += 1;
}
println!("{}", x);
```

Этот код напечатает `6`. Мы создали `y`, изменяемую ссылку на `x`, а затем добавили единицу к значению, на которое указывает `y`. Следует отметить, что `x` также должна быть помечена как `mut`, если бы этого не было, то мы не могли бы взять изменяемый заем неизменяемого значения.

Во всем остальном изменяемые ссылки (`&mut`) такие же, как и неизменяемые (`&`). Однако, существует большая разница между этими двумя концепциями, и тем, как они взаимодействуют. Вы можете сказать, что в приведенном выше примере есть что-то подозрительное, потому что нам зачем-то понадобилась дополнительная область видимости, созданная с помощью `{` и `}`. Если мы уберем эти скобки, то получим ошибку:

```
error: cannot borrow `x` as immutable because it is also borrowed as mutable
    println!("{}", x);
                   ^
note: previous borrow of `x` occurs here; the mutable borrow prevents
subsequent moves, borrows, or modification of `x` until the borrow ends
        let y = &mut x;
                     ^
note: previous borrow ends here
fn main() {

}
^
```

Как оказалось, есть правила.

Правила

Вот правила о заимствовании в Rust:

Во-первых, область видимости любого заема должна находиться в пределах области видимости владельца. Во-вторых, одновременно у вас может быть только один из двух перечисленных ниже видов заимствования, но не оба сразу:

- одна или более неизменяемых ссылок (`&T`) на ресурс,

- ровно одна изменяемая ссылка (`&mut T`) на ресурс.

Вы можете заметить, что они очень похожи, хотя и не совсем то же самое, относительно определения состояния гонки данных:

> Состояние 'гонки данных' возникает, когда два или более указателей имеют доступ к одной и той же области памяти одновременно, и когда по крайней мере один из них производит запись, а операции не синхронизированы.

Что касается неизменяемых ссылок, то вы можете иметь их столько, сколько хотите, так как ни одна из них не производит запись. Если же вы производите запись, и вам нужно два или больше указателей на одну и ту же область памяти, то вы можете иметь только один `&mut` одновременно. Это то как Rust предотвращает возникновение состояния гонки данных во время компиляции: мы получим ошибки, если мы нарушим эти правила.

Имея это в виду, давайте рассмотрим наш пример еще раз.

Мысли об областях видимости (Thinking in scopes)

Вот код:

```
let mut x = 5;
let y = &mut x;

*y += 1;

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

Этот код выдает нам такую ошибку:

```
error: cannot borrow `x` as immutable because it is also borrowed as mutable
    println!("{}", x);
                   ^
```

Это потому, что мы нарушили правила: у нас есть изменяемая ссылка `&mut T`, указывающая на `x`, и поэтому мы не можем создать какую-либо `&T`. Одно из двух. Примечание подсказывает как следует рассматривать эту проблему:

```
note: previous borrow ends here
fn main() {

}
^
```

Другими словами, изменяемая ссылка сохраняется до конца нашего примера. А мы хотим, чтобы изменяемый заем заканчивался *до* того, как мы пытаемся вызвать `println!` и создать неизменяемый заем. В Rust заимствование привязано к области видимости, в которой оно является действительным. И эти области видимости выглядят следующим образом:

```
let mut x = 5;

let y = &mut x;    // -+ &mut borrow of x starts here
                   //  |
*y += 1;           //  |
                   //  |
println!("{}", x); // -+ - try to borrow x here
                   // -+ &mut borrow of x ends here
```

Конфликт областей видимости: мы не можем создать `&x` до тех пор, пока `y` находится в области видимости.

Поэтому, когда мы добавляем фигурные скобки:

```
let mut x = 5;

{
    let y = &mut x; // -+ &mut borrow starts here
    *y += 1;        //  |
}                   // -+ ... and ends here

println!("{}", x);  // <- try to borrow x here
```

Никаких проблем нет. Наш изменяемый заем выходит из области видимости до создания неизменяемого. Но область видимости является ключом к определению того, как долго длится заем.

Проблемы, которые предотвращает заимствование

Почему нужны эти ограничивающие правила? Ну, как мы уже отметили, эти правила предотвращают гонки данных. Какие виды проблем могут привести к состоянию гонки данных? Вот некоторые из них.

Недействительный итератор

Одним из примеров является 'недействительный итератор', такое может произойти, когда вы пытаетесь изменить коллекцию, которую в данный момент итерируете. Проверка заимствования Rust предотвращает это:

```
let mut v = vec![1, 2, 3];

for i in &v {
    println!("{}", i);
}
```

Этот код печатает от одного до трех. Так как мы итерируем по вектору, мы получаем лишь ссылки на элементы. И сам `v` заимствован как неизменяемый, что означает, что мы не можем изменить его в процессе итерирования:

```
let mut v = vec![1, 2, 3];

for i in &v {
    println!("{}", i);
    v.push(34);
}
```

Вот ошибка:

```
error: cannot borrow `v` as mutable because it is also borrowed as immutable
    v.push(34);
    ^
note: previous borrow of `v` occurs here; the immutable borrow prevents
subsequent moves or mutable borrows of `v` until the borrow ends
for i in &v {
          ^
note: previous borrow ends here
for i in &v {
    println!(“{}”, i);
    v.push(34);
}
^
```

Мы не можем изменить `v`, потому что он уже заимствован в цикле.

Использование после освобождения (use after free)

Ссылки должны жить так же долго, как и ресурс, на который они ссылаются. Rust проверяет области видимоси ваших ссылок, чтобы удостовериться, что это правда.

Если Rust не будет проверять это свойство, то мы можем случайно использовать ссылку, которая будет недействительна. Например:

```
let y: &i32;
{
    let x = 5;
    y = &x;
}

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

Мы получим следующую ошибку:

```
error: `x` does not live long enough
    y = &x;
         ^
note: reference must be valid for the block suffix following statement 0 at
2:16...
let y: &i32;
{
    let x = 5;
    y = &x;
}

note: ...but borrowed value is only valid for the block suffix following
statement 0 at 4:18
    let x = 5;
    y = &x;
}
```

Другими словами, `y` действителен только для той области видимости, где существует `x`. Как только `x` выходит из области видимости, ссылка на него становится недействительной. Таким образом, ошибка сообщает, что заем ‘does not live long enough‘ (‘не живет достаточно долго‘), потому что он не является действительным столько времени, сколько требуется.

Время жизни

Эта глава является одной из трех, описывающих систему владения ресурсами Rust. Эта система представляет собой наиболее уникальную и привлекательную фичу Rust, о которой разработчики на Rust должны иметь полное представление. Владение - это то, как Rust достигает своей главной цели - безопасности памяти. Система владения включает в себя несколько различных концепций, каждая из которых рассматривается в своей собственной главе:

- владение, ключевая концепция
- заимствование, и ассоциированная с ним фича ‘ссылки’
- время жизни, ее вы сейчас читаете

Эти три главы взаимосвязаны, и их порядок важен. Вы должны будете освоить все три главы, чтобы полностью понять систему владения.

Мета

Прежде чем перейти к деталям, отметитм два важных нюанса о системе владения.

Rust сфокусирован на безопасности и скорости. Это достигается за счет ‘абстракций с нулевой стоимостью‘ (‘zero-cost abstractions‘), что означает, что в Rust стоимость абстракций должна быть настолько минимальной, насколько это возможно, без ущерба для работоспособности. Система владения ресурсами - это яркий пример абстракции с нулевой стоимостью. Весь анализ, о котором мы будем говорить в этом руководстве, выполняется *во время компиляции*. Вы не платите хоть сколько-нибудь времени рантайма за какую-либо из фич.

Тем не менее, эта система все же имеет определенную стоимость: кривая обучения. Многие новые пользователи Rust испытыват то, что мы называем ‘борьба с проверкой заимствования‘, когда компилятор Rust отказывается компилировать программу, которая по мнению автора является абсолютно правильной. Это часто происходит потому, что мысленное представление программиста о том, как должно работать владение, не совпадает с реальными правилами, которыми оперирует Rust. Вы, наверное, также будете испытывать подобные трудности поначалу. Однако существует и хорошая новость: более опытные разработчики на Rust сообщают, что чем больше они работают с правилами системы владения, тем меньше они борются с проверкой заимствования.

Имея это в виду, давайте перейдем к изучению систему владения.

Время жизни

Одалживание ссылки на ресурс, которым кто-то владеет может быть довольно сложным. Например, представьте себе следующую последовательность операций:

- Я получаю ручку на какой-то ресурс.
- Я одалживаю вам ссылку на это ресурс.
- Я решаю, что ресурс мне больше не требуется, и освобождаю его, в то время как у вас все еще есть на него ссылка.
- Вы решаете использовать этот ресурс.

Ой-ой! Ваша ссылка указывает на недопустимый ресурс. Это называется висячий указатель или ‘использование после освобождения‘, когда ресурсом является память.

Чтобы исправить это, мы должны убедиться, что четвертый шаг никогда не произойдет после третьего. Система владения в Rust делает это через концепцию под названием время жизни, которая описывает область видимости, на протяжении которой ссылка будет действительна.

Когда у нас есть функция, которая принимает ссылку в качестве аргумента, мы можем явно или неявно указать время жизни ссылки:

```
// implicit
fn foo(x: &i32) {
}

// explicit
fn bar<'a>(x: &'a i32) {
}
```

Читается `'a` как ‘время жизни a‘. Технически, все ссылки имеет некоторое время жизни, связанное с ними, но компилятор позволяет опускать его в общих случаях. Прежде чем мы перейдем к этому, давайте разберем пример ниже, с явным указанием времени жизни:

```
fn bar<'a>(...)
```

Эта часть объявляет параметры времени жизни. Она говорит, что `bar` имеет один параметр времени жизни, `'a`. Если бы в качестве параметров функции у нас было две ссылки, то это выглядело бы так:

```
fn bar<'a, 'b>(...)
```

Затем в списке параметров функции, мы используем заданные параметры времени жизни:

```
...(x: &'a i32)
```

Если бы мы хотели `&mut` ссылку, то сделали бы так:

```
...(x: &'a mut i32)
```

Если вы сравните `&mut i32` с `&'a mut i32`, то увидите, что они отличаются только определением времени жизни `'a`, написанным между `&` и `mut i32`. `&mut i32` читается как ‘изменяемая ссылка на i32‘, а `&'a mut i32` - как ‘изменяемая ссылка на i32 со временем жизни 'a‘.

Вы также должны будете явно указать время жизни при работе с [`struct`][structs]:

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let y = &5; // this is the same as `let _y = 5; let y = &_y;`
    let f = Foo { x: y };

    println!("{}", f.x);
}
```

Как вы можете заметить, `struct` также могут иметь время жизни. Так же как и функции

```
struct Foo<'a> {
```

определяют время жизни и

```
x: &'a i32,
```

используют его. Почему же мы должны определять время жизни здесь? Мы должны убедиться, что ссылка на `Foo` не может жить дольше, чем ссылка на `i32`, содержащаяся в нем.

Осмысливаем области видимости (Thinking in scopes)

Один из способов понять, что же такое время жизни, это визуализировать область, в которой ссылка является действительной. Например:

```
fn main() {
    let y = &5;     // -+ y goes into scope
                    //  |
    // stuff        //  |
                    //  |
}                   // -+ y goes out of scope
```

Добавим нашу структуру `Foo`:

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let y = &5;           // -+ y goes into scope
    let f = Foo { x: y }; // -+ f goes into scope
    // stuff              //  |
                          //  |
}                         // -+ f and y go out of scope
```

Наша `f` живет в области видимости `y`, поэтому все работает. Что же произойдет, если это будет не так? Этот код не будет работать:

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let x;                    // -+ x goes into scope
                              //  |
    {                         //  |
        let y = &5;           // ---+ y goes into scope
        let f = Foo { x: y }; // ---+ f goes into scope
        x = &f.x;             //  | | error here
    }                         // ---+ f and y go out of scope
                              //  |
    println!("{}", x);        //  |
}                             // -+ x goes out of scope
```

Уф! Как вы можете видеть здесь, области видимости `f` и `y` меньше, чем область видимости `x`. Но когда мы выполняем `x = &f.x`, мы присваиваем `x` ссылку на что-то, что вот-вот выйдет из области видимости.

Присвоение имени времени жизни - это способ задать имя его области видимости. Задание имени является первым шагом, который дает возможность оперировать этим именованным понятием.

'static

Время жизни с именем ‘static‘ является специальным. Он обозначает, что что-то имеет время жизни, равное времени жизни всей программы. Большинство Rust программистов впервые столкиваются с `'static`, когда имеют дело со строками:

```
let x: &'static str = "Hello, world.";
```

Строковые литералы имеют тип `&'static str`, потому что ссылка должна быть действительна на протяжении работы всей программы: они располагаются в сегменте данных конечного двоичного файла. Другой пример - глобальные переменные:

```
static FOO: i32 = 5;
let x: &'static i32 = &FOO;
```

В этом примере `i32` добавляется в сегмент данных двоичного файла, а `x` ссылается на него.

Опускание времени жизни

В Rust есть мощный локальный вывод типов. Однако, сигнатуры объявлений верхнего уровня не выводятся, чтобы можно было рассуждать о типах на основании одних лишь сигнатур. Из соображений удобства, введён ограниченный механизм вывода типов сигнатур функций, называемый "опускание времени жизни" (“lifetime elision”). Он выводит типы на основании только элементов сигнатуры - тело функции при этом не учитывается. При этом его

назначение - это вывести лишь параметры времени жизни аргументов. Для этого он реализует три простых правила. Таким образом, опускание времени жизни упрощает написание сигнатур, одновременно не скрывая реальные типы аргументов.

Когда речь идет о неявном времени жизни, мы используем термины *входное время жизни* (*input lifetime*) и *выходное время жизни* (*output lifetime*). *Входное время жизни* связано с передаваемыми в функцию параметрами, а *выходное время жизни* связано с возвращаемым функцией значением. Например, эта функция имеет входное время жизни:

```
fn foo<'a>(bar: &'a str)
```

А эта имеет выходное время жизни:

```
fn foo<'a>() -> &'a str
```

Эта же имеет как входное, так и выходное время жизни:

```
fn foo<'a>(bar: &'a str) -> &'a str
```

Ниже представлены три правила:

- Каждое неявное время жизни в аргументах функции становится индивидуальным временем жизни.
- Если есть ровно одно входное время жизни, явное или неявное, то это время жизни назначается всем неявным выходным временам жизни.
- Если есть несколько входных вемен жизни, но один из них это `&self` или `&mut self`, то всем неявным выходным временам жизни назначается время жизни `self`.

В противном случае, неявное задание выходного времени жизни является ошибкой.

Примеры

Вот некоторые примеры функций, представленные в 2 видах: с явно и неявно заданным временем жизни:

```
fn print(s: &str); // неявно
fn print<'a>(s: &'a str); // явно

fn debug(lvl: u32, s: &str); // неявно
fn debug<'a>(lvl: u32, s: &'a str); // явно

// В предыдущем примере, для `lvl` не требуется указывать время жизни, потому
// что это не ссылка (`&`). Только элементы, связанные с ссылками (например,
// такие как `struct`, которая содержит ссылку) требуют указания времени жизни.

fn substr(s: &str, until: u32) -> &str; // неявно
fn substr<'a>(s: &'a str, until: u32) -> &'a str; // явно

fn get_str() -> &str; // НЕКОРРЕКТНО, нет входных параметров

fn frob(s: &str, t: &str) -> &str; // НЕКОРРЕКТНО, два входных параметра
fn frob<'a, 'b>(s: &'a str, t: &'b str) -> &str; // Expanded: Output lifetime is unclear

fn get_mut(&mut self) -> &mut T; // неявно
fn get_mut<'a>(&'a mut self) -> &'a mut T; // явно

fn args<T:ToCStr>(&mut self, args: &[T]) -> &mut Command // неявно
fn args<'a, 'b, T:ToCStr>(&'a mut self, args: &'b [T]) -> &'a mut Command // явно

fn new(buf: &mut [u8]) -> BufWriter; // неявно
fn new<'a>(buf: &'a mut [u8]) -> BufWriter<'a> // явно
```

Изменяемость (mutability)

Изменяемость, то есть возможность изменить что-то, работает в Rust несколько иначе, чем в других языках. Во-первых, по умолчанию связанные имена не изменяемы:

```
let x = 5;
x = 6; // ошибка!
```

Изменяемость можно добавить с помощью ключевого слова `mut`:

```
let mut x = 5;

x = 6; // нет проблем!
```

Это изменяемое связанное имя. Когда связанное имя изменяемо, это означает, что мы можем поменять связанное с ним значение. В примере выше не то, чтобы само значение `x` менялось, просто имя `x` связывается с другим значением типа `i32`.

Если же вы хотите изменить само связанное значение, вам понадобится изменяемая ссылка:

```
let mut x = 5;
let y = &mut x;
```

`y` - это неизменяемое имя для изменяемой ссылки. Это значит, что `y` нельзя связать ещё с чем-то (`y = &mut z`), но можно изменить то, на что указывает связанная ссылка (`*y = 5`). Тонкая разница.

Конечно, вы можете объявить и изменяемое имя для изменяемой ссылки:

```
let mut x = 5;
let mut y = &mut x;
```

Теперь `y` можно связать с другим значением, и само это значение тоже можно менять.

Стоит отметить, что `mut` - это часть образца, поэтому можно делать такие вещи:

```
let (mut x, y) = (5, 6);

fn foo(mut x: i32) {
```

Внутренняя (interior) и внешняя (exterior) изменяемость

Однако, когда мы говорим, что что-либо "неизменяемо" в Rust, это не означает, что оно не может измениться: мы имеем ввиду что-то, что имеет "внешнюю изменяемость". Для примера рассмотрим `Arc<T>`:

```
use std::sync::Arc;

let x = Arc::new(5);
let y = x.clone();
```

Когда мы вызываем метод `clone()`, `Arc<T>` должна обновить счётчик ссылок. Мы не использовали модификатор `mut`, а значит `x` - неизменяемая привязка и мы не можем получить ссылку (`&mut 5`) или что-то подобное. Так что же это даёт?

Для того чтобы понять это, мы должны вернуться назад к основам философии Rust'a, к сохранности памяти и механизму, гарантирующему это, к системе владения, и, в частности, к заимствованию:

> Одновременно у вас может быть только один из двух перечисленных ниже видов заимствования, но не оба сразу:
>
> - одна или более неизменяемых ссылок (`&T`) на ресурс,
> - ровно одна изменяемая ссылка (`&mut T`) на ресурс.

Итак, что же здесь на самом деле является "неизменяемым": безопасно ли иметь два указателя на один объект? В случае с `Arc<T>`, да: изменяемый объект полностью находится внутри самой структуры. По этой причине, метод `clone()` возвращает неизменяемую ссылку (`&T`). Если бы он возвращал изменяемую ссылку (`&mut T`), то у нас были бы проблемы.

Другие типы, например те, что определены в модуле `std::cell`, напротив, имеют "внутреннюю изменяемость". Например:

```
use std::cell::RefCell;

let x = RefCell::new(42);

let y = x.borrow_mut();
```

RefCell возвращает изменяемую ссылку `&mut` при помощи метода `borrow_mut()`. А не опасно ли это? Что, если мы сделаем так:

```
use std::cell::RefCell;

let x = RefCell::new(42);

let y = x.borrow_mut();
let z = x.borrow_mut();
```

Это приведёт к панике во время исполнения. Вот что делает `RefCell`: он принудительно выполняет проверку правил заимствования во время исполнения и вызывает `panic!`, если они были нарушены. Это подводит нас к другим аспектам правил изменяемости Rust'a. Давайте сначала поговорим о них.

Изменяемость на уровне полей

Изменяемость - это свойство либо заимствования (`&mut`), либо связывания (`let mut`). Это значит, что, например, у вас не может быть структуры, часть полей которой изменяется, а другая часть - нет:

```
struct Point {
    x: i32,
    mut y: i32, // нельзя
}
```

Изменяемость структуры определяется при её связывании:

```
struct Point {
    x: i32,
    y: i32,
}

let mut a = Point { x: 5, y: 6 };

a.x = 10;

let b = Point { x: 5, y: 6};

b.x = 10; // error: cannot assign to immutable field `b.x`
```

Однако, используя `Cell<T>`, вы можете эмулировать изменяемость на уровне полей:

```
use std::cell::Cell;

struct Point {
    x: i32,
    y: Cell<i32>,
}

let point = Point { x: 5, y: Cell::new(6) };

point.y.set(7);

println!("y: {:?}", point.y);
```

Это выведет на экран `y: Cell { value: 7 }`. Мы успешно изменили значение `y`.

Перечисления

В Rust есть "типы-суммы", или *перечисления* (тип-сумма - это термин из теории типов). Перечисления - это очень полезная возможность Rust, и они очень много используются в стандартной библиотеке языка. Они объявляются с помощью ключевого слова `enum`. `enum` - это тип, который соотносит набор неких вариантов одному имени. Например, ниже мы определяем перечисление `Character` (символ), представляющее собой или цифру (`Digit`), или что-то другое.

```
enum Character {
    Digit(i32),
    Other,
}
```

Большая часть обычных типов могут быть вариантами перечисления. Вот несколько примеров:

```
struct Empty;
struct Color(i32, i32, i32);
struct Length(i32);
struct Stats { Health: i32, Mana: i32, Attack: i32, Defense: i32 }
struct HeightDatabase(Vec<i32>);
```

Здесь мы видим, что, в зависимости от типа, вариант перечисления может содержать, а может и не содержать вложенные данные. Например, в перечислении `Character`, вариант `Digit` даёт значимое имя числу типа `i32`. А вот вариант `Other` представляет собой лишь имя, без значения. Однако наиболее полезно именно то, что отдельные варианты представляют собой различные виды символов (`Character`).

Как и структуры, варианты перечислений по умолчанию не сравнимы операциями сравнения (`==`, `!=`), не упорядочены (не реализуют `<`, `>=` и другие) и не поддерживают другие двухместные операции, такие как умножение (`*`) и сложение (`+`). Нижеследующий код, как таковой, не верен (если мы используем приведённый выше тип-перечисление `Character`):

```
// Оба этих присваивания успешны
let ten  = Character::Digit(10);
let four = Character::Digit(4);

// Error: `*` is not implemented for type `Character`
let forty = ten * four;

// Error: `<=` is not implemented for type `Character`
let four_is_smaller = four <= ten;

// Error: `==` is not implemented for type `Character`
let four_equals_ten = four == ten;
```

Мы используем `::` синтаксис чтобы использовать имя каждого из вариантов. Их область видимости ограничена именем самого перечисления `enum`. Это позволяет использовать оба варианта из примера ниже совместно:

```
Character::Digit(10);
Hand::Digit;
```

Оба варианта имеют одинаковые имена, `Digit`, но область видимости каждого из них ограничена соответствующим именем `enum`.

То, что эти операции не поддерживаются, может показаться достаточно ограничивающим, но это ограничение, которое мы всегда можем преодолеть. Есть два способа: посредством реализация равенства самостоятельно, или посредством сопоставления вариантов с шаблонами с помощью выражений конструкции `match`, о котором вы узнаете в следующем разделе. Пока мы еще не знаем достаточно о Rust для реализации равенства, но мы узнаем об этом в разделе `traits`.

Конструкция `match` (Сопоставление с шаблоном)

Простого `if`/`else` часто недостаточно, потому что нужно проверить больше, чем два возможных варианта. Да и к тому же условия в `else` часто становятся очень сложными. Как же решить эту проблему?

В Rust есть ключевое слово `match`, позволяющее заменить группы операторов `if`/`else` чем-то более удобным. Смотрите:

```
let x = 5;

match x {
    1 => println!("один"),
    2 => println!("два"),
    3 => println!("три"),
    4 => println!("четыре"),
    5 => println!("пять"),
    _ => println!("что-то ещё"),
}
```

`match` принимает выражение и выбирает одну из ветвей исполнения согласно его значению. Каждая *ветвь* имеет форму `значение => выражение`. Выражение ветви вычисляется, когда значение данной ветви совпадает со значением, принятым оператором `match` (в данном случае, `x`). Эта конструкция называется `match` (сопоставление), потому что она выполняет сопоставление значения неким "шаблонам". Раздел Шаблоны, идущий далее, охватывает все варианты, которые были затронуты здесь.

Так в чём же преимущества данной конструкции? Их несколько. Во-первых, ветви `match` *проверяются на полноту*. Видите последнюю ветвь, со знаком подчёркивания (`_`)? Если мы удалим её, Rust выдаст ошибку:

```
error: non-exhaustive patterns: `_` not covered
```

Другими словами, компилятор сообщает нам, что мы забыли сопоставить какие-то значения. Поскольку `x` - это целое число, оно может принимать разные значения - например, `6`. Однако, если мы убираем ветвь `_`, ни одна ветвь не совпадёт, поэтому такой код не скомпилируется. `_` - это ‘совпадение с любым значением‘. Если ни одна другая ветвь не совпала, совпадёт ветвь с `_`. Поскольку в примере выше есть ветвь с `_`, мы покрываем всё множество значений `x`, и наша программа скомпилируется.

`match` также является выражением. Это значит, что мы можем использовать его в правой части оператора `let` или непосредственно как выражение:

```
let x = 5;

let numer = match x {
    1 => "one",
    2 => "two",
    3 => "three",
    4 => "four",
    5 => "five",
    _ => "something else",
};
```

Иногда это очень удобный вариант преобразования.

Структуры

Структура - это другой вид *агрегатного типа*, как и кортеж. Разница в том, что в структурах у каждого элемента есть имя. Элемент структуры называется *полем* или *членом структуры*. Смотрите:

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let origin = Point { x: 0, y: 0 }; // origin: Point

    println!("Начало координат находится в ({}, {})", origin.x, origin.y);
}
```

Этот код делает много разных вещей, поэтому давайте разберём его по порядку. Мы объявляем структуру с помощью ключевого слова `struct`, за которым следует имя объявляемой структуры. Обычно, имена типов-структур начинаются с заглавной буквы и используют чередующийся регистр букв: название `PointInSpace` выглядит привычно, а `Point_In_Space` - нет.

Как всегда, мы можем создать экземпляр нашей структуры с помощью оператора `let`. Однако в данном случае мы используем синтаксис вида `ключ: значение` для установки значения каждого поля. Порядок инициализации полей не обязательно должен совпадать с порядком их объявления.

Наконец, поскольку у полей есть имена, мы можем получить поле с помощью операции `точка`: `origin.x`.

Значения, хранимые в структурах, неизменяемы по умолчанию. В этом плане они не отличаются от других именованных сущностей. Чтобы они стали изменяемы, используйте ключевое слово `mut`:

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let mut point = Point { x: 0, y: 0 };

    point.x = 5;

    println!("Точка находится в ({}, {})", point.x, point.y);
}
```

Этот код напечатает `Точка находится в (5, 0)`.

Шаблоны сопоставления `match`

Шаблоны достаточно часто используются в Rust. Мы уже использовали их в разделе Связывание переменных, в разделе Конструкция `match`, а также в некоторых других местах. Давайте коротко пробежимся по всем возможностям, которые можно реализовать с помощью шаблонов!

Быстро освежим в памяти: сопоставлять с шаблоном литералы можно либо напрямую, либо с использованием символа `_`, который означает *любой* случай:

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

Этот код напечатает `one`.

Сопоставление с несколькими шаблонами

Вы можете сопоставлять с несколькими шаблонами, используя `|`:

```
let x = 1;

match x {
    1 | 2 => println!("one or two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

Этот код напечатает `one or two`.

Сопоставление с диапазоном

Вы можете сопоставлять с диапазоном значений, используя `...`:

```
let x = 1;

match x {
    1 ... 5 => println!("one through five"),
    _ => println!("anything"),
}
```

Этот код напечатает `one through five`.

Диапазоны в основном используются с числами или одиночными символами (`char`).

```
let x = '';

match x {
    'a' ... 'j' => println!("early letter"),
    'k' ... 'z' => println!("late letter"),
    _ => println!("something else"),
}
```

Этот код напечатает `something else`.

Связывание

Вы можете связать значение с именем с помощью символа `@`:

```
let x = 1;

match x {
    e @ 1 ... 5 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

Этот код напечатает `got a range element 1`. Это полезно, когда вы хотите сделать сложное сопоставление для части структуры данных:

```
#[derive(Debug)]
struct Person {
    name: Option<String>,
}

let name = "Steve".to_string();
let mut x: Option<Person> = Some(Person { name: Some(name) });
match x {
    Some(Person { name: ref a @ Some(_), .. }) => println!("{:?}", a),
    _ => {}
}
```

Этот код напечатает `Some("Steve")`: мы связали внутреннюю `name` с `a`.

Если вы используете `@` совместно с `|`, то вы должны убедиться, что имя связывается в каждой из частей шаблона:

```
let x = 5;

match x {
    e @ 1 ... 5 | e @ 8 ... 10 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

Игнорирование вариантов

Если при сопоставлении вы используете перечисление, содержащее варианты, то вы можете указать `..`, чтобы проигнорировать значение и тип в варианте:

```
enum OptionalInt {
    Value(i32),
    Missing,
}

let x = OptionalInt::Value(5);

match x {
    OptionalInt::Value(..) => println!("Got an int!"),
    OptionalInt::Missing => println!("No such luck."),
}
```

Этот код напечатает `Got an int!`.

Ограничители шаблонов

Вы можете ввести *ограничители шаблонов* (*match guards*) с помощью `if`:

```
enum OptionalInt {
    Value(i32),
    Missing,
}

let x = OptionalInt::Value(5);

match x {
    OptionalInt::Value(i) if i > 5 => println!("Got an int bigger than five!"),
    OptionalInt::Value(..) => println!("Got an int!"),
    OptionalInt::Missing => println!("No such luck."),
}
```

Этот код напечатает `Got an int!`.

ref и ref mut

Если вы хотите получить ссылку, то используйте ключевое слово `ref`:

```
let x = 5;

match x {
    ref r => println!("Got a reference to {}", r),
}
```

Этот код напечатает `Got a reference to 5`.

Здесь `r` внутри `match` имеет тип `&i32`. Другими словами, ключевое слово `ref` *создает* ссылку, для использования в шаблоне. Если вам нужна изменяемая ссылка, то `ref mut` будет работать аналогичным образом:

```
let mut x = 5;

match x {
    ref mut mr => println!("Got a mutable reference to {}", mr),
}
```

Деструктуризация

Если у вас есть сложный тип данных, например `struct`, вы можете деструктурировать его внутри шаблона:

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x: x, y: y } => println!("({},{})", x, y),
}
```

Если нам нужны значения только некоторых из полей структуры, то мы можем не присваивать им всем имена:

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x: x, .. } => println!("x is {}", x),
}
```

Этот код напечатает `x is 0`.

Вы можете сделать это для любого поля, а не только для первого:

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { y: y, .. } => println!("y is {}", y),
}
```

Этот код напечатает `y is 0`.

Такое ‘деструктурирование‘ работает для любых сложных типов данных, таких как кортежи или перечисления.

Mix and Match

Вот так! Существует много разных способов использования конструкции сопоставления с шаблоном, и все они могут быть смешаны и состыкованы, в зависимости от того, что вы хотите сделать:

```
match x {
    Foo { x: Some(ref name), y: None } => ...
}
```

Шаблоны являются очень мощным инструментом. Их использование находит очень широкое применение.

Синтаксис методов

Функции - это хорошо, но если вы хотите вызвать несколько связных функций для каких-либо данных, то это может быть неудобно. Рассмотрим этот код:

```
baz(bar(foo()));
```

Читать данную строку кода следует слева направо, поэтому мы наблюдаем такой порядок: "baz bar foo." Но он противоположен порядку, в котором функции будут вызываться: "foo bar baz." Не было бы неплохо, если бы мы могли использовать нечто вроде нижеприведенного синтаксиса вызова?

```
foo.bar().baz();
```

К счастью, как вы уже наверно догадались, это возможно! Rust предоставляет возможность использовать такой *синтаксис вызова метода* с помощью ключевого слова `impl`.

Вызов методов

Вот как это работает:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

fn main() {
    let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
    println!("{}", c.area());
}
```

Этот код напечатает `12.566371`.

Мы создали структуру, которая представляет собой круг. Затем мы написали блок `impl` и определили метод `area` внутри него.

Методы принимают специальный первый параметр, `&self`. Есть три возможных варианта: `self`, `&self` и `&mut self`. Вы можете думать об этом специальном параметре как о `x` в `x.foo()`. Три варианта соответствуют трем возможным видам элемента `x`: `self` - если это просто значение в стеке, `&self` - если это ссылка и `&mut self` - если это

изменяемая ссылка. Мы передаем параметр `&self` в метод `area`, поэтому мы можем использовать его так же, как и любой другой параметр. Так как мы знаем, что это `Circle`, мы можем получить доступ к полю `radius` так же, как если бы это была любая другая структура.

По умолчанию следует использовать `&self`, также как следует предпочитать заимствование владению, а неизменные ссылки изменяемым. Вот пример, включающий все три варианта:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn reference(&self) {
       println!("taking self by reference!");
    }

    fn mutable_reference(&mut self) {
       println!("taking self by mutable reference!");
    }

    fn takes_ownership(self) {
       println!("taking ownership of self!");
    }
}
```

Цепочка вызовов методов

Итак, теперь мы знаем, как вызвать метод, например `foo.bar()`. Но что насчет нашего первоначального примера, `foo.bar().baz()`? Это называется 'цепочка вызовов', и мы можем сделать это, вернув `self`.

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }

    fn grow(&self, increment: f64) -> Circle {
        Circle { x: self.x, y: self.y, radius: self.radius + increment }
    }
}

fn main() {
    let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
    println!("{}", c.area());

    let d = c.grow(2.0).area();
    println!("{}", d);
}
```

Проверьте тип возвращаемого значения:

```
fn grow(&self) -> Circle {
```

Мы просто указываем, что возвращается `Circle`. С помощью этого метода мы можем создать новый круг, площадь которого будет в 100 раз больше, чем у старого.

Статические методы

Вы также можете определить методы, которые не принимают параметр `self`. Вот паттерн, который очень распространен в Rust коде:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn new(x: f64, y: f64, radius: f64) -> Circle {
        Circle {
            x: x,
            y: y,
            radius: radius,
        }
    }
}

fn main() {
    let c = Circle::new(0.0, 0.0, 2.0);
}
```

Этот *статический метод*, который создает новый `Circle`. Обратите внимание, что статические методы вызываются с помощью синтаксиса: `Struct::method()`, а не `ref.method()`.

Паттерн строитель

Давайте предположим, что нам нужно, чтобы наши пользователи могли создавать круги и чтобы у них была возможность задавать только те свойства, которые им нужны. В противном случае, атрибуты `x` и `y` будут `0.0`, а `radius` будет `1.0`. Rust не поддерживает перегрузку методов, именованные аргументы или переменное количество аргументов. Вместо этого мы используем паттерн строитель. Он выглядит следующим образом:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

struct CircleBuilder {
    x: f64,
    y: f64,
    radius: f64,
}

impl CircleBuilder {
    fn new() -> CircleBuilder {
        CircleBuilder { x: 0.0, y: 0.0, radius: 0.0, }
    }

    fn x(&mut self, coordinate: f64) -> &mut CircleBuilder {
        self.x = coordinate;
        self
    }

    fn y(&mut self, coordinate: f64) -> &mut CircleBuilder {
        self.y = coordinate;
        self
    }

    fn radius(&mut self, radius: f64) -> &mut CircleBuilder {
        self.radius = radius;
        self
    }

    fn finalize(&self) -> Circle {
        Circle { x: self.x, y: self.y, radius: self.radius }
    }
}

fn main() {
    let c = CircleBuilder::new()
                .x(1.0)
                .y(2.0)
                .radius(2.0)
                .finalize();

    println!("area: {}", c.area());
    println!("x: {}", c.x);
    println!("y: {}", c.y);
}
```

Все, что мы сделали здесь, это создали еще одну структуру, `CircleBuilder`. В ней мы определили методы строителя. Также мы определили метод `area()` в `Circle`. Мы также сделали еще один метод в `CircleBuilder`: `finalize()`. Этот метод создает наш окончательный `Circle` из строителя. Таким образом, мы использовали систему типов для обеспечения концепции: мы можем использовать методы `CircleBuilder`, которые каким-либо образом ограничивают создание `Circle`, в зависимости от нашего выбора.

Вектора

"Вектор" - это динамический или, по-другому, "растущий" массив, реализованный в виде стандартного библиотечного типа `Vec<T>` (где `<T>` является дженериком). Вектора всегда размещают данные в куче. Вы можете создавать их с помощью макроса `vec!`:

```
let v = vec![1, 2, 3, 4, 5]; // v: Vec<i32>
```

(Заметьте, что, в отличие от макроса `println!`, который мы использовали ранее, с `vec!` используются квадратные скобки `[]`. Rust разрешает использование и круглых, и квадратных скобок в обеих ситуациях - это просто стилистическое соглашение.)

Для создания вектора из повторяющихся значений есть другая форма `vec!`:

```
let v = vec![0; 10]; // десять нулей
```

Доступ к элементам

Чтобы получить значение по определенному индексу в векторе, мы используем `[]`:

```
let v = vec![1, 2, 3, 4, 5];

println!("Третий элемент вектора v равен {}", v[2]);
```

Индексы отсчитываются от `0`, так что третьим элементом является `v[2]`.

Итерирование

Если у вас есть вектор, то вы можете итерировать по его элементам с помощью `for`. Есть три варианта:

```
let mut v = vec![1, 2, 3, 4, 5];

for i in &v {
    println!("Ссылка {}", i);
}

for i in &mut v {
    println!("Изменяемая ссылка {}", i);
}

for i in v {
    println!("Владение вектором и его элементами {}", i);
}
```

У векторов есть много других полезных методов, о которых вы можете прочитать в документации API.

Строки

Строки являются важным понятием для любого ведущего программиста. Система обработки строк в Rust немного отличается от других языков, в связи с его фокусировкой на системный уровень. Всякий раз, когда вы имеете дело со структурой данных с переменным размером, все может стать слишком запутанным, и строки как раз являются структурой данных с переменным размером. Кроме того, работа со строками в Rust также отличается и от некоторых системных языков, таких как C.

Давайте разбираться в деталях. *string* - это последовательность скалярных значений юникод, закодированных в виде потока UTF-8 байт. Все строки должны быть гарантированно валидными UTF-8 последовательностями. Кроме того, строки не оканчиваются нулем и могут содержать нулевые байты.

В Rust есть два основных типа строк: `&str` и `String`. Сперва поговорим о `&str`. Это произносится как 'строковый срез (слайс)'. Строковые литералы имеют тип `&'static str`:

```
let string = "Hello there."; // string: &'static str
```

Эта строка выделяется статически, что означает, что она сохраняется в нашей скомпилированной программе и существует в течение всего периода ее выполнения. `string` привязка представляет собой ссылку на эту статически размещенную строку. Строковые срезы имеют фиксированный размер и не могут быть изменены.

`String` же, напротив, выделяется в куче. Эта строка расширяема, а также она гарантированно является UTF-8 последовательностью. `String` обычно создается путем преобразования из строкового среза с использованием метода `to_string`.

```
let mut s = "Hello".to_string(); // mut s: String
println!("{}", s);

s.push_str(", world.");
println!("{}", s);
```

`String` преобразуются в `&str` с помощью `&`:

```
fn takes_slice(slice: &str) {
    println!("Got: {}", slice);
}

fn main() {
    let s = "Hello".to_string();
    takes_slice(&s);
}
```

Представление `String` как `&str` является дешевой операцией, но преобразование `&str` в `String` предполагает выделение памяти. Нет причин делать это, если в этом нет необходимости!

Индексация

Поскольку строки являются валидными UTF-8 последовательностями, то они не поддерживают индексацию:

```
let s = "hello";

println!("The first letter of s is {}", s[0]); // ERROR!!!
```

Как правило, доступ к вектору с помощью `[]` является очень быстрой операцией. Но поскольку каждый символ в строке, закодированной UTF-8, может быть представлен несколькими байтами, то при поиске вы должны перебрать n-ое количество литер в строке. Это значительно более дорогая операция, а мы не хотим вводить в заблуждение. Кроме того, ‘литера‘ - это не совсем то, что определено в Unicode. Мы можем выбрать как рассматривать строку: как отдельные байты или как кодовые единицы:

```
let hachiko = "□□□□□";

for b in hachiko.as_bytes() {
    print!("{}, ", b);
}

println!("");

for c in hachiko.chars() {
    print!("{}, ", c);
}

println!("");
```

Этот код напечатает:

```
229, 191, 160, 231, 138, 172, 227, 131, 143, 227, 131, 129, 229, 133, 172,
□, □, □, □, □,
```

Как вы можете видеть, количество байт больше, чем количество символов (`char`).

Вы можете получить что-то наподобие индекса, как показано ниже:

```
let dog = hachiko.chars().nth(1); // kinda like hachiko[1]
```

Это подчеркивает, что мы должны пройти через весь список `chars`.

Конкатенация

Если у вас есть `String`, то вы можете присоединить к ней в конец `&str`:

```
let hello = "Hello ".to_string();
let world = "world!";

let hello_world = hello + world;
```

Но если у вас есть две `String`, то необходимо использовать `&`:

```
let hello = "Hello ".to_string();
let world = "world!".to_string();

let hello_world = hello + &world;
```

Это потому, что `&String` может быть автоматически приведен к `&str`. Эта фича называется ‘Приведение при разыменовании’.

Дженерики (обобщённые типы)

Иногда, при написании функции или типа данных, мы можем захотеть, чтобы они работали для нескольких типов аргументов. К счастью, у Rust есть фича, которая дает нам лучший способ реализовать это: дженерики (обобщённые типы). Дженериками называется ‘параметрический полиморфизм‘ в теории типов. Это означает, что они являются типами или функциями, которые имеют несколько форм (‘poly‘ - кратно, ‘morph‘ - форма) по данному параметру (‘параметрический‘).

В любом случае, хватит о теории типов, давайте рассмотрим какой-нибудь дженерик код. Стандартная библиотека Rust предоставляет тип `Option<T>`, который является дженерик типом:

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

`Часть <T>`, которую вы уже видели несколько раз прежде, указывает, что это обобщённый тип данных. Внутри перечисления, везде, где мы видим `T`, мы подставляем вместо этого абстактного типа тот, который используется в дженерике. Вот пример использования `Option<T>` с некоторыми дополнительными аннотациями типов:

```
let x: Option<i32> = Some(5);
```

В определении типа мы используем `Option<i32>`. Обратите внимание, что это очень похоже на `Option<T>`. С той лишь разницей, что, в данном конкретном `Option`, `T` имеет значение `i32`. В правой стороне выражения, мы используем `Some(T)`, где `T` равно `5`. Так как `5` является представителем типа `i32`, то типы по обе стороны совпадают, отчего Rust счастлив. Если же они не совпадают, то мы получим ошибку:

```
let x: Option<f64> = Some(5);
// error: mismatched types: expected `core::option::Option<f64>`,
// found `core::option::Option<_>` (expected f64 but found integral variable)
```

Но это не значит, что мы не можем сделать `Option<T>`, который содержит `f64`! Просто с левой и с правой сторон выражения объявления переменной типы должны совпадать:

```
let x: Option<i32> = Some(5);
let y: Option<f64> = Some(5.0f64);
```

Это просто прекрасно. Одно определение - многостороннее использование.

Дженерики могут быть обобщёнными не только для одного единственного типа. Рассмотрим другой подобный тип из стандартной библиотеки Rust - `Result<T, E>`:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Этот тип является обобщённым сразу для *двух* типов: `T` и `E`. Кстати, заглавные буквы может быть любыми, какими хотите. Мы могли бы определить `Result<T, E>` как:

```
enum Result<A, Z> {
    Ok(A),
    Err(Z),
}
```

если бы захотели. Соглашение гласит, что первый обобщённый параметр для 'типа' должен быть `T`, и что для 'ошибки' используется `E`. Но Rust не проверяет этого.

Тип `Result<T, E>` предназначен для того, чтобы возвращать результат вычисления, и имеет возможность вернуть ошибку, если произойдет какой-либо сбой.

Дженерик функции

Мы можем задавать функции, которые принимают дженерик типы с помощью аналогичного синтаксиса:

```
fn takes_anything<T>(x: T) {
    // do something with x
}
```

Синтаксис состоит из двух частей: `<T>` говорит о том, что “эта функция является обобщённой по одному типу, `T`”, а `x: T` говорит о том, что “x имеет тип `T`”.

Несколько аргументов могут иметь один и тот же дженерик тип:

```
fn takes_two_of_the_same_things<T>(x: T, y: T) {
    // ...
}
```

Мы можем написать версию, которая принимает несколько типов:

```
fn takes_two_things<T, U>(x: T, y: U) {
    // ...
}
```

Обобщённые функции наиболее полезны в связке с ‘ограничениями по типажам‘, о которых мы расскажем в главе Типажи.

Дженерик структуры

Вы также можете задать дженерик тип для `struct`:

```
struct Point<T> {
    x: T,
    y: T,
}

let int_origin = Point { x: 0, y: 0 };
let float_origin = Point { x: 0.0, y: 0.0 };
```

Аналогично функциям, мы также объявляем дженерик параметры в `<T>`, а затем используем их в объявлении типа `x: T`.

Типажи

Вы помните, ключевое слово `impl`, используемое для вызова функции с синтаксисом метода?

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

Типажи схожи, за исключением того, что мы определяем типаж, содержащий лишь сигнатуру метода, а затем реализуем этот типаж для нужной структуры. Например, как показано ниже:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

Как вы можете видеть, блок `trait` очень похож на блок `impl`. Различие состоит лишь в том, что тело метода не определяется, а определяется только его сигнатура. Когда мы реализуем типаж, мы используем `impl Trait for Item`, а не просто `impl Item`.

Так что же в этом такого грандиозного? Помните ошибку, которую мы получали для нашей дженерик функции `inverse`?

```
error: binary operation `==` cannot be applied to type `T`
```

Мы можем использовать типажи для ограничения дженериков. Рассмотрим похожую функцию, которая также не компилируется, и выводит ошибку:

```
fn print_area<T>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

Rust выводит:

```
error: type `T` does not implement any method in scope named `area`
```

Поскольку `T` может быть любого типа, мы не можем быть уверены, что он реализует метод `area`. Но мы можем добавить **ограничение по типажу** к нашему дженерику `T`, гарантируя, что он будет соответствовать требованиям:

```
fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

Синтаксис `<T: HasArea>` означает **любой тип, реализующий типаж `HasArea`**. Так как типажи определяют сигнатуры типов функций, мы можем быть уверены, что любой тип, который реализует `HasArea` будет иметь метод `.area()`.

Вот расширенный пример того, как это работает:

```
trait HasArea {
    fn area(&self) -> f64;
}

struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

struct Square {
    x: f64,
    y: f64,
    side: f64,
}

impl HasArea for Square {
    fn area(&self) -> f64 {
        self.side * self.side
    }
}

fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}

fn main() {
    let c = Circle {
        x: 0.0f64,
        y: 0.0f64,
        radius: 1.0f64,
    };

    let s = Square {
        x: 0.0f64,
        y: 0.0f64,
        side: 1.0f64,
    };

    print_area(c);
    print_area(s);
}
```

Ниже показан вывод программы:

```
This shape has an area of 3.141593
This shape has an area of 1
```

Как вы можете видеть, теперь `print_area` не только является дженериком, но и гарантирует, что будет получен корректный тип. Если же мы передадим некорректный тип:

```
print_area(5);
```

Мы получим ошибку времени компиляции:

```
error: failed to find an implementation of trait main::HasArea for int
```

До сих пор мы добавляли реализации типажей лишь для структур, но реализовать типаж возможно для любого типа. Технически, мы *могли бы* реализовать `HasArea` для `i32`:

```
trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for i32 {
    fn area(&self) -> f64 {
        println!("this is silly");

        *self as f64
    }
}

5.area();
```

Хотя технически это возможно, но реализация методов для таких примитивных типов считается плохим стилем программирования.

Такой подход может показаться неконтролируемым, однако есть два ограничения, связанные с реализацией типажей, которые мешают коду выйти из-под контроля. Во- первых, типажи должны быть объявлены используемыми, с помощью ключевого слова `use`, в той области видимости, где вы хотите использовать методы этих типажей. Так, например, следующий код не будет работать:

```
mod shapes {
    use std::f64::consts;

    trait HasArea {
        fn area(&self) -> f64;
    }

    struct Circle {
        x: f64,
        y: f64,
        radius: f64,
    }

    impl HasArea for Circle {
        fn area(&self) -> f64 {
            consts::PI * (self.radius * self.radius)
        }
    }
}

fn main() {
    let c = shapes::Circle {
        x: 0.0f64,
        y: 0.0f64,
        radius: 1.0f64,
    };

    println!("{}", c.area());
}
```

Теперь, когда мы переместили структуры и типажи в свой собственный модуль, мы получаем следующую ошибку:

```
error: type `shapes::Circle` does not implement any method in scope named `area`
```

Если мы добавим строку с `use` над функцией `main` и сделаем нужные элементы публичными, все будет в порядке:

```
mod shapes {
    use std::f64::consts;

    pub trait HasArea {
        fn area(&self) -> f64;
    }

    pub struct Circle {
        pub x: f64,
        pub y: f64,
        pub radius: f64,
    }

    impl HasArea for Circle {
        fn area(&self) -> f64 {
            consts::PI * (self.radius * self.radius)
        }
    }
}

use shapes::HasArea;

fn main() {
    let c = shapes::Circle {
        x: 0.0f64,
        y: 0.0f64,
        radius: 1.0f64,
    };

    println!("{}", c.area());
}
```

Это означает, что даже если кто-то сделает что-то плохое, как например добавит методы к `int`, на наш код это не окажет влияния, если вы не объявите `use` для этого типажа.

Вот еще одно ограничение, связанное с реализацией типажей. Либо типаж, либо тип, для которого вы пишете `impl`, должны находиться в вашем крейте. Таким образом, мы могли бы реализовать типаж `HasArea` для `i32`, потому что `HasArea` находится в нашем крейте. Но если бы мы попытались реализовать типаж `Float`, предоставляемый самим Rust, для `i32`, мы не смогли бы этого сделать, потому что оба: типаж и тип отсутствуют в нашем крейте.

Последнее, что нужно сказать о типажах: дженерик функции с ограничением по типажам используют *мономорфизацию* (*mono*: один, *morph*: форма), поэтому они диспетчеризуются статически. Что это значит? Посмотрите главу Статическая и динамическая диспетчеризация, чтобы получить больше информации.

Множественные привязки к типажам

Вы уже видели как можно ограничить обобщенный параметр типа определенным типажом:

```
fn foo<T: Clone>(x: T) {
    x.clone();
}
```

Если вам нужно более одной привязки, то вы можете использовать `+`:

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

fn foo<T: Clone + Debug>(x: T) {
    x.clone();
    println!("{:?}", x);
}
```

Теперь тип `T` должен реализовавать как типаж `Clone`, так и типаж `Debug`.

Утверждение where

Написание функций с несколькими дженерик типами и небольшим количеством ограничений по типажам выглядит не так уж плохо, но, с увеличением количества зависимостей, синтаксис получается более неуклюжим:

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

fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
    x.clone();
    y.clone();
    println!("{:?}", y);
}
```

Имя функции находится слева, а список параметров - далеко справа. Привязки загромождают место.

Rust имеет решение на этот счет, и оно называется 'утверждение `where`':

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

fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
    x.clone();
    y.clone();
    println!("{:?}", y);
}

fn bar<T, K>(x: T, y: K) where T: Clone, K: Clone + Debug {
    x.clone();
    y.clone();
    println!("{:?}", y);
}

fn main() {
    foo("Hello", "world");
    bar("Hello", "workd");
}
```

`foo()` использует синтаксис, показанный ранее, а `bar()` использует утверждение `where`. Все, что нам нужно сделать, это убрать ограничения при определении типов параметров, а затем добавить `where` после списка параметров. Для более длинного списка, могут быть добавлены пробельные символы:

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

fn bar<T, K>(x: T, y: K)
    where T: Clone,
          K: Clone + Debug {

    x.clone();
    y.clone();
    println!("{:?}", y);
}
```

Такая гибкость может добавить ясности в сложных ситуациях.

`where` является более мощным инструментом, чем просто синтаксис. Например:

```
trait ConvertTo<Output> {
    fn convert(&self) -> Output;
}

impl ConvertTo<i64> for i32 {
    fn convert(&self) -> i64 { *self as i64 }
}

// can be called with T == i32
fn normal<T: ConvertTo<i64>>(x: &T) -> i64 {
    x.convert()
}

// can be called with T == i64
fn inverse<T>() -> T
        // this is using ConvertTo as if it were "ConvertFrom<i32>"
        where i32: ConvertTo<T> {
    1i32.convert()
}
```

Этот код демонстрирует дополнительные преимущества использования утверждения `where`: оно позволяет задавать ограничение, где с левой стороны располагается произвольный тип (в данном случае `i32`), а не только простой параметр типа (как например `T`).

Наш пример `inverse`

Вернемся к разделу Дженерики, мы пытались написать такой код:

```
fn inverse<T>(x: T) -> Result<T, String> {
    if x == 0.0 { return Err("x cannot be zero!".to_string()); }

    Ok(1.0 / x)
}
```

Если мы попытаемся скомпилировать его, мы получим такую ошибку:

```
error: binary operation `==` cannot be applied to type `T`
```

Все потому, что тип `T` является слишком общим: мы не можем знать наверняка, что любой случайный `T` можно сравнивать. Для конкретизации мы можем ограничить его типажом. Хотя этот код и не будет работать, но попробуйте сделать следующее:

```
fn inverse<T: PartialEq>(x: T) -> Result<T, String> {
    if x == 0.0 { return Err("x cannot be zero!".to_string()); }

    Ok(1.0 / x)
}
```

Вы получите ошибку:

```
error: mismatched types:
 expected `T`,
    found `_`
(expected type parameter,
    found floating-point variable)
```

Как уже было сказано, этот код не будет работать. Это потому, что наш `T` реализует `PartialEq`, который принимает на вход другой `T`, но, вместо этого, мы передаем переменную с плавающей точкой. Нам нужно другое ограничение. С помощью `Float` можно исправить ошибку:

```
use std::num::Float;

fn inverse<T: Float>(x: T) -> Result<T, String> {
    if x == Float::zero() { return Err("x cannot be zero!".to_string()) }

    let one: T = Float::one();
    Ok(one / x)
}
```

Нам следует заменить `0.0` и `1.0` соответствующими методами из типажа `Float`. И `f32`, и `f64` реализуют типаж `Float`, так что наша функция будет работать просто отлично:

```
println!("the inverse of {} is {:?}", 2.0f32, inverse(2.0f32));
println!("the inverse of {} is {:?}", 2.0f64, inverse(2.0f64));

println!("the inverse of {} is {:?}", 0.0f32, inverse(0.0f32));
println!("the inverse of {} is {:?}", 0.0f64, inverse(0.0f64));
```

Методы по умолчанию

Есть еще одна особенность типажей, которую мы должны охватывать: методы по умолчанию. Проще всего это показать на примере:

```
trait Foo {
    fn bar(&self);

    fn baz(&self) { println!("We called baz."); }
}
```

При реализации типажа `Foo` необходимо реализовать метод `bar()`, но нет необходимости в реализации метода `baz()`. Это поведение будет реализовано по умолчанию. По желанию, можно переопределить значение метода по умолчанию:

```
struct UseDefault;

impl Foo for UseDefault {
    fn bar(&self) { println!("We called bar."); }
}

struct OverrideDefault;

impl Foo for OverrideDefault {
    fn bar(&self) { println!("We called bar."); }

    fn baz(&self) { println!("Override baz!"); }
}

let default = UseDefault;
default.baz(); // prints "We called baz."

let over = OverrideDefault;
over.baz(); // prints "Override baz!"
```

Типаж Drop (сброс)

Мы обсудили типажи. Теперь давайте поговорим о конкретном типаже, предоставляемом стандартной библиотекой Rust. Этот типаж - `Drop` (сброс) - позволяет выполнить некоторый код, когда значение выходит из области видимости. Например:

```
struct HasDrop;

impl Drop for HasDrop {
    fn drop(&mut self) {
        println!("Сбрасываем!");
    }
}

fn main() {
    let x = HasDrop;

    // сделаем что-то

} // тут x выходит из области видимости
```

Когда `x` выходит из области видимости в конце `main()`, исполнится код реализации типажа `Drop`. У него один метод, который тоже называется `drop()`. Он принимает изменяемую ссылку на себя (`self`).

Вот и всё! Работа `Drop` достаточно проста, но есть несколько тонкостей. Например, значения сбрасываются в порядке, обратном порядку их объявления. Вот ещё пример:

```
struct Firework {
    strength: i32,
}

impl Drop for Firework {
    fn drop(&mut self) {
        println!("БАБАХ силой {}!!!", self.strength);
    }
}

fn main() {
    let firecracker = Firework { strength: 1 };
    let tnt = Firework { strength: 100 };
}
```

Этот код выведет следующее:

```
БАБАХ силой 100!!!
БАБАХ силой 1!!!
```

Сначала взрывается тринитротолуоловая бомба (`tnt`), потому что она была объявлена последней. За ней взрывается шутиха (`firecracker`). Первым вошёл, последним вышел.

Так зачем нужен `Drop`? Часто `Drop` используют, чтобы освободить ресурсы, представленные структурой (`struct`). Например, счётчик ссылок `Arc<T>` уменьшает число активных ссылок в `drop()`, и когда оно достигает нуля, освобождает хранимое значение.

if let

Иногда хочется сделать определённые вещи менее неуклюже. Например, скомбинировать `if` и `let` чтобы более удобно сделать сопоставление с образцом. Для этого есть `if let`.

В качестве примера рассмотрим `Option<T>`. Если это `Some<T>`, мы хотим вызвать функцию на этом значении, а если это `None` - не делать ничего. Вроде такого:

```
match option {
    Some(x) => { foo(x) },
    None => {},
}
```

Здесь необязательно использовать `match`. `if` тоже подойдёт:

```
if option.is_some() {
    let x = option.unwrap();
    foo(x);
}
```

Но оба этих варианта выглядят странно. Мы можем исправить это с помощью `if let`:

```
if let Some(x) = option {
    foo(x);
}
```

Если [сопоставление с образцом] успешно, имена в образце связываются с соответствующими частями разбираемого значения, и блок исполняется. Если значение не соответствует образцу, ничего не происходит.

Если вы хотите делать что-то ещё при несовпадении с образцом, используйте `else`:

```
if let Some(x) = option {
    foo(x);
} else {
    bar();
}
```

while let

Похожим образом, `while let` можно использовать для перебора значений, пока они соответствуют образцу. Код вроде такого:

```
loop {
    match option {
        Some(x) => println!("{}", x),
        _ => break,
    }
}
```

Превращается в такой:

```
while let Some(x) = option {
    println!("{}", x);
}
```

Типажи-объекты

Когда код включает в себя полиморфизм, то должен быть механизм, чтобы определить, какая конкретная версия будет фактически вызвана. Это называется 'диспетчерезация.' Есть две основные формы диспетчерезации: статическая и динамическая. Хотя Rust и отдает предпочтение статической диспетчерезации, он также поддерживает динамическую диспетчерезацию через механизм, называемый 'типажи-объекты.'

Подготовка

Для остальной части этой главы нам потребуется типаж и несколько его реализаций. Давайте создадим простой типаж `Foo`. Он содержит один метод, который возвращает `String`.

```
trait Foo {
    fn method(&self) -> String;
}
```

Также мы реализуем этот типаж для `u8` и `String`:

```
impl Foo for u8 {
    fn method(&self) -> String { format!("u8: {}", *self) }
}

impl Foo for String {
    fn method(&self) -> String { format!("string: {}", *self) }
}
```

Статическая диспетчерезация

Мы можем использовать этот типаж для выполнения статической диспетчерезации с помощью ограничения типажом:

```
fn do_something<T: Foo>(x: T) {
    x.method();
}

fn main() {
    let x = 5u8;
    let y = "Hello".to_string();

    do_something(x);
    do_something(y);
}
```

Здесь Rust использует 'мономорфизацию' для статической диспетчерезации. Это означает, что Rust создаст специальную версию `do_something()` для каждого из типов: `u8` и `String`, а затем заменит все места вызовов на вызовы этих специализированных функций. Другими словами, Rust сгенерирует нечто вроде этого:

```
fn do_something_u8(x: u8) {
    x.method();
}

fn do_something_string(x: String) {
    x.method();
}

fn main() {
    let x = 5u8;
    let y = "Hello".to_string();

    do_something_u8(x);
    do_something_string(y);
}
```

Статическая диспетчерезация имеет большой потенциал: она позволяет вызывать функцию, которая будет встроена, потому что вызываемая версия этой функции известна на этапе компиляции, а встраивание - это ключ к хорошей оптимизации. Статическая диспетчерезация быстра, но это достигается путем компромисса: происходит 'раздувание кода' в связи с большим количеством копий одной и той же функции, по одной для каждого типа, расположенных в бинарном файле.

Кроме того, компиляторы не совершенны и могут "оптимизировать" код так, что он станет медленнее. Например, встроенные функции будут слишком охотно раздувать кэш команд (правила кэширования все вокруг нас). Это одна из причин, по которой `#[inline]` и `#[inline(always)]` следует использовать осторожно, и почему использование динамической диспетчеризации иногда более эффективно.

Тем не менее, в общем случае более эффективно использовать статическую диспетчерезацию. Кроме того, всегда можно иметь тонкую статически- диспетчерезуемую обертку для функции, которая выполняет динамическую диспетчеризацию, но не наоборот. То есть статические вызовы являются более гибкими. По этой причине стандартная библиотека старается быть статически диспетчерезуемой везде, где это возможно.

Динамическая диспетчерезация

Rust обеспечивает динамическую диспетчеризацию через механизм под названием 'типажи-объекты'. Типажи-объекты, такие как `&Foo` или `Box<Foo>`, это обычные переменные, хранящие значения *любого* типа, реализующего данный типаж. Конкретный тип типажа-объекта может быть определен только на этапе выполнения.

Типаж-объект может быть получен из указателя на конкретный тип, который реализует этот типаж, путем его `явного приведения` (например, `&x as &Foo`) или `неявного приведения` (например, используя `&x` в качестве аргумента функции, которая принимает `&Foo`).

Явное и неявное приведение типажа-объекта также работает для таких указателей, как `&mut T` в `&mut Foo` и `Box<T>` в `Box<Foo>`, но это все на данный момент. Явное и неявное приведение идентичны.

Эта операция может рассматриваться как "затирание" знания компилятора о конкретном типе указателя, поэтому типажи-объекты иногда называют как "затирание типов".

Возвращаясь к примеру выше, мы можем использовать тот же самый типаж для выполнения динамической диспетчеризации с типажами-объектами путем явного приведения типа:

```
fn do_something(x: &Foo) {
    x.method();
}

fn main() {
    let x = 5u8;
    do_something(&x as &Foo);
}
```

или неявного приведения типа:

```
fn do_something(x: &Foo) {
    x.method();
}

fn main() {
    let x = "Hello".to_string();
    do_something(&x);
}
```

Функция, которая принимает типаж-объект, не обладает специализированнми копиями для каждого из типов, которые реализуют типаж `Foo`: генерируется только одна копия. Часто (но не всегда), в результате происходит уменьшение раздувания кода. Тем не менее, это происходит за счет более медленного вызова виртуальных функций, и, по существу, блокирования любой возможности встраивания и связанных с этим оптимизаций.

Почему указатели?

В отличие от многих управляемых языков, Rust по умолчанию не размещает значения по указателю, так как типы могут иметь различные размеры. Знать размер значения во время компиляции важно прежде всего для выполнения таких задач, как передача значения в качестве аргумента в функцию, что вызывает помещение переданного значения в стек, и выделение (и освобождение) места на куче для сохранения значения там.

Для `Foo` допускается иметь значение, которое может быть либо `String` (24 байт), либо `u8` (1 байт), либо любой другой тип, для которого в соответствующих крейтах может быть реализован `Foo` (возможно абсолютно любое число байт). Так как этот другой тип может быть сколь угодно большими, то нет никакого способа, гарантирующего, что последний вариант будет работать, если значения сохраняются без указателя.

Размещение значения по указателю означает, что, когда мы имеем дело с типажом-объектом, размер самого значения не важен, а важен лишь размер указателя.

Representation Представление

Методы типажа можно вызвать для типажа-объекта с помощью специальной записи указателей на функции, традиционно называемой 'виртуальная таблица' ('vtable') (создается и управляется компилятором).

Типажи-объекты являются одновременно и простыми и сложными: их основное представление и устройство довольно прямолинейно, но есть некоторые тонкости относительно обнаружения сообщений об ошибках и странного поведения.

Давайте начнем с простого, с рантайм представления типажа-объекта. Модуль `std::raw` содержит структуры с макетами, которые являются такими же, как и сложные встроенные типы, в том числе типажи-объекты:

```
pub struct TraitObject {
    pub data: *mut (),
    pub vtable: *mut (),
}
```

То есть типаж-объект, такой как `&Foo`, состоит из указателя на "данные" и указателя на "виртуальную таблицу".

Указатель `data` адрессует данные (какого-то неизвестного типа `T`), которые хранит типаж-объект, а указатель `vtable` указывает на виртуальную таблицу ("таблица виртуальных методов"), которая соответствует реализации `Foo` для `T`.

По существу, виртуальная таблица - это структура указателей на функции, указывающих на конкретный кусок машинного кода для каждого метода в реализации. Вызов метода наподобие `trait_object.method()` возвращает правильный указатель из виртуальной таблицы, а затем динамически вызывает метод по этому указателю. Например:

```
struct FooVtable {
    destructor: fn(*mut ()),
    size: usize,
    align: usize,
    method: fn(*const ()) -> String,
}

// u8:

fn call_method_on_u8(x: *const ()) -> String {
    // the compiler guarantees that this function is only called
    // with `x` pointing to a u8
    let byte: &u8 = unsafe { &*(x as *const u8) };

    byte.method()
}

static Foo_for_u8_vtable: FooVtable = FooVtable {
    destructor: /* compiler magic */,
    size: 1,
    align: 1,

    // cast to a function pointer
    method: call_method_on_u8 as fn(*const ()) -> String,
};


// String:

fn call_method_on_String(x: *const ()) -> String {
    // the compiler guarantees that this function is only called
    // with `x` pointing to a String
    let string: &String = unsafe { &*(x as *const String) };

    string.method()
}

static Foo_for_String_vtable: FooVtable = FooVtable {
    destructor: /* compiler magic */,
    // values for a 64-bit computer, halve them for 32-bit ones
    size: 24,
    align: 8,

    method: call_method_on_String as fn(*const ()) -> String,
};
```

Поле `destructor` в каждой виртуальной таблице указывает на функцию, которая будет очищать любые ресурсы типа этой виртуальной таблицы, для `u8` она тривиальна, но для `String` она будет освобождать память. Это необходимо для владельцев типажей-объектов, таких как `Box<Foo>`, для которых необходимо очищать выделенную память как для `Box`, так и для внутреннего типа, когда они выходят из области видимости. Поля `size` и `align` хранят

размер затёртого типа, и его требования к выравниванию; по существу, они не использовался в момент, так как информация встроенного в деструктор, но будет использоваться в будущем, так как объекты отличительным признакам постепенно становится более гибким.

Предположим, у нас есть несколько значений, которые реализуют `Foo`, тогда явный вид создания и использования типажей-объектов `Foo` может выглядеть примерно как (игнорируются несоответствия типов: в любом случае, они всего лишь указатели):

```
let a: String = "foo".to_string();
let x: u8 = 1;

// let b: &Foo = &a;
let b = TraitObject {
    // store the data
    data: &a,
    // store the methods
    vtable: &Foo_for_String_vtable
};

// let y: &Foo = x;
let y = TraitObject {
    // store the data
    data: &x,
    // store the methods
    vtable: &Foo_for_u8_vtable
};

// b.method();
(b.vtable.method)(b.data);

// y.method();
(y.vtable.method)(y.data);
```

Если `b` или `y` были владельцами типажей-объектов (`Box<Foo>`), то будут вызываны деструкторы `(b.vtable.destructor)(b.data)` или `(y.vtable.destructor)(y.data)` соответственно, как только они выйдут из своей области определения.

Замыкания

Помимо именованных функций Rust предоставляет еще и анонимные функции. Анонимные функции, которые имеют связанное окружение, называются 'замыкания'. Они так называются потому что они замыкают свое окружение. Как мы увидим далее, Rust имеет реально крутую реализацию замыканий.

Синтаксис

Замыкания выглядят следующим образом:

```
let plus_one = |x: i32| x + 1;

assert_eq!(2, plus_one(1));
```

Мы создаем связывание, `plus_one`, и присваиваем ему замыкание. Аргументы замыкания располагаются между двумя символами `|`, а телом замыкания является выражение, в данном случае: `x + 1`. Помните, что `{ }` также является выражением, поэтому тело замыкания может содержать много строк:

```
let plus_two = |x| {
    let mut result: i32 = x;

    result += 1;
    result += 1;

    result
};

assert_eq!(4, plus_two(2));
```

Обратите внимание, что есть несколько небольших различий между замыканиями и обычными функциями, определенными с помощью `fn`. Первое отличие состоит в том, что для замыкания мы не должны указывать ни типы аргументов, которые оно принимает, ни тип возвращаемого им значения. Мы можем:

```
let plus_one = |x: i32| -> i32 { x + 1 };

assert_eq!(2, plus_one(1));
```

Но мы не должны. Почему так? В основном, это было сделано из эргономических соображений (соображений удобства). В то время как для именованных функций явное указание типа является полезным для таких аспектов как документация и вывод типа, типы замыканий редко документируют, поскольку они анонимны. К тому же, они не вызывают "ошибок на расстоянии" (error-at-a-distance), которые могут вызывать именованные функции. Такие ошибки могут возникать, когда локальное изменение (например, в теле одной из функций) вызывает изменение вывода типов. Компилятор пытается подобрать типы в

окружающей программе под уже другие типы в изменённой функции, и часто оказывается, что имена имеют другие типы, нежели мы ожидали. В результате происходит ошибка "на расстоянии" - возможно, в другой функции, использующей изменённую.

Второе отличие - синтаксис очень похож, но все же немного отличается. Я добавил пробелы здесь, чтобы было более наглядно:

```
fn  plus_one_v1   (x: i32 ) -> i32 { x + 1 }
let plus_one_v2 = |x: i32 | -> i32 { x + 1 };
let plus_one_v3 = |x: i32 |          x + 1  ;
```

Есть небольшие различия, но принцип аналогичен.

Замыкания и их окружение

Замыкания называются так потому, что они 'замыкают свое окружение.' Это выглядит следующим образом:

```
let num = 5;
let plus_num = |x: i32| x + num;

assert_eq!(10, plus_num(5));
```

Это замыкание, `plus_num`, ссылается на связанную с помощью оператора `let` переменную `num`, расположенную в своей области видимости. Если говорить более конкретно, то оно заимствует связывание. Если мы сделаем что-то, что противоречило бы связыванию, то получим ошибку. Например этот код:

```
let mut num = 5;
let plus_num = |x: i32| x + num;

let y = &mut num;
```

Который выдаст следующие ошибки:

```
error: cannot borrow `num` as mutable because it is also borrowed as immutable
    let y = &mut num;
                ^~~
note: previous borrow of `num` occurs here due to use in closure; the immutable
  borrow prevents subsequent moves or mutable borrows of `num` until the borrow
  ends
    let plus_num = |x| x + num;
                   ^~~~~~~~~~
note: previous borrow ends here
fn main() {
    let mut num = 5;
    let plus_num = |x| x + num;

    let y = &mut num;
}
^
```

Подробное и к тому же полезное сообщение об ошибке! Как говорится в этом сообщении, мы не можем получить изменяемый заем переменной `num` потому что замыкание уже заимствует его. Если же мы обеспечим выход замыкания из области видимости, то мы сможем:

```
let mut num = 5;
{
    let plus_num = |x: i32| x + num;

} // plus_num goes out of scope, borrow of num ends

let y = &mut num;
```

Однако, Rust также может забирать право владения и перемещать свое окружение, если этого требует замыкание:

```
let nums = vec![1, 2, 3];

let takes_nums = || nums;

println!("{:?}", nums);
```

Этот код выдаст:

```
note: `nums` moved into closure environment here because it has type
  `[closure(()) -> collections::vec::Vec<i32>]`, which is non-copyable
let takes_nums = || nums;
                    ^~~~~~
```

`Vec<T>` обладает правом владения на свое содержимое, и поэтому, когда мы ссылаемся на него в нашем замыкании, мы должны забрать право владения на `nums`. Это тоже самое, как если бы мы передавали `nums` в функцию, которая забирала бы право владения на него.

`move` замыкания

Мы можем заставить наше замыкание забирать право владения на свое окружение с помошью ключевого слова `move`:

```
let num = 5;

let owns_num = move |x: i32| x + num;
```

Теперь, когда указано ключевое слово `move`, переменные следуют нормальной семантике перемещения. В данном примере `5` реализует `Copy`, поэтому `owns_num` становится владельцем копии `num`. Так в чем же разница?

```
let mut num = 5;

{
    let mut add_num = |x: i32| num += x;

    add_num(5);
}

assert_eq!(10, num);
```

Итак, в этом примере наше замыкание принимает изменяемую ссылку на `num`. Затем, когда мы вызываем замыкание `add_num`, то, как мы и ожидали, оно изменяет значение внутри. Нам также необходимо объявить `owns_num` как `mut`, потому что оно изменяет свое окружение.

Если же мы будем использовать `move` замыкание, то получим следующие отличия:

```
let mut num = 5;

{
    let mut add_num = move |x: i32| num += x;

    add_num(5);
}

assert_eq!(5, num);
```

Мы всего лишь получаем `5`. Вместо того, чтобы получать изменяемый заем на `num`, мы получаем право владения на копию.

Вот еще один способ думать о `move` замыканиях: они предоставляют замыкание со своим собственным фреймом стека. Без `move` замыкание может быть связано с фреймом стека, который его создал, в то время как `move` замыкание содержит свой собственный фрейм стека. Это означает, например, что вы не можете вернуть не `move` замыкание из функции.

Но прежде чем говорить о получении в качестве аргумента и возвращении замыкания, мы должны поговорить о том, как реализуются замыкания. Как системный язык программирования, Rust дает вам кучу контроля над тем, что делает ваш код, и замыкания не являются исключением.

Реализация замыканий

Реализация замыканий в Rust немного отличается от других языков. Фактически, она представляет из себя просто синтаксисический сахар для типажей. Перед тем как читать дальше, настоятельно рекомендуем изучить главу Типажи, а также главу Типажи-объекты, в которой говорится о типажах-объектах.

Изучили? Хорошо.

Ключ к пониманию того, как замыкания работают изнутри звучит немного странно: использование `()` для вызова функции, как например `foo()`, представляет собой перегружаемую операцию. Исходя из этого, все остальное встает на свои места. В Rust мы используем систему типажей для перегрузки операций. Вызов функций не является исключением. Существуют три отдельных типажа для их перегрузки:

```
pub trait Fn<Args> : FnMut<Args> {
    extern "rust-call" fn call(&self, args: Args) -> Self::Output;
}

pub trait FnMut<Args> : FnOnce<Args> {
    extern "rust-call" fn call_mut(&mut self, args: Args) -> Self::Output;
}

pub trait FnOnce<Args> {
    type Output;

    extern "rust-call" fn call_once(self, args: Args) -> Self::Output;
}
```

Вы можете заметить некоторые различия между этими типажами, но есть одно главное различие - `self`: `Fn` принимает `&self`, `FnMut` принимает `&mut self`, `FnOnce` принимает `self`. Это покрывает все три вида `self` с помощью обычного синтаксиса вызова методов. Мы разделили их на три типажа, вместо того, чтобы иметь один. Это дает нам большее количество контроля над тем, какого вида замыкания мы можем принять.

Использование `|| {}` при создании замыканий является синтаксическим сахаром для этих трех типажей. Rust будет генерировать структуру для окружения, реализующую (`impl`) соответствующий типаж, а затем использовать его.

Передача замыканий в качестве аргументов

Теперь, когда мы знаем, что замыкания являются типажами, получается, что мы уже знаем, как принимать и возвращать замыкания: как и любой другой типаж!

Это также означает, что мы можем выбирать между статической и динамической диспетчеризацией. Во-первых, давайте напишем функцию, которая принимает что-то вызываемое, вызывает это что-то и возвращает результат:

```
fn call_with_one<F>(some_closure: F) -> i32
    where F : Fn(i32) -> i32 {

    some_closure(1)
}

let answer = call_with_one(|x| x + 2);

assert_eq!(3, answer);
```

Мы передаем наше замыкание `|x| x + 2`, в функцию `call_with_one`. Она же делает то, о чем говорит ее название: вызывает замыкание, передавая ему `1` в качестве аргумента.

Давайте рассмотрим сигнатуру функции `call_with_one` более подробно:

```
fn call_with_one<F>(some_closure: F) -> i32
```

Мы принимаем один параметр, который имеет тип `F`. Мы также возвращаем `i32`. Эта часть не интересна. Следующим важным моментом является:

```
    where F : Fn(i32) -> i32 {
```

Так как `Fn` является типажом, мы можем связать с ним наш дженерик (обобщенный) параметр. В этом примере, замыкание принимает `i32` в качестве аргумента и возвращает `i32`, поэтому дженерик привязка, которую мы используем, выглядит так: `Fn(i32) -> i32`.

Здесь есть еще один ключевой момент: так как мы ограничиваем дженерик параметр с помощью типажа, то будет применена мономорфизация, и поэтому в замыкании будет использоваться статическая диспетчеризация. Это довольно лаконично (аккуратно). Во многих языках для замыканий по существу используется выделение памяти в куче, и поэтому всегда будет использоваться динамическая диспетчеризация. В Rust мы можем выделить память для окружения замыкания в стеке и использовать статическую диспетчеризацию вызова. Это случается довольно часто с итераторами и их адаптерами, которые нередко принимают замыкания в качестве аргументов.

Конечно, если нам нужна динамическая диспетчеризация, мы также можем использовать и ее. Обычно для этого случая используется типаж-объект:

```
fn call_with_one(some_closure: &Fn(i32) -> i32) -> i32 {
    some_closure(1)
}

let answer = call_with_one(&|x| x + 2);

assert_eq!(3, answer);
```

Теперь наша функция в качетве аргумента принимает типаж-объект `&Fn`. Поэтому мы должны создать ссылку на замыкание а затем передать ее в функцию `call_with_one`, для этого мы используем `&||`.

Возврат замыканий

Что очень характерно для кода в функциональном стиле - возвращать замыкания в различных ситуациях. Если вы попытаетесь вернуть замыкание, то можете столкнуться с ошибкой. Сперва это может показаться странным, но мы с этим разберемся. Вот как вы, наверное, попытаетесь вернуть замыкание из функции:

```
fn factory() -> (Fn(i32) -> Vec<i32>) {
    let vec = vec![1, 2, 3];

    |n| vec.push(n)
}

let f = factory();

let answer = f(4);
assert_eq!(vec![1, 2, 3, 4], answer);
```

Это выдаст следующие длинные, взаимосвязанные ошибки:

```
error: the trait `core::marker::Sized` is not implemented for the type
`core::ops::Fn(i32) -> collections::vec::Vec<i32>` [E0277]
f = factory();
^
note: `core::ops::Fn(i32) -> collections::vec::Vec<i32>` does not have a
constant size known at compile-time
f = factory();
^
error: the trait `core::marker::Sized` is not implemented for the type
`core::ops::Fn(i32) -> collections::vec::Vec<i32>` [E0277]
factory() -> (Fn(i32) -> Vec<i32>) {
             ^~~~~~~~~~~~~~~~~~~~
note: `core::ops::Fn(i32) -> collections::vec::Vec<i32>` does not have a constant size know
n at compile-time
fa ctory() -> (Fn(i32) -> Vec<i32>) {
              ^~~~~~~~~~~~~~~~~~~~
```

Для того чтобы вернуть что-то из функции, Rust должен знать, какой размер имеет тип возвращаемого значения. Но так как `Fn` является типажом, то в качестве него могут выступать совершенно разные объекты, с разными размерами: много различных типов могут реализовать `Fn`. Самый простой способ передать что-то неопределенного размера - передать ссылку на это что-то, так как ссылки имеют известный размер. Таким образом, следовало бы написать так:

```
fn factory() -> &(Fn(i32) -> Vec<i32>) {
    let vec = vec![1, 2, 3];

    |n| vec.push(n)
}

let f = factory();

let answer = f(4);
assert_eq!(vec![1, 2, 3, 4], answer);
```

Но тогда мы получим другую ошибку:

```
error: missing lifetime specifier [E0106]
fn factory() -> &(Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~~
```

Верно. Так как у нас используется ссылка, то мы должны задать ее время жизни. Так наша функция `factory()` не принимает никаких аргументов, то элизия (сокрытие) здесь не уместна. Какое время жизни мы должны выбрать? `'static`:

```
fn factory() -> &'static (Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

Но мы получим еще ошибку:

```
error: mismatched types:
 expected `&'static core::ops::Fn(i32) -> i32`,
    found `[closure <anon>:7:9: 7:20]`
(expected &-ptr,
    found closure) [E0308]
         |x| x + num
         ^~~~~~~~~~
```

Эта ошибка сообщает нам, что ожидается использование `&'static Fn(i32) -> i32`, а используется `[closure <anon>:7:9: 7:20]`. Подождите, что?

Поскольку каждое замыкание (в индивидуальном порядке) генерирует свою собственную `struct` для окружения и реализует `Fn` и компанию, то эти типы являются анонимными. Они существуют исключительно для этого замыкания. Поэтому Rust показывает их как `closure <anon>`, а не в виде какого-то автоматически сгенерированного имени.

Но почему же наше замыкание не реализует `&'static Fn`? Как мы обсуждали ранее, замыкание заимствует свое окружение. И в этом случае наше окружение представляет собой выделеную в стеке память, содержащую значение связанной переменной `num - 5`. Из-за этого заем имеет срок жизни фрейма стека. Так что, когда мы вернем это замыкание, то вызов функции будет завершен, а фрейм стека уйдет, и наше замыкание захватит окружение, содержащее в памяти мусор!

Так что же делать? Этот код *почти* работает:

```
fn factory() -> Box<Fn(i32) -> i32> {
    let num = 5;

    Box::new(|x| x + num)
}
let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

Мы используем типаж-объект, полученный в результате упаковки (`Box`) типажа `Fn`. И остаётся только одна, последняя проблема:

```
error: `num` does not live long enough
Box::new(|x| x + num)
         ^~~~~~~~~~
```

Мы все еще по-прежнему ссылаемся на родительский фрейм стека. С этии последним исправлением мы сможем наконец выполнить нашу задачу:

```
fn factory() -> Box<Fn(i32) -> i32> {
    let num = 5;

    Box::new(move |x| x + num)
}
let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

Благодаря изменению внутреннего замыкания на `move Fn` будет создаваться новый фрейм стека для нашего замыкания. А благодаря упаковке (`Box`) замыкания, получается известный размер возвращаемого значения, и позволяет ему избежать (быть независимым от) нашего фрейма стека.

Универсальный синтаксис вызова функций (Universal Function Call Syntax)

Иногда, функции могут иметь одинаковые имена. Рассмотрим этот код:

```
trait Foo {
    fn f(&self);
}

trait Bar {
    fn f(&self);
}

struct Baz;

impl Foo for Baz {
    fn f(&self) { println!("Baz's impl of Foo"); }
}

impl Bar for Baz {
    fn f(&self) { println!("Baz's impl of Bar"); }
}

let b = Baz;
```

Если мы попытаемся вызвать `b.f()`, то получим ошибку:

```
error: multiple applicable methods in scope [E0034]
b.f();
  ^~~
note: candidate #1 is defined in an impl of the trait `main::Foo` for the type
`main::Baz`
    fn f(&self) { println!("Baz's impl of Foo"); }
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
note: candidate #2 is defined in an impl of the trait `main::Bar` for the type
`main::Baz`
    fn f(&self) { println!("Baz's impl of Bar"); }
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Нам нужен способ указать, какой конкретно метод нужен, чтобы устранить неоднозначность. Эта фича называется 'универсальный синтаксис вызова функций', и выглядит это так:

```
Foo::f(&b);
Bar::f(&b);
```

Давайте разберемся.

```
Foo::
Bar::
```

Эти части вызова задают один из двух видов типажей: `Foo` и `Bar`. Это то, что на самом деле устраняет неоднозначность между двумя методами: Rust вызывает метод того типажа, имя которого вы используете.

```
f(&b)
```

Когда мы вызываем метод, используя синтаксис вызова метода, как например `b.f()`, Rust автоматически заимствует `b`, если `f()` принимает в качестве аргумента `&self`. В этом же случае, Rust не будет использовать автоматическое заимствование, и поэтому мы должны явно передать `&b`.

Форма с угловыми скобками

Форма UFCS, о которой мы только что говорили:

```
Trait::method(args);
```

Это сокращенная форма записи. Ниже представлена расширенная форма записи, которая требуется в некоторых ситуациях:

```
<Type as Trait>::method(args);
```

Синтаксис `<>::` является средством предоставления подсказки типа. Тип располагается внутри `<>`. В этом случае типом является `Type as Trait`, указывающий, что мы хотим здесь вызвать `Trait` версию метода. Часть `as Trait` является необязательной, если вызов не является неоднозначным. То же самое что с угловыми скобками, отсюда и короткая форма.

Вот пример использования длинной формы записи.

```
trait Foo {
    fn clone(&self);
}

#[derive(Clone)]
struct Bar;

impl Foo for Bar {
    fn clone(&self) {
        println!("Making a clone of Bar");

        <Bar as Clone>::clone(self);
    }
}
```

Этот код вызвает метод `clone()` типажа `Clone`, а не типажа `Foo`.

Контейнеры (crates) и модули (modules)

Когда проект начинает разрастаться, то хорошей практикой разработки программного обеспечения считается: разбить его на небольшие кусочки, а затем собрать их вместе. Также важно иметь четко определенный интерфейс, так как часть вашей функциональности является приватной, а часть - публичной. Для облегчения такого рода вещей Rust обладает модульной системой.

Основные термины: контейнеры и модули

Rust имеет два различных термина, которые относятся к модульной системе: *контейнер* и *модуль*. Контейнер - это синоним *библиотеки* или *пакета* на других языках. Именно поэтому инструмент управления пакетами в Rust называется "Cargo": вы пересылаете ваши контейнеры другим с помошью Cargo. Контейнеры могут производить исполняемый файл или библиотеку, в зависимости от проекта.

Каждый контейнер имеет неявный *корневой модуль*, содержащий код для этого контейнер. В рамках этого базового модуля можно определить дерево суб-модулей. Модули позволяют разделить ваш код внутри контейнера.

В качестве примера, давайте сделаем контейнер *phrases*, который выдает нам различные фразы на разных языках. Чтобы не усложнять пример, мы будем использовать два вида фраз: "greetings" и "farewells", и два языка для этих фраз: английский и японский ( ). Мы будем использовать следующий шаблон модуля:

```
                                +-----------+
                            +---| greetings |
                            |   +-----------+
                +---------+ |
            +---| english |---+
            |   +---------+ |   +-----------+
            |               +---| farewells |
+---------+ |                   +-----------+
| phrases |---+
+---------+ |                   +-----------+
            |               +---| greetings |
            |   +----------+|   +-----------+
            +---| japanese |--+
                +----------+|
                            |   +-----------+
                            +---| farewells |
                                +-----------+
```

В этом примере, `phrases` - это название нашего контейнера. Все остальное - модули. Вы можете видеть, что они образуют дерево, в основании которого располагается *корень* контейнера - `phrases`.

Теперь, когда у нас есть схема, давайте определим модули в коде. Для начала создайте новый контейнер с помощью Cargo:

```
$ cargo new phrases
$ cd phrases
```

Если вы помните, то эта команда создает простой проект:

```
$ tree .
.
├── Cargo.toml
└── src
    └── lib.rs

1 directory, 2 files
```

`src/lib.rs` - корень нашего контейнера, соответствующий `phrases` в нашей диаграмме выше.

Объявление модулей

Для объявления каждого из наших модулей, мы используем ключевое слово `mod`. Давайте сделаем, чтобы наш `src/lib.rs` выглядел следующим образом:

```
mod english {
    mod greetings {
    }

    mod farewells {
    }
}

mod japanese {
    mod greetings {
    }

    mod farewells {
    }
}
```

После ключевого слова `mod`, вы задаете имя модуля. Имена модулей следуют соглашениям, как и другие идентификаторы Rust: `lower_snake_case`. Содержание каждого модуля обрамляется в фигурные скобки (`{}`).

Внутри `mod` вы можете объявить суб-`mod`. Мы можем обращаться к суб-модулям с помощью нотации (`::`). Так выглядят обращения к нашим четырем вложенным модулям: `english::greetings`, `english::farewells`, `japanese::greetings` и `japanese::farewells`. Так как суб-модули располагаются в пространстве имен своих родительских модулей, то суб-модули `english::greetings` и `japanese::greetings` не конфликтуют, несмотря на то, что они имеют одинаковые имена, `greetings`.

Так как в этом контейнере нет функции `main()`, и называется он `lib.rs`, Cargo соберет этот контейнер в виде библиотеки:

```
$ cargo build
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
$ ls target/debug
build  deps  examples  libphrases-a7448e02a0468eaa.rlib  native
```

`libphrase-hash.rlib` - это скомпилированный контейнер. Прежде чем мы рассмотрим, как его можно использовать из другого контейнера, давайте разобьем его на несколько файлов.

Контейнеры с несколькими файлами

Если бы каждый контейнер мог состоять только из одного файла, тогда этот файл был бы очень большими. Зачастую легче разделить контейнер на несколько файлов, и Rust поддерживает это двумя способами.

Вместо объявления модуля наподобие:

```
mod english {
    // contents of our module go here
}
```

Мы можем объявить наш модуль в виде:

```
mod english;
```

Если мы это сделаем, то Rust будет ожидать, что найдет либо файл `english.rs`, либо файл `english/mod.rs` с содержимым нашего модуля.

Обратите внимание, что в этих файлах вам не требуется заново объявлять модуль: это уже сделано при изначальном объявлении `mod`.

С помощью этих двух приемов мы можем разбить наш контейнер на две директории и семь файлов:

```
$ tree .
.
├── Cargo.lock
├── Cargo.toml
├── src
│   ├── english
│   │   ├── farewells.rs
│   │   ├── greetings.rs
│   │   └── mod.rs
│   ├── japanese
│   │   ├── farewells.rs
│   │   ├── greetings.rs
│   │   └── mod.rs
│   └── lib.rs
└── target
    └── debug
        ├── build
        ├── deps
        ├── examples
        ├── libphrases-a7448e02a0468eaa.rlib
        └── native
```

`src/lib.rs` - корень нашего контейнера, и выглядит он следующим образом:

```
mod english;
mod japanese;
```

Эти два объявления информируют Rust, что следует искать: `src/english.rs` или `src/english/mod.rs`, `src/japanese.rs` или `src/japanese/mod.rs`, в зависимости от нашей структуры. В данном примере мы выбрали второй вариант из-за того, что наши модули содержат суб-модули. И `src/english/mod.rs` и `src/japanese/mod.rs` выглядят следующим образом:

```
mod greetings;
mod farewells;
```

В свою очередь, эти объявления информируют Rust, что следует искать: `src/english/greetings.rs`, `src/japanese/greetings.rs`, `src/english/farewells.rs`, `src/japanese/farewells.rs` или `src/english/greetings/mod.rs`, `src/japanese/greetings/mod.rs`, `src/english/farewells/mod.rs`, `src/japanese/farewells/mod.rs`. Так как эти суб-модули не содержат свои собственные суб-модули, то мы выбрали `src/english/greetings.rs` и `src/japanese/farewells.rs`. Вот так!

Содержание `src/english/greetings.rs` и `src/japanese/farewells.rs` являются пустыми на данный момент. Давайте добавим несколько функций.

Поместите следующий код в `src/english/greetings.rs`:

```
fn hello() -> String {
    "Hello!".to_string()
}
```

Следующий код в `src/english/farewells.rs`:

```
fn goodbye() -> String {
    "Goodbye.".to_string()
}
```

Следующий код в `src/japanese/greetings.rs`:

```
fn hello() -> String {
    "□□□□□".to_string()
}
```

Конечно, вы можете скопировать и вставить этот код с этой страницы, или просто напечатать что-нибудь еще. Вам совершенно не обязательно знать, что на японском языке написано "Konnichiwa", чтобы понять как работает модульная система.

Поместите следующий код в `src/japanese/farewells.rs`:

```
fn goodbye() -> String {
    "□□□□□".to_string()
}
```

(Это "Sayonara", если вам интересно.)

Теперь у нас есть некоторая функциональность в нашем контейнере, давайте попробуем использовать его из другого контейнера.

Импорт внешних контейнеров

У нас есть библиотечный контейнер. Давайте создадим исполняемый контейнер, который импортирует и использует нашу библиотеку.

Создайте файл `src/main.rs` и положите в него следующее: (при этом он не будет компилироваться)

```
extern crate phrases;

fn main() {
    println!("Hello in English: {}", phrases::english::greetings::hello());
    println!("Goodbye in English: {}", phrases::english::farewells::goodbye());

    println!("Hello in Japanese: {}", phrases::japanese::greetings::hello());
    println!("Goodbye in Japanese: {}", phrases::japanese::farewells::goodbye());
}
```

Объявление `extern crate` информирует Rust о том, что для компиляции и компоновки кода нам нужен контейнер `phrases`. После этого объявление мы можем использовать модули контейнера `phrases`. Как мы уже упоминали ранее, вы можете использовать два подряд идущих символа двоеточия для обращения к суб-модулям и функциям внутри них.

Кроме того, Cargo предполагает, что `src/main.rs` - это корень бинарного, а не библиотечного контейнера. Теперь наш пакет содержит два контейнера: `src/lib.rs` и `src/main.rs`. Этот шаблон является довольно распространенным для исполняемых контейнеров: основная функциональность сосредоточена в библиотечном контейнере, а исполняемый контейнер использует эту библиотеку. Таким образом, другие программы также могут использовать библиотечный контейнер, к тому же такой подход обеспечивает отделение интереса (разделение функциональности).

Хотя этот код все еще не работает. Мы получаем четыре ошибки, которые выглядят примерно так:

```
$ cargo build
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/main.rs:4:38: 4:72 error: function `hello` is private
src/main.rs:4     println!("Hello in English: {}", phrases::english::greetings::hello());
                                                    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
note: in expansion of format_args!
<std macros>:2:25: 2:58 note: expansion site
<std macros>:1:1: 2:62 note: in expansion of print!
<std macros>:3:1: 3:54 note: expansion site
<std macros>:1:1: 3:58 note: in expansion of println!
phrases/src/main.rs:4:5: 4:76 note: expansion site
```

По умолчанию все элементы в Rust являются приватными. Давайте поговорим об этом более подробно.

Экспорт публичных интерфейсов

Rust позволяет точно контролировать, какие элементы вашего интерфейса являются публичными, и поэтому по умолчанию все элементы являются приватными. Чтобы сделать элементы публичными, вы используете ключевое слово `pub`. Давайте сначала сосредоточимся на модуле `english`, для чего сократим файл `src/main.rs` до этого:

```
extern crate phrases;

fn main() {
    println!("Hello in English: {}", phrases::english::greetings::hello());
    println!("Goodbye in English: {}", phrases::english::farewells::goodbye());
}
```

В файле `src/lib.rs` в объявлении модуля `english` давайте добавим модификатор `pub`:

```
pub mod english;
mod japanese;
```

В файле `src/english/mod.rs` давайте сделаем оба модуля с модификатором `pub`:

```
pub mod greetings;
pub mod farewells;
```

В файле `src/english/greetings.rs` давайте добавим модификатор `pub` к объявлению нашей функции `fn`:

```
pub fn hello() -> String {
    "Hello!".to_string()
}
```

А также в файле `src/english/farewells.rs`:

```
pub fn goodbye() -> String {
    "Goodbye.".to_string()
}
```

Теперь наши контейнеры компилируются, хотя и с предупреждениями о том, что функции в модуле `japanese` не используются:

```
$ cargo run
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/japanese/greetings.rs:1:1: 3:2 warning: function is never used: `hello`, #[warn(dead_co
de)] on by default
src/japanese/greetings.rs:1 fn hello() -> String {
src/japanese/greetings.rs:2     "□□□□□".to_string()
src/japanese/greetings.rs:3 }
src/japanese/farewells.rs:1:1: 3:2 warning: function is never used: `goodbye`, #[warn(dead_
code)] on by default
src/japanese/farewells.rs:1 fn goodbye() -> String {
src/japanese/farewells.rs:2     "□□□□□".to_string()
src/japanese/farewells.rs:3 }
     Running `target/debug/phrases`
Hello in English: Hello!
Goodbye in English: Goodbye.
```

Теперь, когда функции являются публичными, мы можем их использовать. Отлично! Тем не менее, написание `phrases::english::greetings::hello()` является очень длинным и неудобным. Rust предоставляет другое ключевое слово, для импорта имен в текущую область, чтобы для обращения можно было использовать короткие имена. Давайте поговорим об этом ключевом слове, `use`.

Импорт модулей с помощью `use`

Rust предоставляет ключевое слово `use`, которое позволяет импортировать имена в нашу локальную область видимости. Давайте изменим файл `src/main.rs`, чтобы он выглядел следующим образом:

```
extern crate phrases;

use phrases::english::greetings;
use phrases::english::farewells;

fn main() {
    println!("Hello in English: {}", greetings::hello());
    println!("Goodbye in English: {}", farewells::goodbye());
}
```

Две строки, начинающиеся с `use`, импортируют соответствующие модули в локальную область видимости, поэтому мы можем обратиться к функциям по гораздо более коротким именам. По соглашению, при импорте функции, лучшей практикой считается импортировать модуль, а не функцию непосредственно. Другими словами, вы *могли бы* сделать следующее:

```
extern crate phrases;

use phrases::english::greetings::hello;
use phrases::english::farewells::goodbye;

fn main() {
    println!("Hello in English: {}", hello());
    println!("Goodbye in English: {}", goodbye());
}
```

Но такой подход не является идиоматическим. Он значительно чаще приводит к конфликту имен. Для нашей короткой программы это не так важно, но, как только программа разрастается, это становится проблемой. Если у нас возникает конфликт имен, то Rust выдает ошибку компиляции. Например, если мы сделаем функции `japanese` публичными, и пытаемся скомпилировать этот код:

```
extern crate phrases;

use phrases::english::greetings::hello;
use phrases::japanese::greetings::hello;

fn main() {
    println!("Hello in English: {}", hello());
    println!("Hello in Japanese: {}", hello());
}
```

Rust выдаст нам сообщение об ошибке во время компиляции:

```
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/main.rs:4:5: 4:40 error: a value named `hello` has already been imported in this module
 [E0252]
src/main.rs:4 use phrases::japanese::greetings::hello;
                  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
error: aborting due to previous error
Could not compile `phrases`.
```

Если мы импортируем несколько имен из одного модуля, то нам совсем не обязательно писать одно и то же много раз. Вместо этого кода:

```
use phrases::english::greetings;
use phrases::english::farewells;
```

Вы можете использовать сокращение:

```
use phrases::english::{greetings, farewells};
```

Реэкспорт с помощью `pub use`

Вы можете использовать `use` не просто для сокращения идентификаторов. Вы также можете использовать его внутри вашего контейнера, чтобы реэкспортировать функцию из другого модуля. Это позволяет представить внешний интерфейс, который может не напрямую отображать внутреннюю организацию кода.

Давайте посмотрим на примере. Измените файл `src/main.rs` следующим образом:

```
extern crate phrases;

use phrases::english::{greetings,farewells};
use phrases::japanese;

fn main() {
    println!("Hello in English: {}", greetings::hello());
    println!("Goodbye in English: {}", farewells::goodbye());

    println!("Hello in Japanese: {}", japanese::hello());
    println!("Goodbye in Japanese: {}", japanese::goodbye());
}
```

Затем измените файл `src/lib.rs`, чтобы сделать модуль `japanese` с публичным:

```
pub mod english;
pub mod japanese;
```

Далее, убедитесь, что обе функции публичные, сперва в `src/japanese/greetings.rs`:

```
pub fn hello() -> String {
    "□□□□□".to_string()
}
```

А затем в `src/japanese/farewells.rs`:

```
pub fn goodbye() -> String {
    "□□□□□".to_string()
}
```

Наконец, измените файл `src/japanese/mod.rs` вот так:

```
pub use self::greetings::hello;
pub use self::farewells::goodbye;

mod greetings;
mod farewells;
```

Объявление `pub use` привносит указанную функцию в эту часть области видимости нашей модулной иерархии. Так как мы использовали `pub use` внутри нашего модуля `japanese`, то теперь мы можем вызывать функцию `phrases::japanese::hello()` и функцию `phrases::japanese::goodbye()`, хотя код для них расположен в `phrases::japanese::greetings::hello()` и `phrases::japanese::farewells::goodbye()` соответственно. Наша внутренняя организация не определяет наш внешний интерфейс.

В этом примере мы используем `pub use` отдельно для каждой функции, которую хотим привнести в область `japanese`. В качестве альтернативы, мы могли бы использовать шаблонный синтаксис, чтобы включать в себя все элементы из модуля `greetings` в текущую область: `pub use self::greetings::*`.

Что можно сказать о `self`? По умолчанию объявления `use` используют абсолютные пути, начинающиеся с корня контейнера. `self`, напротив, формирует эти пути относительно текущего места в иерархии. У `use` есть еще одна особая форма: вы можете использовать `use super::`, чтобы подняться по дереву на один уровень вверх от вашего текущего местоположения. Некоторые предпочитают думать о `self` как о `.`, а о `super` как о `..`, что для многих командных оболочек является представлением для текущей директории и для родительской директории соответственно.

Вне `use`, пути относительны: `foo::bar()` ссылаться на функцию внутри `foo` относительно того, где мы находимся. Если же используется префикс `::`, то `::foo::bar()` будет ссылаться на другой `foo`, абсолютный путь относительно корня контейнера.

Кроме того, обратите внимание, что мы использовали `pub use` прежде, чем объявили наши модули с помощью `mod`. Rust требует, чтобы объявления `use` шли в первую очередь.

Следующий код собирается и работает:

```
$ cargo run
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
     Running `target/debug/phrases`
Hello in English: Hello!
Goodbye in English: Goodbye.
Hello in Japanese: □□□□□
Goodbye in Japanese: □□□□□
```

`const` and `static`

В Rust можно определить постоянную с помощью ключевого слова `const`:

```
const N: i32 = 5;
```

В отличие от обычных имён, объявляемых с помощью `let`, тип постоянной надо указывать всегда.

Постоянные живут в течение всего времени работы программы. А именно, у них вообще нет определённого адреса в памяти. Это потому, что они встраиваются (inline) в каждое место, где есть их использование. По этой причине ссылки на одну и ту же постоянную не обязаны указывать на один и тот же адрес в памяти.

`static`

В Rust также можно объявить что-то вроде "глобальной переменной", используя статические значения. Они похожи на постоянные, но статические значения не встраиваются в место их использования. Это значит, что каждое значение существует в единственном экземпляре, и у него есть определённый адрес.

Вот пример:

```
static N: i32 = 5;
```

Так же, как и в случае с постоянными, тип статического значения надо указывать всегда.

Статические значения живут в течение всего времени работы программы, и любая ссылка на постоянную имеет статическое время жизни (`static` lifetime):

```
static NAME: &'static str = "Steve";
```

Изменяемость

Вы можете сделать статическое значение изменяемым с помощью ключевого слова `mut`:

```
static mut N: i32 = 5;
```

Поскольку `N` изменяемо, один поток может изменить его во время того, как другой читает его значение. Это ситуация "гонки" по данным, и она считается небезопасным поведением в Rust. Поэтому и чтение, и изменение статического изменяемого значения (`static mut`) является [небезопасным]unsafe, и обе эти операции должны выполняться в небезопасных блоках (`unsafe` block):

```
unsafe {
    N += 1;

    println!("N: {}", N);
}
```

Более того, любой тип, хранимый в статической переменной, должен иметь ограничение `Sync`.

Инициализация

И постоянные, и статические значения имеют определённые требования к тому, что можно хранить в них. Они могут быть проинициализированы только выражением, значение которого постоянно. Другими словами, вы не можете использовать вызов функции или что-то, вычисляемое во время исполнения.

Какую конструкцию стоит использовать?

Почти всегда стоит предпочитать постоянные. Ситуация, когда вам нужно реальное место в памяти и соответствующий ему адрес довольно редка. А использование постоянных позволяет компилятору провести оптимизации вроде распространения постоянных (constant propagation) не только в вашем контейнере, но и в тех, которые зависят от него.

Постоянная похожа на `#define` в C: это дополнительная мета-информация, но у неё нет дополнительных накладных расходов во время исполнения программы. Вопрос "использовать ли `const` или `static` в Rust" похож на вопрос "использовать ли

define или static в C".

Кортежные структуры

В Rust есть ещё один тип данных, который представляет собой нечто среднее между кортежем и структурой. Он называется *кортежной структурой*. Кортежные структуры именуются, а вот у их полей имён нет:

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);
```

Эти два объекта различны, несмотря на то, что у них одинаковые значения:

```
let black = Color(0, 0, 0);
let origin = Point(0, 0, 0);
```

Почти всегда, вместо кортежной структуры лучше использовать обычную структуру. Мы бы скорее объявили типы `Color` и `Point` вот так:

```
struct Color {
    red: i32,
    blue: i32,
    green: i32,
}

struct Point {
    x: i32,
    y: i32,
    z: i32,
}
```

Теперь у нас есть настоящие имена, а не только позиции. Хорошие имена важны, и при использовании структуры у нас есть эти имена.

Однако, *есть* один случай, когда кортежные структуры очень полезны. Это кортежная структура с всего одним элементом. Такое использование называется *новым типом*, потому что оно позволяет создать новый тип, отличный от типа значения, содержащегося в кортежной структуре. При этом новый тип обозначает что-то другое:

```
struct Inches(i32);

let length = Inches(10);

let Inches(integer_length) = length;
println!("Длина в дюймах: {}", integer_length);
```

Как вы можете видеть в данном примере, извлечь вложенный целый тип можно с помощью деконструирующего `let`. Мы обсуждали это выше, в разделе "кортежи". В данном случае, оператор `let Inches(integer_length)` присваивает `10` имени `integer_length`.

Атрибуты

В Rust объявления могут быть аннотированы с помощью ‘атрибутов‘. Они выглядят так:

```
#[test]
```

или так:

```
#![test]
```

Разница между ними состоит в символе `!`, который изменяет его поведение, определяющее к какому элементу применяется атрибут:

```
#[foo]
struct Foo;

mod bar {
    #![bar]
}
```

Атрибут `#[foo]` относится к следующему за ним элементу, который является объявлением `struct`. Атрибут `#![bar]` относится к элементу охватывающему его, который является объявлением `mod`. В остальном они одинаковы. Оба каким-то образом изменяют значение элемента, к которому они прикрепленны.

Например, рассмотрим такую функцию:

```
#[test]
fn check() {
    assert_eq!(2, 1 + 1);
}
```

Фунуция помечена как `#[test]`. Это означает, что она особенная: эта функция будет выполняться при запуске тестов. При компиляции, как правило, она не будет включена. Теперь эта функция является функцией тестирования.

Атрибуты также могут иметь дополнительные данные:

```
#[inline(always)]
fn super_fast_fn() {
```

Или даже ключи и значения:

```
#[cfg(target_os = "macos")]
mod macos_only {
```

Атрибуты в Rust используются для ряда различных вещей. Вот ссылка на полный список атрибутов. В настоящее время вы не можете создавать свои собственные атрибуты, компилятор Rust определяет их.

Псевдонимы типов

Ключевое слово `type` позволяет объявить псевдоним другого типа:

```
type Name = String;
```

Затем вы можете использовать этот псевдоним вместо реального типа:

```
type Name = String;

let x: Name = "Hello".to_string();
```

Однако, обратите внимание на то что *псевдоним* не объявляет новый тип. Rust строго типизированный язык, например у вас не получится сравнить значения двух различных типов:

```
let x: i32 = 5;
let y: i64 = 5;

if x == y {
   // ...
}
```

Вы получите ошибку при компиляции:

```
error: mismatched types:
 expected `i32`,
    found `i64`
(expected i32,
    found i64) [E0308]
     if x == y {
              ^
```

Но если мы используем псевдоним:

```
type Num = i32;

let x: i32 = 5;
let y: Num = 5;

if x == y {
   // ...
}
```

То этот пример скомпилируется без ошибок. Значения типа `Num` всегда будут такие же как и у типа `i32`.

Вы также можете использовать псевдонимы типов с обобщённым кодом:

```
use std::result;

enum ConcreteError {
    Foo,
    Bar,
}

type Result<T> = result::Result<T, ConcreteError>;
```

В этом примере мы создаем свою версию типа `Result`, который всегда будет использовать перечисление `ConcreteError` в `Result<T, E>` вместо типа `E`. Псевдонимы типов часто используются в модулях стандартной библиотеки для создания своих псевдонимов для `Result<T, E>`. Например, io::Result.

Приведение типов

Rust, со своим акцентом на безопасность, обеспечивает два различных способа преобразования различных типов между собой. Первый - `as`, для безопасного приведения. Второй - `transmute`, в отличие от первого, позволяет произвольное приведение типов и является одной из самых опасных фич Rust!

as

Ключевое слово `as` выполняет обычное приведение типов:

```
let x: i32 = 5;

let y = x as i64;
```

Оно допускает только определенные виды приведения типов:

```
let a = [0u8, 0u8, 0u8, 0u8];

let b = a as u32; // four eights makes 32
```

Это приведет к ошибке:

```
error: non-scalar cast: `[u8; 4]` as `u32`
let b = a as u32; // four eights makes 32
        ^~~~~~~~
```

Это 'нескалярное преобразование', потому что у нас здесь преобразуются множественные значения: четыре элемента массива. Такие виды преобразований очень опасны, потому что они делают предположения о том, как реализованы множественные нижележащие структуры. Поэтому нам нужно что-то более опасное.

transmute

Функция `transmute` предоставляется внутренними средствами компилятора, и то, что она делает, является очень простым, но в то же время очень опасным. Она сообщает Rust, чтобы он воспринимал значение одного типа, как будто это значение другого типа. Это делается независимо от системы проверки типов, и поэтому полностью на ваш страх и риск.

В предыдущем примере, мы знаем, что массив из четырех `u8` отображается в массив `u32` должным образом, и поэтому мы хотим выполнить приведение. Если вместо `as` использовать `transmute`, то Rust позволит это сделать:

```
use std::mem;

unsafe {
    let a = [0u8, 0u8, 0u8, 0u8];

    let b = mem::transmute::<[u8; 4], u32>(a);
}
```

Для того чтобы компиляция прошла успешно, мы должны обернуть эту операцию в `unsafe` блок. Технически, только вызов `mem::transmute` должен быть выполнен в небезопасном блоке, но в данном случае хорошо было бы поместить в этот блок все необходимое, связаное с этим вызовом, чтобы было удобнее искать. В данном примере связаной необходимой переменной является `a`, и поэтому она находится в блоке. Код может быть в любом стиле, иногда контекст расположен слишком далеко, и тогда упаковка всего кода в `unsafe` не будет такой уж хорошей идеей.

Хотя при использовании `transmute` и выполняется очень мало проверок, но как минимум будет проверяться, что типы имеют одинаковый размер. Нижеприведенный код завершится ошибкой:

```
use std::mem;

unsafe {
    let a = [0u8, 0u8, 0u8, 0u8];

    let b = mem::transmute::<[u8; 4], u64>(a);
}
```

со следующим описанием:

```
error: transmute called on types with different sizes: [u8; 4] (32 bits) to u64
(64 bits)
```

Все, кроме этой одной проверки, на ваш страх и риск!

Ассоциированные типы

Ассоциированные (связанные) типы - это мощная часть системы типов в Rust. Они связаны с идеей 'семейства типа', другими словами, группировки различных типов вместе. Это описание немного абстрактно, так что давайте разберем на примере. Если вы хотите написать типаж `Graph`, то нужны два обобщенных параметра типа: тип узел и тип ребро. Исходя из этого, вы можете написать типаж `Graph<N, E>`, который выглядит следующим образом:

```
trait Graph<N, E> {
    fn has_edge(&self, &N, &N) -> bool;
    fn edges(&self, &N) -> Vec<E>;
    // etc
}
```

Такое решение вроде бы достигает своей цели, он, в конечном счете, является неудобным. Например, любая функция, которая принимает `Graph` в качестве параметра, *также* должна быть дженериком с обобщенными параметрами `N` и `E`:

```
fn distance<N, E, G: Graph<N, E>>(graph: &G, start: &N, end: &N) -> u32 { ... }
```

Наша функция расчета расстояния работает независимо от типа `Edge`, поэтому параметр `E` в этой сигнатуре является лишним и только отвлекает.

Что действительно нужно заявить, это чтобы сформировать какого-либо вида `Graph`, нужны соответствующие типы `E` и `N`, собранные вместе. Мы можем сделать это с помощью ассоциированных типов:

```
trait Graph {
    type N;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
    // etc
}
```

Теперь наши клиенты могут абстрагироваться от определенного `Graph`:

```
fn distance<G: Graph>(graph: &G, start: &G::N, end: &G::N) -> uint { ... }
```

Больше нет необходимости иметь дело с типом `E`!

Давайте поговорим обо всем этом более подробно.

Определение ассоциированных типов

Давайте построим наш типаж `Graph`. Вот его определение:

```
trait Graph {
    type N;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
}
```

Достаточно просто. Ассоциированные типы используют ключевое слово `type`, и расположены внутри тела типажа, наряду с функциями.

These `type` declarations can have all the same thing as functions do. For example, if we wanted our `N` type to implement `Display`, so we can print the nodes out, we could do this: Эти объявления `type` могут иметь все то же самое, как функции делают. Например, если бы мы хотели, чтобы тип `N` реализовывал `Display`, чтобы была возможность печатать узлы, мы могли бы сделать следующее:

```
use std::fmt;

trait Graph {
    type N: fmt::Display;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
}
```

Реализация ассоциированных типов

Типаж, который включает ассоциированные типы, как и любой другой типаж, для реализации использует ключевое слово `impl`. Вот простая реализация `Graph`:

```
struct Node;

struct Edge;

struct MyGraph;

impl Graph for MyGraph {
    type N = Node;
    type E = Edge;

    fn has_edge(&self, n1: &Node, n2: &Node) -> bool {
        true
    }

    fn edges(&self, n: &Node) -> Vec<Edge> {
        Vec::new()
    }
}
```

Это глупая реализация, которая всегда возвращает `true` и пустой `Vec<Edge>`, но она дает вам общее представление о том, как реализуются такие вещи. Для начала нужны три `struct`, одна для графа, одна для узла и одна для ребра. В этой реализации используются `struct` для всех трех сущностей, но вполне могли бы использоваться и другие типы, которые работали бы так же хорошо, если бы реализация была более продвинутой.

Затем идет строка с `impl`, которая является такой же, как и при реализации любого другого типажа.

Далее мы используем знак `=`, чтобы определить наши ассоциированные типы. Имя типажа идет слева от знака `=`, а конкретный тип, для которого мы `impl` этот типаж, идет справа. Наконец, мы используем конкретные типы при объявлении функций.

Типажи-объекты и ассоциированные типы

Вот еще немного синтаксиса, о котором следует упомянуть: типажи-объекты. Если вы попытаетесь создать типаж-объект из ассоциированного типа, как в этом примере:

```
let graph = MyGraph;
let obj = Box::new(graph) as Box<Graph>;
```

Вы получите две ошибки:

```
error: the value of the associated type `E` (from the trait `main::Graph`) must
be specified [E0191]
let obj = Box::new(graph) as Box<Graph>;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
24:44 error: the value of the associated type `N` (from the trait
`main::Graph`) must be specified [E0191]
let obj = Box::new(graph) as Box<Graph>;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Мы не сможем создать типаж-объект, подобный этому, потому что у него нет информации об ассоциированных типах. Вместо этого, мы можем написать так:

```
let graph = MyGraph;
let obj = Box::new(graph) as Box<Graph<N=Node, E=Edge>>;
```

Синтаксис `N=Node` позволяет нам предоставлять конкретный тип, `Node`, для параметра типа `N`. То же самое и для `E=Edge`. Если бы мы не предоставляли это ограничение, то не могли бы знать наверняка, какая `impl` соответствует этому типажу-объекту.

Безразмерные типы

Большинство типов имеют определённый размер в байтах. Этот размер обычно известен во время компиляции. Например, `i32` - это 32 бита, или 4 байта. Однако, существуют некоторые полезные типы, которые не имеют определённого размера. Они называются "безразмерными" или "типами динамического размера". Один из примеров таких типов - это `[T]`. Этот тип представляет собой последовательность из определённого числа элементов `T`. Но мы не знаем, как много этих элементов, поэтому размер неизвестен.

Rust понимает несколько таких типов, но их использование несколько ограничено. Есть три ограничения:

1. Мы можем работать с экземпляром безразмерного типа только с помощью указателя. `&[T]` будет работать, а `[T]` - нет.
2. Переменные и аргументы не могут иметь тип динамического размера.
3. Только последнее поле структуры может быть безразмерного типа; другие - нет. Варианты перечислений не могут содержать типы динамического размера в качестве данных.

А зачем это всё? Поскольку мы можем использовать `[T]` только через указатель, если бы язык не поддерживал безразмерные типы, мы бы не смогли написать такой код:

```
impl Foo for str {
```

или

```
impl<T> Foo for [T] {
```

Вместо этого, вам бы пришлось написать:

```
impl Foo for &str {
```

Таким образом, данная реализация работала бы только для ссылок, и не поддерживала бы другие типы указателей. А реализацию для безразмерного типа смогут использовать любые указатели, включая определённые пользователем умные указатели (позже, когда будут исправлены некоторые ошибки).

?Sized

Если вы пишете функцию, принимающую тип динамического размера, вы можете использовать специальное ограничение `?Sized`:

```
struct Foo<T: ?Sized> {
    f: T,
}
```

Этот `?` читается как "T может быть размерным (`Sized`)". Он означает, что это ограничение особенное: оно разрешает использование некоторых типов, которые не могли бы быть использованы в его отсутствие. Таким образом, оно *расширяет* множество подходящих типов, а не сужает его. Это можно представить себе как если бы все типы `T` неявно были размерными (`T: Sized`), а `?` отменял это ограничение по-умолчанию.

Перегрузка операций

Rust позволяет ограниченную форму перегрузки операций. Есть определенные операции, которые могут быть перегружены. Есть специальные типажи, которые вы можете реализовать для поддержки конкретной операции между типами. В результате чего перегружается операция.

Например, операция `+` может быть перегружена с помощью типажа `Add`:

```
use std::ops::Add;

#[derive(Debug)]
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() {
    let p1 = Point { x: 1, y: 0 };
    let p2 = Point { x: 2, y: 3 };

    let p3 = p1 + p2;

    println!("{:?}", p3);
}
```

В `main` мы можем использовать операцию `+` для двух `Point`, так как мы реализовали типаж `Add<Output=Point>` для `Point`.

Есть целый ряд операций, которые могут быть перегружены таким образом, и все связанные с этим типажи расположены в модуле `std::ops`. Проверьте эту часть документации для получения полного списка.

Реализация этих типажей следует паттерну. Давайте посмотрим на типаж `Add` более детально:

```
pub trait Add<RHS = Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}
```

В общей сложности здесь присутствуют три типа: тип `impl Add`, который мы реализуем, тип `RHS`, который по умолчанию равен `Self` и тип `Output`. Для выражения `let z = x + y`: `x` - это тип `Self`, `y` - это тип `RHS`, а `z` - это тип `Self::Output`.

```
impl Add<i32> for Point {
    type Output = f64;

    fn add(self, rhs: i32) -> f64 {
        // add an i32 to a Point and get an f64
    }
}
```

позволит вам сделать следующее:

```
let p: Point = // ...
let x: f64 = p + 2i32;
```

Преобразования при разыменовании (`Deref` coercions)

Стандартная библиотека Rust реализует особый типаж, `Deref`. Обычно его используют, чтобы перегрузить `*`, операцию разыменования:

```
use std::ops::Deref;

struct DerefExample<T> {
    value: T,
}

impl<T> Deref for DerefExample<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.value
    }
}

fn main() {
    let x = DerefExample { value: 'a' };
    assert_eq!('a', *x);
}
```

Это полезно при написании своих указательных типов. Однако, в языке есть возможность, связанная с `Deref`: преобразования при разыменовании. Вот правило: если есть тип `U`, и он реализует `Deref<Target=T>`, значения `&U` будут автоматически преобразованы в `&T`, когда это необходимо. Вот пример:

```
fn foo(s: &str) {
    // позаимствуем строку на секунду
}

// String реализует Deref<Target=str>
let owned = "Hello".to_string();

// Поэтому, такой код работает:
foo(&owned);
```

Амперсанд перед значением означает, что мы берём ссылку на него. Поэтому `owned` - это `String`, а `&owned` - `&String`. Поскольку у нас есть реализация типажа `impl Deref<Target=str> for String`, `&String` разыменуется в `&str`, что устраивает `foo()`.

Вот и всё. Это правило - одно из немногих мест в Rust, где типы преобразуются автоматически. Оно позволяет писать гораздо более гибкий код. Например, тип `Rc<T>` реализует `Deref<Target=T>`, поэтому такой код работает:

```
use std::rc::Rc;

fn foo(s: &str) {
    // позаимствуем строку на секунду
}

// String реализует Deref<Target=str>
let owned = "Hello".to_string();
let counted = Rc::new(owned);

// Поэтому, такой код работает:
foo(&counted);
```

Мы всего лишь обернули наш `String` в `Rc<T>`. Но теперь мы можем передать `Rc<String>` везде, куда мы могли передать `String`. Сигнатура `foo` не поменялась, и работает как с одним, так и с другим типом. Этот пример делает два преобразования: сначала `Rc<String` преобразуется в `String`, а потом `String` в `&str`. Rust сделает столько преобразований, сколько возможно, пока типы не совпадут.

Другая известная реализация, предоставляемая стандартной библиотекой, это `impl Deref<Target=[T]> for Vec<T>`:

```
fn foo(s: &[i32]) {
    // позаимствуем срез на секунду
}

// Vec<T> реализует Deref<Target=[T]>
let owned = vec![1, 2, 3];

foo(&owned);
```

Вектора могут разыменовываться в срезы.

Разыменование и вызов методов

`Deref` также будет работать при вызове метода. Другими словами, возможен такой код:

```
struct Foo;

impl Foo {
    fn foo(&self) { println!("Foo"); }
}

let f = Foo;

f.foo();
```

Несмотря на то, что `f` - это не ссылка, а `foo` принимает `&self`, это будет работать. Более того, все примеры ниже делают одно и то же:

```
f.foo();
(&f).foo();
(&&f).foo();
(&&&&&&&&f).foo();
```

Методы `Foo` можно вызывать и на значении типа `&&&&&&&&&&&&&&&&Foo`, потому что компилятор сделает столько разыменований, сколько нужно для совпадения типов. А разыменование использует `Deref`.

Макросы

К этому моменту вы узнали о многих инструментах Rust, которые нацелены на абстрагирование и повторное использование кода. Эти единицы повторно использованного кода имеют богатую смысловую структуру. Например, функции имеют сигнатуры типа, типы параметров могут имеют ограничения по типажам, перегруженные функции также могут принадлежать к определенному типажу.

Эта структура означает, что ключевые абстракции Rust имеют мощный механизм проверки времени компиляции. Но это достигается за счет снижения гибкости. Если вы визуально определите структуру повторно используемого кода, то вы можете найти трудным или громоздким выражение этой схемы в виде дженерик функции, типажа, или чего-то еще в семантике Rust.

Макросы позволяют абстрагироваться на *синтаксическом* уровне. Вызов макроса является сокращением для "расширенной" синтаксической формы. Это расширение происходит в начале компиляции, до начала статической проверки. В результате, макросы могут охватить много шаблонов повторного использования кода, которые невозможны при использовании лишь ключевых абстракций Rust.

Недостатком является то, что код, основанный на макросах, может быть трудным для понимания, потому что к нему применяется меньше встроенных правил. Подобно обычной функции, качественный макрос может быть использован без понимания его реализации. Тем не менее, может быть трудно разработать качественный макрос! Кроме того, ошибки компилятора в макро коде сложнее интерпретировать, потому что они описывают проблемы в расширенной форме кода, а не в исходной сокращенной форме кода, которую используют разработчики.

Эти недостатки делают макросы чем-то вроде "фичи последней инстанции". Это не означает, что макросы это плохо; они являются частью Rust, потому что иногда они все же нужны для по-настоящему краткой записи хорошо абстрагированной части кода. Просто имейте этот компромисс в виду.

Определение макросов (Макроопределения)

Вы, возможно, видели макрос `vec!`, который используется для инициализации вектора с произвольным количеством элементов.

```
let x: Vec<u32> = vec![1, 2, 3];
```

Его нельзя реализовать в виде обычной функциии, так как он принимает любое количество аргументов. Но мы можем представить его в виде синтаксического сокращения для следующего кода

```
let x: Vec<u32> = {
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(2);
    temp_vec.push(3);
    temp_vec
};
```

Мы можем реализовать это сокращение, используя макрос: [1]

представленной здесь по соображениям эффективности и повторного использования. Некоторые из них упомянуты в главе [продвинутые макросы][advanced macros chapter].

```
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Вау, тут много нового синтаксиса! Давайте разберем его.

```
macro_rules! vec { ... }
```

Тут мы определяем макрос с именем `vec`, аналогично тому, как `fn vec` определяло бы функцию с именем `vec`. При вызове мы неформально пишем имя макроса с восклицательным знаком, например, `vec!`. Восклицательный знак является частью синтаксиса вызова и служит для того, чтобы отличать макрос от обычной функции.

Сопоставление (Matching) (Синтаксис вызова макрокоманды)

Макрос определяется с помощью ряда *правил*, которые представляют собой варианты сопоставления с образцом. Выше у нас было

```
( $( $x:expr ),* ) => { ... };
```

Это очень похоже на конструкцию `match`, но сопоставление происходит на уровне синтаксических деревьев Rust, на этапе компиляции. Точка с запятой не является обязательной для последнего (только здесь) варианта. "Образец" слева от `=>` известен как *шаблон совпадений* (*образец*) (*обнаружитель совпадений*) (*matcher*). Он имеет свою собственную грамматику в рамках языка.

Образец `$x:expr` будет соответствовать любому выражению Rust, связывая его дерево синтаксиса с *метапеременной* `$x`. Идентификатор `expr` является *спецификатором фрагмента*; полные возможности перечислены в главе [продвинутые макросы][advanced

macros chapter]. Образец, окруженный `$(...),*`, будет соответствовать нулю или более выражениям, разделенным запятыми.

За исключением специального синтаксиса сопоставления с образцом, любые другие элементы Rust, которые появляются в образце, должны в точности совпадать. Например,

```
macro_rules! foo {
    (x => $e:expr) => (println!("mode X: {}", $e));
    (y => $e:expr) => (println!("mode Y: {}", $e));
}

fn main() {
    foo!(y => 3);
}
```

выведет

```
mode Y: 3
```

А с

```
foo!(z => 3);
```

мы получим ошибку компиляции

```
error: no rules expected the token `z`
```

Развертывание (Expansion) (Синтаксис преобразования макрокоманды)

С правой стороны макро правил используется, по большей части, обычный синтаксис Rust. Но мы можем соединить кусочки раздробленного синтаксиса, захваченные при сопоставлении с соответствующим образцом. Из предыдущего примера:

```
$(
    temp_vec.push($x);
)*
```

Каждое соответствующее выражение `$x` будет генерировать одиночный оператор `push` в развернутой форме макроса. Повторение в развернутой форме происходит синхронно с повторением в форме образца (более подробно об этом чуть позже).

Поскольку `$x` уже объявлен в образце как выражение, мы не повторяем `:expr` с правой стороны. Кроме того, мы не включаем разделителяющую запятую в качестве части оператора повторения. Вместо этого, у нас есть точка с запятой в пределах повторяемого блока.

Еще одна деталь: макрос `vec!` имеет *две* пары фигурных скобках правой части. Они часто сочетаются таким образом:

```
macro_rules! foo {
    () => {{
        ...
    }}
}
```

Внешние скобки являются частью синтаксиса `macro_rules!`. На самом деле, вы можете использовать `()` или `[]` вместо них. Они просто разграничивают правую часть в целом.

Внутренние скобки являются частью расширенного синтаксиса. Помните, что макрос `vec!` используется в контексте выражения. Мы используем блок, для записи выражения с множественными операторами, в том числе включающее `let` привязки. Если ваш макрос раскрывается в одно единственное выражение, то дополнительной слой скобок не нужен.

Note that we never *declared* that the macro produces an expression. In fact, this is not determined until we use the macro as an expression. With care, you can write a macro whose expansion works in several contexts. For example, shorthand for a data type could be valid as either an expression or a pattern.

Обратите внимание, что мы никогда не *говорили*, что макрос создает выражения. На самом деле, это не определяется, пока мы не используем макрос в качестве выражения. Если соблюдать осторожность, то можно написать макрос, развернутая форма которого будет валидна сразу в нескольких контекстах. Например, сокращенная форма для типа данных может быть валидной и как выражение, и как шаблон.

Повторение (Repetition) (Многовариантность)

Операции повтора всегда сопутствуют два основных правила:

1. `$(...)*` walks through one "layer" of repetitions, for all of the `$name`s it contains, in lockstep, and
2. each `$name` must be under at least as many `$(...)*`s as it was matched against. If it is under more, it'll be duplicated, as appropriate.
3. `$(...)*` проходит через один "слой" повторений, для всех `$name`, которые он содержит, в ногу, и
4. каждое `$name` должно быть под крайней мере, столько `$(...)*`, как это было сопоставляется. Если это в более, это будет дублироваться, при необходимости.

This baroque macro illustrates the duplication of variables from outer repetition levels.

Этот причудливый макрос иллюстрирует дублирования переменных из внешних уровней повторения.

```
macro_rules! o_O {
    (
        $(
            $x:expr; [ $( $y:expr ),* ]
        );*
    ) => {
        &[ $($( $x + $y ),*),* ]
    }
}

fn main() {
    let a: &[i32]
        = o_O!(10; [1, 2, 3];
               20; [4, 5, 6]);

    assert_eq!(a, [11, 12, 13, 24, 25, 26]);
}
```

Это наибольшая синтаксиса совпадений. Эти примеры используют конструкцию `$(...)*`, которая означает "ноль или более" совпадений. Также вы можете написать `$(...)+`, что будет означать "одно или более" совпадений. Обе формы записи включают необязательный разделитель, располагающийся сразу за закрывающей скобкой, который может быть любым символом, за исключением `+` или `*`.

Эта система повторений основана на "Macro-by- Example" (PDF ссылка).

Гигиена (Hygiene)

Некоторые языки реализуют макросы с помощью простой текстовой замены, что приводит к различным проблемам. Например, нижеприведенная C программа напечатает `13` вместо ожидаемого `25`.

```
#define FIVE_TIMES(x) 5 * x

int main() {
    printf("%d\n", FIVE_TIMES(2 + 3));
    return 0;
}
```

После развертывания мы получаем `5 * 2 + 3`, но умножение имеет больший приоритет чем сложение. Если вы часто использовали C макросы, вы, наверное, знаете стандартные идиомы для устранения этой проблемы, а также пять или шесть других проблем. В Rust мы можем не беспокоиться об этом.

```
macro_rules! five_times {
    ($x:expr) => (5 * $x);
}

fn main() {
    assert_eq!(25, five_times!(2 + 3));
}
```

Метапеременная `$x` обрабатывается как единый узел выражения, и сохраняет свое место в дереве синтаксиса даже после замены.

Другой распространенной проблемой в системе макросов является *захват переменной* (*variable capture*). Вот C макрос, использующий GNU C расширение, который эмулирует блоки выражениий в Rust.

```
#define LOG(msg) ({ \
    int state = get_log_state(); \
    if (state > 0) { \
        printf("log(%d): %s\n", state, msg); \
    } \
})
```

Вот простой случай использования, применение которого может плохо кончиться:

```
const char *state = "reticulating splines";
LOG(state)
```

Он раскрывается в

```
const char *state = "reticulating splines";
int state = get_log_state();
if (state > 0) {
    printf("log(%d): %s\n", state, state);
}
```

Вторая переменная с именем `state` затеняет первую. Это проблема, потому что команде печати требуется обращаться к ним обоим.

Эквивалентный макрос в Rust обладает требуемым поведением.

```
macro_rules! log {
    ($msg:expr) => {{
        let state: i32 = get_log_state();
        if state > 0 {
            println!("log({}): {}", state, $msg);
        }
    }};
}

fn main() {
    let state: &str = "reticulating splines";
    log!(state);
}
```

Это работает, потому что Rust имеет систему макросов с соблюдением гигиены. Раскрытие каждого макроса происходит в отдельном *контексте синтаксиса*, и каждая переменная обладает меткой контекста синтаксиса, где она была введена. Это как если бы переменная `state` внутри `main` была бы окрашена в другой "цвет" в отличае от переменной `state` внутри макроса, из-за чего они бы не конфликтовали.

Это также ограничивает возможности макросов для внедрения новых связываний переменных на месте вызова. Код, приведенный ниже, не будет работать:

```
macro_rules! foo {
    () => (let x = 3);
}

fn main() {
    foo!();
    println!("{}", x);
}
```

Вместо этого вы должны передавать имя переменной при вызове, тогда она будет обладать меткой правильного контекста синтаксиса.

```
macro_rules! foo {
    ($v:ident) => (let $v = 3);
}

fn main() {
    foo!(x);
    println!("{}", x);
}
```

Это справедливо для `let` привязок и меток loop, но не для элементов. Код, приведенный ниже, компилируется:

```
macro_rules! foo {
    () => (fn x() { });
}

fn main() {
    foo!();
    x();
}
```

Рекурсия макросов

Раскрытие макроса также может включать в себя вызовы макросов, в том числе вызовы того макроса, который раскрывается. Эти рекурсивные макросы могут быть использованы для обработки древовидного ввода, как показано на этом (упрощенном) HTML сокращение:

```
macro_rules! write_html {
    ($w:expr, ) => (());

    ($w:expr, $e:tt) => (write!($w, "{}", $e));

    ($w:expr, $tag:ident [ $($inner:tt)* ] $($rest:tt)*) => {{
        write!($w, "<{}>", stringify!($tag));
        write_html!($w, $($inner)*);
        write!($w, "</{}>", stringify!($tag));
        write_html!($w, $($rest)*);
    }};
}

fn main() {
    use std::fmt::Write;
    let mut out = String::new();

    write_html!(&mut out,
        html[
            head[title["Macros guide"]]
            body[h1["Macros are the best!"]]
        ]);

    assert_eq!(out,
        "<html><head><title>Macros guide</title></head>\
         <body><h1>Macros are the best!</h1></body></html>");
}
```

Отладка макросов

Чтобы увидеть результаты расширения макросов, выполните команду `rustc --pretty expanded`. Вывод представляет собой целый контейнер, так что вы можете подать его обратно в `rustc`, что иногда выдает лучшие сообщения об ошибках, чем при обычной компиляции. Обратите внимание, что вывод `--pretty expanded` может иметь разное значение, если несколько переменных, имеющих одно и то же имя (но разные контексты синтаксиса), находятся в той же области видимости. В этом случае `--pretty expanded,hygiene` расскажет вам о контекстах синтаксиса.

`rustc`, поддерживает два синтаксических расширения, которые помогают с отладкой макросов. В настоящее время, они неустойчивы и требуют feature gates.

- `log_syntax!(...)` будет печатать свои аргументы в стандартный вывод во время компиляции, и "развертываться" в ничто.
- `trace_macros!(true)` будет выдавать сообщение компилятора каждый раз, когда макрос развертывается. Используйте `trace_macros!(false)` в конце развертывания, чтобы выключить его.

Требования синтаксиса

Код на Rust может быть разобран в синтаксическое дерево, даже когда он содержит неразвёрнутые макросы. Это свойство очень полезно для редакторов и других инструментов, обрабатывающих исходный код. Оно также влияет на вид системы макросов Rust.

Как следствие, когда компилятор разбирает вызов макроса, ему необходимо знать, во что развернётся данный макрос. Макрос может разворачиваться в следующее:

- ноль или больше элементов;
- ноль или больше методов;
- выражение;
- оператор;
- образец.

Вызов макроса в блоке может представлять собой элементы, выражение, или оператор. Rust использует простое правило для разрешения этой неоднозначности. Вызов макроса, производящего элементы, должен либо

- ограничиваться фигурными скобками, т.е. `foo! { ... }`;
- завершаться точкой с запятой, т.е. `foo!(...);`.

Другое следствие разбора перед раскрытием макросов - это то, что вызов макроса должен состоять из допустимых лексем. Более того, скобки всех видов должны быть сбалансированы в месте вызова. Например, `foo!([)` не является разрешённым кодом. Такое поведение позволяет компилятору понимать где заканчивается вызов макроса.

Говоря более формально, тело вызова макроса должно представлять собой последовательность *деревьев лексем*. Дерево лексем определяется рекурсивно и представляет собой либо:

- последовательность деревьев лексем, окружённую согласованными круглыми, квадратными или фигурными скобками (`()`, `[]`, `{}`);
- любую другую одиночную лексему.

Внутри сопоставления каждая метапеременная имеет *указатель фрагмента*, определяющий синтаксическую форму, с которой она совпадает. Вот список этих указателей:

- `ident`: идентификатор. Например: `x`; `foo`.
- `path`: квалифицированное имя. Например: `T::SpecialA`.
- `expr`: выражение. Например: `2 + 2`; `if true then { 1 } else { 2 }`; `f(42)`.
- `ty`: тип. Например: `i32`; `Vec<(char, String)>`; `&T`.
- `pat`: образец. Например: `Some(t)`; `(17, 'a')`; `_`.
- `stmt`: единственный оператор. Например: `let x = 3`.
- `block`: последовательность операторов, ограниченная фигурными скобками. Например: `{ log(error, "hi"); return 12; }`.

- `item`: элемент. Например: `fn foo() { }`; `struct Bar;`.
- `meta`: "мета-элемент", как в атрибутах. Например: `cfg(target_os = "windows")`.
- `tt`: единственное дерево лексем.

Есть дополнительные правила относительно лексем, следующих за метапеременной:

- за `expr` должно быть что-то из этого: `=> , ;`;
- за `ty` и `path` должно быть что-то из этого: `=> , : = > as`;
- за `pat` должно быть что-то из этого : `=> , =`;
- за другими лексемами могут следовать любые символы.

Приведённые правила обеспечивают развитие синтаксиса Rust без необходимости менять существующие макросы.

И ещё: система макросов никак не обрабатывет неоднозначность разбора. Например, грамматика `$($t:ty)* $e:expr` всегда будет выдавать ошибку, потому что синтаксическому анализатору пришлось бы выбирать между разбором `$t` и разбором `$e`. Можно изменить синтаксис вызова так, чтобы грамматика отличалась в начале. В данном случае можно написать `$(T $t:ty)* E $e:exp`.

Области видимости, импорт и экспорт макросов

Макросы разворачиваются на ранней стадии компиляции, перед разрешением имён. Один из недостатков такого подхода в том, что правила видимости для макросов отличны от правил для других конструкций языка.

Компилятор определяет и разворачивает макросы при обходе графа исходного кода контейнера в глубину. При этом определения макросов включаются в граф в порядке их встречи компилятором. Поэтому макрос, определённый на уровне модуля, виден во всём последующем коде модуля, включая тела всех вложенных модулей (`mod`).

Макрос, определённый в теле функции, или где-то ещё не на уровне модуля, виден только внутри этого элемента (например, внутри одной функции).

Если модуль имеет атрибут `macro_use`, то его макросы также видны в его родительском модуле после элемента `mod` данного модуля. Если родитель тоже имеет атрибут `macro_use`, макросы также будут видны в модуле-родителе родителя, после элемента `mod` родителя. Это распространяется на любое число уровней.

Атрибут `macro_use` также можно поставить на подключение контейнера `extern crate`. В этом контексте оно управляет тем, какие макросы будут загружены из внешнего контейнера, т.е.

```
#[macro_use(foo, bar)]
extern crate baz;
```

Если атрибут записан просто как `#[macro_use]`, будут загружены все макросы. Если атрибута нет, никакие макросы не будут загружены. Загружены могут быть только макросы, объявленные с атрибутом `#[macro_export]`.

Чтобы загрузить макросы из контейнера *без* компоновки контейнера в выходной артефакт, можно использовать атрибут `#[no_link]`.

Например:

```
macro_rules! m1 { () => (()) }

// здесь видны: m1

mod foo {
    // здесь видны: m1

    #[macro_export]
    macro_rules! m2 { () => (()) }

    // здесь видны: m1, m2
}

// здесь видны: m1

macro_rules! m3 { () => (()) }

// здесь видны: m1, m3

#[macro_use]
mod bar {
    // здесь видны: m1, m3

    macro_rules! m4 { () => (()) }

    // здесь видны: m1, m3, m4
}

// здесь видны: m1, m3, m4
```

Когда эта библиотека загружается с помощью `#[macro_use] extern crate`, виден только макрос `m2`.

Атрибуты, относящиеся к макросам, перечислены в справочнике Rust.

Переменная `$crate`

Если макрос используется в нескольких контейнерах, всё становится ещё сложнее. Допустим, `mylib` определяет

```
pub fn increment(x: u32) -> u32 {
    x + 1
}

#[macro_export]
macro_rules! inc_a {
    ($x:expr) => ( ::increment($x) )
}

#[macro_export]
macro_rules! inc_b {
    ($x:expr) => ( ::mylib::increment($x) )
}
```

`inc_a` работает только внутри `mylib`, а `inc_b` - только снаружи. Более того, `inc_b` сломается, если пользователь импортирует `mylib` под другим именем.

В Rust пока нет гигиеничных ссылок на контейнеры, но есть простой способ обойти эту проблему. Особая макро-переменная `$crate` раскроется в `::foo` внутри макроса, импортированного из контейнера `foo`. А когда макрос определён и используется в одном и том же контейнере, `$crate` станет пустой. Это означает, что мы можем написать

```
#[macro_export]
macro_rules! inc {
    ($x:expr) => ( $crate::increment($x) )
}
```

чтобы определить один макрос, который будет работать и внутри, и снаружи библиотеки. Имя функции раскроется или в `::increment`, или в `::mylib::increment`.

Чтобы эта система работала просто и правильно, `#[macro_use] extern crate ...` может быть написано только в корне вашего контейнера, но не внутри `mod`. Это обеспечивает, что `$crate` раскроется в единственный идентификатор.

Во тьме глубин

Вводная глава упоминала рекурсивные макросы, но она не рассказывала всей истории. Рекурсивные макросы полезны ещё по одной причине: каждый рекурсивный вызов даёт нам ещё одну возможность сопоставить с образцом аргументы макроса.

Приведём такой радикальный пример использования данной возможности. С помощью рекурсивных макросов можно реализовать конечный автомат типа Bitwise Cyclic Tag. Стоит заметить, что мы не рекомендуем такой подход, а просто иллюстрируем возможности макросов.

```
macro_rules! bct {
    // cmd 0:  d ... => ...
    (0, $($ps:tt),* ; $_d:tt)
        => (bct!($($ps),*, 0 ; ));
    (0, $($ps:tt),* ; $_d:tt, $($ds:tt),*)
        => (bct!($($ps),*, 0 ; $($ds),*));

    // cmd 1p:  1 ... => 1 ... p
    (1, $p:tt, $($ps:tt),* ; 1)
        => (bct!($($ps),*, 1, $p ; 1, $p));
    (1, $p:tt, $($ps:tt),* ; 1, $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; 1, $($ds),*, $p));

    // cmd 1p:  0 ... => 0 ...
    (1, $p:tt, $($ps:tt),* ; $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; $($ds),*));

    // halt on empty data string
    ( $($ps:tt),* ; )
        => (());
}
```

В качестве упражнения предлагаем читателю определить ещё один макрос, чтобы уменьшить степень дублирования кода в определении выше.

Процедурные макросы

Если система макросов не может сделать того, что вам нужно, вы можете написать плагин к компилятору. По сравнению с макросами, это гораздо труднее, там ещё более нестабильные интерфейсы, и ещё сложнее найти ошибки. Зато вы получаете гибкость - внутри плагина может исполняться произвольный код на Rust. Иногда плагины расширения синтаксиса называются *процедурными макросами*.

---

1. Фактическое определение `vec!` в libcollections отличается от ↩

Сырые указатели

Стандартная библиотека Rust содержит ряд различных типов умных указателей, но среди них есть два типа, которые экстра-специальные. Большая часть безопасности в Rust является слкдствием проверок во время компиляции, но сырье указатели не имеют конкретных гарантий и являются небезопасными для использования.

`*const T` и `*mut T` в Rust называются ‘сырыми указателями‘ (‘raw pointers‘). Иногда, при написании определенных видов библиотек, вам по какой-то причине нужно обойти гарантии безопасности Rust. В этом случае, вы можете использовать сырые указатели в реализации вашей библиотеки, вместе с тем предоставляя безопасный интерфейс для пользователей. Например, `*` указатели допускают псевдонимы, позволяя им быть использоваными для записи типов с разделяемой собственности, и даже поточно-безопасные типы памяти (`Rc<T>` и `Arc<T>` типы и реализован полностью в Rust).

Вот некоторые факты о сырых указателях, которые следует помнить и которые отличают их от других типов указателей. Они:

- не гарантируют, что они указывают на действительную область памяти, и не гарантируют, что они является ненулевыми указателями (в отличие от `Box` и `&`);
- не имеют никакой автоматической очистки, в отличие от `Box`, и поэтому требуют ручного управления ресурсами;
- это простые структуры данных (plain-old-data), то есть они не перемещают право собственности, опять же в отличие от `Box`, следовательно, компилятор Rust не может защитить от ошибок, таких как использование освобождённой памяти (use- after-free);
- лишены сроков жизни в какой-либо форме, в отличие от `&`, и поэтому компилятор не может делать выводы о висячих указателях; и
- не имеют никаких гарантий относительно псевдонимизации или изменяемости, за исключением изменений, недопустимых непосредственно для `*const T`.

Основы

Создание сырого указателя совершенно безопасно:

```
let x = 5;
let raw = &x as *const i32;

let mut y = 10;
let raw_mut = &mut y as *mut i32;
```

А вот его разыменование не является. Следующий код не будет работать:

```
let x = 5;
let raw = &x as *const i32;

println!("raw points at {}", *raw);
```

Он выдает такую ошибку:

```
error: dereference of unsafe pointer requires unsafe function or block [E0133]
     println!("raw points at{}", *raw);
                                 ^~~~
```

Когда вы разыменовываете сырой указатель, вы принимаете на себя ответственность, что он не указывает на что-то, что может быть некорректным. Таким образом, вы должны использовать `unsafe`:

```
let x = 5;
let raw = &x as *const i32;

let points_at = unsafe { *raw };

println!("raw points at {}", points_at);
```

Для более подробной информации по операциям с сырыми указателями, обратитесь к API документации о них.

FFI

Сырые указатели полезны для FFI: `*const T` и `*mut T` в Rust приблизительно соответствуют `const T*` и `T*` в C. Для более подробной информации об этом обратитесь к главе FFI.

Ссылки и сырые указатели

Во время выполнения и сырой указатель, `*`, и ссылка, указывающая на тот же кусок данных, имеют одинаковое представление. По факту, ссылка `&T` будет неявно приведена к сырому указателю `*const T` в безопасном коде, аналогично и для вариантов `mut` (оба приведения могут быть выполнены явно, с помощью, соответственно, `value as *const T` и `value as *mut T`).

Переход в обратном направлении, от `*const` к ссылке `&`, не являеся безопасным. Ссылка `&T` всегда валидна, и поэтому, как минимум, сырой указатель `*const T` должен указывать на правильный экземпляр типа `T`. Кроме того, в результате указатель должен удовлетворять правилам псевдонимизации и изменяемости ссылок. Компилятор предполагает, что эти свойства верны для любых ссылок, независимо от того, как они были созданы, и поэтому любое преобразование из сырых указателей равносильно утверждению, что они соответствуют этим правилам. Программист *должен* гарантировать это.

Рекомендуемым методом преобразования является

```
let i: u32 = 1;

// explicit cast
let p_imm: *const u32 = &i as *const u32;
let mut m: u32 = 2;

// implicit coercion
let p_mut: *mut u32 = &mut m;

unsafe {
    let ref_imm: &u32 = &*p_imm;
    let ref_mut: &mut u32 = &mut *p_mut;
}
```

Разыменование с помощью конструкции `&*x` является более предпочтительным, чем с использованием `transmute`. Последнее является гораздо более мощным инструментом, чем необходимо, а более ограниченное поведение сложнее использовать неправильно. Например, она требует, чтобы `x` представляет собой указатель (в отличие от `transmute`).

Небезопасный код

Введение

Rust стремится обеспечить безопасные абстракции над низкоуровневыми деталями процессора и операционной системы, но иногда бывает нужно спуститься и писать код именно на низком уровне. Цель этого руководства - предоставить обзор опасностей и мощи (эффективности), получаемых при использовании небезопасного подмножества в Rust.

Rust обеспечивает аварийный люк в виде блока `unsafe { ... }`, который позволяет программисту обойти некоторые из проверок компилятора и выполнить широкий спектр операций, таких как:

- разыменование raw pointers
- вызов функций посредством FFI (covered by the FFI guide)
- побитовое (поразрядное) преобразование типов (приведение типов, которое может оказаться небезопасным и зависящим от реализации) (`transmute`, также известное как "reinterpret cast")
- встраивание ассемблерного кода inline assembly

Обратите внимание, что блок `unsafe` не дает послабления для правил, относящихся к срокам жизни для `&` и замораживанию (фиксации) позаимствованных данных.

Любое использование `unsafe` равносильно заявлению программиста: "Я знаю больше, чем ты", для компилятора. И таким образом, программист должен быть абсолютно уверен в том, что он на самом деле знает что-то, почему этот кусок кода является валидным. В целом, следует попытаться свести к минимуму количество опасного кода в кодовой базе; предпочтительно использовать допустимо минимальное количество `unsafe` блоков при создания безопасных интерфейсов.

> **Примечание**: низкоуровневые детали языка Rust еще могут меняться, поэтому нет никакой гарантии стабильности или обратной совместимости. В частности, могут быть изменения, которые не вызывают ошибок компиляции, но вызывают смысловые изменения (например, приводящие к неопределенному (двусмысленному) поведению). Таким образом, требуется крайняя осторожность.

Указатели

Ссылки

Одной из важнейших особенностей Rust является безопасность памяти. Она достигается, в частности с помощью системы владения, а именно: благодаря ей компилятор может гарантировать, что каждая ссылка `&` всегда валидна, и никогда не указывает на освобожденную память.

Эти ограничения для `&` имеют огромные преимущества. Тем не менее, они также ограничивают и то, как мы можем их использовать. Например, поведение ссылок `&` отличается от поведения указателей языка C, и поэтому ссылки не могут быть использованы в качестве указателей в интерфейсах внешних функций (FFI). Кроме того, и неизменяемые (`&`), и изменяемые (`&mut`) ссылки обладают некоторыми гарантиями относительно псевдонимизации и замораживания (фиксации), необходимыми для обеспечения безопасности памяти.

В частности, если у вас есть ссылка `&T`, то `T` не должен быть изменен с помощью этой или любой другой ссылки. В стандартной библиотеке есть несколько типов, например `Cell` и `RefCell`, которые обеспечивают внутреннюю изменчивость, заменяя гарантии во время компиляции на динамические проверки во время выполнения.

Ссылка `&mut` имеет различные ограничения: когда объект имеет ссылку `&mut T`, указывающую на него, тогда эта `&mut` ссылка должна быть единственным возможным способом обращения к этому объекту во всей программе. То есть, ссылки `&mut` не могу быть псевдонимизированны (иметь псевдоним) любыми другими ссылками.

Использование `unsafe` кода, для того чтобы некорректно обойти и нарушить эти ограничения, приведет к неопределеному (двусмысленному) поведению. Например, следующий код создает два псевдонима для `&mut` указателя, и не является валидным.

```
use std::mem;
let mut x: u8 = 1;

let ref_1: &mut u8 = &mut x;
let ref_2: &mut u8 = unsafe { mem::transmute(&mut *ref_1) };

// oops, ref_1 and ref_2 point to the same piece of data (x) and are
// both usable
*ref_1 = 10;
*ref_2 = 20;
```

Сырье указатели (Raw pointers)

Rust предлагает два дополнительных вида указателя (*сырье указатели* (*raw pointers*), написание которых `*const T` и `*mut T`. Они приблизительно соответствуют `const T*` и `T*` языка C; действительно, одно из самых распространенных их применений - для FFI, при взаимодействии с внешними библиотеками C.

Сырье указатели дают гораздо меньше гарантий, чем другие типы указателей, предоставляемые языком Rust и библиотеками. Например, они

- не гарантируют, что они указывают на действительную область памяти, и не

гарантируют, что они является ненулевыми указателями (в отличие от `Box` и `&`);
- не имеют никакой автоматической очистки, в отличие от `Box`, и поэтому требуют ручного управления ресурсами;
- это простые структуры данных (plain-old-data), то есть они не перемещают право собственности, опять же в отличие от `Box`, следовательно, компилятор Rust не может защитить от ошибок, таких как использование освобождённой памяти (use- after-free);
- лишены сроков жизни в какой-либо форме, в отличие от `&`, и поэтому компилятор не может делать выводы о висячих указателях; и
- не имеют никаких гарантий относительно псевдонимизации или изменяемости, за исключением изменений, недопустимых непосредственно для `*const T`.

К счастью, им присуща и положительная особенность: слабые гарантии означают и слабые ограничения. Отсутствие ограничений делает сырые указатели подходящими в качестве строительного блока для реализации внутри библиотек таких вещей как смарт-указатели (умные указатели) и векторы. Например, для `*` указателей разрешается задавать псевдонимы, что позволяет им быть использованными при написании типов для множественного права собственности, таких как указатели со счетчиком ссылок и указатели со сборкой мусора и даже типов для потоко- безопасного совместного использования памяти (типы `Rc` и `Arc` реализованы в Rust в полной мере).

При работе с сырыми указателями есть две вещи, которые требуют большей осторожности (т.е. требуют блок `unsafe { ... }`):

- разыменование: указатели могут содержать любое значение: поэтому возможные результаты включают: аварии, чтение неинициализированной памяти, использование памяти после освобождения или обычное чтение данных.
- арифметика указателей с помощью `offset` intrinsic (или `.offset` метода): внутреннее используется так называемая "in-bounds" арифметика, то есть, он определен только поведение, если результат находится внутри (или один байт после конца (one-byte-past-the-end)) объекта, из которого первоначально указатель пришел.

Последнее предположение позволяет компилятору более эффективно оптимизировать код. Как можно видеть, фактически *создание* сырого указателя не является небезопасным, и не происходит преобразование указателя в целое число.

Ссылки и сырые указатели

Во время выполнения и сырой указатель, `*`, и ссылка, указывающая на тот же кусок данных, имеют одинаковое представление. По факту, ссылка `&T` будет неявно приведена к сырому указателю `*const T` в безопасном коде, аналогично и для вариантов `mut` (оба приведения могут быть выполнены явно, с помощью, соответственно, `value as *const T` и `value as *mut T`).

Переход в обратном направлении, от `*const` к ссылке `&`, не являеся безопасным. Ссылка `&T` всегда валидна, и поэтому, как минимум, сырой указатель `*const T` должен указывать на правильный экземпляр типа `T`. Кроме того, в результате указатель должен удовлетворять правилам псевдонимизации и изменяемости ссылок. Компилятор предполагает, что эти свойства верны для любых ссылок, независимо от того, как они были созданы, и поэтому любое преобразование из сырых указателей равносильно утверждению, что они соответствуют этим правилам. Программист *должен* гарантировать это.

Рекомендуемым методом преобразования является

```
let i: u32 = 1;
// explicit cast
let p_imm: *const u32 = &i as *const u32;
let mut m: u32 = 2;
// implicit coercion
let p_mut: *mut u32 = &mut m;

unsafe {
    let ref_imm: &u32 = &*p_imm;
    let ref_mut: &mut u32 = &mut *p_mut;
}
```

Разыменование с помощью конструкции `&*x` является более предпочтительным, чем с использованием `transmute`. Последнее является гораздо более мощным инструментом, чем необходимо, а более ограниченное поведение сложнее использовать неправильно. Например, она требует, чтобы `x` представляет собой указатель (в отличие от `transmute`).

Делаем небезопасный код безопасным

Есть различные способы создать безопасный интерфейс вокруг некоторого небезопасного кода:

- хранить указатели приватно (т.е. не в публичных полях публичных структур), так чтобы вы могли видеть и контролировать всех, кто читает и пишет по указателю, в одном месте.
- использовать макрос `assert!()` повсеместно: так как вы не можете полагаться на защиту компилятора и систему типов, чтобы удостовериться в корректности вашего `unsafe` кода во время компиляции, то используйте `assert!()`, чтобы убедиться, что он работает правильно во время выполнения.
- реализовывать `Drop` для очистки ресурсов с помощью деструктора, и использовать RAII (получение ресурса есть инициализация). Это уменьшает потребность в каком-либо ручном управлении памятью пользователями, и автоматически гарантирует, что очистка всегда выполняется, даже когда в потоке произошла паника.
- гарантировать, что любые данные, хранящиеся по сырому указателю разрушаются в соответствующее время.

Нестабильные фичи Rust

Rust обеспечивает три канала распространения для Rust: nightly, beta и stable. Нестабильные функции доступны только в nightly Rust. Для более подробной информации об этом процессе смотрите ‘Стабильность как результат’.

Чтобы установить nightly Rust, вы можете использовать `rustup.sh`:

```
$ curl -s https://static.rust-lang.org/rustup.sh | sh -s -- --channel=nightly
```

Если вы беспокоитесь о потенциальной безопасности использования данной команды `curl | sh`, то продолжайте читать далее. Вы также можете использовать двухступенчатый вариант установки и изучить наш установочный скрипт:

```
$ curl -f -L https://static.rust-lang.org/rustup.sh -O
$ sh rustup.sh --channel=nightly
```

Если же вы используете Windows, то, пожалуйста, скачайте один из установочных пакетов: 32-битный или 64-битный и запустите его.

Удаление

Если вы решили, что Rust вам больше не нужен, то мы будем чуть-чуть огорчены, но это нормально. Не каждый язык программирования отлично подходит для всех. Просто запустите скрипт деинсталляции:

```
$ sudo /usr/local/lib/rustlib/uninstall.sh
```

Если вы использовали установщик Windows, то просто повторно запустите `.msi`, который предложит вам возможность удаления.

Некоторые люди, причём не безосновательно, насторожились, когда мы сказали использовать `curl | sh`. Когда вы делаете так, вы должны доверять тем хорошим людям, которые поддерживают Rust, и не бояться, что они попытаются взломать ваш компьютер и сделать какие-либо плохие вещи. Озабоченность своей безопасностью - это очень хорошо. Если вы один из таких людей, пожалуйста посмотрите в документации как собрать Rust из исходных кодов или скачайте уже скомпилированный Rust. Мы обещаем, что данный способ не будет использоваться для установки Rust'а всегда: скрипт был сделан для быстрого обновления пока Rust находится в стадии alpha.

Мы так же должны упомянуть официально поддерживаемые платформы:

- Windows (7, 8, Server 2008 R2)
- Linux (2.6.18 и более новые, разные дистрибутивы), x86 и x86-64
- OSX 10.7 (Lion) и более новые, x86 и x86-64

Rust активно тестируется на всех этих платформах, а также на некоторых других, например на Android. Но мы указали те, на которых Rust точно должен работать, ибо для этих платформ он тестируется больше всего.

Напоследок, замечание о Windows. Rust считает, что Windows - это первоклассная платформа для релиза, но если быть честными, то опыт разработки для Windows не на столько хорош, как для Linux/OS X. Мы работаем над этим! Если что-то не работает, то это ошибка. Пожалуйста, дайте нам знать, если такое произойдёт. Каждый коммит тестируется на Windows, впрочем так же, как и на любой другой платформе.

Если вы уже установили Rust, то откройте терминал и введите это:

```
$ rustc --version
```

Вы должны увидеть версию, хэш коммита, дату коммита и дату сборки:

```
rustc 1.0.0-nightly (f11f3e7ba 2015-01-04) (built 2015-01-06)
```

Итак, теперь у вас есть установленный Rust! Поздравляем!

Установщик также устанавливает документацию, которая доступна без подключения к сети. На UNIX системах она располагается в каталоге `/usr/local/share/doc/rust`. В Windows - в директории `share/doc`, относительно того куда вы установили Rust.

Также есть ещё ряд мест, где можно получить помощь. Канал #rust на irc.mozilla.org, к которому вы можете подключиться через Mibbit. Нажмите на эту ссылку, и вы будете общаться в чате с другими Rustaceans (это дурашливое прозвище, которым мы себя называем), и мы поможем вам. Другие полезные ресурсы, посвящённые Rust: форум пользователей, /r/rust subreddit, stack overflow. Русскоязычные ресурсы: канал #rust-ru на irc.mozilla.org, google groups.

Compiler Plugins

Introduction

`rustc` can load compiler plugins, which are user-provided libraries that extend the compiler's behavior with new syntax extensions, lint checks, etc.

A plugin is a dynamic library crate with a designated *registrar* function that registers extensions with `rustc`. Other crates can load these extensions using the crate attribute `#![plugin(...)]`. See the `rustc::plugin` documentation for more about the mechanics of defining and loading a plugin.

If present, arguments passed as `#![plugin(foo(... args ...))]` are not interpreted by rustc itself. They are provided to the plugin through the `Registry`'s `args` method.

In the vast majority of cases, a plugin should *only* be used through `#![plugin]` and not through an `extern crate` item. Linking a plugin would pull in all of libsyntax and librustc as dependencies of your crate. This is generally unwanted unless you are building another plugin. The `plugin_as_library` lint checks these guidelines.

The usual practice is to put compiler plugins in their own crate, separate from any `macro_rules!` macros or ordinary Rust code meant to be used by consumers of a library.

Syntax extensions

Plugins can extend Rust's syntax in various ways. One kind of syntax extension is the procedural macro. These are invoked the same way as ordinary macros, but the expansion is performed by arbitrary Rust code that manipulates syntax trees at compile time.

Let's write a plugin `roman_numerals.rs` that implements Roman numeral integer literals.

```
#![crate_type="dylib"]
#![feature(plugin_registrar, rustc_private)]

extern crate syntax;
extern crate rustc;

use syntax::codemap::Span;
use syntax::parse::token;
use syntax::ast::{TokenTree, TtToken};
use syntax::ext::base::{ExtCtxt, MacResult, DummyResult, MacEager};
use syntax::ext::build::AstBuilder;  // trait for expr_usize
use rustc::plugin::Registry;

fn expand_rn(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree])
        -> Box<MacResult + 'static> {

    static NUMERALS: &'static [(&'static str, u32)] = &[
        ("M", 1000), ("CM", 900), ("D", 500), ("CD", 400),
        ("C",  100), ("XC",  90), ("L",  50), ("XL",  40),
        ("X",   10), ("IX",   9), ("V",   5), ("IV",   4),
        ("I",    1)];

    let text = match args {
        [TtToken(_, token::Ident(s, _))] => token::get_ident(s).to_string(),
        _ => {
            cx.span_err(sp, "argument should be a single identifier");
            return DummyResult::any(sp);
        }
    };

    let mut text = &*text;
    let mut total = 0;
    while !text.is_empty() {
        match NUMERALS.iter().find(|&&(rn, _)| text.starts_with(rn)) {
            Some(&(rn, val)) => {
                total += val;
                text = &text[rn.len()..];
            }
            None => {
                cx.span_err(sp, "invalid Roman numeral");
                return DummyResult::any(sp);
            }
        }
    }

    MacEager::expr(cx.expr_u32(sp, total))
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_macro("rn", expand_rn);
}
```

Then we can use `rn!()` like any other macro:

```
#![feature(plugin)]
#![plugin(roman_numerals)]

fn main() {
    assert_eq!(rn!(MMXV), 2015);
}
```

The advantages over a simple `fn(&str) -> u32` are:

- The (arbitrarily complex) conversion is done at compile time.
- Input validation is also performed at compile time.
- It can be extended to allow use in patterns, which effectively gives a way to define new literal syntax for any data type.

In addition to procedural macros, you can define new `derive`-like attributes and other kinds of extensions. See `Registry::register_syntax_extension` and the `SyntaxExtension` enum. For a more involved macro example, see `regex_macros`.

Tips and tricks

Some of the macro debugging tips are applicable.

You can use `syntax::parse` to turn token trees into higher-level syntax elements like expressions:

```
fn expand_foo(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree])
        -> Box<MacResult+'static> {

    let mut parser = cx.new_parser_from_tts(args);

    let expr: P<Expr> = parser.parse_expr();
```

Looking through `libsyntax` parser code will give you a feel for how the parsing infrastructure works.

Keep the `Span`s of everything you parse, for better error reporting. You can wrap `Spanned` around your custom data structures.

Calling `ExtCtxt::span_fatal` will immediately abort compilation. It's better to instead call `ExtCtxt::span_err` and return `DummyResult`, so that the compiler can continue and find further errors.

To print syntax fragments for debugging, you can use `span_note` together with `syntax::print::pprust::*_to_string`.

The example above produced an integer literal using `AstBuilder::expr_usize`. As an alternative to the `AstBuilder` trait, `libsyntax` provides a set of quasiquote macros. They are undocumented and very rough around the edges. However, the implementation may be a good starting point for an improved quasiquote as an ordinary plugin library.

Lint plugins

Plugins can extend Rust's lint infrastructure with additional checks for code style, safety, etc. You can see `src/test/auxiliary/lint_plugin_test.rs` for a full example, the core of which is reproduced here:

```
declare_lint!(TEST_LINT, Warn,
              "Warn about items named 'lintme'")

struct Pass;

impl LintPass for Pass {
    fn get_lints(&self) -> LintArray {
        lint_array!(TEST_LINT)
    }

    fn check_item(&mut self, cx: &Context, it: &ast::Item) {
        let name = token::get_ident(it.ident);
        if name.get() == "lintme" {
            cx.span_lint(TEST_LINT, it.span, "item is named 'lintme'");
        }
    }
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_lint_pass(box Pass as LintPassObject);
}
```

Then code like

```
#![plugin(lint_plugin_test)]

fn lintme() { }
```

will produce a compiler warning:

```
foo.rs:4:1: 4:16 warning: item is named 'lintme', #[warn(test_lint)] on by default
foo.rs:4 fn lintme() { }
         ^~~~~~~~~~~~~~~
```

The components of a lint plugin are:

- one or more `declare_lint!` invocations, which define static `Lint` structs;
- a struct holding any state needed by the lint pass (here, none);
- a `LintPass` implementation defining how to check each syntax element. A single `LintPass` may call `span_lint` for several different `Lint`s, but should register them all through the `get_lints` method.

Lint passes are syntax traversals, but they run at a late stage of compilation where type information is available. `rustc`'s built-in lints mostly use the same infrastructure as lint plugins, and provide examples of how to access type information.

Lints defined by plugins are controlled by the usual attributes and compiler flags, e.g. `#[allow(test_lint)]` or `-A test-lint`. These identifiers are derived from the first argument to `declare_lint!`, with appropriate case and punctuation conversion.

You can run `rustc -W help foo.rs` to see a list of lints known to `rustc`, including those provided by plugins loaded by `foo.rs`.

Inline Assembly

For extremely low-level manipulations and performance reasons, one might wish to control the CPU directly. Rust supports using inline assembly to do this via the `asm!` macro. The syntax roughly matches that of GCC & Clang:

```
asm!(assembly template
   : output operands
   : input operands
   : clobbers
   : options
   );
```

Any use of `asm` is feature gated (requires `#![feature(asm)]` on the crate to allow) and of course requires an `unsafe` block.

> **Note**: the examples here are given in x86/x86-64 assembly, but all platforms are supported.

Assembly template

The `assembly template` is the only required parameter and must be a literal string (i.e. `""`)

```
#![feature(asm)]

#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn foo() {
    unsafe {
        asm!("NOP");
    }
}

// other platforms
#[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
fn foo() { /* ... */ }

fn main() {
    // ...
    foo();
    // ...
}
```

(The `feature(asm)` and `#[cfg]`s are omitted from now on.)

Output operands, input operands, clobbers and options are all optional but you must add the right number of `:` if you skip them:

```
asm!("xor %eax, %eax"
    :
    :
    : "eax"
   );
```

Whitespace also doesn't matter:

```
asm!("xor %eax, %eax" ::: "eax");
```

Operands

Input and output operands follow the same format: `: "constraints1"(expr1), "constraints2"(expr2), ..."`. Output operand expressions must be mutable lvalues:

```
fn add(a: i32, b: i32) -> i32 {
    let mut c = 0;
    unsafe {
        asm!("add $2, $0"
             : "=r"(c)
             : "0"(a), "r"(b)
             );
    }
    c
}

fn main() {
    assert_eq!(add(3, 14159), 14162)
}
```

Clobbers

Some instructions modify registers which might otherwise have held different values so we use the clobbers list to indicate to the compiler not to assume any values loaded into those registers will stay valid.

```
// Put the value 0x200 in eax
asm!("mov $$0x200, %eax" : /* no outputs */ : /* no inputs */ : "eax");
```

Input and output registers need not be listed since that information is already communicated by the given constraints. Otherwise, any other registers used either implicitly or explicitly should be listed.

If the assembly changes the condition code register `cc` should be specified as one of the clobbers. Similarly, if the assembly modifies memory, `memory` should also be specified.

Options

The last section, `options` is specific to Rust. The format is comma separated literal strings (i.e. `:"foo", "bar", "baz"`). It's used to specify some extra info about the inline assembly:

Current valid options are:

1. *volatile* - specifying this is analogous to `__asm__ __volatile__ (...)` in gcc/clang.
2. *alignstack* - certain instructions expect the stack to be aligned a certain way (i.e. SSE) and specifying this indicates to the compiler to insert its usual stack alignment code
3. *intel* - use intel syntax instead of the default AT&T.

Без stdlib

По умолчанию, `std` компонуется с каждым контейнером Rust. В некоторых случаях это нежелательно, и этого можно избежать с помощью атрибута `#![no_std]`, примененного (привязанного) к контейнеру.

```
// a minimal library
#![crate_type="lib"]
#![feature(no_std)]
#![no_std]
```

Очевидно, должно быть нечто большее, чем просто библиотеки: `#[no_std]` можно использовать с исполняемыми контейнерами, а управлять точкой входа можно двумя способами: с помощью атрибута `#[start]`, или с помощью переопределения прокладки (shim) для C функции `main` по умолчанию на вашу собственную.

В функцию, помеченную атрибутом `#[start]`, передаются параметры командной строки в том же формате, что и в C:

```
#![feature(lang_items, start, no_std, libc)]
#![no_std]

// Pull in the system libc library for what crt0.o likely requires
extern crate libc;

// Entry point for this program
#[start]
fn start(_argc: isize, _argv: *const *const u8) -> isize {
    0
}

// These functions and traits are used by the compiler, but not
// for a bare-bones hello world. These are normally
// provided by libstd.
#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] fn panic_fmt() -> ! { loop {} }
```

Чтобы переопределить вставленную компилятором прокладку `main`, нужно сначала отключить ее с помощью `#![no_main]`, а затем создать соответствующий символ с правильным ABI и правильным именем, что также потребует переопределение искажения (коверкания) имен компилятором (`#[no_mangle]`):

```
#![feature(no_std)]
#![no_std]
#![no_main]
#![feature(lang_items, start)]

extern crate libc;

#[no_mangle] // для уверенности в том, что этот символ будет называться `main` на выходе
pub extern fn main(argc: i32, argv: *const *const u8) -> i32 {
    0
}

#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] fn panic_fmt() -> ! { loop {} }
```

В настоящее время компилятор делает определенные предположения о символах, которые доступны для вызова в исполняемом контейнере. Как правило, эти функции предоставляются стандартной библиотекой, но если она не используется, то вы должны определить их самостоятельно.

Первая из этих трех функций, `stack_exhausted`, вызывается тогда, когда обнаруживается (происходит) переполнение стека. Эта функция имеет ряд ограничений, касающихся того, как она может быть вызывана и того, что она должна делать, но если регистр предела стека не поддерживается, то поток всегда имеет "бесконечный стек" и эта функция не должна быть вызвана (получить управление, срабатывать).

Вторая из этих трех функций, `eh_personality`, используется в механизме обработки ошибок компилятора. Она часто отображается на функцию personality (специализации) GCC (для получения дополнительной информации смотри реализацию libstd), но можно с уверенностью сказать, что для контейнеров, которые не вызывают панику, эта функция никогда не будет вызвана. Последняя функция, `panic_fmt`, также используются в механизме обработки ошибок компилятора.

Использование основной библиотеки (libcore)

> **Примечание**: структура основной библиотеки (core) является нестабильной, и поэтому рекомендуется использовать стандартную библиотеку (std) там, где это возможно.

С учетом указанных выше методов, у нас есть чисто-металлический исполняемый код работает Rust. Стандартная библиотека предоставляет немало функциональных возможностей, однако, для Rust также важна производительность. Если стандартная библиотека не соответствует этим требованиям, то вместо нее может быть использована libcore.

Основная библиотека имеет очень мало зависимостей и гораздо более компактна, чем стандартная библиотека. Кроме того, основная библиотека имеет большую часть необходимой функциональности для написания идиоматического и эффективного кода на Rust.

В качестве примера приведем программу, которая вычисляет скалярное произведение двух векторов, предоставленных из кода C, и использует идиоматические практики Rust.

```
#![feature(lang_items, start, no_std, core, libc)]
#![no_std]

extern crate core;

use core::prelude::*;

use core::mem;

#[no_mangle]
pub extern fn dot_product(a: *const u32, a_len: u32,
                          b: *const u32, b_len: u32) -> u32 {
    use core::raw::Slice;

    // Convert the provided arrays into Rust slices.
    // The core::raw module guarantees that the Slice
    // structure has the same memory layout as a &[T]
    // slice.
    //
    // This is an unsafe operation because the compiler
    // cannot tell the pointers are valid.
    let (a_slice, b_slice): (&[u32], &[u32]) = unsafe {
        mem::transmute((
            Slice { data: a, len: a_len as usize },
            Slice { data: b, len: b_len as usize },
        ))
    };

    // Iterate over the slices, collecting the result
    let mut ret = 0;
    for (i, j) in a_slice.iter().zip(b_slice.iter()) {
        ret += (*i) * (*j);
    }
    return ret;
}

#[lang = "panic_fmt"]
extern fn panic_fmt(args: &core::fmt::Arguments,
                    file: &str,
                    line: u32) -> ! {
    loop {}
}

#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
#[lang = "eh_personality"] extern fn eh_personality() {}
```

Обратите внимание, что здесь, в отличае от примеров, рассмотренных выше, есть один дополнительный "lang" элемент `panic_fmt`. Он должен быть определён потребителями "libcore", потому что основная библиотека объявляет панику, но не определяет её. "lang" элемент `panic_fmt` определяет панику для этого контейнера, и необходимо гарантировать, что он никогда не возвращает значение.

Как видно в этом примере, основная библиотека предназначена для предоставления всей мощи Rust при любых обстоятельствах, независимо от требований платформы. Дополнительные библиотеки, такие как "liballoc", добавляют функциональность для "libcore", из-за чего делаются другие платформно-зависимые предположения, но это по-прежнему является более компактным (переносимым), чем стандартная библиотека.

Внутренние средства (Intrinsics)

> **Примечание**: внутренние средства всегда будут иметь нестабильный интерфейс, рекомендуется использовать стабильные интерфейсы libcore, а не внутренние напрямую.

Они импортируются как если бы они были FFI функциями, со специальным `rust-intrinsic` ABI. Например, если, находясь в отдельныом (автономном) контексте, хочется иметь возможность `transmute` между типами, а также использовать эффективную арифметику указателей, то можно импортировать эти функции через объявление, такое как

```
extern "rust-intrinsic" {
    fn transmute<T, U>(x: T) -> U;

    fn offset<T>(dst: *const T, offset: isize) -> *const T;
}
```

Как и с любыми другими FFI функциями, их вызов всегда `unsafe`.

Lang items

> **Note**: lang items are often provided by crates in the Rust distribution, and lang items themselves have an unstable interface. It is recommended to use officially distributed crates instead of defining your own lang items.

The `rustc` compiler has certain pluggable operations, that is, functionality that isn't hard-coded into the language, but is implemented in libraries, with a special marker to tell the compiler it exists. The marker is the attribute `#[lang="..."]` and there are various different values of `...`, i.e. various different 'lang items'.

For example, `Box` pointers require two lang items, one for allocation and one for deallocation. A freestanding program that uses the `Box` sugar for dynamic allocations via `malloc` and `free`:

```
#![feature(lang_items, box_syntax, start, no_std, libc)]
#![no_std]

extern crate libc;

extern {
    fn abort() -> !;
}

#[lang = "owned_box"]
pub struct Box<T>(*mut T);

#[lang="exchange_malloc"]
unsafe fn allocate(size: usize, _align: usize) -> *mut u8 {
    let p = libc::malloc(size as libc::size_t) as *mut u8;

    // malloc failed
    if p as usize == 0 {
        abort();
    }

    p
}
#[lang="exchange_free"]
unsafe fn deallocate(ptr: *mut u8, _size: usize, _align: usize) {
    libc::free(ptr as *mut libc::c_void)
}

#[start]
fn main(argc: isize, argv: *const *const u8) -> isize {
    let x = box 1;

    0
}

#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] fn panic_fmt() -> ! { loop {} }
```

Note the use of `abort`: the `exchange_malloc` lang item is assumed to return a valid pointer, and so needs to do the check internally.

Other features provided by lang items include:

- overloadable operators via traits: the traits corresponding to the `==`, `<`, dereferencing (`*`) and `+` (etc.) operators are all marked with lang items; those specific four are `eq`, `ord`, `deref`, and `add` respectively.
- stack unwinding and general failure; the `eh_personality`, `fail` and `fail_bounds_checks` lang items.
- the traits in `std::marker` used to indicate types of various kinds; lang items `send`, `sync` and `copy`.
- the marker types and variance indicators found in `std::marker`; lang items `covariant_type`, `contravariant_lifetime`, etc.

Lang items are loaded lazily by the compiler; e.g. if one never uses `Box` then there is no need to define functions for `exchange_malloc` and `exchange_free`. `rustc` will emit an error when an item is needed but not found in the current crate or any that it depends on.

Аргументы компоновки (Link args)

Еще один способ указать rustc как настроить линковку - с помощью атрибута `link_args`. Этот атрибут применяется к блокам `extern` и определяет исходные флаги, которые нужно передать компоновщику при производстве артефакта. Пример использования:

```
#![feature(link_args)]

#[link_args = "-foo -bar -baz"]
extern {}
```

Обратите внимание, что эта фича в настоящее время находится в закрытых фичах (feature gate), `feature(link_args)`, потому что это не санкционированный способ выполнения компоновки. Сейчас rustc выходит на (использует) системный компоновщик, поэтому имеет смысл предоставить дополнительные аргументы командной строки, но это не всегда будет так. В будущем rustc может использовать LLVM непосредственно, чтобы компоновать нативные библиотеки, и в этом случае `link_args` не будет иметь никакого смысла.

Настоятельно рекомендуется не использовать этот атрибут, вместо него лучше использовать более формальный атрибут `#[link(...)]` для блоков `extern`.

Тесты производительности

Rust поддерживает тесты производительности, которые помогают измерить производительность вашего кода. Давайте изменим наш `src/lib.rs`, чтобы он выглядел следующим образом (комментарии опущены):

```
#![feature(test)]

extern crate test;

pub fn add_two(a: i32) -> i32 {
    a + 2
}

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

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }

    #[bench]
    fn bench_add_two(b: &mut Bencher) {
        b.iter(|| add_two(2));
    }
}
```

Обратите внимание на включение фичи (feature gate) `test`, что включает эту нестабильную фичу.

Мы импортировали контейнер `test`, который включает поддержку измерения производительности. У нас есть новая функция, аннотированная с помощью атрибута `bench`. В отличие от обычных тестов, которые не принимают никаких аргументов, тесты производительности в качестве аргумента принимают `&mut Bencher`. `Bencher` предоставляет метод `iter`, который в качестве аргумента принимает замыкание. Это замыкание содержит код, производительность которого мы хотели бы протестировать.

Запуск тестов производительности осуществляется командой `cargo bench`:

```
$ cargo bench
   Compiling adder v0.0.1 (file:///home/steve/tmp/adder)
     Running target/release/adder-91b3e234d4ed382a

running 2 tests
test tests::it_works ... ignored
test tests::bench_add_two ... bench:         1 ns/iter (+/- 0)

test result: ok. 0 passed; 0 failed; 1 ignored; 1 measured
```

Все тесты, не относящиеся к тестам производительности, были проигнорированы. Вы, наверное, заметили, что выполнение `cargo bench` занимает немного больше времени чем `cargo test`. Это происходит потому, что Rust запускает наш тест несколько раз, а затем выдает среднее значение. Так как мы выполняем слишком мало полезной работы в этом примере, у нас получается `1 ns/iter (+/- 0)`, но была бы выведена дисперсия, если бы был один.

Советы по написанию тестов производительности:

- Внутри `iter` цикла пишите только тот код, производительность которого вы хотите измерить; инициализацию выполняйте за пределами `iter` цикла
- Внутри `iter` цикла пишите код, который будет идемпотентным (будет делать "то же самое" на каждой итерации); не накапливайте и не изменяйте состояние
- Вне `iter` цикла пишите код который также будет идемпотентным; скорее всего, он будет запущен много раз во время теста
- Внутри `iter` цикла пишите код, который будет коротким и быстрым, так чтобы запуски тестов происходили быстро и калибратор мог настроить длину пробега с точным разрешением
- Внутри `iter` цикла пишите код, делающий что-то простое, чтобы помочь в выявлении улучшения (или уменьшения) производительности

Особенности оптимизации

А вот другой сложный момент, относящийся к написанию тестов производительности: тесты, скомпилированные с оптимизацией, могут быть значительно изменены оптимизатором, после чего тест будет мерить производительность не так, как мы этого ожидаем. Например, компилятор может определить, что некоторые выражения не оказывают каких-либо внешних эффектов и просто удалит их полностью.

```
#![feature(test)]

extern crate test;
use test::Bencher;

#[bench]
fn bench_xor_1000_ints(b: &mut Bencher) {
    b.iter(|| {
        (0..1000).fold(0, |old, new| old ^ new);
    });
}
```

выведет следующие результаты

```
running 1 test
test bench_xor_1000_ints ... bench:         0 ns/iter (+/- 0)

test result: ok. 0 passed; 0 failed; 0 ignored; 1 measured
```

Движок для запуска тестов производительности оставляет две возможности, позволяющие этого избежать. Либо использовать замыкание, передаваемое в метод `iter`, которое возвращает какое-либо значение; тогда это заставит оптимизатор думать, что возвращаемое значение будет использовано, из-за чего удалить вычисления полностью будет не возможно. Для примера выше этого можно достигнуть, изменив вызова `b.iter`

```
b.iter(|| {
    // note lack of `;` (could also use an explicit `return`).
    (0..1000).fold(0, |old, new| old ^ new)
});
```

Либо использовать вызов функции `test::black_box`, которая представляет собой "черный ящик", непрозрачный для оптимизатора, тем самым заставляя его рассматривать любой аргумент как используемый.

```
#![feature(test)]

extern crate test;

b.iter(|| {
    let n = test::black_box(1000);

    (0..n).fold(0, |a, b| a ^ b)
})
```

В этом примере не происходит ни чтения, ни измения значения, что очень дешево для малых значений. Большие значения могут быть переданы косвенно для уменьшения издержек (например, `black_box(&huge_struct)`).

Выполнение одного из вышеперечисленных изменений дает следующие результаты измерения производительности

```
running 1 test
test bench_xor_1000_ints ... bench:       131 ns/iter (+/- 3)

test result: ok. 0 passed; 0 failed; 0 ignored; 1 measured
```

Тем не менее, оптимизатор все еще может вносить нежелательные изменения в определенных случаях, даже при использовании любого из вышеописанных приемов.

Синтаксис упаковки и образцы (шаблоны)

В настоящее время единственный стабильный способ создания `Box` - это создание с помощью метода `Box::new`. В стабильной сборке Rust также невозможно деструктурировать `Box` при использованиии сопоставления с шаблоном. В нестабильной сборке может быть использовано ключевое слово `box`, как для создания, так и для деструктуризации `Box`. Ниже представлен пример использования:

```
#![feature(box_syntax, box_patterns)]

fn main() {
    let b = Some(box 5);
    match b {
        Some(box n) if n < 0 => {
            println!("Box contains negative number {}", n);
        },
        Some(box n) if n >= 0 => {
            println!("Box contains non-negative number {}", n);
        },
        None => {
            println!("No box");
        },
        _ => unreachable!()
    }
}
```

Обратите внимание, что эти фичи в настоящее время являются скрытыми: `box_syntax` (создание упаковки) и `box_patterns` (деструктурирование и сопоставление с образцом), потому что синтаксис все еще может измениться в будущем.

Возвращение указателей

Во многих языках с указателями, вы можете вернуть указатель из функции, чтобы таким образом избежать копирования большой структуры данных. Например:

```
struct BigStruct {
    one: i32,
    two: i32,
    // etc
    one_hundred: i32,
}

fn foo(x: Box<BigStruct>) -> Box<BigStruct> {
    Box::new(*x)
}

fn main() {
    let x = Box::new(BigStruct {
        one: 1,
        two: 2,
        one_hundred: 100,
    });

    let y = foo(x);
}
```

Идея состоит в том, что, при передаче упаковки, происходит копирование только указателя, а не всех `int`, из которых состоит `BigStruct`.

Это антипаттерн в Rust. Вместо этого следует написать так:

```
#![feature(box_syntax)]

struct BigStruct {
    one: i32,
    two: i32,
    // etc
    one_hundred: i32,
}

fn foo(x: Box<BigStruct>) -> BigStruct {
    *x
}

fn main() {
    let x = Box::new(BigStruct {
        one: 1,
        two: 2,
        one_hundred: 100,
    });

    let y: Box<BigStruct> = box foo(x);
}
```

Это дает вам гибкость без ущерба для производительности.

Вы можете подумать, что такое использование даст нам ужасную производительность: возвращается значение, а затем оно сразу упаковывается?! Разве это не паттерн худшего из двух миров? Rust намного умнее. В этом коде не происходит копирование. `main` выделяет

достаточно места для `box`, передает указатель на эту память в `foo` в виде `x`, а затем `foo` записывает значение прямо в `Box<T>`.

Это достаточно важно, поэтому стоит повторить: указатели не для оптимизации возвращаемых значений в коде. Позвольте вызывающей стороне самой выбрать, как она хочет использовать выход.

Шаблоны `match` для срезов

Если вы хотите в качестве шаблона для сопоставления использовать срез или массив, то вы можете использовать `&` и активировать фичу `slice_patterns`:

```
#![feature(slice_patterns)]

fn main() {
    let v = vec!["match_this", "1"];

    match &v[..] {
        ["match_this", second] => println!("The second element is {}", second),
        _ => {},
    }
}
```

Ассоциированные константы

С включенной фичей `associated_consts` вы можете определить константы вроде этой:

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32;
}

impl Foo for i32 {
    const ID: i32 = 1;
}

fn main() {
    assert_eq!(1, i32::ID);
}
```

Любая реализация `Foo` долна будет определить `ID`. Без этого определения:

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32;
}

impl Foo for i32 {
}
```

выдаст ошибку

```
error: not all trait items implemented, missing: `ID` [E0046]
     impl Foo for i32 {
     }
```

Также может быть реализовано значение по умолчанию:

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32 = 1;
}

impl Foo for i32 {
}

impl Foo for i64 {
    const ID: i32 = 5;
}

fn main() {
    assert_eq!(1, i32::ID);
    assert_eq!(5, i64::ID);
}
```

Как вы можете видеть, при реализации `Foo`, можно оставить константу неопределенной, как в случае для `i32`. Тогда будет использовано значение по умолчанию. Но также можно и добавить собственное определение, как в случае для `i64`.

Ассоциированные константы могут быть ассоциированы не только с типажом. Это также прекрасно работает и с блоком `impl` для `struct`:

```
#![feature(associated_consts)]

struct Foo;

impl Foo {
    pub const FOO: u32 = 3;
}
```

Глоссарий

Не каждый пользователь Rust имеет опыт работы с системами программирования, или необходимые знания в области компьютерной науки, поэтому мы добавили разъяснения терминов, которые могут быть незнакомы.

Арность

Арность относится к числу аргументов функции или операции, котое они принимают.

```
let x = (2, 3);
let y = (4, 6);
let z = (8, 2, 6);
```

В приведенном выше примере `x` и `y` имеют арность 2. `z` имеет арность 3.

Абстрактное синтаксическое дерево (Дерево абстрактного синтаксического анализа)

Когда компилятор компилирует программу, он делает целый ряд различных вещей. Одна из вещей, которые он делает, это преобразует текст вашей программы в 'Абстрактное синтаксическое дерево,' или 'AST.' Это дерево является представлением структуры вашей программы. Например, `2 + 3` может быть преобразовано в дерево:

```
  +
 / \
2   3
```

А `2 + (3 * 4)` будет выглядеть следующим образом:

```
  +
 / \
2   *
   / \
  3   4
```

Академические исследования

Неполный перечень работ, которые оказали какое-то влияние на Rust.

Рекомендуется для вдохновения и лучшего понимания предпосылок Rust.

Система типов

- Region based memory management in Cyclone
- Safe manual memory management in Cyclone
- Typeclasses: making ad-hoc polymorphism less ad hoc
- Macros that work together
- Traits: composable units of behavior
- Alias burying - We tried something similar and abandoned it.
- External uniqueness is unique enough
- Uniqueness and Reference Immutability for Safe Parallelism
- Region Based Memory Management

Многозадачность

- Singularity: rethinking the software stack
- Language support for fast and reliable message passing in singularity OS
- Scheduling multithreaded computations by work stealing
- Thread scheduling for multiprogramming multiprocessors
- The data locality of work stealing
- Dynamic circular work stealing deque - The Chase/Lev deque
- Work-first and help-first scheduling policies for async-finish task parallelism - More general than fully-strict work stealing
- A Java fork/join calamity - critique of Java's fork/join library, particularly its application of work stealing to non-strict computation
- Scheduling techniques for concurrent systems
- Contention aware scheduling
- Balanced work stealing for time-sharing multicores
- Three layer cake
- Non-blocking steal-half work queues
- Reagents: expressing and composing fine-grained concurrency
- Algorithms for scalable synchronization of shared-memory multiprocessors

Другое

- Crash-only software

- Composing High-Performance Memory Allocators
- Reconsidering Custom Memory Allocation

Научные доклады *о* Rust

- GPU programming in Rust
- Parallel closures: a new twist on an old idea - not exactly about rust, but by nmatsakis