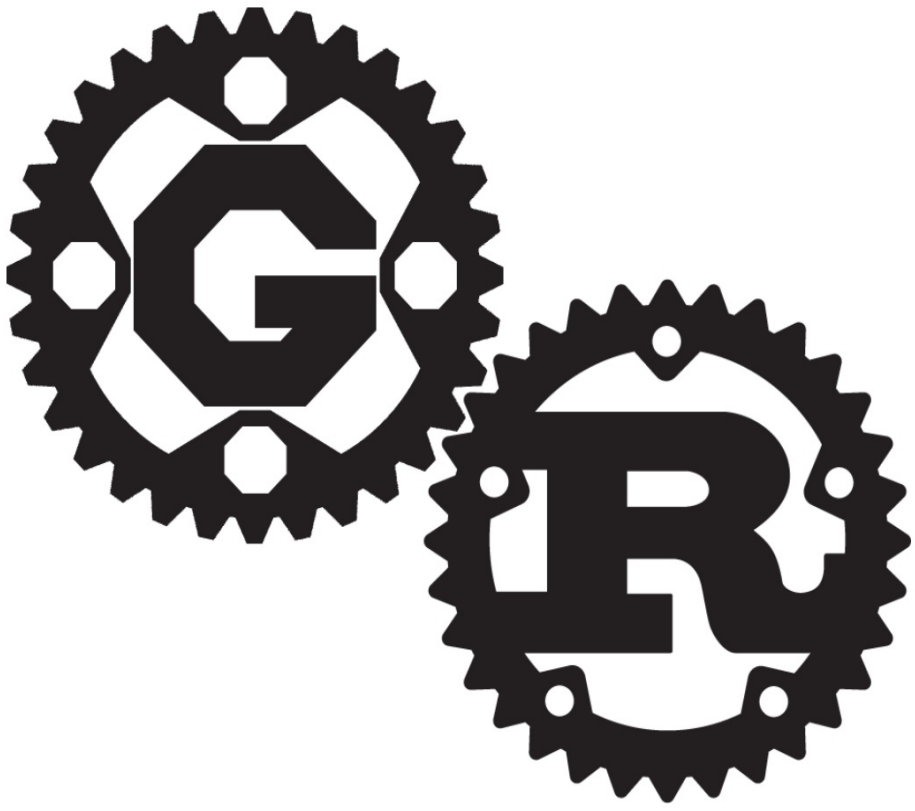
Язык программирования Rust

Перевод на русский язык
"The Rust Programming Language"



Полезные ссылки

Чаты	Ссылки
для обсуждения языка, получения помощи	gitter join chat
для обсуждения самой книги и вопросов перевода	gitter join chat

pull requests closed in about 6 hours issues closed in 3 days

Мы на Хабре

Введение к русскоязычному переводу

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

ВНИМАНИЕ! Перевод окончен и соответствует stable версии книги на момент выхода Rust 1.2 stable. Если вы видите несоответствие примеров или текста реальному поведению или оригиналу книги, пожалуйста, создайте задачу или сразу делайте Pull Request с исправлениями. Мы не кусаемся и рады исправлениям! :wink:

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

Соавторам

С чего начать

Есть некоторое количество очень простых проблем. Это опечатки, и, взяв одну из таких задач, вы сможете легко поучаствовать в переводе и очень нам поможете.

Не бойтесь code review, у нас не принято наезжать на новичков. :smile:

Где получить помощь

У этого репозитория есть чат-комната на Gitter. Если у вас возник вопрос по задаче или по тому, что вы взялись делать, как перевести какой-то термин или как собрать книгу локально - вам сюда.

Для опытных

Правила перевода.

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.

Ресурсы

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

Ревизия исходного кода данной версии книги

`3a7b8aa`

Введение

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 нецелесообразно: при встраивании в другие языки, при написании программ с особыми 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 возникает накладных расходов во время выполнения приложения и устраняются все гонки данных. Это даёт Rust преимущество над другими языками программирования, имеющими аналогичную направленность. Rust также направлен на достижение «абстракции с нулевой стоимостью». Хотя некоторые из этих абстракций и ведут себя как в языках высокого уровня, но даже тогда Rust по-прежнему обеспечивает точный контроль, как делал бы язык низкого уровня.

Книга «Язык программирования Rust» делится на восемь 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 кода на Rust.
- Синтаксис и семантика — Каждое понятие Rust разбивается на небольшие кусочки.
- Нестабильные возможности Rust — Передовые возможности, которые пока не добавлены в стабильную сборку.
- Глоссарий — Ссылки на термины, используемые в книге.
- Академические исследования — Литература, которая оказала влияние на Rust.

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 каждое понятие языка отдельно, перед тем как двигаться дальше. Большое количество перекрёстных ссылок соединяет эти части воедино.

Содействие

Исходные файлы, из которых генерируется оригинал этой книги, могут быть найдены на Github: github.com/rust-lang/rust/tree/master/src/doc/trpl

Исходные файлы перевода этой книги на русский язык также находятся на GitHub: github.com/kgv/rust_book_ru

Краткое введение в Rust

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

Основное понятие, которое делает Rust уникальным, называется «владение». Рассмотрим следующий небольшой пример:

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

Эта программа создаёт связанное имя `x`. Его значением 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 вызовы функций, и поэтому они визуально отличаются. Символ `!` также помогает при разборе, что облегчает написание инструментов, а это тоже важно.

Мы использовали `mut`, чтобы сделать `x` изменяемым: связанные имена в Rust по умолчанию неизменяемы. 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. Rust может выводить типы, что позволяет достичь компромисса между мощью статической типизации и 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, можете пока 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» — это то, что делает Rust уникальным. 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 наподобие сборщика мусора. Другими словами, в Rust вы не вызываете функции вроде `malloc` и `free` собственноручно: 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 одно имя, `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 бы два раза — 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("foo");
    ^
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, несмотря на то, что мы и сделали наше имя изменяемым, мы всё ещё 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 двух элементов. Добавление третьего элемента будет означать выделение новой области 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. Компилятор обнаружил данную проблему.

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()` прекрасно работает.

Если нам всё же нужна ссылка, 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 владения хороша не только для предотвращения проблемы повисших указателей, но также и для всей совокупности связанных с этим проблем, таких как: недействительность итератора, параллелизм и многое другое.

С чего начать

Первый раздел книги рассказывает о том, как начать работать с Rust и его инструментами. Сначала мы установим Rust, затем напишем классическую программу «Hello World», и, наконец, поговорим о Cargo, который представляет из себя 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:

> Примечание: вам не нужно вводить символы `$`, они просто обозначают начало команд. В интернете вы найдёте много руководств и примеров, которые следуют этому соглашению: `$` для команд, запускаемых из-под обычного пользователя, и `#` для команд, которые нужно выполнять из-под администратора.

```
$ 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 соответствующий установщик:

Удаление

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

```
$ 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 также 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. Windows для Rust — это такая же первоклассная целевая платформа. К сожалению, если честно, разрабатывать программы на Rust на Windows не так приятно, как на Linux и OS X. Но мы занимаемся этим! Если что-то не работает, то это ошибка. Пожалуйста, дайте нам знать, если такое произойдёт. Каждый коммит тестируется на Windows, впрочем, как и на всех остальных платформах.

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 директории `/usr/local/share/doc/rust`. В Windows — в директории `share/doc`, относительно того куда вы установили Rust.

Есть ещё несколько мест, где можно получить помощь. Есть 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, веб-подключение к #rust-ru, google groups.

Привет, мир!

Теперь, когда вы установили Rust, давайте напишем первую программу на Rust. Традиционно, в любом новом изучаемом языке программирования, первая программа просто выводит на экран текст «Привет, мир!». 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 директорию `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, созданных сообществом. Команда разработчиков Rust 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 поддиректорию 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 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!("Привет, мир!");
}
```

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

```
$ rustc main.rs
$ ./main # или main.exe в Windows
Привет, мир!
```

Работает! Разберём подробнее что же произошло. (Примечание переводчика: вам нужен терминал с поддержкой юникода, чтобы вывод русских символов работал правильно. Большинство современных терминалов поддерживают юникод из коробки.)

```
fn main() {

}
```

Эти строки определяют «функцию» в Rust. Функция `main` особенна: это начало каждой программы на Rust. Первая строка говорит: «Мы объявляем 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. Также хорошим стилем считается ставить открывающую фигурную скобку на той же строке, что и объявление функции, отделённую от него одним пробелом.

Теперь эта строка:

```
    println!("Привет, мир!");
```

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. 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, и мы обсудим этот момент позже. И последнее, что нужно отметить: макросы Rust 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. `"Привет, мир!"` — это «строка». Строки — это удивительно сложная тема в системном языке программирования. Это «статически расположенная в памяти» строка. Если вы хотите больше узнать про расположении в памяти, рекомендуем почитать про стек и кучу, но в принципе вы можете пока не заботиться о таких деталях. Мы передаём строку в качестве аргумента в `println!`, который выводит строки на экран. Это достаточно просто!

В завершение, строка заканчивается точкой с запятой (`;`). Rust — это `язык с ориентацией на выражения`, 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 `;`.

Наконец, скомпилируем и запустим нашу программу. 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: файл с нашим исходным кодом, с расширением `.rs`, и исполняемый файл (`main.exe` в Windows, `main` в остальных случаях).

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

Мы вывели наш текст `"Привет, мир!"` в окне терминала.

Если вы пришли из динамических языков программирования вроде Ruby, Python или JavaScript, необходимость предварительной компиляции программы может показаться вам необычной. Rust — это язык, программы на котором *компилируются перед исполнением*. 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 компиляция и запуск этих программ делается одной командой (например, `python main.py`). В мире языков программирования много компромиссов, и Rust сделал свой выбор.

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

Дальше мы познакомимся с новым инструментом `Cargo`, который используется для написания настоящих программ на Rust. Использовать `rustc` удобно лишь для небольших программ, но по мере роста проекта, потребуется инструмент, который поможет 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. 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
Привет, мир!
```

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`
Привет, мир!
```

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`
Привет, мир!
```

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!("Привет, мир!");
}
```

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 сильно углубляться в детали.

Если вы хотите более основательно изучить язык, 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 `Привет, мир!`. Посмотрите файл `src/main.rs`:

```
fn main() {
    println!("Привет, мир!")
}
```

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`
Привет, мир!
```

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!("Ваша попытка: {}", 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 те 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`, неизменяемы 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 уберем вызовы 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!("Ваша попытка: {}", 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
Ваша попытка: 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 (crate) `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.)

Много нового! Теперь, когда у нас есть внешние зависимости, Cargo скачал последние версии каждой из них из своего реестра, являющегося копией реестра с Crates.io. Crates.io — это место, где программисты на Rust могут публиковать свои проекты с открытым исходным кодом, чтобы их использовали в других проектах.

После обновления реестра, Cargo проверяет раздел `[dependencies]` и скачивает всё, что нам необходимо. В нашем случае, мы сказали, что наш проект зависит от `rand`. Самому контейнеру `rand` для работы нужен контейнер `libc`. По этой причине Cargo скачал и `libc`. После загрузки всего необходимого, оно компилируется, а затем компилируется и наш проект.

Если мы запустим `cargo build` снова, текст вывода будет другим:

```
$ cargo build
```

Всё правильно, ничего не будет выведено! Cargo знает, что уже собраны и наш проект, и все его зависимости, а значит незачем делать это снова. Раз делать ничего не надо, Cargo просто завершил работу. Если мы снова откроем файл `src/main.rs`, сделаем какие-нибудь изменения и затем сохраним их, мы увидим только одну строку:

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

Итак, мы сказали Cargo, что нам нужна библиотека `rand` с любой версией ветки `0.3.x`, и он взял последнюю версию, на тот момент, когда его запустили - `v0.3.8`. Но что делать, когда на следующей неделе выйдет версия `v0.3.9`, содержащая важные изменения? Что если исправления настолько масштабны, что версия `0.3.9` становится несовместимой с нашим кодом?

Решением этой проблемы является файл `Cargo.lock`, который находится в директории с нашим проектом. Когда мы в первый раз собирали наш проект, Cargo подобрал версии, подходящие под наши условия, и записал их в файл `Cargo.lock`. Когда мы в будущем будем собирать наш проект, Cargo будет проверять, существует ли `Cargo.lock`, и затем использовать указанные в нём версии контейнеров. Благодаря этому мы автоматически получаем повторяемые сборки. Другими словами, мы будем использовать контейнер версии `0.3.8` до тех пор, пока явно не обновим информацию о его версии в `Cargo.lock`.

А что, если мы захотим использовать версию `v0.3.9`? У Cargo есть другая команда, `update`, которая скажет «игнорируй Cargo.lock, найди последние версии библиотек из той ветки, которую мы указали в Cargo.toml. Когда всё сделаешь, запиши информацию о версиях в Cargo.lock». Но по умолчанию, Cargo смотрит только версию больше, чем `0.3.0`, и меньше `0.4.0`. Если мы хотим перейти на версии `0.4.x`, мы должны указать это в `Cargo.toml`. Потом, когда мы запустим `cargo build`, Cargo обновит индекс и пересмотрит наши требования к `rand`.

В документации по Cargo можно узнать о нём, а так же о его экосистеме намного больше, но пока что это всё, что нам нужно знать. Cargo делает повторное использование библиотек намного проще, и программисты на Rust, как правило, пишут небольшие проекты, которые

входят в состав других более крупных проектов.

Давайте использовать `rand`. Вот наш следующий шаг:

```
extern crate rand;

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

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

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

    println!("Загаданное число: {}", secret_number);

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

    let mut guess = String::new();

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

    println!("Ваша попытка: {}", guess);
}
```

Первое, что мы сделали — изменили первую строку. Теперь она выглядит так: `extern crate rand`. Так как мы указали `rand` в разделе `[dependencies]`, мы можем использовать `extern crate` для того, чтобы Rust знал, что мы собираемся использовать эту зависимость. `extern crate` также выполняет эквивалент оператора `use rand;`, т.е. теперь мы можем использовать всё, что есть в контейнере `rand`, используя префикс `rand::`.

Далее, мы добавили новую строку `use`: `use rand::Rng`. Мы собираемся использовать метод, а ему нужно, чтобы Rng был в области видимости. Основная идея такова: методы, объявленные где-то в другом месте, называются «типажами» (traits), и для того, чтобы этот метод можно было использовать, необходимо чтобы типаж был в области видимости. Чтобы узнать об этом более подробно, можно прочитать секцию о типажах.

Мы добавили две новые строки в середину кода:

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

    println!("Загаданное число: {}", secret_number);
```

Мы используем функцию `rand::thread_rng()` для получения копии генератора случайных чисел, который будет локальным для текущего потока выполнения. Выше мы добавили `use rand::Rng` и теперь можем использовать метод `gen_range()`. Этот метод получает два аргумента и генерирует число, которое может быть больше либо равно первому аргументу и меньше, чем второй аргумент. Таким образом, если мы укажем числа 1 и 101, то от генератора можно получить числа от 1 до 100 включительно.

Вторая строка печатает наше секретное число. Это поможет нам во время тестирования, пока мы разрабатываем нашу программу. Но мы обязательно удалим эту строчку в финальной версии. Будет не интересно играть в игру, если она сразу печатает ответ!

Давайте запустим изменённую программу:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/debug/guessing_game`
Угадайте число!
Загаданное число: 7
Пожалуйста, введите предположение.
4
Ваша попытка: 4
$ cargo run
     Running `target/debug/guessing_game`
Угадайте число!
Загаданное число: 83
Пожалуйста, введите предположение.
5
Ваша попытка: 5
```

Замечательно! Следующий шаг: сравнение нашего предположения с «загаданным» числом.

Сравнение

Теперь, когда мы знаем, что ввёл пользователь, давайте сравним «загаданное» число с предполагаемым ответом. Здесь приведён наш следующий шаг, который, к сожалению, не будет работать:

```
extern crate rand;

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

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

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

    println!("Загаданное число: {}", secret_number);

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

    let mut guess = String::new();

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

    println!("Ваша попытка: {}", guess);

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

Здесь мы видим что-то новое. Первое — это ещё один `use`. Мы ввели в область видимости тип `std::cmp::Ordering`. Далее, ещё пять новых строк в конце, которые используют его:

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

Метод `cmp()` может быть вызван у чего-либо, что может сравниваться, и получает ссылку на то, с чем мы хотим его сравнить. Результатом сравнения будет тип `Ordering`, который мы добавили выше. Мы используем оператор `match` для определения `Ordering` — результата сравнения. `Ordering` — перечисление. Они обозначаются `enum`, сокращённо от `enumeration` (перечисление). Перечисления выглядят следующим образом:

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

С таким определением, всё, что имеет тип `Foo` может иметь значение либо `Foo::Bar`, либо `Foo::Baz`. Мы используем `::` для обозначения пространства имён для вариантов перечисления.

У перечисления `Ordering` есть три возможных варианта: `Less`, `Equal` и `Greater`. Выражение `match` получает переменную какого-либо типа и предлагает вам создать «ветви» для каждого возможного значения. Так как у нас есть три возможных значения `Ordering`, у нас будет три ветви:

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

Если результатом сравнения будет значение `Less`, мы выведем на экран `Слишком маленькое!`; если будет `Greater`, то `Слишком большое!`; и если `Equal`, то `Вы выиграли!`. `match` очень удобен и он часто используется в Rust.

Мы упоминали, что это не совсем корректный код, но всё же давайте попробуем:

```
$ 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`.
```

У-у-у! Это большая ошибка. Суть этой ошибки в «несоответствии типов» (mismatched types). В Rust строгая статическая система типов. Однако, у нас также есть вывод типов. Когда мы пишем `let guess = String::new()`, Rust понимает, что `guess` должна быть типа `String`, благодаря чему мы можем не указывать тип явно. `secret_number` — число, которое может иметь значение от одного до ста. Оно может иметь тип `i32` — 32-битное целое, или `u32` — 32-битное целое без знака, или `i64` — 64-битное целое, или какой-нибудь другой. По умолчанию, Rust сделает его 32-битным целым, `i32`. Однако, здесь Rust не знает как сравнить `guess` и `secret_number`. Они должны быть одного типа. В итоге, чтобы можно было сравнить `guess` и `secret_number`, мы должны преобразовать переменную `guess`, которую мы прочитали с ввода, из типа `String` в настоящий числовой тип. Мы можем сделать это, добавив несколько строчек. Вот как будет выглядеть наша программа:

```
extern crate rand;

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

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

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

    println!("Загаданное число: {}", secret_number);

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

    let mut guess = String::new();

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

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

    println!("Ваша попытка: {}", guess);

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

Вот строки, которые мы добавили:

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

Подождите минутку, у нас ведь уже есть `guess`? Rust позволил нам «затенить» (скрыть) предыдущее `guess` новым. Это часто используется в подобных случаях, когда `guess` изначально бывает типа `String`, но нам требуется преобразовать её в `u32`. Затенение позволяет нам переиспользовать имя `guess`, а не создавать для каждого типа новое уникальное имя, такое как `guess_str` и `guess` или какое-нибудь другое.

Мы связали `guess` с выражением, которое похоже на то, что мы писали ранее:

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

За которым следует вызов `ok().expect()`. Здесь `guess` ссылается на старый `guess`, который ещё является строкой, которую мы получили с ввода. Метод `trim()` у типа `String` удаляет всё пустое пространство с начала и конца нашей строки. Это важно, ведь для нормальной работы `read_line()` нам необходимо нажать `Enter` после окончания ввода.

Это значит, что если мы набрали `5` и нажали `Enter`, `guess` выглядит следующим образом: `5\n`. `\n` обозначает «новую строку» (newline) — значение клавиши `Enter`. `trim()` удалит его и оставит только `5`. Метод `parse()`, применяемый к строке, преобразует её в число. Он может анализировать различные числа, но мы можем указать Rust какой именно тип нам нужен. Поэтому мы указали `let guess: u32`. Двоеточие `:`, идущее после `guess`, говорит Rust, что мы указали тип значения. `u32` - 32-битное беззнаковое целое число. У Rust есть несколько встроенных числовых типов, но мы выбрали именно `u32`. Это достаточно хороший тип, чтобы хранить небольшие положительные числа.

Как и `read_line()`, вызов `parse()` может вызвать проблемы. Что, если наша строка будет содержать `A👍%`? Мы не сможем преобразовать её в число. Как и в случае с `read_line()`, мы будем использовать методы `ok()` и `expect()` на случай, если `parse()` не сможет преобразовать строку.

Давайте запустим нашу программу!

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/guessing_game`
Угадайте число!
Загаданное число: 58
Пожалуйста, введите предположение.
  76
Ваша попытка: 76
Слишком большое!
```

Замечательно! Вы можете видеть, что мы добавили пробел перед нашим числом, но программа поняла, что мы хотели сказать `76`. Запустим программу ещё несколько раз и проверим, что загадывание числа работает.

Теперь большая часть нашей игры работает, но мы можем сделать только одно предположение. Давайте изменим это, добавив циклы!

Зацикливание

Ключевое слово `loop` создаёт бесконечный цикл. Давайте добавим его:

```
extern crate rand;

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

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

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

    println!("Загаданное число: {}", secret_number);

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

        let mut guess = String::new();

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

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

        println!("Ваша попытка: {}", guess);

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

И посмотрим на работу приложения. Но подождите, мы же добавили бесконечный цикл? Всё верно. Помните что мы говорили о `parse()`? Если мы введём не числовой ответ, мы просто выйдем из программы. Посмотрите:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/guessing_game`
Угадайте число!
Загаданное число: 59
Пожалуйста, введите предположение.
45
Ваша попытка: 45
Слишком маленькое!
Пожалуйста, введите предположение.
60
Ваша попытка: 60
Слишком большое!
Пожалуйста, введите предположение.
59
Ваша попытка: 59
Вы выиграли!
Пожалуйста, введите предположение.
quit
thread '<main>' panicked at 'Пожалуйста, введите число!'
```

Ха! Если мы введём `quit`, то действительно выйдем из программы. Как и при вводе любого другого не числового значения. Что ж, это, мягко говоря, не очень хорошо. Для начала, давайте сделаем выход из программы, если мы выиграли игру:

```
extern crate rand;

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

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

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

    println!("Загаданное число: {}", secret_number);

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

        let mut guess = String::new();

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

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

        println!("Ваша попытка: {}", guess);

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

С добавлением строки `break` после вывода `Вы выиграли!`, мы получили возможность выхода из цикла, когда мы угадали загаданное число. Выход из цикла также означает и завершение нашей программы, так как это последнее, что есть в `main()`. Нам надо сделать ещё одно улучшение — при любом не числовом вводе, мы не должны выходить из программы, мы просто должны проигнорировать ввод. Мы можем сделать это следующим образом:

```
extern crate rand;

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

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

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

    println!("Загаданное число: {}", secret_number);

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

        let mut guess = String::new();

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

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

        println!("Ваша попытка: {}", guess);

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

Это строка, которую мы изменили:

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

Здесь показано, как мы можем перейти от «сбоя при ошибке» к «обработке ошибки» заменив `ok().expect()` на инструкцию `match`. `Result`, возвращённый функцией `parse()`, как и `Ordering`, является перечислением. Однако в данном случае каждый из вариантов имеет некоторые ассоциированные с ним данные: `Ok` — успех, `Err` — ошибку. У каждого есть некоторая дополнительная информация: преобразованное число, либо тип ошибки. Здесь мы проверили значение результата работы `parse()` при помощи `match`. В случае, если результат равен `Ok`, то `match` привяжет внутреннее значение результата

(`Ok(num)`) к имени `num` и вернёт в привязку `guess`. Когда происходит ошибка (Err), нам не важно, какая именно это ошибка, поэтому мы используем вместо имени _. Так мы проигнорируем ошибку и вызовем `continue`, который отправит нас на следующую итерацию цикла.

Теперь всё должно быть нормально! Давайте посмотрим:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/guessing_game`
Угадайте число!
Загаданное число: 61
Пожалуйста, введите предположение.
10
Ваша попытка: 10
Слишком маленькое!
Пожалуйста, введите предположение.
99
Ваша попытка: 99
Слишком большое!
Пожалуйста, введите предположение.
foo
Пожалуйста, введите предположение.
61
Ваша попытка: 61
Вы выиграли!
```

Замечательно! Если мы ещё чуть-чуть подкрутим нашу программу, игра будет готова. Догадываетесь, что нужно поменять? Всё правильно, мы не должны выводить наше секретное число. Знание этого числа хорошо для тестирования, но оно портит всю игру. Так выглядит окончательный вариант нашего кода:

```
extern crate rand;

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

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

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

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

        let mut guess = String::new();

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

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

        println!("Ваша попытка: {}", guess);

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

Готово!

Вы сделали «Угадайку»! Поздравляем!

Этот первый проект показал вам следующее: `let`, `match`, методы, привязанные функции, использование внешних контейнеров и многое другое. Наш следующий проект покажет ещё больше.

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, а в центре стола стояла большая миска со спагетти, которая постоянно пополнялась. Как подобает философам, они большую часть своего времени проводили 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 сущность запутанных спагетти такова, что необходима вторая вилка, чтобы отправлять спагетти в рот. То есть философу требовалась еще и вилка справа от него. Философы положили свои вилки и встали из-за стола, продолжая думать. Ведь 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. 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("Анна Шмидт");
}
```

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

Продолжим:

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

Этот блок `impl` позволяет объявить что-либо 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. Она возвращает новый экземпляр нашей структуры `Philosopher`.

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

Этот код создаёт новый экземпляр `Philosopher` и присваивает его полю `name` значение переданного аргумента `name`. Но используется не сам аргумент, а результат вызова его метода `.to_string()`. Этот вызов создаёт копию строки, на которую указывает наш `&str`, и возвращает новый экземпляр `String`, который и будет присвоен полю `name` структуры `Philosopher`.

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

И последнее на что следует обратить внимание: мы просто объявляем структуру `Philosopher` и кажется, что ничего больше не делаем. Rust — это язык программирования, «ориентированный на выражения», что означает, что каждое выражение возвращает значение. Это верно и для функций, у которых автоматически возвращается последнее выражение. Так как в нашем примере в последнем выражении функции мы создаем структуру `Philosopher`, то она и будет возвращена функцией.

Имя функции `new()` не регламентируется Rust. Это просто соглашение об именовании функций, которые возвращают новые экземпляры структур. Давайте снова 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("Анна Шмидт");
}
```

Здесь мы связываем пять имен 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: "Анна Шмидт".to_string() };
}
```

Этот код выглядит не слишком изящно. Использование статической функции `new` имеет и другие преимущества, но даже в этом простом случае, её использование было оправдано.

Теперь у нас уже есть каркас программы, и 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 приема пищи, 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 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("Анна Шмидт"),
    ];

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

Давайте сначала рассмотрим функцию `main()`. Вместо того чтобы создавать пять отдельных связанных имен для философов, мы создаем для них `Vec<T>`. `Vec<T>` называют «вектор», он является расширяемой версией массива. Затем в цикле `for` мы перебираем вектор, получая ссылку на очередного философа на каждой итерации.

В теле цикла мы вызываем метод `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 метода `eat()` мы выводим только 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.
Анна Шмидт закончила есть.
```

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

Дальше нам нужно сделать так, чтобы философы не только заканчивали, но и начинали

есть. 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("Анна Шмидт"),
    ];

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

Появились некоторые небольшие изменения. Давайте посмотрим, что же изменилось:

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

Конструкция `use` предоставляет доступ к области видимости модуля `thread` из стандартной библиотеки. Мы собираемся использовать этот модуль далее в коде, и поэтому нам нужно объявить о его использовании.

```
    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 вызываем функцию `sleep_ms` между ними. Эта функция останавливает рабочий поток на 1000 миллисекунд, что симулирует процесс приема пищи философа.

Если вы запустите программу теперь, то увидите, что каждый философ, по очереди, начинает есть, ест какое-то время и заканчивает есть:

```
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.
Анна Шмидт начала есть.
Анна Шмидт закончила есть.
```

Превосходно! Теперь у нас осталась только одна проблема: наши философы едят по очереди, а не одновременно, то есть мы пока не решили задачу параллелизма.

Для того, чтобы наши философы начали есть одновременно, нам нужно внести некоторые изменения 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("Анна Шмидт"),
    ];

    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 связанное имя `handles`. Мы задали такое имя, потому что собираемся создать несколько потоков, в результате чего получим для них дескрипторы, с помощью которых сможем контролировать их выполнение. Здесь нам нужно явно указать тип, а зачем это необходимо, мы расскажем чуть позже. `_` - это заполнитель типа. Мы говорим компилятору «`handles` — это вектор, содержащий элементы, тип которых Rust должен вывести самостоятельно».

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

Мы берем наш список философов и вызываем метод `into_iter()`. Этот метод создаёт итератор, который при каждой итерации забирает право владения на соответствующий элемент. Это нужно для передачи элемента вектора в поток. Мы берем этот итератор и вызываем метод `map`, который принимает замыкание в качестве аргумента и вызывает это замыкание для каждого из элементов итератора.

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

Вот здесь происходит сам параллелизм. Функция `thread::spawn` принимает в качестве аргумента замыкание и исполняет это замыкание в новом потоке. Это замыкание дополнительно нуждается в указании ключевого слова `move`, которое сообщает, что это замыкание получает владение переменными, которые оно захватывает. В данном случае — переменной `p` функции `map`.

Внутри потока мы всего лишь вызываем метод `eat()` переменной `p`. Также обратите внимание, что вызов `thread::spawn` не оканчивается точкой с запятой, что превращает его в выражение. Этот нюанс важен, так как возвращается правильное значение. Для получения более подробной информации, прочитайте главу Выражения и операторы.

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

По завершении мы получаем результат вызова `map` и собираем полученный результат в коллекцию с помощью метода `collect()`. Метод `collect()` создаёт коллекцию какого-то типа, и для того, чтобы Rust понял, коллекцию какого типа мы хотим получить, мы указали для `handle` тип принимаемого значения `Vec<T>`. Элементами коллекции будут возвращаемые из методов `thread::spawn` значения, которые являются дескрипторами этих потоков. Вот так!

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

В конце функции `main()` мы в цикле перебираем каждый дескриптор и вызываем для него метод `join()`, который блокирует дальнейшее исполнение основного потока, пока не завершится дочерний поток. Это позволяет нам быть уверенными, что потоки завершат работу до того как произойдет выход из программы.

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

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

Но как же быть с вилками? Их мы пока ещё не смоделировали.

Давайте же начнем. Сначала сделаем новую структуру:

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

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

Структура `Table` содержит вектор мьютексов (`Mutex`). Мьютекс — способ управления доступом к данным для параллельно выполняющихся потоков: только один поток может получить доступ к данным в конкретный момент времени. Это именно то свойство, которое нужно для реализации наших вилок. В коде мы используем пустой кортеж, `()`, внутри мьютекса, так как не собираемся использовать это значение, а мьютекс используется только для организации доступа.

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("Анна Шмидт", 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, с этими изменениями мы получили корректно работающую программу. 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 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,
    }
}
```

Мы 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. Мы также добавили один аргумент, `table`. Мы получаем доступ к списку вилок через структуру `Table`. Затем используем идентификаторы вилок `self.left` и `self.right` для получения доступа к вилке по определенному индексу. В результате чего мы получаем `Mutex`, который регулирует доступ к вилке, и вызываем для него метод `lock()`, блокируя доступ к вилке. Если в настоящее время доступ к вилке уже предоставлен кому-то еще, то мы будем блокированы, пока вилка не станет доступной.

Вызов метода `lock()` может потерпеть неудачу, и если это случается, то мы аварийно завершаем работу программы. Может возникнуть ситуация, когда поток аварийно завершит свою работу, а мьютекс при этом останется заблокированным. Такой мьютекс называется «отравленным (poisoned)». Но в нашем случае это не может произойти, потому как мы просто используем метод `unwrap()`.

Результаты выполнения этих двух строк имеют имена `_left` и `_right` соответственно. Зачем мы используем знаки подчеркивания в начале имён? Это для того, чтобы сказать компилятору, что мы хотим получить значения, которые далее *не планируем использовать*. Таким образом Rust не будет выводить предупреждение о неиспользуемых именах.

Когда же мьютекс будет освобождён? Это произойдет автоматически, когда `_left` и `_right` выйдут из области видимости, то есть по окончании работы функции.

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

Далее в `main()` мы создаем новый экземпляр структуры `Table` и оборачиваем его в `Arc<T>`. Это «атомарный счетчик ссылок» (atomic reference count). Он нужен для обеспечения доступа к нашей структуре `Table` из нескольких потоков. Когда он передается в новый поток, то счетчик увеличивается, а когда этот поток завершает работу, то счетчик уменьшается.

```
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("Анна Шмидт", 0, 4),
];
```

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`. Конструктор Анны Шмидт должен был бы принимать в качестве аргументов значения `4` и `0`, но вместо этого он принимает значения `0` и `4`. Это помешает нашей программе попасть в безвыходное состояние, если каждый возьмет по одной вилке одновременно. Так что давайте представим, что один из философов у нас левша! Это один из способов решить данную проблему, и, на мой взгляд, самый простой.

```
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 метод `table.clone()`. Метод `clone()` структуры `Arc<T>` клонирует значение и инкрементирует счетчик, который автоматически декрементируется, когда клонированное значение покинет область видимости. Это необходимо для того, чтобы мы знали, как много ссылок на `table` существуют в рамках наших потоков на данный момент времени. Если бы у нас не было подсчета ссылок, то мы бы не знали, как и когда освободить хранимое значение.

Вы можете заметить, что здесь мы выполняем новое связывание с именем `table`, затеняя старое связанное имя `table`. Это позволяет нам не вводить новое уникальное имя.

Теперь наша программа работает! Только два философа могут обедать одновременно. 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.
Анна Шмидт начала есть.
Зарубина Наталья закончила есть.
Анна Шмидт 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 использовать определенный язык там, где проявляется его сильные стороны, и использовать другой язык там, где первый 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!".

Мы выбрали пять миллионов руководствуясь тем, сколько времени занимает эта работа на современном 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 выбранной нами системе эта программа работает `2.156` секунд. И если мы воспользуемся какой-нибудь утилитой для мониторинга процессов (например, `top`), то увидим, что она использует только одно ядро. 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
```

Эту программу легко переписать на Rust:

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

    for h in handles {
        println!("Thread finished with count={}",
        h.join().map_err(|_| "Could not join a thread!").unwrap());
    }
    println!("done!");
```

Мы уже знакомы с частью этого кода из предыдущих примеров. Мы создаем десять потоков, собирая их в вектор `handles`. Внутри каждого потока мы осуществляем пять миллионов повторений в цикле, и прибавляем к `x` единицу каждый раз. Наконец, мы воссоединяем все потоки.

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 динамическая библиотека (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 — это не язык, но, в настоящее время, это доминирующая реализация исполнения JavaScript на сервере.

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 написанием *какого-то* кода на Rust и написанием *хорошего* кода на Rust.

Этот раздел состоит из относительно самостоятельных уроков, которые показывают, как повысить уровень вашего кода на Rust. В нем представлены общие шаблоны и стандартные функции библиотеки. Главы 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 этом понятии, 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 связанное имя, `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?

Когда функция вызывается, то выделяется некоторый объем памяти для всех её локальных переменных и некоторой дополнительной информации. Это называется «стековый кадр» (stack frame). В этом руководстве мы будем игнорировать эту дополнительную информацию, и будем рассматривать лишь локальные переменные, которые мы определяем. 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 для этого.

Когда функция завершается, её стековый кадр освобождается. Это происходит автоматически — для этого нам не надо предпринимать никаких действий.

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

Недостатком является то, что мы не можем хранить необходимые значения дольше, чем в рамках одной функции.

А ещё мы 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()` и 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 `1 073 741 823`. Это число равно 2^{30}, количеству байтов в гигабайте.

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

Адрес	Имя	Значение
0	x	42

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

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

Адрес	Имя	Значение
2	z	100
1	y	5
0	x	42

Поскольку `0` было задействовано в первом кадре, для кадра `foo()` используются `1` и `2`. При дальнейших вызовах функций стек будет расти вверх.

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 выравнивающие разделители, состоящие из некоторого числа байтов, которые отделяют каждый из адресов. Размер этого разделителя может даже превышать размер хранящегося значения.

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

Адрес	Имя	Значение
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 трёх-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();
}
```

Сначала вызывается `main()`:

Адрес	Имя	Значение
0	x	42

Затем из `main()` вызывается `foo()`:

Адрес	Имя	Значение
3	c	1
2	b	100
1	a	5

0	x	42

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

Адрес	Имя	Значение
4	i	6
3	c	1
2	b	100
1	a	5
0	x	42

Вот что мы имели ввиду раньше, говоря, что наш стек растет вверх.

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

Адрес	Имя	Значение
3	c	1
2	b	100
1	a	5
0	x	42

А затем завершается `foo()`, оставляя только `main()`:

Адрес	Имя	Значение
0	x	42

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

Куча

Такой способ выделения памяти работает очень хорошо, но он может быть использован не всегда. 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 упаковки, т.е. типа `Box<T>`. (Примечание переводчика: мы называем `Box<T>` упаковкой, потому что `T` как бы «упакован» в `Box`: упаковка знает размер того, что лежит внутри. Эта информация закодирована в типе `T`, поэтому во время исполнения, для размерных типов, это просто указатель.) Вот пример:

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

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

Адрес	Имя	Значение
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`? Наш `x` представляет собой `Box<i32>`, а упаковка выделяет память в куче. Фактическое значение упаковки — структура, которая хранит указатель на «кучу». Когда начинает выполняться функция, осуществляется вызов `Box::new()`, который выделяет некоторый объем памяти в куче, и кладет туда `5`. Теперь память выглядит следующим образом:

Адрес	Имя	Значение
$(2^{30}) - 1$		5
...	...	...
1	y	42
0	x	→ $(2^{30}) - 1$

В нашем гипотетическом компьютере с 1Гб оперативной памяти имеется 2^{30} адресов. 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. Поскольку структура `x` хранит сырой указатель (raw pointer) на адрес, который мы выделили в куче, то значение `x` равно $(2^{30}) - 1$ — это то самое местоположение в памяти.

Мы не слишком много говорили о том, что на самом деле означает «выделить» и «освободить память» в этом контексте. 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 дальнейших примерах в этой книге, 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:

Адрес	Имя	Значение
$(2^{30}) - 1$		5
$(2^{30}) - 2$		
$(2^{30}) - 3$		
$(2^{30}) - 4$		42
...	...	...
3	y	→ $(2^{30}) - 4$
2	y	42

1	y	42
0	x	→ (2^{30}) - 1

В этом примере мы выделили четыре элемента в куче, но освободили лишь два из них. Отсюда разрыв между (2^{30}) - 1 и (2^{30}) - 4, 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. Программы на Rust используют для этого jemalloc.

Ладно, вернемся к нашему примеру. Так как эта память расположена в куче, то она может оставаться 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:

Адрес	Имя	Значение
1	y	42
0	x	??????

[moving]: Мы можем продлить время жизни памяти 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:

Адрес	Имя	Значение
1	y	→ 0
0	x	5

Значением `x` является `5`, а `y` представляет собой ссылку на `x`. То есть, ее значением является адрес памяти, по которому расположен `x`. В данном случае это `0`.

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

Адрес	Имя	Значение
3	z	42
2	i	→ 0
1	y	→ 0
0	x	5

Кадры стека используются не только для локальных имён, но также и для аргументов. Таким образом, в этом случае, наш кадр должен содержать как `i`, наш аргумент, так и `z`, наше локальное имя. `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 находящуюся по этому указателю память, то это может привести к ошибкам в дальнейшей работе.

Сложный пример

Хорошо, давайте рассмотрим следующую, более сложную 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()`:

Адрес	Имя	Значение
$(2^{30}) - 1$		20
...	...	...
2	j	→ 0
1	i	→ $(2^{30}) - 1$
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()`:

Адрес	Имя	Значение
$(2^{30}) - 1$		20
...	...	...
5	z	→ 4
4	y	10
3	x	→ 0

2	j	→ 0
1	i	→ (2^{30}) - 1
0	h	3

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

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

Адрес	Имя	Значение
(2^{30}) - 1		20
...	...	...
7	g	100
6	f	→ 4
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ (2^{30}) - 1
0	h	3

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

Адрес	Имя	Значение
(2^{30}) - 1		20
...	...	...
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ (2^{30}) - 1
0	h	3

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

Адрес	Имя	Значение
(2^{30}) - 1		20
(2^{30}) - 2		5

...	...	...
10	e	→ 9
9	d	→ (2^{30}) - 2
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}) - 1
0	h	3

Тут мы выделяем другое значение в куче, и поэтому мы вычитаем единицу из (2^{30}) - 1. Это выражение написать легче, чем `1 073 741 822`. В любом случае, переменные создаются, как обычно.

В конце `bar()` вызывает `baz()`:

Адрес	**Имя**	**Значение**
(2^{30}) - 1		20
(2^{30}) - 2		5
...	...	...
12	g	100
11	f	→ 9
10	e	→ 9
9	d	→ (2^{30}) - 2
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}) - 1
0	h	3

Сейчас мы на наибольшей глубине! Поздравляем с достижением данной точки.

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

Адрес	Имя	Значение
$(2^{30}) - 1$		20
$(2^{30}) - 2$		5
...	...	...
10	e	→ 9
9	d	→ $(2^{30}) - 2$
8	c	5
7	b	→ 4
6	a	→ 0
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ $(2^{30}) - 1$
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: $(2^{30}) - 2$.

Адрес	Имя	Значение
$(2^{30}) - 1$		20
...	...	...
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ $(2^{30}) - 1$
0	h	3

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

Адрес	Имя	Значение
$(2^{30}) - 1$		20
...	...	...
2	j	→ 0

1	i	→ (2^{30}) - 1
0	h	3

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

А что делают другие языки?

Большинство языков со 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, не достигают своей цели во всех 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 («последним пришёл — первым вышел») фундаментально проще. Это значит, что программа быстрее исполняется, и проще по смыслу.

Эффективность во время выполнения

Управление памятью для стека тривиально: машина просто увеличивает или уменьшает одно значение, так называемый «указатель стека» (stack pointer). Управление памятью для кучи сложнее: память, выделенная в куче, освобождается в произвольные моменты, а каждая область выделенной в куче памяти может быть произвольного размера. Распределителю памяти, как правило, требуется приложить гораздо больше усилий для определения областей, которые 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 счётчиками 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)
>
> Тестирование программы может быть очень эффективным способом показать наличие ошибок, но оно безнадёжно неподходяще для доказательства их отсутствия.
>
> Дейкстра, Эдсгер Вибе, «The Humble Programmer» (1972)

Давайте поговорим о том, как тестировать код на Rust. 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 атрибутом `test`. Давайте создадим новый проект 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, а любой тест, который вызывает `panic!`, выполняется неудачно. Давайте сделаем тест, который выполнится неудачно:

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

`assert!` — это макрос, определенный в Rust, и принимающий 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. Можно использовать `$?` на OS X и Linux:

```
$ echo $?
101
```

На Windows, если вы используете `cmd`:

```
    echo %ERRORLEVEL%
```

И если вы используете PowerShell:

```
    echo $LASTEXITCODE # сам код
    echo $? # логическое, успешно или не успешно
```

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

Можно 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 равенство двух аргументов:

```
#[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
```

Тесты `should_panic` могут быть хрупкими, поскольку 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 «настоящий» 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 её вызова с ожидаемым результатом.

Тесты с атрибутом `ignore`

Некоторые тесты могу занимать много времени на выполнение. Такие тесты могут быть отключены по умолчанию с помощью атрибута `ignore`:

```
fn it_works() {
    assert_eq!(4, add_two(2));
}

#[test]
#[ignore]
fn expensive_test() {
    // код, который занимает час на выполнение
}
```

Теперь запустим наши тесты и видим, что `it_works` запускается, а `expensive_test` нет:

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

running 2 tests
test expensive_test ... ignored
test it_works ... ok

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

   Doc-tests adder

running 0 tests

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

Дорогостоящие тесты могут быть запущены с помощью команды `cargo test -- --ignored`:

```
$ cargo test -- --ignored
     Running target/adder-91b3e234d4ed382a

running 1 test
test expensive_test ... 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
```

Аргумент `--ignored` — это аргумент для тестового исполняемого файла, а не для Cargo, именно поэтому команда выглядит так `cargo test -- --ignored`.

Тесты в модуле `test`

Есть один нюанс, из-за которого наш пример нельзя назвать идиоматичным: отсутствует модуль тестирования. Идиоматичный вариант нашего примера будет выглядеть 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, если нужно, определить вспомогательные функции, 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 возможность `glob`. Давайте изменим `src/lib.rs` соответствующим образом:

```
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 модульные тесты (unit tests). Любой код, задачей которого является только лишь тестирование небольшого кусочка функциональности, имеет смысл перенести в этот модуль. Но что если мы хотим написать «интеграционные тесты» (integration tests)? 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 `extern crate adder`. Это потому, что тесты в директории `tests` — это отдельный контейнер, и, следовательно, мы должны компоноваться с нашей библиотекой. 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 старый модульный тест, а также новый интеграционный тест.

Это все, что касается директории `tests`. Модуль `test` здесь не нужен, так как здесь всё относится к тестам.

Давайте, наконец, перейдем к третьей части: тесты в 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 была написана документация. Для того, чтобы такой ситуации не возникало, Rust поддерживает автоматический запуск примеров в документации (имейте ввиду, что это работает только с библиотеками). Вот дополненный `src/lib.rs` с примерами:

```
//! Контейнер `adder` предоставляет функции сложения чисел.
//!
//! # Examples
//!
//! ```
//! assert_eq!(4, adder::add_two(2));
//! ```

/// Эта функция прибавляет 2 к своему аргументу.
///
/// # 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. (Примечание переводчика: заголовок `# Examples` имеет особое значение: его нельзя написать по-другому или написать на русском языке, иначе Rust не найдёт примеров кода в документации.)

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 увеличиваться автоматически, если вы добавите еще примеров. Например, при добавлении ещё одного функционального теста, он получит имя `add_two_1`.

Мы не рассмотрели все детали написания тестов в документации. Подробнее смотрите главу Документация.

Последнее замечание: тесты в документации *не работают* для исполняемых файлов. Подробнее об организации файлов можно узнать в главе Контейнеры и модули.

Условная компиляция

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

```
#[cfg(foo)]

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

Над атрибутами конфигурации определены логические операции:

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

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

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

Они 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]
# по умолчанию, никаких дополнительных возможностей
default = []

# возможность «secure-password» зависит от пакета bcrypt
secure-password = ["bcrypt"]
```

Если вы определите такие возможности, Cargo передаст флаг в `rustc`:

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

Совокупность этих флагов конфигурации (`cfg`) 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 в `rustc` будет передан флаг `--cfg feature="foo"`, и результат 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` будет отсутствовать.

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!");
}
```

Значение флага будет заменено на `true` или `false` во время компиляции, в зависимости от настройки конфигурации.

Документация

Документация является важной частью любого программного проекта, и в Rust ей уделяется не меньше внимания, чем самому коду. 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 отдельных файлов в формате Markdown.

Документирование исходного кода

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

```
/// Создаёт новый `Rc<T>`.
///
/// # Examples
///
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
pub fn new(value: T) -> Rc<T> {
    // здесь реализация
}
```

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, это на использование `///` вместо `//`. Символы `///` указывают, что это документирующий комментарий.

Документирующие комментарии пишутся на Markdown.

Rust отслеживает такие комментарии, и использует их при создании документации.

При документировании таких вещей, как перечисления, нужно учитывать некоторые особенности работы `rustdoc`. Такой код работает:

```
/// Тип `Option`. Подробнее смотрите [документацию уровня модуля](http://doc.rust-lang.org/
).
enum Option<T> {
    /// Нет значения
    None,
    /// Некоторое значение `T`
    Some(T),
}
```

А такой — нет:

```
/// Тип `Option`. Подробнее смотрите [документацию уровня модуля](http://doc.rust-lang.org/
).
enum Option<T> {
    None, /// Нет значения
    Some(T), /// Некоторое значение `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:

```
/// Создаёт новый `Rc<T>`.
```

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

```
///
/// Подробности создания `Rc<T>`, возможно, описывающие сложности семантики,
/// дополнительные опции, и всё остальное.
///
```

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 обознаются 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
```

Раздел `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 — т.е. обнаруживает/вызывает панику, то очень важно задокументировать это.

```
/// # Failures
```

Раздел `Failures`. Если ваша функция или метод возвращает `Result<T, E>`, то хорошим тоном является описание условий, при которых она возвращает `Err(E)`. Это чуть менее важно, чем описание `Panics`, потому как неудача кодируется в системе типов, но это не значит, что стоит пренебрегать данной возможностью.

```
/// # Safety
```

Раздел `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. Примеры должны размещаться внутри блоков кода, о которых мы сейчас поговорим. Этот раздел может иметь более одного подраздела:

```
/// # Examples
///
/// Простые образцы типа `&str`:
///
/// ```
/// let v: Vec<&str> = "И была у них курочка Ряба".split(' ').collect();
/// assert_eq!(v, vec!["И", "была", "у", "них", "курочка", "Ряба"]);
/// ```
///
/// Более сложные образцы с замыканиями:
///
/// ```
/// let v: Vec<&str> = "абв1где2жзи".split(|c: char| c.is_numeric()).collect();
/// assert_eq!(v, vec!["абв", "где", "жзи"]);
/// ```
```

Давайте подробно обсудим блоки кода.

Блок кода

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

```
/// ```
/// println!("Привет, мир");
/// ```
```

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 создания документации. Это позволяет избежать устаревания примеров. 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 он пожалуется при попытке создания документации.

Тесты в документации

Давайте обсудим наш пример документации:

```
/// ```
/// println!("Привет, мир");
/// ```
```

Заметьте, что здесь нет нужды в `fn main()` или чём-нибудь подобном. `rustdoc` автоматически добавит оборачивающий `main()` 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 `rustdoc` использует для обработки примеров:

1. Любые ведущие (leading) атрибуты `#![foo]` остаются без изменений в качестве атрибутов контейнера.
2. Будут вставлены некоторые общие атрибуты `allow`, в том числе: `unused_variables`, `unused_assignments`, `unused_mut`, `unused_attributes`, `dead_code`. Небольшие примеры часто приводят к срабатыванию этих анализов.
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, что насчёт всех этих примеров кода с `///`, о которых мы говорили? Простой текст, обработанный `rustdoc`, выглядит так:

```
/// Некоторая документация.
# fn foo() {}
```

А исходный текст на Rust после обработки выглядит так:

```
/// Некоторая документация.
```

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 документирующем комментарии 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, вот код:

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

Ниже приведено отрисованное объяснение этого кода.

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);
```

А 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 кусочки кода, которые как-то относятся к той или иной части вашего объяснения.

Документирование макросов

Вот пример документирования макроса:

```
/// Паниковать с данным сообщением, если только выражение не является истиной.
///
/// # Examples
///
/// ```
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(1 + 1 == 2, "Математика сломалась.");
/// # }
/// ```
///
/// ```should_panic
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(true == false, "Я сломан.");
/// # }
/// ```
#[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 выводе.

Запуск тестов в документации

Для запуска тестов можно использовать одну из двух комманд

```
$ rustdoc --test path/to/my/crate/root.rs
# или
$ 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 компонуется с библиотекой, которую надо протестировать, но в случае с исполняемым файлом компоноваться не с чем.

Есть еще несколько полезных аннотаций, которые помогают `rustdoc` работать правильно 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 неё 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!("Привет, мир");
/// }
/// ```
```

Аннотация `no_run` указывает, что код должен компилироваться, но запускать его на выполнение не требуется. Это важно для таких примеров, которые должны успешно компилироваться, но выполнение которых оказывается бесконечным циклом! Например: «Вот как запустить сетевой сервис».

Документирование модулей

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

```
mod foo {
    //! Это документация для модуля `foo`.
    //!
    //! # Examples

    // ...
}
```

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

```
//! Модуль использования разных `foo`.
//!
//! Модуль `foo` содержит много полезной функциональности ла-ла-ла
```

Стиль документирующих комментариев

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 файлах Markdown, вам не нужно добавлять префикс документирующего комментария, `///`. 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 файле Markdown. Однако есть один недостаток: файлы Markdown должны иметь заголовок наподобие этого:

```
% Заголовок

Это пример документации.
```

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"]
```

Т.е. 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 для изменения некоторых настроек, или при написании макроса.

Ре-экспорт

`rustdoc` будет показывать документацию для общедоступного (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 в обоих местах будет использована одна и та же документация.

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 генерирует `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 настройки:

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

Замечание по безопасности

Комментарии в документации в формате Markdown 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 изменяемое имя, 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 перебор элементов закончен. Если мы получаем `Some(i32)`, то печатаем его, а если `None`, то прекращаем выполнение цикла оператором `break`.

Этот пример, по большому счету, делает то же самое, что 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 заключается 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 плохой практике в Rust, связанной с использованием диапазонов. Она продемонстрирована в примере ниже.

Вот, только что мы говорили о том, какие диапазоны крутые. Но диапазоны также и очень примитивны. Например, если вам нужно перебрать 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 пройти по элементам векторов напрямую, как показано ниже:

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

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

Есть две причины предпочесть прямое использование итератора. Во-первых, это яснее выражает наше намерение. Мы обходим элементы вектора, а не индексы 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 работы `println!`. На самом деле `num` имеет тип `&i32`. То есть, это ссылка на `i32`, а не сам `i32`. `println!` выполняет разыменование переменной за нас, поэтому мы не видим его в исходном коде. 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 владельцем, 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 нужно, давайте поговорим о том, что же можно использовать 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 и 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 — это диапазоны.

Потребители

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

```
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, в какую коллекцию (например, вектор, список, и т.д.) вы хотите собрать элементы, и поэтому тип необходимо указать явно. Вот версия, которая компилируется:

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

Если помните, 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 вывод типа `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!("У нас есть несколько чисел!"),
    None => println!("Числа не найдены :("),
}
```

`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, `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 схематично можно представить в виде: `fold(base, |accumulator, element| ...)`. Он принимает два аргумента: первый - это элемент, называемый *базой*; 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 запутанно. Давайте рассмотрим значения всех элементов итератора:

база	аккумулятор	элемент	результат замыкания
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 значением 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 большое значение в связи с одним свойством итераторов, о котором мы еще не говорили: ленивость. Давайте ещё немного поговорим об итераторах, и вы поймете, почему потребители так важны.

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* — они не обязаны создавать все значения заранее. Например, этот код 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 создавал последовательность. Давайте добавим потребителя:

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

Теперь `collect()` потребует, чтобы диапазон выдавал ему какие-нибудь числа, поэтому он сгенерирует последовательность.

Диапазоны — это один из двух основных типов итераторов. Другой часто используемый итератор — `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 хорошо послужат вам. Есть и более продвинутые итераторы, в том числе и те, которые генерируют бесконечную последовательность.

Вот и все, что касается итераторов. Последнее понятие в этой теме, о котором мы хотели бы рассказать — адаптеры итераторов. Давайте перейдем к нему!

Адаптеры итераторов

Адаптеры итераторов получают итератор и изменяют его каким-то образом, выдавая новый итератор. 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);
 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Причина этого — ленивость итераторов! То замыкание никогда не будет выполнено. Пример ниже не напечатает ни одного значения:

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

Если вы пытаетесь выполнить замыкание ради побочных эффектов (вроде печати), 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, которые мы упоминали раньше:

```
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 мы используем образец `&x`, чтобы извлечь само целое число. Это необходимо, поскольку `filter` не потребляет элементы, которые выдаются во время итерации, а лишь выдаёт ссылку.)

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, и вы также можете написать свой собственный итератор. Итераторы обеспечивают безопасный и эффективный способ работы со всеми видами списков. Сперва работать с ними немного непривычно, но чем больше вы с ними сталкиваетесь, тем больше они вас цепляют. Для получения полного списка различных итераторов, адаптеров и потребителей смотрите документацию модуля iter.

Многозадачность

Многозадачность и параллелизм являются невероятно важными проблемами в информатике. Это актуальная тема для современной индустрии. У компьютеров все больше и больше ядер, но многие программисты не готовы в полной мере использовать их.

Средства Rust для безопасной работы с памятью в полной мере применимы и при работе в многозадачной среде. Даже многозадачные программы на Rust должны безопасно работать с памятью, и не создавать состояний гонок по данным. Система типов Rust достаточно мощна, чтобы справиться с этими задачами на этапе компиляции.

Прежде чем мы поговорим об особенностях многозадачности в Rust, важно понять вот что: Rust — достаточно низкоуровневый язык, поэтому вся поддержка многозадачности реализована в стандартной библиотеке, а не в самом языке. Это означает, что если вам не нравится какой-то аспект реализации многозадачности в Rust, вы всегда можете создать альтернативную библиотеку. mio — реально существующий пример такого подхода.

Справочная информация: `Send` и `Sync`

Рассуждать о многозадачности довольно трудно. Rust строго статически типизирован, и это помогает нам делать выводы о коде. В связи с этим Rust предоставляет два типажа, помогающих нам разбираться в любом коде, который вообще может быть многозадачным.

`Send`

Первый типаж, о котором мы будем говорить, называется `Send`. Когда тип `T` реализует `Send`, это указывает компилятору, что владение переменными этого типа можно безопасно перемещать между потоками.

Это важно для соблюдения некоторых ограничений. Например, это имеет значение, когда у нас есть канал, соединяющий два потока, и мы хотим 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 мы оборачиваем библиотеку чужого кода (FFI), и она не является потокобезопасной, то нам не следует реализовывать типаж `Send`, и компилятор поможет нам убедиться в невозможности покинуть текущий поток.

`Sync`

Второй из этих типажей называется `Sync`. Когда тип `T` реализует `Sync`, это указывает компилятору, что использование переменных этого типа не приводит к небезопасной работе с памятью в многопоточной среде.

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 получить надежные гарантии о свойствах вашего кода в условиях многозадачности. Прежде чем мы покажем, как этого достигнуть, сначала мы должны узнать, как вообще написать многозадачную программу в Rust!

Потоки

Стандартная библиотека Rust предоставляет библиотеку многопоточности, которая позволяет запускать код на Rust 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 используется 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

Вчитайтесь: «безопасное совместное использование изменяемого состояния». Похоже на ложь, не так ли? Многие программисты считают, что организовать многопоточную работу с изменяемым состоянием очень сложно и почти невозможно. Но благодаря системе типов Rust, это всё же правда — безопасно работать с изменяемыми данными можно.

Кто-то однажды сказал это:

> Совместно используемое изменяемое состояние является корнем всех зол. Большинство языков пытаются решить эту проблему через часть, отвечающую за «изменяемое», но Rust решает ее через часть, отвечающую за «совместно используемое».

Та же самая система владения, которая помогает предотвратить неправильное использование указателей, также помогает исключить гонки по данным, один из худших видов ошибок многозадачности.

В качестве примера приведем программу на Rust, которая входила бы в состояние гонки по данным на многих языках. На Rust она не скомпилируется:

```
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: мы работаем с `data` в каждом потоке. При этом, поток становится владельцем того, что он получает как часть окружения замыкания. А это значит, что 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 обмениваться между потоками.

Чтобы гарантировать, что его можно безопасно использовать из нескольких потоков, `Arc<T>` предполагает наличие еще одного свойства у вложенного типа. Он предполагает, что `T` реализует типаж `Sync`. В нашем случае мы также хотим, чтобы была возможность изменять вложенное значение. Нам нужен тип, который может обеспечить изменение своего

содержимого лишь одним пользователем одновременно. 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 охранное значение мьютекса 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 полученная ссылка 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, 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 ждём какое-то время, пока потоки отработают. Это не идеальный способ дождаться окончания их работы: возможно, мы выбрали разумное время ожидания но, скорее всего, мы будем ждать либо больше чем нужно, либо меньше чем нужно, в зависимости от того, сколько на самом деле времени потребуется потокам, чтобы закончить вычисления.

Есть более точные способы синхронизации потоков, и несколько из них реализовано в стандартной библиотеке Rust. Давайте поговорим об одном из них: каналах.

Каналы

Вот версия нашего кода, которая использует для синхронизации каналы, 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 создать новый канал. В этом примере мы в каждом из десяти потоков вызываем метод `send`, который передает по каналу пустой кортеж `()`, а затем в главном потоке ждем, пока не будут приняты все десять значений.

Хотя по этому каналу посылается просто сигнал (пустой кортеж `()` не несёт никаких данных), в общем случае мы можем отправить по каналу любое значение, которое реализует 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, создаётся поток, в котором вычисляется ответ, а затем этот ответ с помощью метода `send()` передаётся 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());
```

Используемый в коде выше метод `join()` структуры `Thread` возвращает `Result`, что позволяет нам проверить, паниковал ли поток, или он завершился нормально.

Обработка ошибок

Как и многие языки программирования, Rust призывает разработчика определенным способом обрабатывать ошибки. Вообще, существует два общих подхода обработки ошибок: с помощью исключений и через возвращаемые значения. И Rust предпочитает возвращаемые значения.

В этой главе мы намерены подробно изложить работу с ошибками в Rust. Более того, мы попробуем раз за разом погружаться в обработку ошибок с различных сторон, так что под конец у вас будет уверенное практическое представление о том, как все это сходится воедино.

В наивной реализации обработка ошибок в Rust может выглядеть многословной и раздражающей. Мы рассмотрим основные камни преткновения, а также продемонстрируем, как сделать обработку ошибок лаконичной и удобной, пользуясь стандартной библиотекой.

Содержание

Эта глава очень длинная, в основном потому, что мы начнем с самого начала — рассмотрения типов-сумм (sum type) и комбинаторов, и далее попытаемся последовательно объяснить подход Rust к обработке ошибок. Так что разработчики, которые имеют опыт работы с другими выразительными системами типов, могут свободно перескакивать от раздела к разделу.

Основы

Обработку ошибок можно рассматривать как *вариативный анализ* того, было ли некоторое вычисление выполнено успешно или нет. Как будет показано далее, ключом к удобству обработки ошибок является сокращение количества явного вариативного анализа, который должен выполнять разработчик, сохраняя при этом код легко сочетаемым с другим кодом (composability).

(Примечание переводчика: Вариативный анализ – это один из наиболее общеприменимых методов аналитического мышления, который заключается в рассмотрении проблемы, вопроса или некоторой ситуации с точки зрения каждого возможного конкретного случая. При этом рассмотрение по отдельности каждого такого случая является достаточным для того, чтобы решить первоначальный вопрос.

Важным аспектом такого подхода к решению проблем является то, что такой анализ должен быть исчерпывающим (exhaustive). Другими словами, при использовании вариативного анализа должны быть рассмотрены все возможные случаи.

В Rust вариативный анализ реализуется с помощью синтаксической конструкции `match`. При этом компилятор гарантирует, что такой анализ будет исчерпывающим: если разработчик не рассмотрит все возможные варианты заданного значения, программа не будет скомпилирована.)

Сохранять сочетаемость кода важно, потому что без этого требования мы могли бы просто получать `panic` всякий раз, когда мы сталкивались бы с чем-то неожиданным. (`panic` вызывает прерывание текущего потока и, в большинстве случаев, приводит к завершению всей программы.) Вот пример:

```
// Попробуйте угадать число от 1 до 10.
// Если заданное число соответствует тому, что мы загадали, возвращается true.
// В противном случае возвращается false.
fn guess(n: i32) -> bool {
    if n < 1 || n > 10 {
        panic!("Неверное число: {}", n);
    }
    n == 5
}

fn main() {
    guess(11);
}
```

Если попробовать запустить этот код, то программа аварийно завершится с сообщением вроде этого:

```
thread '<main>' panicked at 'Неверное число: 11', src/bin/panic-simple.rs:6
```

Вот другой, менее надуманный пример. Программа, которая принимает число в качестве аргумента, удваивает его значение и печатает на экране.

```
use std::env;

fn main() {
    let mut argv = env::args();
    let arg: String = argv.nth(1).unwrap(); // ошибка 1
    let n: i32 = arg.parse().unwrap();      // ошибка 2
    println!("{}", 2 * n);
}
```

Если вы запустите эту программу без параметров (ошибка 1) или если первый параметр будет не целым числом (ошибка 2), программа завершится паникой, так же, как и в первом примере.

Обработка ошибок в подобном стиле подобна слону в посудной лавке. Слон будет нестись в направлении, в котором ему вздумается, и крушить все на своем пути.

Объяснение `unwrap`

В предыдущем примере мы утверждали, что программа будет просто паниковать, если будет выполнено одно из двух условий для возникновения ошибки, хотя, в отличии от первого примера, в коде программы нет явного вызова `panic`. Тем не менее, вызов `panic` встроен в вызов `unwrap`.

Вызывать `unwrap` в Rust подобно тому, что сказать: "Верни мне результат вычислений, а если произошла ошибка, просто паникуй и останавливай программу". Мы могли бы просто показать исходный код функции `unwrap`, ведь это довольно просто, но перед этим мы должны разобрать с типами `Option` и `Result`. Оба этих типа имеют определенный для них метод `unwrap`.

Тип `Option`

Тип `Option` объявлен в стандартной библиотеке:

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

Тип `Option` — это способ выразить *возможность отсутствия* чего бы то ни было, используя систему типов Rust. Выражение *возможности отсутствия* через систему типов является важной концепцией, поскольку такой подход позволяет компилятору требовать от разработчика обрабатывать такое отсутствие. Давайте взглянем на пример, который пытается найти символ в строке:

```
// Поиск Unicode-символа `needle` в `haystack`. Когда первый символ найден,
// возвращается побайтовое смещение для этого символа. Иначе возвращается `None`.
fn find(haystack: &str, needle: char) -> Option<usize> {
    for (offset, c) in haystack.char_indices() {
        if c == needle {
            return Some(offset);
        }
    }
    None
}
```

Обратите внимание, что когда эта функция находит соответствующий символ, она возвращает не просто `offset`. Вместо этого она возвращает `Some(offset)`. `Some` — это вариант или *конструктор значения* для типа `Option`. Его можно интерпретировать как функцию типа `fn<T>(value: T) -> Option<T>`. Соответственно, `None` — это также конструктор значения, только у него нет параметров. Его можно интерпретировать как функцию типа `fn<T>() -> Option<T>`.

Может показаться, что мы подняли много шума из ничего, но это только половина истории. Вторая половина — это *использование* функции `find`, которую мы написали. Давайте попробуем использовать ее, чтобы найти расширение в имени файла.

```
fn main() {
    let file_name = "foobar.rs";
    match find(file_name, '.') {
        None => println!("Расширение файла не найдено."),
        Some(i) => println!("Расширение файла: {}", &file_name[i+1..]),
    }
}
```

Этот код использует сопоставление с образцом чтобы выполнить *вариативный анализ* для возвращаемого функцией `find` значения `Option<usize>`. На самом деле, вариативный анализ является единственным способом добраться до значения, сохраненного внутри `Option<T>`. Это означает, что вы, как разработчик, обязаны обработать случай, когда значение `Option<T>` равно `None`, а не `Some(t)`.

Но подождите, как насчет `unwrap`, который мы до этого использовали? Там не было никакого вариативного анализа! Вместо этого, вариативный анализ был перемещен внутрь метода `unwrap`. Вы можете сделать это самостоятельно, если захотите:

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

impl<T> Option<T> {
    fn unwrap(self) -> T {
        match self {
            Option::Some(val) => val,
            Option::None =>
              panic!("called `Option::unwrap()` on a `None` value"),
        }
    }
}
```

Метод `unwrap` *абстрагирует вариативный анализ*. Это именно то, что делает `unwrap` удобным в использовании. К сожалению, `panic!` означает, что `unwrap` неудобно сочетать с другим кодом: это слон в посудной лавке.

Совмещение значений `Option<T>`

В предыдущем примере мы рассмотрели, как можно воспользоватся `find` для того, чтобы получить расширение имени файла. Конечно, не во всех именах файлов можно найти `.`, так что существует вероятность, что имя некоторого файла не имеет расширения. Эта *возможность отсутствия* интерпретируется на уровне типов через использование `Option<T>`. Другими словами, компилятор заставит нас рассмотреть возможность того, что расширение не существует. В нашем случае мы просто печатаем сообщение об этом.

Получение расширения имени файла — довольно распространенная операция, так что имеет смысл вынести код в отдельную функцию:

```
// Возвращает расширение заданного имени файла, а именно все символы,
// идущие за первым вхождением `.` в имя файла.
// Если в `file_name` нет ни одного вхождения `.`, возвращается `None`.
fn extension_explicit(file_name: &str) -> Option<&str> {
    match find(file_name, '.') {
        None => None,
        Some(i) => Some(&file_name[i+1..]),
    }
}
```

(Подсказка: не используйте этот код. Вместо этого используйте метод `extension` из стандартной библиотеки.)

Код выглядит простым, но его важный аспект заключается в том, что функция `find` заставляет нас рассмотреть вероятность отсутствия значения. Это хорошо, поскольку это означает, что компилятор не позволит нам случайно забыть о том варианте, когда в имени файла отсутствует расширение. С другой стороны, каждый раз выполнять явный вариативный анализ, подобно тому, как мы делали это в `extension_explicit`, может стать немного утомительным.

На самом деле, вариативный анализ в `extension_explicit` является очень распространенным паттерном: если `Option<T>` владеет определенным значением `T`, то выполнить его преобразование с помощью функции, а если нет — то просто вернуть `None`.

Rust поддерживает параметрический полиморфизм, так что можно очень легко объявить комбинатор, который абстрагирует это поведение:

```
fn map<F, T, A>(option: Option<T>, f: F) -> Option<A> where F: FnOnce(T) -> A {
    match option {
        None => None,
        Some(value) => Some(f(value)),
    }
}
```

В действительности, `map` определен в стандартной библиотеке как метод `Option<T>`.

Вооружившись нашим новым комбинатором, мы можем переписать наш метод `extension_explicit` так, чтобы избавиться от вариативного анализа:

```
// Возвращает расширение заданного имени файла, а именно все символы,
// идущие за первым вхождением `.` в имя файла.
// Если в `file_name` нет ни одного вхождения `.`, возвращается `None`.
fn extension(file_name: &str) -> Option<&str> {
    find(file_name, '.').map(|i| &file_name[i+1..])
}
```

Есть еще одно поведение, которое можно часто встретить — это использование значения по-умолчанию в случае, когда значение `Option` равно `None`. К примеру, ваша программа может считать, что расширение файла равно `rs` в случае, если на самом деле оно отсутствует.

Легко представить, что этот случай вариативного анализа не специфичен только для расширений файлов — такой подход может работать с любым `Option<T>`:

```
fn unwrap_or<T>(option: Option<T>, default: T) -> T {
    match option {
        None => default,
        Some(value) => value,
    }
}
```

Хитрость только в том, что значение по-умолчанию должно иметь тот же тип, что и значение, которое может находится внутри `Option<T>`. Использование этого метода элементарно:

```
fn main() {
    assert_eq!(extension("foobar.csv").unwrap_or("rs"), "csv");
    assert_eq!(extension("foobar").unwrap_or("rs"), "rs");
}
```

(Обратите внимание, что `unwrap_or` объявлен как метод `Option<T>` в стандартной библиотеке, так что мы воспользовались им вместо функции, которую мы объявили ранее. Не забудьте также изучить более общий метод `unwrap_or_else`).

Существует еще один комбинатор, на который, как мы думаем, стоит обратить особое внимание: `and_then`. Он позволяет легко сочетать различные вычисления, которые допускают *возможность отсутствия*. Пример — большая часть кода в этом разделе, который связан с определением расширения заданного имени файла. Чтобы делать это, нам для начала необходимо узнать имя файла, которое как правило извлекается из *файлового пути*. Хотя большинство файловых путей содержат имя файла, подобное нельзя сказать обо *всех* файловых путях. Примером могут послужить пути `.`, `..` или `/`.

Таким образом, мы определили задачу нахождения расширения заданного *файлового пути*. Начнем с явного вариативного анализа:

```
fn file_path_ext_explicit(file_path: &str) -> Option<&str> {
    match file_name(file_path) {
        None => None,
        Some(name) => match extension(name) {
            None => None,
            Some(ext) => Some(ext),
        }
    }
}

fn file_name(file_path: &str) -> Option<&str> {
  unimplemented!() // опустим реализацию
}
```

Можно подумать, мы могли бы просто использовать комбинатор `map`, чтобы уменьшить вариативный анализ, но его тип не совсем подходит. Дело в том, что `map` принимает функцию, которая делает что-то только с внутренним значением. Результат такой функции *всегда* оборачивается в `Some`. Вместо этого, нам нужен метод, похожий `map`, но который позволяет вызывающему передать еще один `Option`. Его общая реализация даже проще, чем `map`:

```
fn and_then<F, T, A>(option: Option<T>, f: F) -> Option<A>
        where F: FnOnce(T) -> Option<A> {
    match option {
        None => None,
        Some(value) => f(value),
    }
}
```

Теперь мы можем переписать нашу функцию `file_path_ext` без явного вариативного анализа:

```
fn file_path_ext(file_path: &str) -> Option<&str> {
    file_name(file_path).and_then(extension)
}
```

Т и п `Option` имеет много других комбинаторов определенных в стандартной библиотеке. Очень полезно просмотреть этот список и ознакомиться с доступными методами — они не раз помогут вам сократить количество вариативного анализа. Ознакомление с этими комбинаторами окупится еще и потому, что многие из них определены с аналогичной семантикой и для типа `Result`, о котором мы поговорим далее.

Комбинаторы упрощают использование типов вроде `Option`, ведь они сокращают явный вариативный анализ. Они также соответствуют требованиям сочетаемости, поскольку они позволяют вызывающему обрабатывать возможность отсутствия результата собственным способом. Такие методы, как `unwrap`, лишают этой возможности, ведь они будут паниковать в случае, когда `Option<T>` равен `None`.

Тип `Result`

Тип `Result` также [определен в стандартной библиотеке]:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Тип `Result` — это продвинутая версия `Option`. Вместо того, чтобы выражать возможность *отсутствия*, как это делает `Option`, `Result` выражает возможность *ошибки*. Как правило, *ошибки* необходимы для объяснения того, почему результат определенного вычисления не был получен. Строго говоря, это более общая форма `Option`. Рассмотрим следующий псевдоним типа, который во всех смыслах семантически эквивалентен реальному `Option<T>`:

```
type Option<T> = Result<T, ()>;
```

Здесь второй параметр типа `Result` фиксируется и определяется через `()` (произносится как "unit" или "пустой кортеж"). Тип `()` имеет ровно одно значение — `()`. (Да, это тип и значение этого типа, которые выглядят одинаково!)

Тип `Result` — это способ выразить один из двух возможных исходов вычисления. По соглашению, один исход означает ожидаемый результат или "`Ok`", в то время как другой исход означает исключительную ситуацию или "`Err`".

Подобно `Option`, тип `Result` имеет метод `unwrap`, [определенный в стандартной библиотеке]. Давайте объявим его самостоятельно:

```
impl<T, E: ::std::fmt::Debug> Result<T, E> {
    fn unwrap(self) -> T {
        match self {
            Result::Ok(val) => val,
            Result::Err(err) =>
              panic!("called `Result::unwrap()` on an `Err` value: {:?}", err),
        }
    }
}
```

Это фактически то же самое, что и [определение `Option::unwrap`], за исключением того, что мы добавили значение ошибки в сообщение `panic!`. Это упрощает отладку, но это также вынуждает нас требовать от типа-параметра `E` (который представляет наш тип ошибки) реализации `Debug`. Поскольку подавляющее большинство типов должны реализовывать

`Debug`, обычно на практике такое ограничение не мешает. (Реализация `Debug` для некоторого типа просто означает, что существует разумный способ печати удобочитаемого описания значения этого типа.)

Окей, давайте перейдем к примеру.

Преобразование строки в число

Стандартная библиотека Rust позволяет элементарно преобразовывать строки в целые числа. На самом деле это настолько просто, что возникает соблазн написать что-то вроде:

```
fn double_number(number_str: &str) -> i32 {
    2 * number_str.parse::<i32>().unwrap()
}

fn main() {
    let n: i32 = double_number("10");
    assert_eq!(n, 20);
}
```

Здесь вы должны быть скептически настроены по-поводу вызова `unwrap`. Если строку нельзя преобразовать в число, вы получите панику:

```
thread '<main>' panicked at 'called `Result::unwrap()` on an `Err` value: ParseIntError { k
ind: InvalidDigit }', /home/rustbuild/src/rust-buildbot/slave/beta-dist-rustc-linux/build/s
rc/libcore/result.rs:729
```

Это довольно неприятно, и если бы подобное произошло в используемой вами библиотеке, вы могли бы небезосновательно разгневаться. Так что нам стоит попытаться обработать ошибку в нашей функции, и пусть вызывающий сам решит что с этим делать. Это означает необходимость изменения типа, который возвращается `double_number`. Но на какой? Чтобы понять это, необходимо посмотреть на сигнатуру `метода parse` из стандартной библиотеки:

```
impl str {
    fn parse<F: FromStr>(&self) -> Result<F, F::Err>;
}
```

Хмм. По крайней мере мы знаем, что должны использовать `Result`. Вполне возможно, что метод мог возвращать `Option`. В конце концов, строка либо парсится как число, либо нет, не так ли? Это, конечно, разумный путь, но внутренняя реализация знает *почему* строка не может быть преобразована в целое число. (Это может быть пустая строка, или неправильные цифры, слишком большая или слишком маленькая длина и т.д.) Таким образом, использование `Result` имеет смысл, ведь мы хотим предоставить больше информации, чем просто "отсутствие". Мы хотим сказать, *почему* преобразование не удалось. Вам стоит рассуждать похожим образом, когда вы сталкиваетесь с выбором между `Option` и `Result`. Если вы можете предоставить подробную информацию об ошибке, то вам, вероятно, следует это сделать. (Позже мы поговорим об этом подробнее.)

Хорошо, но как мы запишем наш тип возвращаемого значения? Метод `parse` является обобщенным (generic) для всех различных типов чисел из стандартной библиотеки. Мы могли бы (и, вероятно, должны) также сделать нашу функцию обобщенной, но давайте пока остановимся на конкретной реализации. Нас интересует только тип `i32`, так что нам стоит найти его реализацию `FromStr` (выполните поиск в вашем браузере по строке "FromStr") и посмотреть на его ассоциированный тип `Err`. Мы делаем это, чтобы определить конкретный тип ошибки. В данном случае, это `std::num::ParseIntError`. Наконец, мы можем переписать нашу функцию:

```
use std::num::ParseIntError;

fn double_number(number_str: &str) -> Result<i32, ParseIntError> {
    match number_str.parse::<i32>() {
        Ok(n) => Ok(2 * n),
        Err(err) => Err(err),
    }
}

fn main() {
    match double_number("10") {
        Ok(n) => assert_eq!(n, 20),
        Err(err) => println!("Error: {:?}", err),
    }
}
```

Неплохо, но нам пришлось написать гораздо больше кода! И нас опять раздражает вариативный анализ.

Комбинаторы спешат на помощь! Подобно `Option`, `Result` имеет много комбинаторов, определенных в качестве методов. Существует большой список комбинаторов, общих между `Result` и `Option`. И `map` входит в этот список:

```
use std::num::ParseIntError;

fn double_number(number_str: &str) -> Result<i32, ParseIntError> {
    number_str.parse::<i32>().map(|n| 2 * n)
}

fn main() {
    match double_number("10") {
        Ok(n) => assert_eq!(n, 20),
        Err(err) => println!("Error: {:?}", err),
    }
}
```

Все ожидаемые методы реализованы для `Result`, включая `unwrap_or` и `and_then`. Кроме того, поскольку `Result` имеет второй параметр типа, существуют комбинаторы, которые влияют только на значение ошибки, такие как `map_err` (аналог `map`) и `or_else` (аналог `and_then`).

Создание псевдонима типа `Result`

В стандартной библиотеке можно часто увидеть типы вроде `Result<i32>`. Но постойте, ведь мы определили `Result` с двумя параметрами типа. Как мы можем обойти это, указывая только один из них? Ответ заключается в определении псевдонима типа `Result`, который *фиксирует* один из параметров конкретным типом. Обычно фиксируется тип ошибки. Например, наш предыдущий пример с преобразованием строк в числа можно переписать так:

```
use std::num::ParseIntError;
use std::result;

type Result<T> = result::Result<T, ParseIntError>;

fn double_number(number_str: &str) -> Result<i32> {
    unimplemented!();
}
```

Зачем мы это делаем? Что ж, если у нас есть много функций, которые могут вернуть `ParseIntError`, то гораздо удобнее определить псевдоним, который всегда использует `ParseIntError`, так что мы не будем повторяться все время.

Самый заметный случай использования такого подхода в стандартной библиотеке — псевдоним `io::Result`. Как правило, достаточно писать `io::Result<T>`, чтобы было понятно, что вы используете псевдоним типа из модуля `io`, а не обычное определение из `std::result`. (Этот подход также используется для `fmt::Result`)

Короткое отступление: `unwrap` — не обязательно зло

Если вы были внимательны, то возможно заметили, что я занял довольно жесткую позицию по отношению к методам вроде `unwrap`, которые могут вызвать `panic` и прервать исполнение вашей программы. *В основном*, это хороший совет.

Тем не менее, `unwrap` все-таки можно использовать разумно. Факторы, которые оправдывают использование `unwrap`, являются несколько туманными, и разумные люди могут со мной не согласиться. Я кратко изложу свое *мнение* по этому вопросу:

- **Примеры и "грязный" код.** Когда вы пишете просто пример или быстрый скрипт, обработка ошибок просто не требуется. Для подобных случаев трудно найти что-либо удобнее чем `unwrap`, так что здесь его использование очень привлекательно.
- **Паника указывает на ошибку в программе.** Если логика вашего кода должна предотвращать определенное поведение (скажем, получение элемента из пустого стека), то использование `panic` также допустимо. Дело в том, что в этом случае паника будет сообщать о баге в вашей программе. Это может происходить явно, например от неудачного вызова `assert!`, или происходить потому, что индекс по массиву находится за пределами выделенной памяти.

Вероятно, это не исчерпывающий список. Кроме того, при использовании `Option` зачастую лучше использовать метод `expect`. Этот метод делает ровно то же, что и `unwrap`, за исключением того, что в случае паники напечатает ваше сообщение. Это позволит лучше

понять причину ошибки, ведь будет показано конкретное сообщение, а не просто "called unwrap on a `None` value".

Мой совет сводится к следующему: используйте здравый смысл. Есть причины, по которым слова вроде "никогда не делать X" или "Y считается вредным" не появятся в этой статье. У любых решений существуют компромиссы, и это ваша задача, как разработчика, определить, что именно является приемлемым для вашего случая. Моя цель состоит только в том, чтобы помочь вам оценить компромиссы как можно точнее.

Теперь, когда мы рассмотрели основы обработки ошибок в Rust и разобрались с `unwrap`, давайте подробнее изучим стандартную библиотеку.

Работа с несколькими типами ошибок

До этого момента мы расматривали обработку ошибок только для случаев, когда все сводилось либо только к `Option<T>`, либо только к `Result<T, SomeError>`. Но что делать, когда у вас есть и `Option`, и `Result`? Или если у вас есть `Result<T, Error1>` и `Result<T, Error2>`? Наша следующая задача — обработка *композиции различных типов ошибок*, и это будет главной темой на протяжении всей этой главы.

Совмещение `Option` и `Result`

Пока что мы говорили о комбинаторах, определенных для `Option`, и комбинаторах, определенных для `Result`. Эти комбинаторы можно использовать для того, чтобы сочетать результаты различных вычислений, не делая подробного вариативного анализа.

Конечно, в реальном коде все происходит не так гладко. Иногда у вас есть сочетания типов `Option` и `Result`. Должны ли мы прибегать к явному вариативному анализу, или можно продолжить использовать комбинаторы?

Давайте на время вернемся к одному из первых примеров в этой главе:

```
use std::env;

fn main() {
    let mut argv = env::args();
    let arg: String = argv.nth(1).unwrap(); // ошибка 1
    let n: i32 = arg.parse().unwrap(); // ошибка 2
    println!("{}", 2 * n);
}
```

Учитывая наши знания о типах `Option` и `Result`, а также их различных комбинаторах, мы можем попытаться переписать этот код так, чтобы ошибки обрабатывались должным образом, и программа не паниковала в случае ошибки.

Ньюанс заключается в том, что `argv.nth(1)` возвращает `Option`, в то время как `arg.parse()` возвращает `Result`. Они не могут быть скомпонованы непосредственно. Когда вы сталкиваетесь одновременно с `Option` и `Result`, *обычно* наилучшее решение —

преобразовать `Option` в `Result`. В нашем случае, отсутствие параметра командной строки (из `env::args()`) означает, что пользователь не правильно вызвал программу. Мы могли бы просто использовать `String` для описания ошибки. Давайте попробуем:

```
use std::env;

fn double_arg(mut argv: env::Args) -> Result<i32, String> {
    argv.nth(1)
        .ok_or("Please give at least one argument".to_owned())
        .and_then(|arg| arg.parse::<i32>().map_err(|err| err.to_string()))
}

fn main() {
    match double_arg(env::args()) {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

Рассмотрим пару новых моментов на этом примере. Во-первых, использование комбинатора `Option::ok_or`. Это один из способов преобразования `Option` в `Result`. Такое преобразование требует явного определения ошибки, которую необходимо вернуть в случае, когда значение `Option` равно `None`. Как и для всех комбинаторов, которые мы рассматривали, его объявление очень простое:

```
fn ok_or<T, E>(option: Option<T>, err: E) -> Result<T, E> {
    match option {
        Some(val) => Ok(val),
        None => Err(err),
    }
}
```

Второй новый комбинатор, который мы использовали — `Result::map_err`. Это то же самое, что и `Result::map`, за исключением того, функция применяется к *ошибке* внутри `Result`. Если значение `Result` равно `Ok(...)`, то оно возвращается без изменений.

Мы используем `map_err`, потому что нам необходимо привести все ошибки к одинаковому типу (из-за нашего использования `and_then`). Поскольку мы решили преобразовывать `Option<String>` (из `argv.nth(1)`) в `Result<String, String>`, мы также обязаны преобразовывать `ParseIntError` из `arg.parse()` в `String`.

Ограничения комбинаторов

Работа с IO и анализ входных данных — очень типичные задачи, и это то, чем лично я много занимаюсь с Rust. Так что мы будем использовать IO и различные процедуры анализа как примеры обработки ошибок.

Давайте начнем с простого. Поставим задачу открыть файл, прочесть все его содержимое и преобразовать это содержимое в число. После этого нужно будет умножить значение на `2` и распечатать результат.

Хоть я и пытался убедить вас не использовать `unwrap`, иногда бывает полезным для начала написать код с `unwrap`. Это позволяет сосредоточиться на проблеме, а не на обработке ошибок, и это выявляет места, где надлежащая обработка ошибок необходима. Давайте начнем с того, что напишем просто работающий код, а затем отрефакторим его для лучшей обработки ошибок.

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> i32 {
    let mut file = File::open(file_path).unwrap(); // ошибка 1
    let mut contents = String::new();
    file.read_to_string(&mut contents).unwrap();   // ошибка 2
    let n: i32 = contents.trim().parse().unwrap(); // ошибка 3
    2 * n
}

fn main() {
    let doubled = file_double("foobar");
    println!("{}", doubled);
}
```

(Замечание: Мы используем `AsRef` по тем же причинам, почему он используется в `std::fs::File::open`. Это позволяет удобно использовать любой тип строки в качестве пути к файлу.)

У нас есть три потенциальные ошибки, которые могут возникнуть:

1. Проблема при открытии файла.
2. Проблема при чтении данных из файла.
3. Проблема при преобразовании данных в число.

Первые две проблемы определяются типом `std::io::Error`. Мы знаем это из типа возвращаемого значения методов `std::fs::File::open` и `std::io::Read::read_to_string`. (Обратите внимание, что они оба используют концепцию с псевдонимом типа `Result`, описанную ранее. Если вы кликните на тип `Result`, вы увидите псевдоним типа, и следовательно, лежащий в основе тип `io::Error`.) Третья проблема определяется типом `std::num::ParseIntError`. Кстати, тип `io::Error` *часто* используется по всей стандартной библиотеке. Вы будете видеть его снова и снова.

Давайте начнем рефакторинг функции `file_double`. Для того, чтобы эту функцию можно было сочетать с остальным кодом, она *не должна* паниковать, если какие-либо из перечисленных выше ошибок действительно произойдут. Фактически, это означает, что функция должна *возвращать ошибку*, если любая из возможных операций завершилась неудачей. Проблема состоит в том, что тип возвращаемого значения сейчас `i32`, который не дает нам никакого разумного способа сообщить об ошибке. Таким образом, мы должны начать с изменения типа возвращаемого значения с `i32` на что-то другое.

Первое, что мы должны решить: какой из типов использовать: `Option` или `Result`? Мы, конечно, могли бы с легкостью использовать `Option`. Если какая-либо из трех ошибок происходит, мы могли бы просто вернуть `None`. Это будет работать, и *это лучше, чем просто паниковать*, но мы можем сделать гораздо лучше. Вместо этого, мы будем сообщать некоторые детали о возникшей проблеме. Поскольку мы хотим выразить *возможность ошибки*, мы должны использовать `Result<i32, E>`. Но каким должен быть тип `E`? Поскольку может возникнуть два *разных* типа ошибок, мы должны преобразовать их к общему типу. Одним из таких типов является `String`. Давайте посмотрим, как это отразится на нашем коде:

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    File::open(file_path)
         .map_err(|err| err.to_string())
         .and_then(|mut file| {
              let mut contents = String::new();
              file.read_to_string(&mut contents)
                  .map_err(|err| err.to_string())
                  .map(|_| contents)
         })
         .and_then(|contents| {
              contents.trim().parse::<i32>()
                      .map_err(|err| err.to_string())
         })
         .map(|n| 2 * n)
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Ошибка: {}", err),
    }
}
```

Выглядит немного запутанно. Может потребоваться довольно много практики, прежде вы сможете писать такое. Написание кода в таком стиле называется *следованием за типом*. Когда мы изменили тип возвращаемого значения `file_double` на `Result<i32, String>`, нам пришлось начать подбирать правильные комбинатороы. В данном случае мы использовали только три различных комбинатора: `and_then`, `map` и `map_err`.

Комбинатор `and_then` используется для объединения по цепочке нескольких вычислений, где каждое вычисление может вернуть ошибку. После открытия файла есть еще два вычисления, которые могут завершиться неудачей: чтение из файла и преобразование содержимого в число. Соответственно, имеем два вызова `and_then`.

Комбинатор `map` используется, чтобы применить функцию к значению `Ok(...)` типа `Result`. Например, в самом последнем вызове, `map` умножает значение `Ok(...)` (типа `i32`) на `2`. Если ошибка произошла до этого момента, эта операция была бы пропущена. Это

следует из определения `map`.

Комбинатор `map_err` — это уловка, которая позволяют всему этому заработать. Этот комбинатор, такой же, как и `map`, за исключением того, что применяет функцию к `Err(...)` значению `Result`. В данном случае мы хотим привести все наши ошибки к одному типу — `String`. Поскольку как `io::Error`, так и `num::ParseIntError` реализуют `ToString`, мы можем вызвать метод `to_string`, чтобы выполнить преобразование.

Не смотря на все сказанное, код по-прежнему выглядит запутанным. Мастерство использования комбинаторов является важным, но у них есть свои недостатки. Давайте попробуем другой подход: преждевременный возврат.

Преждевременный `return`

Давайте возьмем код из предыдущего раздела и перепишем его с применением *раннего возврата*. Ранний `return` позволяет выйти из функции досрочно. Мы не можем выполнить `return` для `file_double` внутри замыкания, поэтому нам необходимо вернуться к явному вариативному анализу.

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    let mut file = match File::open(file_path) {
        Ok(file) => file,
        Err(err) => return Err(err.to_string()),
    };
    let mut contents = String::new();
    if let Err(err) = file.read_to_string(&mut contents) {
        return Err(err.to_string());
    }
    let n: i32 = match contents.trim().parse() {
        Ok(n) => n,
        Err(err) => return Err(err.to_string()),
    };
    Ok(2 * n)
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Ошибка: {}", err),
    }
}
```

Кто-то может обосновано не согласиться с тем, что этот код лучше, чем тот, который использует комбинаторы, но если вы не знакомы с комбинаторами, на мой взгляд, этот код будет выглядеть проще. Он выполняет явный вариативный анализ с помощью `match` и `if let`. Если происходит ошибка, мы просто прекращаем выполнение функции и возвращаем ошибку (после преобразования в строку).

Разве это не шаг назад? Ранее мы говорили, что ключ к удобной обработке ошибок — сокращение явного вариативного анализа, но здесь мы вернулись к тому, с чего начинали. Оказывается, существует *несколько* способов его уменьшения. И комбинаторы — не единственный путь.

Макрос `try!`

Краеугольный камень обработки ошибок в Rust — это макрос `try!`. Этот макрос абстрагирует анализ вариантов так же, как и комбинаторы, но в отличие от них, он также абстрагирует *поток выполнения*. А именно, он умеет абстрагировать идею *досрочного возврата*, которую мы только что реализовали.

Вот упрощенное определение макроса `try!:

```
macro_rules! try {
    ($e:expr) => (match $e {
        Ok(val) => val,
        Err(err) => return Err(err),
    });
}
```

(Реальное определение выглядит немного сложнее. Мы обсудим это далее).

Использование макроса `try!` может очень легко упростить наш последний пример. Поскольку он выполняет анализ вариантов и досрочной возврат из функции, мы получаем более плотный код, который легче читать:

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    let mut file = try!(File::open(file_path).map_err(|e| e.to_string()));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents).map_err(|e| e.to_string()));
    let n = try!(contents.trim().parse::<i32>().map_err(|e| e.to_string()));
    Ok(2 * n)
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Ошибка: {}", err),
    }
}
```

Вызов `map_err` по-прежнему необходим, учитывая наше определение `try!`, поскольку ошибки все еще должны быть преобразованы в `String`. Хорошей новостью является то, что в ближайшее время мы узнаем, как убрать все эти вызовы `map_err`! Плохая новость состоит в том, что для этого нам придется кое-что узнать о паре важных типажей из стандартной библиотеки.

Объявление собственного типа ошибки

Прежде чем мы погрузимся в аспекты некоторых типажей из стандартной библиотеки, связанных с ошибками, я бы хотел завершить этот раздел отказом от использования `String` как типа ошибки в наших примерах.

Использование `String` в том стиле, в котором мы использовали его в предыдущих примерах удобно потому, что достаточно легко конвертировать любые ошибки в строки, или даже создавать свои собственные ошибки на ходу. Тем не менее, использование типа `String` для ошибок имеет некоторые недостатки.

Первый недостаток в том, что сообщения об ошибках, как правило, загромождают код. Можно определять сообщения об ошибках в другом месте, но это поможет только если вы необыкновенно дисциплинированны, поскольку очень заманчиво вставлять сообщения об ошибках прямо в код. На самом деле, мы именно этим и занимались в предыдущем примере.

Второй и более важный недостаток заключается в том, что использование `String` чревато *потерей информации*. Другими словами, если все ошибки будут преобразованы в строки, то когда мы будем возвращать их вызывающей стороне, они не будут иметь никакого смысла. Единственное разумное, что вызывающая сторона может сделать с ошибкой типа `String` — это показать ее пользователю. Безусловно, можно проверить строку по значению, чтобы определить тип ошибки, но такой подход не может похвастаться надежностью. (Правда, в гораздо большей степени это недостаток для библиотек, чем для конечных приложений).

Например, тип `io::Error` включает в себя тип `io::ErrorKind`, который является *структурированными данными*, представляющими то, что пошло не так во время выполнения операции ввода-вывода. Это важно, поскольку может возникнуть необходимость по-разному реагировать на различные причины ошибки. (Например, ошибка `BrokenPipe` может изящно завершать программу, в то время как ошибка `NotFound` будет завершать программу с кодом ошибки и показывать соответствующее сообщение пользователю.) Благодаря `io::ErrorKind`, вызывающая сторона может исследовать тип ошибки с помощью вариативного анализа, и это значительно лучше попытки вычленить детали об ошибке из `String`.

Вместо того, чтобы использовать `String` как тип ошибки в нашем предыдущем примере про чтение числа из файла, мы можем определить свой собственный тип, который представляет ошибку в виде *структурированных данных*. Мы постараемся не потерять никакую информацию от изначальных ошибок на тот случай, если вызывающая сторона захочет исследовать детали.

Идеальным способом представления *одного варианта из многих* является определение нашего собственного типа-суммы с помощью `enum`. В нашем случае, ошибка представляет собой либо `io::Error`, либо `num::ParseIntError`, из чего естественным образом вытекает определение:

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

// Мы реализуем `Debug` поскольку, по всей видимости, все типы должны реализовывать `Debug`
.
// Это дает нам возможность получить адекватное и читаемое описание значения CliError
#[derive(Debug)]
enum CliError {
    Io(io::Error),
    Parse(num::ParseIntError),
}
```

Осталось только немного подогнать наш код из примера. Вместо преобразования ошибок в строки, мы будем просто конвертировать их в наш тип `CliError`, используя соответствующий конструктор значения:

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, CliError> {
    let mut file = try!(File::open(file_path).map_err(CliError::Io));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents).map_err(CliError::Io));
    let n: i32 = try!(contents.trim().parse().map_err(CliError::Parse));
    Ok(2 * n)
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Ошибка: {:?}", err),
    }
}
```

Единственное изменение здесь — замена вызова `map_err(|e| e.to_string())` (который преобразовывал ошибки в строки) на `map_err(CliError::Io)` или `map_err(CliError::Parse)`. Теперь *вызывающая сторона* определяет уровень детализации сообщения об ошибке для конечного пользователя. В действительности, использование `String` как типа ошибки лишает вызывающего возможности выбора, в то время использование собственного типа `enum`, на подобие `CliError`, дает вызывающему тот же уровень удобства, который был ранее, и кроме этого *структурированные данные*, описывающие ошибку.

Практическое правило заключается в том, что необходимо определять свой собственный тип ошибки, а тип `String` для ошибок использовать в крайнем случае, в основном когда вы пишете конечное приложение. Если вы пишете библиотеку, определение своего собственного типа ошибки наиболее предпочтительно. Таким образом, вы не лишите пользователя вашей библиотеки возможности выбирать наиболее предпочтительное для его конкретного случая поведение.

Типажи из стандартной библиотеки, используемые для обработки ошибок

Стандартная библиотека определяет два встроенных типажа, полезных для обработки ошибок `std::error::Error` и `std::convert::From`. И если `Error` разработан специально для создания общего описания ошибки, то типаж `From` играет широкую роль в преобразовании значений между различными типами.

Типаж `Error`

Типаж `Error` объявлен в стандартной библиотеке:

```
use std::fmt::{Debug, Display};

trait Error: Debug + Display {
  /// A short description of the error.
  fn description(&self) -> &str;

  /// The lower level cause of this error, if any.
  fn cause(&self) -> Option<&Error> { None }
}
```

Этот типаж очень обобщенный, поскольку предполагается, что он должен быть реализован для *всех* типов, которые представляют собой ошибки. Как мы увидим дальше, он нам очень пригодится для написания сочетаемого кода. Этот типаж, как минимум, позволяет выполнять следующие вещи:

- Получать строковое представление ошибки для разработчика (`Debug`).
- Получать понятное для пользователя представление ошибки (`Display`).
- Получать краткое описание ошибки (метод `description`).
- Изучать по цепочке первопричину ошибки, если она существует (метод `cause`).

Первые две возможности возникают в результате того, что типаж `Error` требует в свою очередь реализации типажей `Debug` и `Display`. Последние два факта исходят из двух методов, определенных в самом `Error`. Мощь `Error` заключается в том, что все существующие типы ошибок его реализуют, что в свою очередь означает что любые ошибки могут быть сохранены как типажи-объекты (trait object). Обычно это выглядит как `Box<Error>`, либо `&Error`. Например, метод `cause` возвращает `&Error`, который как раз является типажом-объектом. Позже мы вернемся к применению `Error` как типажа-объекта.

В настоящее время достаточно показать пример, реализующий типаж `Error`. Давайте воспользуемся для этого типом ошибки, который мы определили в предыдущем разделе:

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

// Мы реализуем `Debug` поскольку, по всей видимости, все типы должны реализовывать `Debug`
.
// Это дает нам возможность получить адекватное и читаемое описание значения CliError
#[derive(Debug)]
enum CliError {
    Io(io::Error),
    Parse(num::ParseIntError),
}
```

Данный тип ошибки отражает возможность возникновения двух других типов ошибок: ошибка работы с IO или ошибка преобразования строки в число. Определение ошибки может отражать столько других видов ошибок, сколько необходимо, за счет добавления новых вариантов в объявлении `enum`.

Реализация `Error` довольно прямолинейна и главным образом состоит из явного анализа вариантов:

```
use std::error;
use std::fmt;

impl fmt::Display for CliError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            // Оба изначальных типа ошибок уже реализуют `Display`,
            // так что мы можем использовать их реализации
            CliError::Io(ref err) => write!(f, "IO error: {}", err),
            CliError::Parse(ref err) => write!(f, "Parse error: {}", err),
        }
    }
}

impl error::Error for CliError {
    fn description(&self) -> &str {
        // Оба изначальных типа ошибок уже реализуют `Error`,
        // так что мы можем использовать их реализацией
        match *self {
            CliError::Io(ref err) => err.description(),
            CliError::Parse(ref err) => err.description(),
        }
    }

    fn cause(&self) -> Option<&error::Error> {
        match *self {
            // В обоих случаях просходит неявное преобразование значения `err`
            // из конкретного типа (`&io::Error` или `&num::ParseIntError`)
            // в типаж-обьект `&Error`. Это работает потому что оба типа реализуют `Error`.
            CliError::Io(ref err) => Some(err),
            CliError::Parse(ref err) => Some(err),
        }
    }
}
```

Хочется отметить, что это очень типичная реализация `Error`: реализация методов `description` и `cause` в соответствии с каждым возможным видом ошибки.

Типаж `From`

Типаж `std::convert::From` объявлен в стандартной библиотеке:

```
trait From<T> {
    fn from(T) -> Self;
}
```

Очень просто, не правда ли? Типаж `From` чрезвычайно полезен, поскольку создает общий подход для преобразования *из* определенного типа `T` в какой-то другой тип (в данном случае, "другим типом" является тип, реализующий данный типаж, или `Self`). Самое важное в типаже `From` — множество его реализаций, предоставляемых стандартной библиотекой.

Вот несколько простых примеров, демонстрирующих работу `From`:

```
let string: String = From::from("foo");
let bytes: Vec<u8> = From::from("foo");
let cow: ::std::borrow::Cow<str> = From::from("foo");
```

Итак, `From` полезен для выполнения преобразований между строками. Но как насчет ошибок? Оказывается, существует одна важная реализация:

```
impl<'a, E: Error + 'a> From<E> for Box<Error + 'a>
```

Эта реализация говорит, что *любой* тип, который реализует `Error`, можно конвертировать в типаж-объект `Box<Error>`. Выглядит не слишком впечатляюще, но это очень полезно в общем контексте.

Помните те две ошибки, с которыми мы имели дело ранее, а именно, `io::Error` and `num::ParseIntError`? Поскольку обе они реализуют `Error`, они также работают с `From`:

```
use std::error::Error;
use std::fs;
use std::io;
use std::num;

// Получаем значения ошибок
let io_err: io::Error = io::Error::last_os_error();
let parse_err: num::ParseIntError = "not a number".parse::<i32>().unwrap_err();

// Собственно, конвертация
let err1: Box<Error> = From::from(io_err);
let err2: Box<Error> = From::from(parse_err);
```

Здесь нужно разобрать очень важный паттерн. Переменные `err1` и `err2` имеют *одинаковый тип* — типаж-объект. Это означает, что их реальные типы скрыты от компилятора, так что по факту он рассматривает `err1` и `err2` как одинаковые сущности. Кроме того, мы

создали `err1` и `err2`, используя один и тот же вызов функции — `From::from`. Мы можем так делать, поскольку функция `From::from` перегружена по ее аргументу и возвращаемому типу.

Эта возможность очень важна для нас, поскольку она решает нашу предыдущую проблему, позволяя эффективно конвертировать разные ошибки в один и тот же тип, пользуясь только одной функцией.

Настало время вернуться к нашему старому другу — макросу `try!`.

Настоящий макрос `try!`

До этого мы привели такое определение `try!`:

```
macro_rules! try {
    ($e:expr) => (match $e {
        Ok(val) => val,
        Err(err) => return Err(err),
    });
}
```

Но это не настоящее определение. Реальное определение можно найти в стандартной библиотеке:

```
macro_rules! try {
    ($e:expr) => (match $e {
        Ok(val) => val,
        Err(err) => return Err(::std::convert::From::from(err)),
    });
}
```

Здесь есть одно маленькое, но очень важное изменение: значение ошибки пропускается через вызов `From::from`. Это делает макрос `try!` очень мощным инструментом, поскольку он дает нам возможность бесплатно выполнять автоматическое преобразование типов.

Вооружившись более мощным макросом `try!`, давайте взглянем на код, написанный нами ранее, который читает файл и конвертирует его содержимое в число:

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    let mut file = try!(File::open(file_path).map_err(|e| e.to_string()));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents).map_err(|e| e.to_string()));
    let n = try!(contents.trim().parse::<i32>().map_err(|e| e.to_string()));
    Ok(2 * n)
}
```

Ранее мы говорили, что мы можем избавиться от вызовов `map_err`. На самом деле, все что мы должны для этого сделать — это найти тип, который работает с `From`. Как мы увидели в предыдущем разделе, `From` имеет реализацию, которая позволяет преобразовать любой тип

ошибки в `Box<Error>`:

```
use std::error::Error;
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, Box<Error>> {
    let mut file = try!(File::open(file_path));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents));
    let n = try!(contents.trim().parse::<i32>());
    Ok(2 * n)
}
```

Мы уже очень близки к идеальной обработке ошибок. Наш код имеет очень мало накладных расходов из-за обработки ошибок, ведь макрос `try!` инкапсулирует сразу три вещи:

1. Вариативный анализ.
2. Поток выполнения.
3. Преобразование типов ошибок.

Когда все эти три вещи объединены вместе, мы получаем код, который не обременен комбинаторами, вызовами `unwrap` или постоянным анализом вариантов.

Но осталась одна маленькая деталь: тип `Box<Error>` *не несет никакой информации*. Если мы возвращаем `Box<Error>` вызывающей стороне, нет никакой возможности (легко) узнать базовый тип ошибки. Ситуация, конечно, лучше, чем со `String`, посольку появилась возможность вызывать методы, вроде `description` или `cause`, но ограничение остается: `Box<Error>` не предоставляет никакой информации о сути ошибки. (Замечание: Это не совсем верно, поскольку в Rust есть инструменты рефлексии во время выполнения, которые полезны при некоторых сценариях, но их рассмотрение выходит за рамки этой главы).

Настало время вернуться к нашему собственному типу `CliError` и связать все в одно целое.

Совмещение собственных типов ошибок

В последнем разделе мы рассмотрели реальный макрос `try!` и то, как он выполняет автоматическое преобразование значений ошибок с помощью вызова `From::from`. В нашем случае мы конвертировали ошибки в `Box<Error>`, который работает, но его значение скрыто для вызывающей стороны.

Чтобы исправить это, мы используем средство, с которым мы уже знакомы: создание собственного типа ошибки. Давайте вспомним код, который считывает содержимое файла и преобразует его в целое число:

```
use std::fs::File;
use std::io::{self, Read};
use std::num;
use std::path::Path;

// Мы реализуем `Debug` поскольку, по всей видимости, все типы должны реализовывать `Debug`
.
// Это дает нам возможность получить адекватное и читаемое описание значения CliError
#[derive(Debug)]
enum CliError {
    Io(io::Error),
    Parse(num::ParseIntError),
}

fn file_double_verbose<P: AsRef<Path>>(file_path: P) -> Result<i32, CliError> {
    let mut file = try!(File::open(file_path).map_err(CliError::Io));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents).map_err(CliError::Io));
    let n: i32 = try!(contents.trim().parse().map_err(CliError::Parse));
    Ok(2 * n)
}
```

Обратите внимание, что здесь у нас еще остались вызовы `map_err`. Почему? Вспомните определения `try!` и `From`. Проблема в том, что не существует такой реализации `From`, которая позволяет конвертировать типы ошибок `io::Error` и `num::ParseIntError` в наш собственный тип `CliError`. Но мы можем легко это исправить! Поскольку мы определили тип `CliError`, мы можем также реализовать для него типаж `From`:

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

impl From<io::Error> for CliError {
    fn from(err: io::Error) -> CliError {
        CliError::Io(err)
    }
}

impl From<num::ParseIntError> for CliError {
    fn from(err: num::ParseIntError) -> CliError {
        CliError::Parse(err)
    }
}
```

Все эти реализации позволяют `From` создавать значения `CliError` из других типов ошибок. В нашем случае такое создание состоит из простого вызова конструктора значения. *Как правило*, это все что нужно.

Наконец, мы можем переписать `file_double`:

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, CliError> {
    let mut file = try!(File::open(file_path));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents));
    let n: i32 = try!(contents.trim().parse());
    Ok(2 * n)
}
```

Единственное, что мы сделали — это удалили вызовы `map_err`. Они нам больше не нужны, поскольку макрос `try!` выполняет `From::from` над значениями ошибок. И это работает, поскольку мы предоставили реализации `From` для всех типов ошибок, которые могут возникнуть.

Если бы мы изменили нашу функцию `file_double` таким образом, чтобы она начала выполнять какие-то другие операции, например, преобразовать строку в число с плавающей точкой, то мы должны были бы добавить новый вариант к нашему типу ошибок:

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

enum CliError {
    Io(io::Error),
    ParseInt(num::ParseIntError),
    ParseFloat(num::ParseFloatError),
}
```

И добавить новую реализацию для `From`:

```
use std::num;

impl From<num::ParseFloatError> for CliError {
    fn from(err: num::ParseFloatError) -> CliError {
        CliError::ParseFloat(err)
    }
}
```

Вот и все!

Рекомендации для авторов библиотек

Если в вашей библиотеке могут возникать специфические ошибки, то вы наверняка должны определить для них свой собственный тип. На ваше усмотрение вы можете сделать его внутреннее представление публичным (как `ErrorKind`), или оставить его скрытым (подобно `ParseIntError`). Независимо от того, что вы предпримете, считается хорошим тоном обеспечить по крайней мере некоторую информацию об ошибке помимо ее строкового представления. Но, конечно, все зависит от конкретных случаев использования.

Как минимум, вы скорее всего должны реализовать типаж `Error`. Это даст пользователям вашей библиотеки некоторую минимальную гибкость при совмещении ошибок. Реализация типажа `Error` также означает, что пользователям гарантируется возможность получения строкового представления ошибки (это следует из необходимости реализации `fmt::Debug` и `fmt::Display`).

Кроме того, может быть полезным реализовать `From` для ваших типов ошибок. Это позволит вам (как автору библиотеки) и вашим пользователям совмещать более детальные ошибки. Например, `csv::Error` реализует `From` для `io::Error` и `byteorder::Error`.

Наконец, на свое усмотрение, вы также можете определить псевдоним типа `Result`, особенно, если в вашей библиотеке определен только один тип ошибки. Такой подход используется в стандартной библиотеке для `io::Result` и `fmt::Result`.

Заключение

Поскольку это довольно длинная глава, не будет лишним составить короткий конспект по обработке ошибок в Rust. Ниже будут приведены некоторые практические рекомендации. Это совсем *не* заповеди. Наверняка существуют веские причины для того, чтобы нарушить любое из этих правил.

- Если вы пишете короткий пример кода, который может быть перегружен обработкой ошибок, это, вероятно, отличная возможность использовать `unwrap` (будь-то `Result::unwrap`, `Option::unwrap` или `Option::expect`). Те, для кого предназначен пример, должны осознавать, что необходимо реализовать надлежащую обработку ошибок. (Если нет, отправляйте их сюда!)
- Если вы пишете одноразовую программу, также не зазорно использовать `unwrap`. Но будьте внимательны: если ваш код попадет в чужие руки, не удивляйтесь, если кто-то будет расстроен из-за скудных сообщений об ошибках!
- Если вы пишете одноразовый код, но вам все-равно стыдно из-за использования `unwrap`, воспользуйтесь либо `String` в качестве типа ошибки, либо `Box<Error + Send + Sync>` (из-за доступных реализаций `From`.)
- В остальных случаях, определяйте свои собственные типы ошибок с соответствующими реализациями `From` и `Error`, делая использование `try!` более удобным.
- Если вы пишете библиотеку и ваш код может выдавать ошибки, определите ваш собственный тип ошибки и реализуйте типаж `std::error::Error`. Там, где это уместно, реализуйте `From`, чтобы вам и вашим пользователям было легче с ними работать. (Из-за правил когерентности в Rust, пользователи вашей библиотеки не смогут реализовать `From` для ваших ошибок, поэтому это должна сделать ваша библиотека.)
- Изучите комбинаторы, определенные для `Option` и `Result`. Писать код, пользуясь

только ними может быть немного утомительно, но я лично нашел для себя хороший баланс между использованием `try!` и комбинаторами (`and_then`, `map` и `unwrap_or` — мои любимые).

Выбор гарантий

Одна из важных черт языка Rust — это то, что он позволяет нам управлять накладными расходами и гарантиями программы.

В стандартной библиотеке Rust есть различные «обёрточные типы», которые реализуют множество компромиссов между накладными расходами, эргономикой, и гарантиями. Многие позволяют выбирать между проверками во время компиляции и проверками во время исполнения. Эта глава подробно объяснит несколько избранных абстракций.

Перед тем, как продолжить, крайне рекомендуем познакомиться с владением и заимствованием в Rust.

Основные типы указателей

`Box<T>`

`Box<T>` — «владеющий» указатель, или, по-другому, «упаковка». Хотя он и может выдавать ссылки на содержащиеся в нём данные, он — единственный владелец этих данных. В частности, когда происходит что-то вроде этого:

```
let x = Box::new(1);
let y = x;
// x больше не доступен
```

Здесь упаковка была *перемещена* в `y`. Поскольку `x` больше не владеет ею, с этого момента компилятор не позволит использовать `x`. Упаковка также может быть перемещена *из* функции — для этого функция возвращает её как свой результат.

Когда упаковка, которая не была перемещена, выходит из области видимости, выполняются деструкторы. Эти деструкторы освобождают содержащиеся данные.

Мы абстрагируемся от динамического выделения памяти, и это абстракция без накладных расходов. Это идеальный способ выделить память в куче и безопасно передавать указатель на эту память. Заметьте, что вы можете создавать ссылки на упаковку по обычным правилам заимствования, которые проверяются во время компиляции.

`&T` и `&mut T`

Это неизменяемые и изменяемые ссылки, соответственно. Они реализуют шаблон «read-write lock», т.е. вы можете создать или одну изменяемую ссылку на данные, или любое число неизменяемых, но не оба вида ссылок одновременно. Эта гарантия проверяется во время компиляции, и ничего не стоит во время исполнения. В большинстве случаев эти два типа указателей покрывают все нужды по передаче дешёвых ссылок между частями кода.

При копировании эти указатели сохраняют связанное с ними время жизни — они всё равно не могут прожить дольше, чем исходное значение, на которое они ссылаются.

`*const T` и `*mut T`

Это сырые указатели в стиле C, не имеющие связанной информации о времени жизни и владельце. Они просто указывают на какое-то место в памяти, без дополнительных ограничений. Они гарантируют только то, что они могут быть разыменованы только в коде, помеченном как «небезопасный».

Они полезны при создании безопасных низкоуровневых абстракций вроде `Vec<T>`, но их следует избегать в безопасном коде.

`Rc<T>`

Это первая рассматриваемая обёртка, использование которой влечёт за собой накладные расходы во время исполнения.

`Rc<T>` — это указатель со счётчиком ссылок. Другими словами, он позволяет создавать несколько «владеющих» указателей на одни и те же данные, и эти данные будут уничтожены, когда все указатели выйдут из области видимости.

Собственно, внутри у него счётчик ссылок (reference count, или сокращённо refcount), который увеличивается каждый раз, когда происходит клонирование `Rc`, и уменьшается когда `Rc` выходит из области видимости. Основная ответственность `Rc<T>` — удостовериться в том, что для разделяемых данных вызываются деструкторы.

Хранимые данные при этом неизменяемы, и если создаётся цикл ссылок, данные утекут. Если нам нужно отсутствие утечек в присутствие циклов, нужно использовать сборщик мусора.

Гарантии

Здесь главная гарантия в том, что данные не будут уничтожены, пока все ссылки на них не исчезнут.

Счётчик ссылок нужно использовать, когда мы хотим динамически выделить какие-то данные и предоставить ссылки на эти данные только для чтения, и при этом неясно, какая часть программы последней закончит использование ссылки. Это подходящая альтернатива `&T`, когда невозможно статически доказать правильность `&T`, или когда это создаёт слишком большие неудобства в написании кода, на который разработчик не хочет тратить своё время.

Этот указатель *не* является потокобезопасным, и Rust не позволяет передавать его или делиться им с другими потоками. Это позволяет избежать накладных расходов от использования атомарных операций там, где они не нужны.

Есть похожий умный указатель, `Weak<T>`. Это невладеющий, но и не заимствуемый, умный указатель. Он тоже похож на `&T`, но не ограничен временем жизни — `Weak<T>` можно не отпускать. Однако, возможна ситуация, когда попытка доступа к хранимым в нём данным провалится и вернёт `None`, поскольку `Weak<T>` может пережить владеющие `Rc`. Его удобно использовать в случае циклических структур данных и некоторых других.

Накладные расходы

Что касается памяти, `Rc<T>` — это одно выделение, однако оно будет включать два лишних слова (т.е. два значения типа `usize`) по сравнению с обычным `Box<T>`. Это верно и для «сильных», и для «слабых» счётчиков ссылок.

Расходы на `Rc<T>` заключаются в увеличении и уменьшении счётчика ссылок каждый раз, когда `Rc<T>` клонируется или выходит из области видимости, соответственно. Отметим, что клонирование не выполняет глубокое копирование, а просто увеличивает счётчик и возвращает копию `Rc<T>`.

Типы-ячейки (cell types)

Типы `Cell` предоставляют «внутреннюю» изменяемость. Другими словами, они содержат данные, которые можно изменять даже если тип не может быть получен в изменяемом виде (например, когда он за указателем `&` или за `Rc<T>`).

Документация модуля `cell` довольно хорошо объясняет эти вещи.

Эти типы *обычно* используют в полях структур, но они не ограничены таким использованием.

`Cell<T>`

`Cell<T>` — это тип, который обеспечивает внутреннюю изменяемость без накладных расходов, но только для типов, реализующих типаж `Copy`. Поскольку компилятор знает, что все данные, вложенные в `Cell<T>`, находятся на стеке, их можно просто заменять без страха утечки ресурсов.

Нарушить инварианты с помощью этой обёртки всё равно можно, поэтому будьте осторожны при её использовании. Если поле обёрнуто в `Cell`, это индикатор того, что эти данные изменяемы и поле может не сохранить своё значение с момента чтения до момента его использования.

```
use std::cell::Cell;

let x = Cell::new(1);
let y = &x;
let z = &x;
x.set(2);
y.set(3);
z.set(4);
println!("{}", x.get());
```

Заметьте, что здесь мы смогли изменить значение через различные ссылки без права изменения.

В плане затрат во время исполнения, такой код аналогичен нижеследующему:

```
let mut x = 1;
let y = &mut x;
let z = &mut x;
x = 2;
*y = 3;
*z = 4;
println!("{}", x);
```

но имеет преимущество в том, что он действительно компилируется.

Гарантии

Этот тип ослабляет правило отсутствия совпадающих указателей с правом записи там, где оно не нужно. Однако, он также ослабляет гарантии, которые предоставляет такое ограничение; поэтому если ваши инварианты зависят от данных, хранимых в `Cell`, будьте осторожны.

Это применяется при изменении примитивов и других типов, реализующих `Copy`, когда нет лёгкого способа сделать это в соответствии с статическими правилами `&` и `&mut`.

`Cell` не позволяет получать внутрение ссылки на данные, что позволяет безопасно менять его содержимое.

Накладные расходы

Накладные расходы при использовании `Cell<T>` отсутствуют, однако если вы оборачиваете в него большие структуры, есть смысл вместо этого обернуть отдельные поля, поскольку иначе каждая запись будет производить полное копирование структуры.

`RefCell<T>`

`RefCell<T>` также предоставляет внутреннюю изменяемость, но не ограничен только типами, реализующими `Copy`.

Однако, у этого решения есть накладные расходы. `RefCell<T>` реализует шаблон «read-write lock» во время исполнения, а не во время компиляции, как `&T` / `&mut T`. Он похож на однопоточный мьютекс. У него есть функции `borrow()` и `borrow_mut()`, которые изменяют внутренний счётчик ссылок и возвращают умный указатель, который может быть разыменован без права изменения или с ним, соответственно. Счётчик ссылок восстанавливается, когда умные указатели выходят из области видимости. С этой системой мы можем динамически гарантировать, что во время заимствования с правом изменения никаких других ссылок на значение больше нет. Если программист пытается позаимствовать значение в этот момент, поток запаникует.

```
use std::cell::RefCell;

let x = RefCell::new(vec![1,2,3,4]);
{
    println!("{:?}", *x.borrow())
}

{
    let mut my_ref = x.borrow_mut();
    my_ref.push(1);
}
```

Как и `Cell`, это в основном применяется в ситуациях, когда сложно или невозможно удовлетворить статическую проверку заимствования. В целом мы знаем, что такие изменения не будут происходить вложенным образом, но это стоит дополнительно проверить.

Для больших, сложных программ, есть смысл положить некоторые вещи в `RefCell`, чтобы упростить работу с ними. Например, многие словари в структуре `ctxt` ctxt в компиляторе Rust обёрнуты в этот тип. Они изменяются только однажды — во время создания, но не во время инициализации, или несколько раз в явно отдельных местах. Однако, поскольку эта структура повсеместно используется везде, жонглирование изменяемыми и неизменяемыми указателями было бы очень сложным (или невозможным), и наверняка создало бы мешанину указателей `&`, которую сложно было бы расширять. С другой стороны, `RefCell` предоставляет дешёвый (но не бесплатный) способ обращаться к таким данным. В будущем, если кто-то добавит код, который пытается изменить ячейку, пока она заимствована, это вызывет панику, источник которой можно отследить. И такая паника обычно происходит детерминированно.

Похожим образом, в DOM Servo много изменения данных, большая часть которого происходит внутри типа DOM, но часть выходит за его границы и изменяет произвольные вещи. Использование `RefCell` и `Cell` для ограждения этих изменений позволяет нам избежать необходимости беспокоиться об изменяемости везде, и одновременно обозначает места, где изменение *действительно* происходит.

Заметьте, что стоит избегать использования `RefCell`, если возможно достаточно простое решение с помощью указателей `&`.

Гарантии

`RefCell` ослабляет *статические* ограничения, предотвращающие совпадение изменяемых указателей, и заменяет их на *динамические* ограничения. Сами гарантии при этом не изменяются.

Накладные расходы

`RefCell` не выделяет память, но содержит дополнительный индикатор «состояния заимствования» (размером в одно слово) вместе с данными.

Во время исполнения каждое заимствование вызывает изменение и проверку счётчика ссылок.

Синхронизированные типы

Многие из вышеперечисленных типов не могут быть использованы потокобезопасным образом. В частности, `Rc<T>` и `RefCell<T>`, оба из которых используют не-атомарные счётчики ссылок, не могут быть использованы так. (*Атомарные* счётчики ссылок — это такие, которые могут быть увеличены из нескольких потоков, не вызывая при этом гонку данных.) Благодаря этому они привносят меньше накладных расходов, но нам также потребуются и потокобезопасные варианты этих типов. Они существуют — это `Arc<T>` и `Mutex<T>`/`RWLock<T>`.

Заметьте, что не-потокобезопасные типы *не могут* быть переданы между потоками, и это проверяется во время компиляции.

В модуле sync много полезных обёрточных типов для многопоточного программирования, но мы затронем только главные из них.

`Arc<T>`

`Arc<T>` — это вариант `Rc<T>`, который использует атомарный счётчик ссылок (поэтому «Arc»). Его можно свободно передавать между потоками.

`shared_ptr` из C++ похож на `Arc`, но в случае C++ вложенные данные всегда изменяемы. Чтобы получить семантику, похожую на семантику C++, нужно использовать `Arc<Mutex<T>>`, `Arc<RwLock<T>>`, или `Arc<UnsafeCell<T>>`[1]. (`UnsafeCell<T>` — это тип-ячейка, который может содержать любые данные и не имеет накладных расходов, но доступ к его содержимому производится только внутри небезопасных блоков.) Последний стоит использовать только тогда, когда мы уверены в том, что наша работа не вызывет нарушения безопасности памяти. Учитывайте, что запись в структуру не атомарна, а многие функции вроде `vec.push()` могут выделять память заново в процессе работы, и тем самым вызывать небезопасное поведение.

Гарантии

Как и `Rc`, этот тип гарантирует, что деструктор хранимых в нём данных будет вызван, когда последний `Arc` выходит из области видимости (за исключением случаев с циклами). В отличие от `Rc`, `Arc` предоставляет эту гарантию и в многопоточном окружении.

Накладные расходы

Накладные расходы увеличиваются по сравнению с `Rc`, т.к. теперь для изменения счётчика ссылок используются атомарные операции (которые происходят каждый раз при клонировании или выходе из области видимости). Когда вы хотите поделиться данными в пределах одного потока, предпочтительнее использовать простые ссылки `&`.

`Mutex<T>` and `RwLock<T>`

`Mutex<T>` и `RwLock<T>` предоставляют механизм взаимоисключения с помощью охранных значений RAII. Охранные значения — это объекты, имеющие некоторое состояние, как замок, пока не выполнится их деструктор. В обоих случаях, мьютекс непрозрачен, пока на нём не вызовут `lock()`, после чего поток остановится до момента, когда мьютекс может быть закрыт, после чего возвращается охранное значение. Оно может быть использовано для доступа к вложенным данным с правом изменения, а мьютекс будет снова открыт, когда охранное значение выйдет из области видимости.

```
{
    let guard = mutex.lock();
    // охранное значение разыменовывается в изменяемое значение
    // вложенного в мьютекс типа
    *guard += 1;
} // мьютекс открывается когда выполняется деструктор
```

`RwLock` имеет преимущество — он эффективно работает в случае множественных чтений. Ведь читать из общих данных всегда безопасно, пока в эти данные никто не хочет писать; и `RwLock` позволяет читающим получить «право чтения». Право чтения может быть получено многими потоками одновременно, и за читающими следит счётчик ссылок. Тот же, кто хочет записать данные, должен получить «право записи», а оно может быть получено только когда все читающие вышли из области видимости.

Гарантии

Оба этих типа предоставляют безопасное изменение данных из разных потоков, но не защищают от взаимной блокировки (deadlock). Некоторая дополнительная безопасность протокола работы с данными может быть получена с помощью системы типов.

Накладные расходы

Для поддержания состояния прав чтения и записи эти типы используют в своей реализации конструкции, похожие на атомарные типы, и они довольно дороги. Они могут блокировать все межпроцессорные чтения из памяти, пока не закончат работу. Ожидание

возможности закрытия этих примитивов синхронизации тоже может быть медленным, когда производится много одновременных попыток доступа к данным.

Сочетание

Распространённая жалоба на код на Rust — это сложность чтения типов вроде `Rc<RefCell<Vec<T>>>` (или ещё более сложных сочетаний похожих типов). Не всегда понятно, что делает такая комбинация, или почему автор решил использовать именно такой тип. Не ясно и то, в каких случаях сам программист должен использовать похожие сочетания типов.

Обычно, вам понадобятся такие типы, когда вы хотите сочетать гарантии разных типов, но не хотите переплачивать за то, что вам не нужно.

Например, одно из таких сочетаний — это `Rc<RefCell<T>>`. Сам по себе `Rc<T>` не может быть разыменован с правом изменения; поскольку `Rc<T>` позволяет делиться данными и одновременная попытка изменения данных может привести к небезопасному поведению, мы кладём внутрь `RefCell<T>`, чтобы получить динамическую проверку одновременных попыток изменения. Теперь у нас есть разделяемые изменяемые данные, но одновременный доступ к ним предоставляется только на чтение, а запись всегда исключительна.

Далее мы можем развить эту мысль и получить `Rc<RefCell<Vec<T>>>` или `Rc<Vec<RefCell<T>>>`. Это — изменяемые, разделяемые между потоками вектора, но они не одинаковы.

В первом типе `RefCell<T>` оборачивает `Vec<T>`, поэтому изменяем весь `Vec<T>` целиком. В то же время, это значит, что в каждый момент времени может быть только одна ссылка на `Vec<T>` с правом изменения. Поэтому код не может одновременно работать с разными элементами вектора, обращаясь к ним через разные `Rc`. Однако, мы сможем добавлять и удалять элементы вектора в произвольные моменты времени. Этот тип похож на `&mut Vec<T>`, с тем различием, что проверка заимствования делается во время исполнения.

Во втором типе заимствуются отдельные элементы, а вектор в целом неизменяем. Поэтому мы можем получить ссылки на отдельные элементы, но не можем добавлять или удалять элементы. Это похоже на `&mut [T]`[2], но, опять-таки, проверка заимствования производится во время исполнения.

В многопоточных программах возникает похожая ситуация с `Arc<Mutex<T>>`, который обеспечивает разделяемое владение и одновременное изменение.

Когда вы читаете такой код, рассматривайте гарантии и накладные расходы каждого вложенного типа шаг за шагом.

Когда вы выбираете сложный тип, поступайте наоборот: решите, какие гарантии вам нужны, и в каком «слое» сочетания они понадобятся. Например, если у вас стоит выбор между `Vec<RefCell<T>>` и `RefCell<Vec<T>>`, найдите компромисс путём рассуждений, как мы делали выше по тексту, и выберите нужный вам тип.

1. На самом деле, `Arc<UnsafeCell<T>>` не скомпилируется, поскольку `UnsafeCell<T>` не реализует `Send` или `Sync`, но мы можем обернуть его в тип и реализовать для него `Send`/`Sync` вручную, чтобы получить `Arc<Wrapper<T>>`, где `Wrapper` — это `struct Wrapper<T>(UnsafeCell<T>)`. ↩

2. `&[T]` и `&mut [T]` — это *срезы*; они состоят из указателя и длины, и могут ссылаться на часть вектора или массива. `&mut [T]` также позволяет изменять свои элементы, но его длину изменить нельзя. ↩

Интерфейс внешних функций (foreign function interface)

Введение

В данном руководстве в качестве примера мы будем использовать snappy, библиотеку для сжатия/распаковки данных. Мы реализуем Rust-интерфейс к этой библиотеке через вызов внешних функций. Rust в настоящее время не в состоянии делать вызовы напрямую в 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!("максимальный размер сжатого буфера длиной 100 байт: {}", x);
}
```

Блок `extern` содержит список сигнатур функций из внешней библиотеки, в данном случае для C ABI (application binary interface; двоичный интерфейс приложений) данной платформы. 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 внутри этого блока в действительности безопасно. Библиотеки C часто предоставляют интерфейсы, которые не являются потоко-безопасными. И почти любая функция, которая принимает в качестве аргумента указатель, не может принимать любое входное значений, поскольку указатель может быть висячим; 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 API snappy:

```
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 (application programming interface; интерфейс программирования приложений) необходимо обернуть, чтобы обеспечить безопасность памяти. Тогда мы сможем использовать концепции более высокого уровня, такие как векторы. Библиотека может выборочно открывать только безопасный, высокоуровневый интерфейс и скрывать небезопасные внутренние детали.

Оборачивание 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 ёмкость — общее количество выделенной памяти в элементах. Длина меньше или равна ёмкости.

```
pub fn validate_compressed_buffer(src: &[u8]) -> bool {
    unsafe {
        snappy_validate_compressed_buffer(src.as_ptr(), src.len() as size_t) == 0
    }
}
```

Обёртка `validate_compressed_buffer` использует блок `unsafe`, но это гарантирует, что её вызов будет безопасен для всех входных данных, поскольку модификатор `unsafe` отсутствует в сигнатуре функции. Т.е. небезопасность скрыта внутри функции и не видна вызывающему.

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 ёмкости, требуемой для хранения сжатых выходных данных. Затем этот вектор может быть передан в функцию `snappy_compress` в качестве выходного параметра. Ещё один параметр передается, чтобы получить настоящую длину после сжатия и установить соответствующую длину вектора.

```
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 несжатых 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 владение ресурсами в вызывающий 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 типаж Drop.

Обратные вызовы функций Rust кодом на C (Callbacks from C code to Rust functions)

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 подходящее соглашение о вызове.

Затем функция обратного вызова может быть передана в библиотеку C через регистрационный вызов, и уже затем может быть вызвана оттуда.

Простой пример:

Код на Rust:

```
extern fn callback(a: i32) {
    println!("Меня вызывают из C со значением {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(); // Активация функции обратного вызова
    }
}
```

Код на C:

```
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); // Вызовет callback(7) в 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 передавать указатель на объект Rust при обратном вызове. Это позволит получить небезопасный доступ к объекту Rust, на которой сослались в обратном вызове.

Код на Rust:

```
#[repr(C)]
struct RustObject {
    a: i32,
    // другие поля
}

extern "C" fn callback(target: *mut RustObject, a: i32) {
    println!("Меня вызывают из C со значением {0}", a);
    unsafe {
        // Меняем значение в RustObject на значение, полученное через функцию обратного выз
ова
        (*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() {
    // Создаём объект, на который будем ссылаться в функции обратного вызова
    let mut rust_object = Box::new(RustObject { a: 5 });

    unsafe {
        register_callback(&mut *rust_object, callback);
        trigger_callback();
    }
}
```

Код на C:

```
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); // Вызовет callback(&rustObject, 7) в Rust
}
```

Асинхронные обратные вызовы

В приведённых примерах обратные вызовы выполняются как непосредственная реакция на вызов функции внешней библиотеки на C. Для выполнения обратного вызова поток исполнения 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.

Более сложная ситуация — это когда внешняя библиотека порождает свои собственные потоки и осуществляет обратные вызовы из них. В этих случаях доступ к структурам данных Rust из обратных вызовов особенно опасен, и поэтому нужно использовать соответствующие механизмы синхронизации. Помимо классических механизмов синхронизации, таких как мьютексы, в Rust есть еще одна возможность: использовать каналы (`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, чтобы обратные вызовы не выполнялись библиотекой C после уничтожения этого объекта Rust. Для этого следует, во-первых, проектировать библиотеку таким образом, чтобы отмена регистрации обратного вызова гарантировала, что он больше не будет выполняться. Во-вторых, нужно отменить регистрацию обратного вызова в деструкторе объекта Rust, которому адресован обратный вызов.

Компоновка

Атрибут `link` для блоков `extern` предоставляет `rustc` основные инструкции относительно того, как он должен 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 компонуемся. Во втором случае `bar` — это тип нативной библиотеки, с которой происходит компоновка. В настоящее время `rustc` 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/статическая библиотека) и конечный (динамическая библиотека/исполняемый файл). (Прим. переводчика: rlib — это формат статической библиотеки с метаданными в формате Rust) 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 распространение 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 результата (промежуточный или конечный) контейнера, нативная статическая библиотека будет включена в него на выходе, поэтому нет необходимости в распространении этой нативной статической библиотеки отдельно.

- Обычная динамическая зависимость. Общие системные библиотеки (такие, как `readline`) доступны на большом количестве систем, и статическую копию этих библиотек часто сложно найти. Когда такая зависимость включена в контейнер Rust, промежуточные артефакты (например, rlib'ы) 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 так же, как 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. Небезопасные блоки изолируют опасные ситуации и дают гарантии

компилятору, что опасности не вытекут за пределы блока.

Небезопасные функции же, наоборот, показывают свою опасность всем. 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 отслеживания 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. И Rust при вызове внешних функций по умолчанию использует соглашение о вызове C 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`. Вот список поддерживаемых ограничений для ABI:

- `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 для взаимодействия с нативными библиотеками данной платформы. Например, на платформе win32 с архитектурой x86, это означает, что будет использован ABI `stdcall`. Однако, на windows x86_64 используется соглашение о вызове `C`, 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 размещение полей `struct` совместимо с представлением в C только в том случае, если к ней применяется атрибут `#[repr(C)]`. Атрибут `#[repr(C, packed)]` может быть использован для размещения полей структуры без выравнивания.

Атрибут `#[repr(C)]` также может быть применен и к перечислениям.

Владеющие упаковки в Rust (`Box<T>`) используют указатели, не допускающие нулевое значение (non-nullable), как дескрипторы содержащихся в них объектов. Тем не менее, эти дескрипторы не должны создаваться вручную, так как они управляются внутренними средствами выделения памяти. Ссылки можно без риска считать ненулевыми указателями непосредственно на тип. Однако нарушение правил проверки заимствования или изменяемости может быть небезопасным. Но компилятор не может сделать так много предположений о сырых указателях. Например, он не полагается на настоящую неизменяемость данных под неизменяемым сырым указателем. Поэтому используйте сырые указатели (`*`), если вам необходимо намеренно нарушить правила (но так, что при этом всё работает). Это нужно, чтобы компилятор «случайно» не предположил относительно ссылок чего-то, что мы собираемся нарушать (возможно, нам нужны несколько указателей с правом изменения, что не допускается обычными ссылками).

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. Однако, 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 кода на Rust из кода на C

Вы можете скомпилировать код на Rust таким образом, чтобы он мог быть вызван из кода на C. 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 соглашения о вызове C, как описано выше в разделе «Соглашение о вызове внешних функций». Атрибут `no_mangle` выключает изменение имён, применяемое в Rust, чтобы было легче компоноваться с этим кодом.

Типажи `Borrow` и `AsRef`

Типажи `Borrow` и `AsRef` очень похожи, но в то же время отличаются. Ниже приводится небольшая памятка об этих двух типажах.

Типаж Borrow

Типаж `Borrow` используется, когда вы пишете структуру данных и хотите использовать владение и заимствование типа как синонимы.

Например, `HashMap` имеет метод `get`, который использует `Borrow`:

```
fn get<Q: ?Sized>(&self, k: &Q) -> Option<&V>
    where K: Borrow<Q>,
          Q: Hash + Eq
```

Эта сигнатура является довольно сложной. Параметр `K` — это то, что нас здесь интересует. Он ссылается на параметр самого `HashMap`:

```
struct HashMap<K, V, S = RandomState> {
```

Параметр `K` представляет собой тип *ключа*, который использует `HashMap`. Взглянем на сигнатуру `get()` еще раз. Использовать `get()` возможно, когда ключ реализует `Borrow<Q>`. Таким образом, мы можем сделать `HashMap`, который использует ключи `String`, но использовать `&str`, когда мы выполняем поиск:

```
use std::collections::HashMap;

let mut map = HashMap::new();
map.insert("Foo".to_string(), 42);

assert_eq!(map.get("Foo"), Some(&42));
```

Это возможно, так как стандартная библиотека содержит `impl Borrow<str> for String`.

Для большинства типов, когда вы хотите получить право собственности или позаимствовать значений, достаточно использовать просто `&T`. `Borrow` же становится полезен, когда есть более одного вида занимаемого значения. Это особенно верно для ссылок и срезов: у вас может быть как `&T`, так и `&mut T`. Если мы хотим принимать оба этих типа, `Borrow` как раз для этого подходит:

```
use std::borrow::Borrow;
use std::fmt::Display;

fn foo<T: Borrow<i32> + Display>(a: T) {
    println!("a заимствовано: {}", a);
}

let mut i = 5;

foo(&i);
foo(&mut i);
```

Это выведет `a заимствовано: 5` дважды.

Типаж AsRef

Типаж `AsRef` является преобразующим типажом. Он используется в обобщённом коде для преобразования некоторого значения в ссылку. Например:

```
let s = "Hello".to_string();

fn foo<T: AsRef<str>>(s: T) {
    let slice = s.as_ref();
}
```

Что в каком случае следует использовать?

Мы видим, что они вроде одинаковы: имеют дело с владением и заимствованием значения некоторого типа. Тем не менее, эти типажи немного отличаются.

Используйте `Borrow`, когда вы хотите абстрагироваться от различных видов заимствований, или когда вы строите структуру данных, которая использует владеющие и заимствованные значения как эквивалентные. Например, это может пригодиться в хэшировании и сравнении.

Используйте `AsRef`, когда вы пишете обобщённый код и хотите непосредственно преобразовать что-либо в ссылку.

Каналы сборок

Проект Rust использует концепцию под названием «каналы сборок» для управления сборками. Важно понять этот процесс, чтобы выбрать, какую версию Rust использовать в вашем проекте.

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 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. Это произойдёт 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

А что насчёт 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

Эта часть разбита на небольшие главы, каждая из которых описывает определённое понятие Rust.

Если вы хотите изучить Rust «от и до», продолжайте чтение данной части по порядку - вы на верном пути!

Эти главы также являются справочником понятий, так что если при чтении другого материала вам будет что-то непонятно, вы всегда сможете найти объяснение здесь.

Связывание имён

Любая реальная программа на Rust посложнее, чем «Hello World», использует *связывание имён*. 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 *переменной*. Но у связывания переменных в Rust есть пара трюков в рукаве. В левой части 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 о них написана отдельная глава. Но на данный момент нам не нужны эти возможности, 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 компиляции. Так почему же наш первый пример скомпилировался? В Rust есть нечто, называемое *выводом 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 мы попросили вас прочитать это вслух, вы бы сказали «`x` - это связывание типа `int` со значением `пять`».

В этом случае мы указали, что `x` у нас будет 32-битным целым числом со знаком. В Rust есть и другие целочисленные типы. Их имена начинаются с `i` для целых чисел со знаком и с `u` для целых чисел без знака. Целые числа могут иметь размер 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 идиоматичным 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 незачем делать связывание неизменяемым по умолчанию. Но вспомните, на чём в первую очередь фокусируется Rust: на безопасности. Если вы случайно забыли указать `mut` и изменили связывание, компилятор заметит это, и сообщит вам, что вы попытались изменить не то, что собирались. 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 того, чтобы по возможности избегать изменяемого состояния, но это выходит за рамки данной книги. 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. Связывание переменных в Rust имеет ещё одно отличие от других языков: оно требует инициализации перед использованием.

Давайте приступим к рассмотрению вышесказанного. 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 «Привет, мир!»:

```
   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 не используем связанную переменную, но от того, что мы её не используем, не будет никакого вреда, поэтому это не ошибка. 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`, чтобы 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 способами форматирования строк (англ.). На данный момент мы просто используем способ по умолчанию: печатать целые числа не очень сложно.

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 язык, ориентированный на выражения, и во-вторых, смысл точки с запятой отличается от смысла аналогичного символа в других языках с синтаксисом на основе фигурных скобок и точки с запятой. Эти две особенности связаны.

Выражения и операторы

Rust — в первую очередь язык, ориентированный на выражения. 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 «операторы выражения». Все остальное — выражения. Давайте сначала поговорим об операторах объявления.

Оператор объявления — это 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 результатом присваивания является присваиваемое значение (например, `5` из предыдущего примера), в Rust значением присваивания 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 отделения выражений друг от друга. Rust выглядит как многие другие языки, которые требуют использовать точку с запятой в конце каждой строки. 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;

    // дальнейший код не будет исполнен!
    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 числа используют представление «дополнение до двух» (дополнительный код). 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-последовательности, для представления последовательностей неких вещей. Самый простой из них — это *массив*, то есть последовательность элементов одного и того же типа, имеющая 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 обобщённое программирование. `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 значений. 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 обобщённое программирование.

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`. Пока просто остановимся на этом.

Вы можете найти больше информации о строках (`str`) в документации к стандартной библиотеке (англ.).

Кортежи

Кортеж — это последовательность фиксированного размера. Вроде такой:

```
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;
```

Стоит отметить и ещё один момент, касающийся длины кортежей: кортеж нулевой длины (`()`; пустой кортеж) часто называют «единичным значением». Соответственно, тип такого значения — «единичный тип».

Доступ к полям кортежа можно получить с помощью *деконструирующего let*. Вот пример:

```
let (x, y, z) = (1, 2, 3);

println!("x это {}", x);
```

Помните, мы говорили, что левая часть оператора `let` может больше, чем просто присваивать имена? Мы имели ввиду то, что приведено выше. Мы можем написать слева от `let` шаблон, 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 основном», потому что есть документирующие комментарии и примеры в документации).

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.
///
/// # Examples
///
/// ```
/// 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 аргументов, возвращаемых значений и привести некоторые примеры использования. Заметьте, 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 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, 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 также 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

На данный момент в Rust есть три способа организовать циклическое исполнение кода. Это `loop`, `while` и `for`. У каждого подхода своё применения.

Циклы `loop`

Бесконечный цикл (`loop`) — простейшая форма цикла в Rust. С помощью этого ключевого слова можно организовать цикл, который продолжается, пока не выполнится какой-либо оператор, прерывающий его. Бесконечный цикл в Rust выглядит так:

```
loop {
    println!("Зациклились!");
}
```

Циклы `while`

Цикл `while` — это ещё один вид конструкции цикла в Rust. Выглядит он так:

```
let mut x = 5; // mut x: i32
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, и 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`. Компилятор сможет транслировать ваш код в более эффективный и безопасный машинный код.

Циклы `for`

Цикл `for` нужен для повторения блока кода определённое количество 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, 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 намеренно нет цикла `for` в стиле C. Управлять каждым элементом цикла вручную сложно, и это может приводить к ошибкам даже у опытных программистов на C.

Перечисление

Если вы хотите отслеживать число прошедших итераций, используйте функцию `.enumerate()`.

С интервалами

```
for (i,j) in (5..10).enumerate() {
    println!("i = {} и j = {}", i, j);
}
```

Выводит:

```
i = 0 и j = 5
i = 1 и j = 6
i = 2 и j = 7
i = 3 и j = 8
i = 4 и j = 9
```

Не забудьте написать скобки вокруг интервала.

С итераторами

```
for (linenumber, line) in lines.enumerate() {
    println!("{}: {}", linenumber, line);
}
```

Outputs:

```
0: привет
1: мир
2: hello
3: world
```

Раннее прерывание цикла

Давайте ещё раз посмотрим на цикл `while`:

```
let mut x = 5;
let mut done = false;

while !done {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 {
        done = true;
    }
}
```

В этом примере в условии для выхода из цикла используется изменяемое имя `done` логического типа. В Rust имеются два ключевых слова, которые помогают работать с итерациями цикла: `break` и `continue`.

Мы можем переписать цикл 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. Использование явного `return` также остановит выполнение 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);
}
```

Метки циклов

Когда у вас много вложенных циклов, вы можете захотеть указать, к какому именно циклу относится `break` или `continue`. Как и во многих других языках, по умолчанию эти операторы будут относиться к самому внутреннему циклу. Если вы хотите прервать внешний цикл, вы можете использовать метку. Так, этот код будет печатать на экране только когда и `x`, и `y` нечётны:

```
'outer: for x in 0..10 {
    'inner: for y in 0..10 {
        if x % 2 == 0 { continue 'outer; } // продолжает цикл по x
        if y % 2 == 0 { continue 'inner; } // продолжает цикл по y
        println!("x: {}, y: {}", x, y);
    }
}
```

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 читаете сейчас
- заимствование, и связанная с ним возможность «ссылки»
- время жизни, расширение понятия 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 подробностям, отметим два важных момента в системе владения.

Rust сфокусирован на безопасности и скорости. Это достигается за счёт «абстракций с нулевой стоимостью» (zero-cost abstractions). Это значит, что в Rust стоимость абстракций должна быть настолько малой, 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*. Во время исполнения вы не платите за какую-либо из возможностей ничего.

Тем не менее, эта система всё же имеет определённую стоимость: кривая обучения. Многие новые пользователи Rust «борются с проверкой заимствования» — 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 говорят, что чем больше они работают с правилами системы владения, тем меньше они борются с компилятором.

Имея это в виду, давайте перейдём к изучению системы владения.

Владение

Связанные имена имеют одну особенность в Rust: они «владеют» тем, с чем они связаны. Это означает, что, когда имя выходит за пределы области видимости, ресурс, с которым оно связано, будет 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 одно* связывание какого-либо ресурса. Например, если у нас есть вектор, то мы можем присвоить этот вектор другому имени:

```
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 владение, и попробуем использовать значение после того, как мы передали это значение 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.

Подробности

Причина, по которой мы не можем использовать значение после того, как мы его переместили, неочевидна, но очень важна. 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 по данным. Поэтому Rust запрещает использование `v` после того, как мы выполнили его перемещение.

Важно также отметить, что оптимизация может удалить саму копию байтов на стеке, 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 владение передаётся другому имени, вы больше не можете использовать исходное. 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 значение `v` имени `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 владение обратно из функции, то мы бы написали:

```
fn foo(v: Vec<i32>) -> Vec<i32> {
    // делаем что-либо с v

    // возвращаем владение
    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 владение 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, строка возврата, и вызов функции получается намного более сложным.

К счастью, Rust предлагает такую возможность, как заимствование, которая помогает нам решить эту проблему. Это тема следующего раздела!

Ссылки и заимствование

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, ключевая концепция
- заимствование, её вы читаете сейчас
- время жизни, расширение понятия 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 подробностям, отметим два важных момента в системе владения.

Rust сфокусирован на безопасности и скорости. Это достигается за счёт «абстракций с нулевой стоимостью» (zero-cost abstractions). Это значит, что в Rust стоимость абстракций должна быть настолько малой, 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*. Во время исполнения вы не платите за какую-либо из возможностей ничего.

Тем не менее, эта система всё же имеет определённую стоимость: кривая обучения. Многие новые пользователи Rust «борются с проверкой заимствования» — 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 говорят, что чем больше они работают с правилами системы владения, тем меньше они борются с компилятором.

Имея это в виду, давайте перейдём к изучению системы владения.

Заимствование

В конце главы Владение у нас была убогая 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 идиоматичным 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 владение ресурсом, она его заимствует. Имена, которые заимствуют что- то, не освобождают ресурс, когда они выходят из области видимости. Это означает, что, после вызова `foo()`, мы снова можем использовать наши исходные имена.

Ссылки являются неизменяемыми, как и имена. Это означает, что внутри `foo()` векторы не могут быть изменены:

```
fn foo(v: &Vec<i32>) {
     v.push(5);
}

let v = vec![];

foo(&v);
```

выдаёт 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.

Ссылки &mut

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 должно быть помечено как `mut`. Если бы этого не было, то мы не могли бы получить изменяемую ссылку 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 приведённом 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() {

}
^
```

Оказывается, есть определённые правила создания ссылок.

Правила

Вот правила заимствования в Rust.

Во-первых, область видимости любой ссылки 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 это похоже, хотя и не соответствует точно, определению состояния гонки данных:

> Состояние «гонки данных» возникает, когда два или более указателей осуществляют доступ к одной и той же области памяти одновременно, по крайней мере один из них производит запись, и 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 одну `&mut` одновременно. Так Rust предотвращает возникновение состояния гонки данных во время компиляции: мы получим ошибку компиляции, если нарушим эти правила.

Имея это в виду, давайте рассмотрим наш пример еще раз.

Осмысливаем области 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 изменяемое заимствование заканчивалось *до* того, как мы пытаемся вызвать `println!` и создать неизменяемое заимствование. 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;    // -+ заимствование x через &mut начинается здесь
                   //  |
*y += 1;           //  |
                   //  |
println!("{}", x); // -+ - пытаемся позаимствовать x здесь
                   // -+ заимствование x через &mut заканчивается здесь
```

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; // -+ заимствование через &mut начинается здесь
    *y += 1;        //  |
}                   // -+ ... и заканчивается здесь

println!("{}", x);  // <- пытаемся позаимствовать x здесь
```

Никаких проблем нет. Наша изменяемая ссылка выходит из области видимости до создания неизменяемой. Но область видимости является ключом к определению того, как долго длится заимствование.

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 обходите. Проверка заимствования Rust предотвращает это:

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

for i in &v {
    println!("{}", i);
}
```

Этот код печатает числа от одного до трёх. Когда мы обходим вектор, мы получаем лишь ссылки на элементы. И сам `v` заимствован как неизменяемый, что означает, что мы не можем изменить его в процессе обхода:

```
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 видимости 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 заимствование «не живет достаточно долго» («does not live long enough»), потому что оно не является действительным столько времени, сколько требуется.

Такая же проблема возникает, когда ссылка объявлена *перед* значением, на которое она ссылается. Это происходит потому что ресурсы в одном блоке освобождаются в порядке, противоположном порядку их объявления:

```
let y: &i32;
let x = 5;
y = &x;

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

Мы получим такую 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;

    println!("{}", y);
}

note: ...but borrowed value is only valid for the block suffix following
statement 1 at 3:14
    let x = 5;
    y = &x;

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

В примере выше `y` объявлена перед `x`, т.е. живёт дольше `x`, а это запрещено.

Время жизни

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, ключевая концепция
- заимствование, и связанная с ним возможность «ссылки»
- время жизни, её вы читаете сейчас

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 подробностям, отметим два важных момента в системе владения.

Rust сфокусирован на безопасности и скорости. Это достигается за счёт «абстракций с нулевой стоимостью» (zero-cost abstractions). Это значит, что в Rust стоимость абстракций должна быть настолько малой, 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*. Во время исполнения вы не платите за какую-либо из возможностей ничего.

Тем не менее, эта система всё же имеет определённую стоимость: кривая обучения. Многие пользователи Rust занимаются тем, что мы зовём «борьбой с проверкой заимствования» — 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 говорят, что чем больше они работают с правилами системы владения, тем меньше они борются с компилятором.

Имея это в виду, давайте перейдём к изучению системы владения.

Время жизни

Одалживание ссылки на ресурс, которым кто-то владеет, может быть довольно сложным. Например, представьте себе следующую последовательность операций:

- Мы получаем абстрактную ссылку на какой-то ресурс.
- Мы одалживаем вам ссылку на этот ресурс.
- Мы решаем, что ресурс нам больше не требуется, и освобождаем 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 понятие времени жизни, которое 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:

```
// неявно
fn foo(x: &i32) {
}

// явно
fn bar<'a>(x: &'a i32) {
}
```

Читается `'a` как «время жизни a». Технически, все ссылки имеют 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».

Внутри `struct`'ов

Вы также должны будете явно указать время жизни при работе со `struct`'ми:

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let y = &5; // то же самое, что и `let _y = 5; let y = &_y;`
    let f = Foo { x: y };

    println!("{}", f.x);
}
```

Как вы можете заметить, структуры также могут иметь время жизни. Так же как и функции,

```
struct Foo<'a> {
```

объявляет время жизни и

```
x: &'a i32,
```

использует 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.

Блоки `impl`

Давайте реализуем метод для `Foo`:

```
struct Foo<'a> {
    x: &'a i32,
}

impl<'a> Foo<'a> {
    fn x(&self) -> &'a i32 { self.x }
}

fn main() {
    let y = &5; // то же самое, что и `let _y = 5; let y = &_y;`
    let f = Foo { x: y };

    println!("x is: {}", f.x());
}
```

Как вы можете видеть, нам нужно объявить время жизни для `Foo` в строке с `impl`. Мы повторяем `'a` дважды, как в функциях: `impl<'a>` определяет время жизни `'a`, и `Foo<'a>` использует его.

Несколько времён жизни (Multiple lifetimes)

Если вы имеете несколько ссылок, вы можете использовать одно и то же время жизни несколько раз:

```
fn x_or_y<'a>(x: &'a str, y: &'a str) -> &'a str {
```

Этот код говорит, что `x` и `y` находятся в одной области видимости друг с другом, и что возвращаемое значение живо на протяжении той же области видимости. Если вы хотите, чтобы `x` и `y` имели разные времена жизни, вы должны использовать параметры нескольких времён жизни:

```
fn x_or_y<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
```

В этом примере `x` и `y` имеют различные области видимости, но возвращаемое значение имеет то же время жизни, что и `x`.

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 входит в область видимости
                    //  |
    // что-то       //  |
                    //  |
}                   // -+ y выходит из области видимости
```

Добавим нашу структуру `Foo`:

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let y = &5;           // -+ y входит в область видимости
    let f = Foo { x: y }; // -+ f входит в область видимости
    // что-то             //  |
                          //  |
}                         // -+ f и y выходят из области видимости
```

Наша `f` живет в области видимости `y`, поэтому все работает. Что же произойдёт, если это будет не так? Этот код не будет работать:

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let x;                    // -+ x входит в область видимости
                              //  |
    {                         //  |
        let y = &5;           // ---+ y входит в область видимости
        let f = Foo { x: y }; // ---+ f входит в область видимости
        x = &f.x;             //  | | здесь ошибка
    }                         // ---+ f и y выходят из области видимости
                              //  |
    println!("{}", x);        //  |
}                             // -+ x выходит из области видимости
```

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 области видимости. Чтобы думать о чём-то, нужно иметь название для этого.

'static

Время жизни с именем «static» — особенное. Оно обозначает, что что-то имеет время жизни, равное времени жизни всей программы. Большинство программистов на Rust впервые сталкиваются с `'static`, когда имеют дело со строками:

```
let x: &'static str = "Привет, мир.";
```

Строковые литералы имеют тип `&'static str`, потому что ссылка всегда действительна: строки 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 отдельным 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 времён жизни, но одно 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 двух видах: с явно и неявно заданным временем жизни:

```rust
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 структура, содержащая 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; // Развёрнуто: Выходное время жизни неясно

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 шаблона, 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 совсем не может измениться. Мы говорим о «внешней изменяемости». 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` — неизменяемое имя. Мы не можем получить ссылку (`&mut 5`) или сделать что-то подобное. И что же?

Для того чтобы понять это, мы должны вернуться назад к основам философии Rust, 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. Таким образом, `let mut z = Arc::new(5);` объявляет атомарный счётчик ссылок с внешней изменяемостью.

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.

Стоит отметить, что тип изменяемости — внутренняя или внешняя — определяется самим типом. Нет способа волшебно превратить значение с внутренней изменяемостью в значение со внешней, и наоборот.

Всё это подводит нас к другим аспектам правил изменяемости Rust. Давайте поговорим о них.

Изменяемость на уровне полей

Изменяемость — это свойство либо ссылки (`&mut`), либо имени (`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`.

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)`.

Rust не поддерживает изменяемость отдельных полей, поэтому вы не можете написать что-то вроде такого:

```
struct Point {
    mut x: i32,
    y: i32,
}
```

Изменяемость — это свойство имени, а не самой структуры. Если вы привыкли к управлению изменяемостью на уровне полей, сначала это может показаться непривычным, но на самом деле такое решение сильно упрощает вещи. Оно даже позволяет вам делать имена изменяемыми только на короткое время:

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let mut point = Point { x: 0, y: 0 };

    point.x = 5;

    let point = point; // это новое имя неизменяемо

    point.y = 6; // это вызывает ошибку
}
```

Синтаксис обновления (update syntax)

Вы можете включить в описание структуры `..` чтобы показать, что вы хотите использовать значения полей какой-то другой структуры. Например:

```
struct Point3d {
    x: i32,
    y: i32,
    z: i32,
}

let mut point = Point3d { x: 0, y: 0, z: 0 };
point = Point3d { y: 1, .. point };
```

Этот код присваивает `point` новое `y`, но оставляет старые `x` и `z`. Это не обязательно должна быть та же самая структура — вы можете использовать этот синтаксис когда создаёте новые структуры, чтобы скопировать значения неуказанных полей:

```
let origin = Point3d { x: 0, y: 0, z: 0 };
let point = Point3d { z: 1, x: 2, .. origin };
```

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`.

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.

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 пользовательские типы по умолчанию не поддерживают операции, может показаться довольно ограниченным. Но это ограничение, которое мы всегда можем преодолеть. Есть два способа: реализовать операцию самостоятельно, или воспользоваться сопоставлением с образцом с помощью `match`, о котором вы узнаете в следующем разделе. Пока мы еще недостаточно знаем Rust, чтобы реализовывать операции, но мы научимся делать это в разделе `traits`.

Конструкция `match`

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». Глава «Шаблоны» описывает все шаблоны, которые можно использовать в `match`.

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 с помощью `match` можно удобно преобразовать значения одного типа в другой.

Сопоставление с образцом для перечислений

Другой полезный способ использования `match` — обработка возможных вариантов перечисления:

```
enum Message {
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 },
    Write(String),
}

fn quit() { /* ... */ }
fn change_color(r: i32, g: i32, b: i32) { /* ... */ }
fn move_cursor(x: i32, y: i32) { /* ... */ }

fn process_message(msg: Message) {
    match msg {
        Message::Quit => quit(),
        Message::ChangeColor(r, g, b) => change_color(r, g, b),
        Message::Move { x: x, y: y } => move_cursor(x, y),
        Message::Write(s) => println!("{}", s),
    };
}
```

Как обычно, компилятор Rust проверяет полноту, поэтому в `match` должна быть ветвь для каждого варианта перечисления. Если какой-то вариант отсутствует, программа не скомпилируется и вам придётся использовать `_`.

Здесь мы не можем использовать обычный `if` вместо `match`, в отличие от кода, который мы видели раньше. Но мы могли бы использовать `if let` — его можно воспринимать как сокращённую форму записи `match`.

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!("один"),
    2 => println!("два"),
    3 => println!("три"),
    _ => println!("что угодно"),
}
```

Этот код напечатает `один`.

Сопоставление с несколькими шаблонами

Вы можете сопоставлять с несколькими шаблонами, используя `|`:

```
let x = 1;

match x {
    1 | 2 => println!("один или два"),
    3 => println!("три"),
    _ => println!("что угодно"),
}
```

Этот код напечатает `один или два`.

Деструктуризация

Если вы работаете с составным типом данных, вроде `struct`, вы можете разобрать его на части («деструктурировать») внутри шаблона:

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x, y } => println!("({},{})", x, y),
}
```

Мы можем использовать `:`, чтобы привязать значение к новому имени.

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x: x1, y: y1 } => println!("({},{})", x1, y1),
}
```

Если нас интересуют только некоторые значения, мы можем не давать имена всем составляющим:

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x, .. } => println!("x равен {}", x),
}
```

Этот код напечатает `x равен 0`.

Вы можете использовать это в любом сопоставлении: не обязательно игнорировать именно первый элемент:

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { y, .. } => println!("y равен {}", y),
}
```

Этот код напечатает `y равен 0`.

Можно произвести деструктуризацию любого составного типа данных — например, кортежей и перечислений.

Игнорирование связывания

Вы можете использовать в шаблоне `_`, чтобы проигнорировать соответствующее значение. Например, вот сопоставление `Result<T, E>`:

```
match some_value {
    Ok(value) => println!("получили значение: {}", value),
    Err(_) => println!("произошла ошибка"),
}
```

В первой ветви мы привязываем значение варианта `Ok` к имени `value`. А в ветви обработки варианта `Err` мы используем `_`, чтобы проигнорировать конкретную ошибку, и просто печатаем общее сообщение.

`_` допустим в любом шаблоне, который связывает имена. Это можно использовать, чтобы проигнорировать части большой структуры:

```
fn coordinate() -> (i32, i32, i32) {
    // создаём и возвращаем какой-то кортеж из трёх элементов
}

let (x, _, z) = coordinate();
```

Здесь мы связываем первый и последний элемент кортежа с именами `x` и `z` соответственно, а второй элемент игнорируем.

Похожим образом, в шаблоне можно использовать `..`, чтобы проигнорировать несколько значений.

```
enum OptionalTuple {
    Value(i32, i32, i32),
    Missing,
}

let x = OptionalTuple::Value(5, -2, 3);

match x {
    OptionalTuple::Value(..) => println!("Получили кортеж!"),
    OptionalTuple::Missing => println!("Вот неудача."),
}
```

Этот код печатает `Получили кортеж!`.

ref и ref mut

Если вы хотите получить ссылку, то используйте ключевое слово `ref`:

```
let x = 5;

match x {
    ref r => println!("Получили ссылку на {}", r),
}
```

Этот код напечатает `Получили ссылку на 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!("Получили изменяемую ссылку на {}", mr),
}
```

Сопоставление с диапазоном

Вы можете сопоставлять с диапазоном значений, используя `...`:

```
let x = 1;

match x {
    1 ... 5 => println!("от одного до пяти"),
    _ => println!("что угодно"),
}
```

Этот код напечатает `от одного до пяти`.

Диапазоны в основном используются с числами или одиночными символами (`char`).

```
let x = '';

match x {
    'а' ... 'и' => println!("ранняя буква"),
    'к' ... 'я' => println!("поздняя буква"),
    _ => println!("что-то ещё"),
}
```

Этот код напечатает `что-то ещё`.

Связывание

Вы можете связать значение с именем с помощью символа `@`:

```
let x = 1;

match x {
    e @ 1 ... 5 => println!("получили элемент диапазона {}", e),
    _ => println!("что угодно"),
}
```

Этот код напечатает `получили элемент диапазона 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!("получили элемент диапазона {}", e),
    _ => println!("что угодно"),
}
```

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!("Получили целое больше пяти!"),
    OptionalInt::Value(..) => println!("Получили целое!"),
    OptionalInt::Missing => println!("Неудача."),
}
```

Этот код напечатает `Получили целое!`.

Если вы используете `if` с несколькими шаблонами, он применяется к обоим частям:

```
let x = 4;
let y = false;

match x {
    4 | 5 if y => println!("да"),
    _ => println!("нет"),
}
```

Этот код печатает `нет`, потому что `if` применяется ко всему `4 | 5`, а не только к `5`. Другими словами, приоритет `if` выглядит так:

```
(4 | 5) if y => ...
```

а не так:

```
4 | (5 if y) => ...
```

Заключение

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 — это очень мощный инструмент. Используйте их.

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». Было бы классно записать вызовы в том порядке, в котором они происходят, не так ли?

```
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!("принимаем self по ссылке!");
    }

    fn mutable_reference(&mut self) {
       println!("принимаем self по изменяемой ссылке!");
    }

    fn takes_ownership(self) {
       println!("принимаем владение 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 шаблон программирования, который очень распространен в коде на Rust:

```
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()`.

Шаблон «строитель» (Builder Pattern)

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 шаблон «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!("площадь: {}", c.area());
    println!("x: {}", c.x);
    println!("y: {}", c.y);
}
```

Всё, что мы сделали здесь — это создали ещё 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 создаёт наш окончательный `Circle` из строителя. Таким образом, мы можем использовать методы `CircleBuilder` чтобы уточнить создание `Circle`.

Вектора

«Вектор» — это динамический или, по-другому, «растущий» массив, реализованный в виде стандартного библиотечного типа `Vec<T>` (где `<T>` является обобщённым типом). 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]`.

Обход

Вы можете обойти элементы вектора 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 — важное понятие для любого программиста. Система обработки строк в Rust немного отличается от других языков, потому что это язык системного программирования. Работать со структурами данных с переменным размером довольно сложно, и строки — как раз такая структура данных. 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 байт UTF-8. Все строки должны быть гарантированно валидными UTF-8 последовательностями. Кроме того, строки не оканчиваются нулём и могут содержать нулевые байты.

В Rust есть два основных типа строк: `&str` и `String`. Сперва поговорим о `&str` — это «строковый срез». Строковые срезы имеют фиксированный размер и не могут быть изменены. Они представляют собой ссылку на последовательность байт UTF-8:

```
let greeting = "Всем привет."; // greeting: &'static str
```

`"Всем привет."` — это строковый литерал, его тип — `&'static str`. Строковые литералы являются статически размещенными строковыми срезами. Это означает, что они сохраняются внутри нашей скомпилированной программы и существуют в течение всего периода ее выполнения. Имя `greeting` представляет собой ссылку на эту статически размещенную строку. Любая функция, ожидающая строковый срез, может также принять в качестве аргумента строковый литерал.

Строковые литералы могут состоять из нескольких строк. Такие литералы можно записывать в двух разных формах. Первая будет включать в себя перевод на новую строку и ведущие пробелы:

```
let s = "foo
    bar";

assert_eq!("foo\n        bar", s);
```

Вторая форма, включающая в себя `\`, вырезает пробелы и перевод на новую строку:

```
let s = "foo\
    bar";

assert_eq!("foobar", s);
```

Но в Rust есть не только `&str`. Тип `String` представляет собой строку, размещенную в куче. Эта строка расширяема, и она также гарантированно является последовательностью UTF-8. `String` обычно создаётся путем преобразования из строкового среза с использованием метода `to_string`.

```
let mut s = "Привет".to_string(); // mut s: String
println!("{}", s);

s.push_str(", мир.");
println!("{}", s);
```

`String` преобразуются в `&str` с помощью `&`:

```
fn takes_slice(slice: &str) {
    println!("Получили: {}", slice);
}

fn main() {
    let s = "Привет".to_string();
    takes_slice(&s);
}
```

Это преобразование не происходит в случае функций, которые принимают какой-то типаж `&str`, а не сам `&str`. Например, у метода `TcpStream::connect` есть параметр типа `ToSocketAddrs`. Сюда можно передать `&str`, но `String` нужно явно преобразовать с помощью `&*`.

```
use std::net::TcpStream;

TcpStream::connect("192.168.0.1:3000"); // параметр &str

let addr_string = "192.168.0.1:3000".to_string();
TcpStream::connect(&*addr_string); // преобразуем addr_string в &str
```

Представление `String` как `&str` — дешёвая операция, но преобразование `&str` в `String` предполагает выделение памяти. Не стоит делать это без необходимости!

Индексация

Поскольку строки являются валидными UTF-8 последовательностями, то они не поддерживают индексацию:

```
let s = "привет";

println!("Первая буква s — {}", s[0]); // ОШИБКА!!!
```

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 (codepoints):

```
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); // что-то вроде hachiko[1]
```

Это подчеркивает, что мы должны пройти по списку `chars` от его начала.

Срезы

Вы можете получить срез строки с помощью синтаксиса срезов:

```
let dog = "hachiko";
let hachi = &dog[0..5];
```

Но заметьте, что это индексы *байтов*, а не *символов*. Поэтому этот код запаникует:

```
let dog = "□□□□□";
let hachi = &dog[0..2];
```

с такой ошибкой:

```
thread '<main>' panicked at 'index 0 and/or 2 in `□□□□□` do not lie on
character boundary'
```

Конкатенация

Если у вас есть `String`, то вы можете присоединить к нему 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 возможность называется «Приведение при разыменовании».

Обобщённое программирование

Иногда, при написании функции или типа данных, мы можем захотеть, чтобы они работали для нескольких типов аргументов. К счастью, у Rust есть возможность, которая даёт нам лучший способ реализовать это: обобщённое программирование. Обобщённое программирование называется «параметрическим полиморфизмом» в теории типов. Это означает, что типы или функции имеют несколько форм (poly — кратно, morph — форма) по данному параметру («параметрический»).

В любом случае, хватит о теории типов; давайте рассмотрим какой-нибудь обобщённый код. Стандартная библиотека Rust предоставляет тип `Option<T>`, который является обобщённым типом:

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

Часть `<T>`, которую вы раньше уже видели несколько раз, указывает, что это обобщённый тип данных. Внутри перечисления, везде, где мы видим `T`, мы подставляем вместо этого абстрактного типа тот, который используется в обобщении. 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, поэтому компилятор 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 типы должны совпадать:

```
let x: Option<i32> = Some(5);
let y: Option<f64> = Some(5.0f64);
```

Это просто прекрасно. Одно определение — многостороннее использование.

Обобщать можно более, чем по одному параметру. Рассмотрим другой обобщённый 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 могут быть любыми. 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 произойдёт какой-либо сбой.

Обобщённые функции

Мы можем задавать функции, которые принимают обобщённые типы, с помощью аналогичного синтаксиса:

```
fn takes_anything<T>(x: T) {
    // делаем что-то с 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 обобщённый 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.

Обобщённые структуры

Вы также можете задать обобщённый 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 обобщённые параметры в `<T>`, а затем используем их в объявлении типа `x: T`.

Типажи

Типаж --- это возможность объяснить компилятору, что данный тип должен предоставлять определённую функциональность.

Вы помните ключевое слово `impl`, используемое для вызова функции через синтаксис 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`.

Мы можем использовать типажи для ограничения обобщённых типов. 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 обобщённому типу `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!("Площадь этой фигуры равна {}", 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:

```
Площадь этой фигуры равна 3.141593
Площадь этой фигуры равна 1
```

Как вы можете видеть, теперь `print_area` не только является обобщённой функцией, но и гарантирует, что будет получен корректный тип. Если же мы передадим некорректный тип:

```
print_area(5);
```

Мы получим ошибку времени компиляции:

```
error: the trait `HasArea` is not implemented for the type `_` [E0277]
```

До сих пор мы добавляли реализации типажей лишь для структур, но реализовать типаж можно 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!("это нелепо");

        *self as f64
    }
}

5.area();
```

Хотя технически это возможно, реализация методов для примитивных типов считается плохим стилем программирования.

Может показаться, что такой подход легко приводит к бардаку в коде, однако есть два ограничения, связанные с реализацией типажей, которые мешают коду выйти из-под контроля. Во-первых, если типаж не определён в нашей области видимости, он не применяется. Например, стандартная библиотека предоставляет типаж `Write`, который добавляет типу `File` функциональность ввода-вывода. По умолчанию у `File` не будет этих методов:

```
let mut f = std::fs::File::open("foo.txt").ok().expect("Не могу открыть foo.txt");
let buf = b"whatever"; // литерал строки байт. buf: &[u8; 8]
let result = f.write(buf);
```

Вот ошибка:

```
error: type `std::fs::File` does not implement any method in scope named `write`
let result = f.write(buf);
               ^~~~~~~~~~
```

Сначала мы должны сделать `use` для типажа `Write`:

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

let mut f = std::fs::File::open("foo.txt").ok().expect("Не могу открыть foo.txt");
let buf = b"whatever";
let result = f.write(buf);
```

Это скомпилируется без ошибки.

Благодаря такой логике работы, даже если кто-то сделает что-то страшное — например, добавит методы `i32`, это не коснётся вас, пока вы не импортируете типаж.

Второе ограничение реализации типажей --- это то, что или типаж, или тип, для которого вы реализуете типаж, должен быть реализован вами. Мы могли бы определить `HasArea` для `i32`, потому что `HasArea` — это наш код. Но если бы мы попробовали реализовать для `i32` `ToString` — типаж, предоставляемый Rust — мы бы не смогли сделать это, потому что ни типаж, ни тип не реализован нами.

Последнее, что нужно сказать о типажах: обобщённые функции с ограничением по типажам используют *мономорфизацию* (*mono*: один, *morph*: форма), поэтому они диспетчеризуются статически. Что это значит? Посмотрите главу Типажи-объекты, чтобы получить больше информации.

Множественные ограничения по типажам

Вы уже видели, как можно ограничить обобщённый параметр типа определённым типажом:

```
fn foo<T: Clone>(x: T) {
    x.clone();
}
```

Если вам нужно больше одного ограничения, 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 обобщёнными 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. Ограничения загромождают место.

Есть решение и для этой проблемы, 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("Привет", "мир");
    bar("Привет", "мир");
}
```

`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. В более длинных списках можно использовать пробелы:

```
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.

На самом деле `where` не только упрощает написание, это более мощная возможность. Например:

```
trait ConvertTo<Output> {
    fn convert(&self) -> Output;
}

impl ConvertTo<i64> for i32 {
    fn convert(&self) -> i64 { *self as i64 }
}

// может быть вызван с T == i32
fn normal<T: ConvertTo<i64>>(x: &T) -> i64 {
    x.convert()
}

// может быть вызван с T == i64
fn inverse<T>() -> T
        // использует ConvertTo как если бы это было «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 (вроде `T`).

Методы по умолчанию

Есть еще одна особенность типажей, о которой стоит поговорить: методы по умолчанию. Проще всего показать это на примере:

```
trait Foo {
    fn is_valid(&self) -> bool;

    fn is_invalid(&self) -> bool { !self.is_valid() }
}
```

В типах, реализующих типаж `Foo`, нужно реализовать метод `is_valid()`, а `is_invalid()` будет реализован по-умолчанию. Его поведение можно переопределить:

```
struct UseDefault;

impl Foo for UseDefault {
    fn is_valid(&self) -> bool {
        println!("Вызван UseDefault.is_valid.");
        true
    }
}

struct OverrideDefault;

impl Foo for OverrideDefault {
    fn is_valid(&self) -> bool {
        println!("Вызван OverrideDefault.is_valid.");
        true
    }

    fn is_invalid(&self) -> bool {
        println!("Вызван OverrideDefault.is_invalid!");
        true // эта реализация противоречит сама себе!
    }
}

let default = UseDefault;
assert!(!default.is_invalid()); // печатает «Вызван UseDefault.is_valid.»

let over = OverrideDefault;
assert!(over.is_invalid()); // печатает «Вызван OverrideDefault.is_invalid!»
```

Наследование

Иногда чтобы реализовать один типаж, нужно реализовать типажи, от которых он зависит:

```
trait Foo {
    fn foo(&self);
}

trait FooBar : Foo {
    fn foobar(&self);
}
```

Типы, реализующие `FooBar`, должны реализовывать `Foo`:

```
struct Baz;

impl Foo for Baz {
    fn foo(&self) { println!("foo"); }
}

impl FooBar for Baz {
    fn foobar(&self) { println!("foobar"); }
}
```

Если мы забудем реализовать `Foo`, компилятор скажет нам об этом:

```
error: the trait `main::Foo` is not implemented for the type `main::Baz` [E0277]
```

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.

Конструкция `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 'диспетчеризация.' Есть две основные формы диспетчеризации: статическая и динамическая. Хотя Rust и отдает предпочтение статической диспетчеризации, он также поддерживает динамическую диспетчеризацию 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 диспетчеризация

Мы можем использовать этот типаж для выполнения статической диспетчеризации 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 диспетчеризации. 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 диспетчеризация 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 диспетчеризация 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 диспетчеризацию. Кроме того, всегда можно иметь тонкую статически- диспетчеризуемую 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 диспетчеризуемой везде, где это возможно.

Динамическая диспетчеризация

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 «затиранием 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 специализированными 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.

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` адресует 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 {
    // компилятор гарантирует, что эта функция вызывается только
    // с `x`, указывающим на u8
    let byte: &u8 = unsafe { &*(x as *const u8) };

    byte.method()
}

static Foo_for_u8_vtable: FooVtable = FooVtable {
    destructor: /* магия компилятора */,
    size: 1,
    align: 1,

    // преобразование в указатель на функцию
    method: call_method_on_u8 as fn(*const ()) -> String,
};


// String:

fn call_method_on_String(x: *const ()) -> String {
    // компилятор гарантирует, что эта функция вызывается только
    // с `x`, указывающим на String
    let string: &String = unsafe { &*(x as *const String) };

    string.method()
}

static Foo_for_String_vtable: FooVtable = FooVtable {
    destructor: /* магия компилятора */,
    // значения для 64-битного компьютера, для 32-битного они в 2 раза меньше
    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);
```

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. Мы добавили пробелы здесь, чтобы было нагляднее:

```
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.

Перемещающие замыкания (`move` closures)

Мы можем заставить наше замыкание забирать право владения на свое окружение с помощью 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 `add_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 синтаксический 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 обобщенный параметр. В этом примере, замыкание принимает `i32` в качестве аргумента и возвращает `i32`, поэтому связывание, которое мы используем, выглядит так: `Fn(i32) -> i32`.

Здесь есть еще один ключевой момент: так как мы ограничиваем обобщённый 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) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

Это выдаст следующие длинные, взаимосвязанные ошибки:

```
error: the trait `core::marker::Sized` is not implemented for the type
`core::ops::Fn(i32) -> i32` [E0277]
fn factory() -> (Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~
note: `core::ops::Fn(i32) -> i32` does not have a constant size known at compile-time
fn factory() -> (Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~
error: the trait `core::marker::Sized` is not implemented for the type `core::ops::Fn(i32)
-> i32` [E0277]
let f = factory();
    ^
note: `core::ops::Fn(i32) -> i32` does not have a constant size known at compile-time
let f = factory();
    ^
```

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) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, 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: closure may outlive the current function, but it borrows `num`,
which is owned by the current function [E0373]
Box::new(|x| x + num)
         ^~~~~~~~~~
```

Мы все еще по-прежнему ссылаемся на родительский фрейм стека. С этим 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 возможность 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 вызывает 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 помощью Cargo. Контейнеры могут производить исполняемый файл или библиотеку, в зависимости от проекта.

Каждый контейнер имеет неявный *корневой модуль*, содержащий код для этого контейнера. 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` и `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 быть ограничен `Sync` и не может иметь реализации `Drop`.

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.

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 прикреплены.

Например, рассмотрим такую функцию:

```
#[test]
fn check() {
    assert_eq!(2, 1 + 1);
}
```

Функция 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 возможностей 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, но, в конечном счете, является неудобным. Например, любая функция, которая принимает `Graph` в качестве параметра, *также* должна быть обобщённой с 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) -> u32 { ... }
```

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.

Эти объявления `type` могут иметь все то же самое, как и при работе с функциями. 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;
```

Использование типажей операций в обобщённых структурах

Теперь, когда мы знаем, как реализованы типажи операций, мы можем реализовать наш типаж `HasArea` и структуру `Square` из главы о типажах более общим образом:

```
use std::ops::Mul;

trait HasArea<T> {
    fn area(&self) -> T;
}

struct Square<T> {
    x: T,
    y: T,
    side: T,
}

impl<T> HasArea<T> for Square<T>
        where T: Mul<Output=T> + Copy {
    fn area(&self) -> T {
        self.side * self.side
    }
}

fn main() {
    let s = Square {
        x: 0.0f64,
        y: 0.0f64,
        side: 12.0f64,
    };

    println!("Площадь s: {}", s.area());
}
```

Мы просто объявляем тип-параметр `T` и используем его вместо `f64` в определении `HasArea` и `Square`. В реализации нужно сделать более хитрые изменения:

```
impl<T> HasArea<T> for Square<T>
        where T: Mul<Output=T> + Copy { ... }
```

Чтобы реализовать `area`, мы должны мочь умножить операнды друг на друга, поэтому мы объявляем `T` как реализующий `std::ops::Mul`. Как и `Add`, `Mul` принимает параметр `Output`: т.к. мы знаем, что числа не меняют своего типа, когда их умножают, `Output` также объявлен как `T`. `T` также должен поддерживать копирование, чтобы Rust не пытался переместить `self.side` в возвращаемое значение.

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 обобщённой 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 «возможности 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 функции, 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]

```
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Ого, 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 далее в этой главе. 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.

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. `$(...)*` проходит через один «слой» повторений, для всех `$name`, которые он содержит, в ногу, и
2. каждое `$name` должно быть под, по крайней мере, стольким количеством `$(...)*`, сколько было использовано при сопоставлении. Если оно под большим числом `$(...)*`, `$name` будет дублироваться, при необходимости.

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 выраженииий 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.

Распространённые макросы

Вот некоторые распространённые макросы, которые вы увидите в коде на Rust.

panic!

Этот макрос вызывает панику текущего потока. Вы можете указать сообщение, с которым поток завершится:

```
panic!("о нет!");
```

vec!

Макрос `vec!` используется по всей книге, поэтому вы наверняка уже видели его. Он упрощает создание `Vec<T>`:

```
let v = vec![1, 2, 3, 4, 5];
```

Он также позволяет вам создавать векторы с повторяющимися значениями. Например, вот сто нолей:

```
let v = vec![0; 100];
```

assert! and assert_eq!

Эти два макроса используются в тестах. `assert!` принимает логическое значение. `assert_eq!` принимает два значения и проверяет, что они равны. `true` засчитывается как успех, а `false` вызывает панику и проваливает тест. Вот так:

```
// Работает!

assert!(true);
assert_eq!(5, 3 + 2);

// а это нет :(

assert!(5 < 3);
assert_eq!(5, 3);
```

try!

`try!` используется для обработки ошибок. Он принимает нечто возвращающее `Result<T, E>` и возвращает `T` если было возвращено `Ok<T>`; иначе он делает возврат из функции со значением `Err(E)`. Вроде такого:

```
use std::fs::File;

fn foo() -> std::io::Result<()> {
    let f = try!(File::create("foo.txt"));

    Ok(())
}
```

Такой код читается легче, чем этот:

```
use std::fs::File;

fn foo() -> std::io::Result<()> {
    let f = File::create("foo.txt");

    let f = match f {
        Ok(t) => t,
        Err(e) => return Err(e),
    };

    Ok(())
}
```

unreachable!

Этот макрос применяется, когда вы хотите пометить какой-то код, который никогда не должен исполняться:

```
if false {
    unreachable!();
}
```

Иногда вам придётся определять ветви условных конструкций, которые точно никогда не исполнятся. В таком случае, используйте этот макрос, чтобы в случае ошибки программа запаниковала:

```
let x: Option<i32> = None;

match x {
    Some(_) => unreachable!(),
    None => println!("Я знаю, что x — это None!"),
}
```

unimplemented!

Макрос `unimplemented!` можно использовать, когда вы хотите, чтобы ваш код прошёл проверку типов, но пока не хотите реализовывать его настоящую логику. Один из примеров — это реализация типажа с несколькими требуемыми методами. Возможно, вы хотите разбираться с типажом постепенно — по одному методу за раз. В таком случае, определите остальные методы как `unimplemented!`, пока не захотите наконец реализовать их.

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 представленного здесь по соображениям эффективности и повторного использования. ↩

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 следствием 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 использованными 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 является 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

Главная сила Rust — в мощных статических гарантиях правильности поведения программы во время исполнения. Но проверки безопасности очень осторожны: на самом деле, существуют безопасные программы, правильность которых компилятор доказать не в силах. Чтобы писать такие программы, нужен способ немного ослабить ограничения. Для этого в Rust есть ключевое слово `unsafe`. Код, использующий `unsafe`, ограничен меньше, чем обычный код.

Давайте рассмотрим синтаксис, а затем поговорим о семантике. `unsafe` используется в четырёх контекстах. Первый — это объявление того, что функция небезопасна:

```
unsafe fn beregis_avtomobilya() {
    // страшные вещи
}
```

Например, все функции, вызываемые через FFI, должны быть помечены как небезопасные. Другое использование `unsafe` — это отметка небезопасного блока:

```
unsafe {
    // страшные вещи
}
```

Третье — небезопасные типажи:

```
unsafe trait Scary { }
```

И четвёртое — реализация (`impl`) таких типажей:

```
unsafe impl Scary for i32 {}
```

Важно явно выделить код, ошибки в котором могут вызвать большие проблемы. Если программа на Rust падает с "segmentation fault", можете быть уверены — проблема в участке, помеченном как небезопасный.

Что значит "безопасный"?

В контексте Rust "безопасный" значит "не делает ничего небезопасного". Также важно знать, что некоторое поведение скорее всего нежелательно, но явно *не* считается небезопасным:

- Deadlock'и
- Утечка памяти или других ресурсов
- Выход без вызова деструкторов
- Целочисленное переполнение

Rust не может предотвратить все виды проблем в программах. Код с ошибками может и будет написан на Rust. Вышеперечисленные вещи неприятны, но они не считаются именно что небезопасными.

В дополнение к этому, ниже представлен список неопределённого поведения (undefined behavior) в Rust. Избегайте этих вещей, даже когда пишете небезопасный код:

- Гонка данных
- Разыменование нулевого или висячего указателя
- Чтение неинициализированной памяти
- Нарушение правил о совпадении указателей с помощью сырых указателей
- `&mut T` и `&T` следуют модели LLVM noalias, кроме случаев, когда `&T` содержит `UnsafeCell<U>`. Небезопасный код не должен нарушать эти гарантии совпадения указателей.
- Изменение неизменяемого значения или ссылки без использования `UnsafeCell<U>`
- Получение неопределённого поведения с помощью intrinsic-операций компилятора:
    - Индексация вне границ объекта с помощью `std::ptr::offset` (`offset` intrinsic), кроме разрешённого случая "один байт за концом объекта".
    - Использование `std::ptr::copy_nonoverlapping_memory` (intrinsic-операции `memcpy32`/`memcpy64`) с пересекающимися буферами
- Неправильные значения примитивных типов, даже в скрытых полях:
    - Нулевые или висячие ссылки или упаковки (boxes)
    - Любое значение логического типа, кроме `false` (0) или `true` (1)
    - Вариант перечисления, не включённый в его определение
    - Суррогатное значение `char` или значение `char`, превыщающее `char::MAX`
    - Последовательности байт, не являющиеся UTF-8, в `str`
- Размотка стека в код на Rust из чужого кода (через границы FFI), или размотка из кода на Rust в чужой код

Сверхспособности небезопасного кода

В небезопасном блоке или функции, Rust разрешает три ситуации, которые обычно запрещены. Всего три. Вот они:

1. Доступ к или изменение статической изменяемой переменной.
2. Разыменование сырого указателя.
3. Вызов небезопасных функций. Это самая мощная возможность.

Это всё. Важно отметить, что `unsafe`, например, не "выключает проверку заимствования". Объявление какого-то кода небезопасным не изменяет его семантику; небезопасность не означает принятие компилятором любого кода. Но она позволяет писать вещи, которые *нарушают* некоторые из правил.

Вы также встретите ключевое слово `unsafe`, когда будете реализовывать интерфейс к чужому коду не на Rust. Идиоматичным считается написание безопасных обёрток вокруг небезопасных библиотек.

Давайте поговорим о трёх упомянутых возможностях, доступных в небезопасном коде.

Доступ или изменение `static mut`

Rust позволяет пользоваться глобальным изменяемым состоянием с помощью `static mut`. Это может вызвать гонку по данным, и в сущности небезопасно. Подробнее смотрите раздел о static.

Разыменование сырого указателя

Сырые указатели поддерживают произвольную арифметику указетелей, и могут вызвать целый ряд проблем безопасности памяти и безопасности в целом. В каком-то смысле, возможность разыменовать произвольный указатель — одна из самых опасных вещей, которые вы можете сделать. Подробнее смотрите раздел о сырых указателях.

Вызов небезопасных функций

Эта возможность затрагивает то, откуда можно делать вызов небезопасного кода: небезопасные функции могут вызываться только из небезопасных блоков.

Мощь и полезность этой возможности сложно переоценить. Rust предоставляет некоторые intrinsic-операции компилятора в виде небезопасных функций, а некоторые небезопасные функции обходят проверки безопасности для достижения большей скорости исполнения.

В заключение, повторимся: хотя вы и *можете* делать в небезопасных участках почти что угодно, это не значит, что стоит это делать. Компилятор будет предполагать выполнение оговоренных инвариантов, так что будьте осторожны!

Нестабильные возможности 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 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.

Плагины к компилятору

Введение

`rustc`, компилятор Rust, поддерживает плагины. Плагины — это разработанные пользователями библиотеки, которые добавляют новые возможности в компилятор: это могут быть расширения синтаксиса, дополнительные статические проверки (lints), и другое.

Плагин — это контейнер, собираемый в динамическую библиотеку, и имеющий отдельную функцию для регистрации расширения в `rustc`. Другие контейнеры могут загружать эти расширения с помощью атрибута `#![plugin(...)]`. Также смотрите раздел `rustc::plugin` с подробным описанием механизма определения и загрузки плагина.

Передаваемые в `#![plugin(foo(... args ...))]` аргументы не обрабатываются самим `rustc`. Они передаются плагину с помощью метода `args` структуры `Registry`.

В подавляющем большинстве случаев плагин должен использоваться *только* через конструкцию `#![plugin]`, а не через `extern crate`. Компоновка потянула бы внутренние библиотеки `libsyntax` и `librustc` как зависимости для вашего контейнера. Обычно это нежелательно, и может потребоваться только если вы собираете ещё один, другой, плагин. Статический анализ `plugin_as_library` проверяет выполнение этой рекомендации.

Обычная практика — помещать плагины в отдельный контейнер, не содержащий определений макросов (`macro_rules!`) и обычного кода на Rust, предназначенного для непосредственно конечных пользователей библиотеки.

Расширения синтаксиса

Плагины могут по-разному расширять синтаксис Rust. Один из видов расширения синтаксиса — это процедурные макросы. Они вызываются так же, как и обычные макросы, но их раскрытие производится произвольным кодом на Rust, который оперирует синтаксическими деревьями во время компиляции.

Давайте напишем плагин `roman_numerals.rs`, который реализует целочисленные литералы с римскими цифрами.

```
#![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;  // типаж для 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, "аргумент должен быть единственным идентификатором");
            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, "неправильное римское число");
                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);
}
```

Теперь мы можем использовать `rn!()` как любой другой макрос:

```
#![feature(plugin)]
#![plugin(roman_numerals)]

fn main() {
    assert_eq!(rn!(MMXV), 2015);
}
```

У этого подхода есть преимущества относительно простой функции `fn(&str) -> u32`:

- Преобразование (в общем случае, произвольной сложности) выполняется во время компиляции;
- Проверка правильности записи литерала также производится во время компиляции;
- Можно добавить возможность использования литерала в образцах (patterns), что по сути позволяет создавать литералы для любого типа данных.

В дополнение к процедурным макросам, вы можете определять новые атрибуты `derive` и другие виды расширений. Смотрите раздел `Registry::register_syntax_extension` и документацию перечисления `SyntaxExtension`. В качестве более продвинутого примера с макросами, можно ознакомиться с макросами регулярных выражений `regex_macros`.

Советы и хитрости

Некоторые советы по отладке макросов применимы и в случае плагинов.

Можно использовать `syntax::parse`, чтобы преобразовать деревья токенов в высокоуровневые элементы синтаксиса, вроде выражений:

```
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();
```

Можно просмотреть код парсера `libsyntax`, чтобы получить представление о работе инфраструктуры разбора.

Сохраняйте `Span`ы всего, что вы разбираете, чтобы лучше сообщать об ошибках. Вы можете обернуть ваши структуры данных в `Spanned`.

Вызов `ExtCtxt::span_fatal` сразу прервёт компиляцию. Вместо этого, лучше вызвать `ExtCtxt::span_err` и вернуть `DummyResult`, чтобы компилятор мог продолжить работу и обнаружить дальнейшие ошибки.

Вы можете использовать `span_note` и `syntax::print::pprust::*_to_string` чтобы напечатать синтаксический фрагмент для отладки.

Пример выше создавал целочисленный литерал с помощью `AstBuilder::expr_usize`. В качестве альтернативы типажу `AstBuilder`, `libsyntax` предоставляет набор макросов квазицитирования. Они не документированы и совсем не отполированы. Однако, эта реализация может стать неплохой основой для улучшенной библиотеки квазицитирования, которая работала бы как обычный плагин.

Плагины статических проверок

Плагины могут расширять инфраструктуру статических проверок Rust, предоставляя новые проверки стиля кодирования, безопасности, и т.д. Полный пример можно найти в `src/test/auxiliary/lint_plugin_test.rs`. Здесь мы приводим его суть:

```
declare_lint!(TEST_LINT, Warn,
              "Предупреждать об элементах, названных '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, "элемент называется 'lintme'");
        }
    }
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_lint_pass(box Pass as LintPassObject);
}
```

Тогда код вроде

```
#![plugin(lint_plugin_test)]

fn lintme() { }
```

выдаст предупреждение компилятора:

```
foo.rs:4:1: 4:16 warning: item is named 'lintme', #[warn(test_lint)] on by default
foo.rs:4 fn lintme() { }
         ^~~~~~~~~~~~~~~
```

Плагин статического анализа состоит из следующих частей:

- один или больше вызовов `declare_lint!`, которые определяют статические структуры `Lint`;

- структура, содержащая состояние, необходимое анализатору (в данном случае, его нет);
- реализация типажа `LintPass`, определяющая, как проверять каждый элемент синтаксиса. Один `LintPass` может вызывать `span_lint` для нескольких различных `Lint`, но он должен зарегистрировать их все через метод `get_lints`.

Проходы статического анализатора — это обходы синтаксического дерева, но они выполняются на поздних стадиях компиляции, когда уже доступна информация о типах. Встроенные в `rustc` анализы в основном используют ту же инфрастуктуру, что и плагины статического анализа. Смотрите их исходный код, чтобы понять, как получать информацию о типах.

Статические проверки, определяемые плагинами, управляются обычными атрибутами и флагами компилятора, т.е. `#[allow(test_lint)]` или `-A test-lint`. Эти идентификаторы выводятся из первого аргумента `declare_lint!`, с учётом соответствующих преобразований регистра букв и пунктуации.

Вы можете выполнить команду `rustc -W help foo.rs`, чтобы увидеть весь список статических проверок, известных `rustc`, включая те, что загружаются из `foo.rs`.

Встроенный ассемблерный код

Если вам нужно работать на самом низком уровне или повысить производительность программы, то у вас может возникнуть необходимость управлять процессором напрямую. Rust поддерживает использование встроенного ассемблера и делает это с помощью с помощью макроса `asm!`. Синтаксис примерно соответствует синтаксису GCC и Clang:

```
asm!(assembly template
   : output operands
   : input operands
   : clobbers
   : options
   );
```

Использование `asm` является закрытой возможностью (требуется указать `#![feature(asm)]` для контейнера, чтобы разрешить ее использование) и, конечно же, требует `unsafe` блока.

> **Примечание**: здесь примеры приведены для x86/x86-64 ассемблера, но поддерживаются все платформы.

Шаблон инструкции ассемблера

Шаблон инструкции ассемблера (assembly template) является единственным обязательным параметром, и он должен быть представлен строкой символов (т.е. `""`)

```
#![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();
    // ...
}
```

(Далее атрибуты `feature(asm)` и `#[cfg]` будут опущены.)

Выходные операнды (output operands), входные операнды (input operands), затираемое (clobbers) и опции (options) не являются обязательными, но вы должны будете добавить соответствующее количество `:` если хотите пропустить их:

```
asm!("xor %eax, %eax"
    :
    :
    : "{eax}"
   );
```

Пробелы и отступы также не имеют значения:

```
asm!("xor %eax, %eax" ::: "{eax}");
```

Операнды

Входные и выходные операнды имеют одинаковый формат: `:"ограничение1" (выражение1), "ограничение2"(выражение2), ..."`. Выражения для выходных операндов должны быть либо изменяемыми, либо неизменяемыми, но еще не инициализированными, L-значениями:

```
fn add(a: i32, b: i32) -> i32 {
    let c: i32;
    unsafe {
        asm!("add $2, $0"
             : "=r"(c)
             : "0"(a), "r"(b)
             );
    }
    c
}

fn main() {
    assert_eq!(add(3, 14159), 14162)
}
```

Однако, если вы захотите использовать реальные операнды (регистры) в этой позиции, то вам потребуется заключить используемый регистр в фигурные скобки `{}`, и вы должны будете указать конкретный размер операнда. Это полезно для очень низкоуровневого программирования, когда важны регистры, которые вы используете:

```
let result: u8;
asm!("in %dx, %al" : "={al}"(result) : "{dx}"(port));
result
```

Затираемое (Clobbers)

Некоторые инструкции могут изменять значения регистров, поэтому мы используем список затираемого. Он указывает компилятору, что тот не должен допускать какого-либо изменение значений этих регистров, чтобы они оставались корректными.

```
// Put the value 0x200 in eax
asm!("mov $$0x200, %eax" : /* no outputs */ : /* no inputs */ : "{eax}");
```

Если входные и выходные регистры уже заданы в ограничениях, то их не нужно перечислять здесь. В противном случае, любые другие регистры, используемые явно или неявно, должны быть перечислены.

Если ассемблер изменяет регистр кода условия `cc`, то он должен быть указан в качестве одного из затираемых. Точно так же, если ассемблер модифицирует память, то должно быть указано `memory`.

Опции

Последний раздел, `options`, специфичен для Rust. Формат представляет собой разделенные запятыми текстовые строки (т.е. `:"foo", "bar", "baz"`). Он используется для того, чтобы задать некоторые дополнительные данные для встроенного ассемблера:

На текущий момент разрешены следующие опции:

1. *volatile* — эта опция аналогична `__asm__ __volatile__ (...)` в gcc/clang;
2. *alignstack* — некоторые инструкции ожидают, что стек был выровнен определенным образом (т.е. SSE), и эта опция указывает компилятору вставить свой обычный код выравнивания стека;
3. *intel* — эта опция указывает использовать синтаксис Intel вместо используемого по умолчанию синтаксиса AT&T.

```
let result: i32;
unsafe {
   asm!("mov eax, 2" : "={eax}"(result) : : : "intel")
}
println!("eax is currently {}", result);
```

Больше информации

Текущая реализация макроса `asm!` --- это прямое связывание с встроенным ассемблером LLVM, поэтому изучите и их документацию, чтобы лучше понять список затираемого, ограничения и др.

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 вызвана 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, для работы которой нужно сделать некоторые платформо-зависимые предположения; но эти библиотеки всё равно более переносимы, чем стандартная библиотека в целом.

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 отдельном (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 небезопасен и помечен как `unsafe`.

Элементы языка (lang items)

> **Замечание**: многие элементы языка предоставляются контейнерами в стандартной поставке Rust, а у самих элементов языка нестабильный интерфейс. Рекомендуется использовать официально распространяемые контейнеры, вместо того, чтобы определять свои собственные элементы языка.

У компилятора `rustc` есть некоторые подключаемые операции, т.е. функционал, не встроенный жёстко в язык, а реализованный в библиотеках и специально помеченный как элемент языка. Метка — это атрибут `#[lang="..."]`. Есть различные значения `...`, т.е. разные «элементы языка».

Например, для указателей `Box` нужны два элемента языка — для выделения памяти и для освобождения. Вот программа, не использующая стандартную библиотеку, и реализующая `Box` через `malloc` и `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 завершился ошибкой
    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 {} }
```

Заметьте, что `exchange_malloc` должен возвращать допустимый указатель, поэтому он производит проверку внутри и делает `abort`, если она не прошла.

Ниже перечислены другие возможности, предоставляемые элементами языка:

- перегружаемые операторы через типажи: типажи, соответствующие `==`, `<`, разыменованию (`*`), `+` и другим операторам, помечены как элементы языка; конкретно эти типажи помечены как `eq`, `ord`, `deref` и `add`;
- раскрутка стека и общая ошибка; это элементы `eh_personality`, `fail` и `fail_bounds_check`;
- типажи в модуле `std::marker`, используемые чтобы помечать различные типы; элементы `send`, `sync` и `copy`;
- типы-метки и индикаторы вариантности из `std::marker`; это элементы `covariant_type`, `contravariant_lifetime` и другие.

Элементы языка загружаются компилятором лениво, т.е. если программа не использует `Box`, вам не нужно определять элементы `exchange_malloc` и `exchange_free`. `rustc` выдаст ошибку, если элемент языка необходим, но не найден ни в текущем контейнере, ни в его зависимостях.

Продвинутое руководстве по компоновке (advanced linking)

Распространённые ситуации, в которых требовалась компоновка с кодом на Rust, уже были рассмотрены в предыдущих главах книги. Однако для поддержки прозрачного взаимодействия с нативными библиотеками требуется более широкая поддержка разных вариантов компоновки.

Аргументы компоновки (link args)

Есть только один способ тонкой настройки компоновки — атрибут `link_args`. Этот атрибут применяется к блокам `extern`, и указывает сырые аргументы, которые должны быть переданы компоновщику при создании артефакта. Например:

```
#![feature(link_args)]

#[link_args = "-foo -bar -baz"]
extern {}
```

Обратите внимание, что эта возможность скрыта за `feature(link_args)`, так как это нештатный способ компоновки. В данный момент `rustc` вызывает системный компоновщик (на большинстве систем это `gcc`, на Windows — `link.exe`), поэтому передача аргументов командной строки имеет смысл. Но реализация не всегда будет такой — в будущем `rustc` может напрямую использовать LLVM для связывания с нативными библиотеками, и тогда `link_args` станет бессмысленным. Того же эффекта можно достигнуть с пощощью передачи `rustc` аргумента `-C link-args`.

Крайне рекомендуется *не* использовать этот атрибут, и пользоваться вместо него более точно определённым атрибутом `#link(...)` для блоков `extern`.

Статическое связывание

Статическое связывание — это процесс создания артефакта, который содержит все нужные библиотеки, и потому не потребует установленных библиотек на целевой системе. Библиотеки на Rust по умолчанию связываются статически, поэтому приложения и библиотеки на Rust можно использовать без установки Rust повсюду. Напротив, нативные библиотеки (например, `libc` и `libm`) обычно связываются динамически, но это можно изменить, и сделать чтобы они также связывались статически.

Компоновка — это процесс, который реализуется по-разному на разных платформах. На некоторых из них статическое связывание вообще не возможно! Этот раздел предполагает знакомство с процессом компоновки на вашей платформе.

Linux

По умолчанию, программы на Rust для Linux компонуются с системной `libc` и ещё некоторыми библиотеками. Давайте посмотрим на пример на 64-битной машине с Linux, GCC и `glibc` (самой популярной `libc` на Linux):

```
$ cat example.rs
fn main() {}
$ rustc example.rs
$ ldd example
        linux-vdso.so.1 =>  (0x00007ffd565fd000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fa81889c000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fa81867e000)
        librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007fa818475000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fa81825f000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fa817e9a000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fa818cf9000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fa817b93000)
```

Иногда динамическое связывание на Linux нежелательно: например, если вы хотите использовать возможности из новых библиотек на старых системах или на целевых системах нет таких библиотек.

Статическое связывание возможно с альтернативной `libc`, `musl`. Вы можете скомпилировать свою версию Rust, которая будет использовать `musl`, и установить её в отдельную директорию, с помощью инструкции, приведённой ниже:

```
$ mkdir musldist
$ PREFIX=$(pwd)/musldist
$
$ # Build musl
$ wget http://www.musl-libc.org/releases/musl-1.1.10.tar.gz
[...]
$ tar xf musl-1.1.10.tar.gz
$ cd musl-1.1.10/
musl-1.1.10 $ ./configure --disable-shared --prefix=$PREFIX
[...]
musl-1.1.10 $ make
[...]
musl-1.1.10 $ make install
[...]
musl-1.1.10 $ cd ..
$ du -h musldist/lib/libc.a
2.2M    musldist/lib/libc.a
$
$ # Build libunwind.a
$ wget http://llvm.org/releases/3.6.1/llvm-3.6.1.src.tar.xz
$ tar xf llvm-3.6.1.src.tar.xz
$ cd llvm-3.6.1.src/projects/
llvm-3.6.1.src/projects $ svn co http://llvm.org/svn/llvm-project/libcxxabi/trunk/ libcxxab
i
llvm-3.6.1.src/projects $ svn co http://llvm.org/svn/llvm-project/libunwind/trunk/ libunwin
d
llvm-3.6.1.src/projects $ sed -i 's#^\(include_directories\).*$#\0\n\1(../libcxxabi/include
)#' libunwind/CMakeLists.txt
llvm-3.6.1.src/projects $ mkdir libunwind/build
llvm-3.6.1.src/projects $ cd libunwind/build
llvm-3.6.1.src/projects/libunwind/build $ cmake -DLLVM_PATH=../../.. -DLIBUNWIND_ENABLE_SHA
RED=0 ..
llvm-3.6.1.src/projects/libunwind/build $ make
llvm-3.6.1.src/projects/libunwind/build $ cp lib/libunwind.a $PREFIX/lib/
llvm-3.6.1.src/projects/libunwind/build $ cd cd ../../../../
$ du -h musldist/lib/libunwind.a
164K    musldist/lib/libunwind.a
$
$ # Build musl-enabled rust
$ git clone https://github.com/rust-lang/rust.git muslrust
$ cd muslrust
muslrust $ ./configure --target=x86_64-unknown-linux-musl --musl-root=$PREFIX --prefix=$PRE
FIX
muslrust $ make
muslrust $ make install
muslrust $ cd ..
$ du -h musldist/bin/rustc
12K     musldist/bin/rustc
```

Теперь у вас есть сборка Rust с `musl`! Поскольку мы установили её в отдельную корневую директорию, надо удостовериться в том, что система может найти исполняемые файлы и библиотеки:

```
$ export PATH=$PREFIX/bin:$PATH
$ export LD_LIBRARY_PATH=$PREFIX/lib:$LD_LIBRARY_PATH
```

Давайте попробуем!

```
$ echo 'fn main() { println!("hi!"); panic!("failed"); }' > example.rs
$ rustc --target=x86_64-unknown-linux-musl example.rs
$ ldd example
        not a dynamic executable
$ ./example
hi!
thread '<main>' panicked at 'failed', example.rs:1
```

Успех! Эта программа может быть скопирована на почти любую машину с Linux с той же архитектурой процессора и будет работать без проблем.

`cargo build` также принимает опцию `--target`, так что вы можете собирать контейнеры как обычно. Однако, возможно вам придётся пересобрать нативные библиотеки с `musl`, чтобы иметь возможность скомпоноваться с ними.

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 возможности (feature gate) `test`, что включает эту нестабильную возможность.

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 изменения 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 шаблоны `match`

В настоящее время единственный стабильный способ создания `Box` — это создание с помощью метода `Box::new`. В стабильной сборке Rust также невозможно деструктурировать `Box` при использовании 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 возможности 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.

Возврат 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 возможность `slice_patterns`:

```
#![feature(slice_patterns)]

fn main() {
    let v = vec!["match_this", "1"];

    match &v[..] {
        ["match_this", second] => println!("The second element is {}", second),
        _ => {},
    }
}
```

Отключаемая возможность `advanced_slice_patterns` позволяет использовать `..`, чтобы обозначить любое число элементов в шаблоне. Этот символ подстановки можно использовать в массиве один раз. Если перед `..` есть идентификатор, результат среза будет связан с этим именем. Например:

```
#![feature(advanced_slice_patterns, slice_patterns)]

fn is_symmetric(list: &[u32]) -> bool {
    match list {
        [] | [_] => true,
        [x, inside.., y] if x == y => is_symmetric(inside),
        _ => false
    }
}

fn main() {
    let sym = &[0, 1, 4, 2, 4, 1, 0];
    assert!(is_symmetric(sym));

    let not_sym = &[0, 1, 7, 2, 4, 1, 0];
    assert!(!is_symmetric(not_sym));
}
```

Ассоциированные константы

С включенной возможностью `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` должна 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;
}
```

Пользовательские менеджеры памяти

Выделение памяти — это не самая простая задача, и Rust обычно заботится об этом сам, но часто нужно тонко управлять выделением памяти. Компилятор и стандартная библиотека в настоящее время позволяют глобально переключить используемый менеджер во время компиляции. Описание сейчас находится в RFC 1183, но здесь мы рассмотрим как сделать ваш собственный менеджер.

Стандартный менеджер памяти

В настоящее время компилятор содержит два стандартных менеджера: `alloc_system` и `alloc_jemalloc` (однако у некоторых платформ отсутствует jemalloc). Эти менеджеры стандартны для контейнеров Rust и содержат реализацию подпрограмм для выделения и освобождения памяти. Стандартная библиотека не компилируется специально для использования только одного из них. Компилятор будет решать какой менеджер использовать во время компиляции в зависимости от типа производимых выходных артефактов.

По умолчанию исполняемые файлы сгенерированные компилятором будут использовать `alloc_jemalloc` (там где возможно). В таком случае компилятор "контролирует весь мир", в том смысле что у него есть власть над окончательной компоновкой.

Однако динамические и статические библиотеки по умолчанию будут использовать `alloc_system`. Здесь Rust обычно в роли гостя в другом приложении или вообще в другом мире, где он не может авторитетно решать какой менеджер использовать. В результате он возвращается назад к стандартным API (таких как `malloc` и `free`), для получения и освобождения памяти.

Переключение менеджеров памяти

Несмотря на то что в большинстве случаев нам подойдёт то, что компилятор выбирает по умолчанию, часто бывает необходимо настроить определенные аспекты. Для того, чтобы переопределить решение компилятора о том, какой именно менеджер использовать, достаточно просто скомпоновать с желаемым менеджером:

```
#![feature(alloc_system)]

extern crate alloc_system;

fn main() {
    let a = Box::new(4); // выделение памяти с помощью системного менеджера
    println!("{}", a);
}
```

В этом примере сгенерированный исполняемый файл будет скомпонован с системным менеджером, вместо менеджера по умолчанию — jemalloc. И наоборот, чтобы сгенерировать динамическую библиотеку, которая использует jemalloc по умолчанию нужно написать:

```
#![feature(alloc_jemalloc)]
#![crate_type = "dylib"]

extern crate alloc_jemalloc;

pub fn foo() {
    let a = Box::new(4); // выделение памяти с помощью jemalloc
    println!("{}", a);
}
```

Написание своего менеджера памяти

Иногда даже выбора между jemalloc и системным менеджером недостаточно и необходим совершенно новый менеджер памяти. В этом случае мы напишем наш собственный контейнер, который будет предоставлять API менеджера памяти (также как и `alloc_system` или `alloc_jemalloc`). Для примера давайте рассмотрим упрощенную и аннотированную версию `alloc_system`:

```
// Компилятору нужно указать, что этот контейнер является менеджером памяти, для
// того что бы при компоновке он не использовал другой менеджер.
#![feature(allocator)]
#![allocator]

// Менеджерам памяти не позволяют зависеть от стандартной библиотеки, которая в
// свою очередь зависит от менеджера, чтобы избежать циклической зависимости.
// Однако этот контейнер может использовать все из libcore.
#![no_std]

// Давайте дадим какое-нибудь уникальное имя нашему менеджеру.
#![crate_name = "my_allocator"]
#![crate_type = "rlib"]

// Наш системный менеджер будет использовать поставляемый вместе с компилятором
// контейнер libc для связи с FFI. Имейте ввиду, что на данный момент внешний
// (crates.io) libc не может быть использован, поскольку он компонуется со
// стандартной библиотекой (`#![no_std]` все еще нестабилен).
#![feature(libc)]
extern crate libc;

// Ниже перечислены пять функций, необходимые пользовательскому менеджеру памяти.
// Их сигнатуры и имена на данный момент не проверяются компилятором, но это
// вскоре будет реализовано, так что они должны соответствовать тому, что
// находится ниже.
//
// Имейте ввиду, что стандартные `malloc` и `realloc` не предоставляют опций для
// выравнивания, так что эта реализация должна быть улучшена и поддерживать
// выравнивание.
#[no_mangle]
pub extern fn __rust_allocate(size: usize, _align: usize) -> *mut u8 {
```

```
    unsafe { libc::malloc(size as libc::size_t) as *mut u8 }
}

#[no_mangle]
pub extern fn __rust_deallocate(ptr: *mut u8, _old_size: usize, _align: usize) {
    unsafe { libc::free(ptr as *mut libc::c_void) }
}

#[no_mangle]
pub extern fn __rust_reallocate(ptr: *mut u8, _old_size: usize, size: usize,
                                _align: usize) -> *mut u8 {
    unsafe {
        libc::realloc(ptr as *mut libc::c_void, size as libc::size_t) as *mut u8
    }
}

#[no_mangle]
pub extern fn __rust_reallocate_inplace(_ptr: *mut u8, old_size: usize,
                                        _size: usize, _align: usize) -> usize {
    old_size // libc не поддерживает этот API
}

#[no_mangle]
pub extern fn __rust_usable_size(size: usize, _align: usize) -> usize {
    size
}
```

После того как мы скомпилировали этот контейнер, мы можем использовать его следующим образом:

```
extern crate my_allocator;

fn main() {
    let a = Box::new(8); // выделение памяти с помощью нашего контейнера
    println!("{}", a);
}
```

Ограничения пользовательских менеджеров памяти

Несколько ограничений при работе с пользовательским менеджером памяти, которые могут быть причиной ошибок компиляции:

- Любой артефакт может быть скомпонован только с одним менеджером. Исполняемые файлы, динамические библиотеки и статические библиотеки должны быть скомпонованы с одним менеджером, и если не один не был указан, то компилятор сам выберет один. В то же время Rust библиотеки (rlibs) не нуждаются в компоновке с менеджером (но это возможно).

- Потребитель какого-либо менеджера памяти имеет пометку `#![needs_allocator]` (в данном случае контейнер `liballoc`) и какой-либо контейнер `#[allocator]` не может транзитивно зависеть от контейнера, которому нужен менеджер (т.е. циклическая зависимость не допускается). Это означает, что менеджеры памяти в данный момент должны ограничить себя только libcore.

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.

Абстрактное синтаксическое дерево

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
```

Арность

Арность означает число аргументов, которые принимает функция или операция.

```
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.

Выражение

В программировании, выражение — это комбинация значений, постоянных, переменных и функций, которая вычисляется в одно значение. Например, `2 + (3 * 4)` — выражение, вычисляющееся в значение `14`. Стоит заметить, что у выражений могут быть побочные эффекты. Например, функция, участвующая в выражении, может делать что-то ещё помимо непосредственно возврата значения.

Язык, ориентированный на выражения

В ранних языках программирования выражения и операторы были двумя отдельными видами синтаксиса: выражения вычислялись в выражение, а операторы производили действия с побочными эффектами. Однако поздние языки уже не имели такого чёткого разделения по этому критерию. В языке, ориентированном на выражения, почти любой оператор — это выражение, а значит, оно возвращает значение. Следовательно, эти выражения могут сами являться частью ещё больших выражений.

Оператор

В программировании, оператор — это наименьший отдельный элемент языка, который обозначает выполнение компьютером законченного действия. Например, в языке C `printf("42");` — это оператор.

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 for shared-memory programming
- 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

Статьи *о* Rust

- GPU Programming in Rust: Implementing High Level Abstractions in a Systems Level Language. Early GPU work by Eric Holk.
- Parallel closures: a new twist on an old idea
    - not exactly about rust, but by nmatsakis
- Patina: A Formalization of the Rust Programming Language. Early formalization of a subset of the type system, by Eric Reed.
- Experience Report: Developing the Servo Web Browser Engine using Rust. By Lars Bergstrom.
- Implementing a Generic Radix Trie in Rust. Undergrad paper by Michael Sproul.
- Reenix: Implementing a Unix-Like Operating System in Rust. Undergrad paper by Alex Light.
- Evaluation of performance and productivity metrics of potential programming languages in the HPC environment. Bachelor's thesis by Florian Wilkens. Compares C, Go and Rust.
- Nom, a byte oriented, streaming, zero copy, parser combinators library in Rust. By Geoffroy Couprie, research for VLC.
- Graph-Based Higher-Order Intermediate Representation. An experimental IR implemented in Impala, a Rust-like language.
- Code Refinement of Stencil Codes. Another paper using Impala.