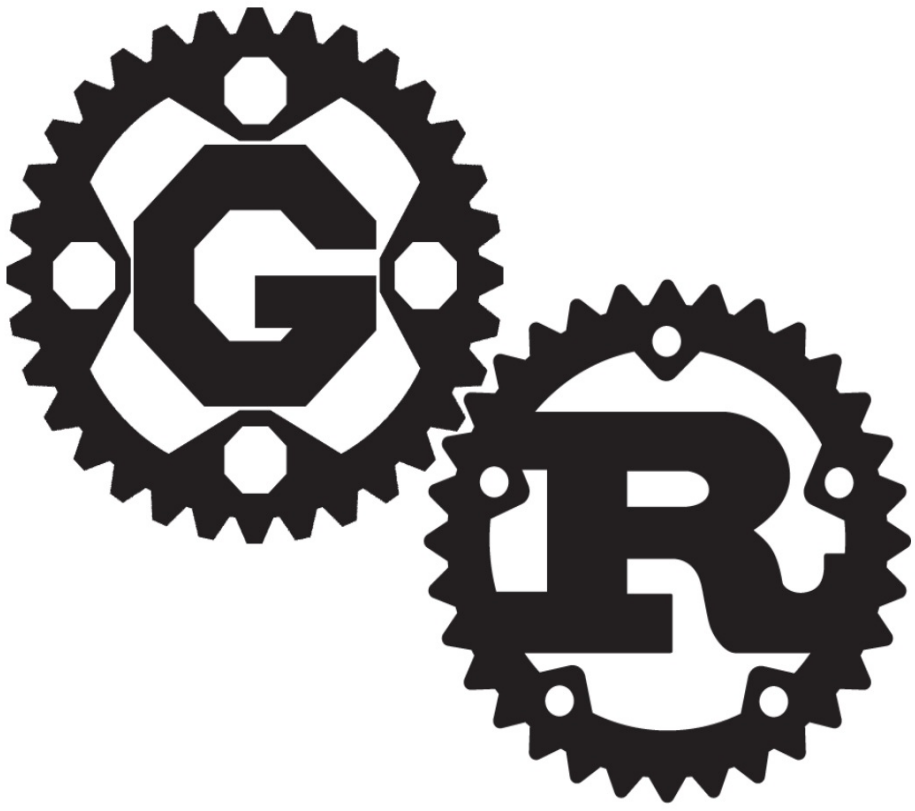
Язык программирования Rust

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 10 minutes

issues closed in not available

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

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 —

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 «Привет, мир!», 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. В этой главе нам понадобится интернет соединение для выполнения команд, с помощью которых мы загрузим Rust из интернета.

Мы воспользуемся несколькими командами в терминале, и они все будут начинаться с $. Вам не нужно вводить $, они используются только для того чтобы обозначить начало каждой команды. В интернете можно увидеть множество руководств и примеров, которые следуют этому правилу: $ обозначает команды, которые выполняются с правами обычного пользователя и # для команд, которые выполняются с правами администратора.

Поддерживаемые платформы

Перечень платформ, на которых работает и для которых компилирует компилятор Rust довольно большой, однако, не все платформы поддерживаются одинаково. Уровни поддержки Rust разбиты на три уровня, у каждого из которых свой набор гарантий.

Платформы идентифицируются по их "целевой тройке", которая является строкой, сообщающей компилятору, какие выходные данные должны быть произведены. Столбцы ниже указывают, работает ли соответствующий компонент на указанной платформе.

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».

3. С чего начать4.1. Угадайка

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`:

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

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,

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.

4.3. Вызов кода на Rust из других языков**5.1.** Стек и куча

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; }

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

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`:

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

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

Условная компиляция

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

fn main() { #[cfg(foo)] fn foo() {} #[cfg(bar = "baz")] fn bar() {} }

```
#[cfg(foo)]

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

Над атрибутами конфигурации определены логические операции:

fn main() { #[cfg(any(unix, windows))] fn foo() {} #[cfg(all(unix, target_pointer_width = "32"))] fn bar() {} #[cfg(not(foo))] fn not_foo() {} }

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

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

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

Они могут быть как угодно вложены:

fn main() { #[cfg(any(not(unix), all(target_os="macos", target_arch = "powerpc")))] fn foo() {} }

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

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 возмо
default = []

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

Если вы определите такие возможности, Cargo

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:

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

```
/// 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

Итераторы

Давайте поговорим о циклах.

Помните цикл `for` в Rust? Вот пример:

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

```
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:

fn main() { let mut range = 0..10; loop { match range.next() { Some(x) => { println!("{}", x); }, None => { break } } } }

```
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`.

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

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.

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:

fn main() { let x = Box::new(1); let y = x; // x больше не доступен }

```
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,

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:

fn main() { let x = Box::new(1); let y = x; // x больше не доступен }

```
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,

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:

#![feature(libc)] 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); }

```
extern crate libc;
use libc::size_t;

#[link(name = "snappy")]
extern {
    fn snappy_max_compressed_length(source_l
}

fn main() {
    let x = unsafe { snappy_max_compressed_l
    println!("максимальный размер сжатого бу
}
```

Блок `extern` содержит список сигнатур функций из внешней библиотеки, в данном случае для C ABI (application binary interface; двоичный интерфейс приложений) данной

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

```
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`:

fn main() { struct HashMap<K, V, S = RandomState> { } }

```
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:

fn main() { use std::collections::HashMap; let mut map = HashMap::new(); map.insert("Foo".to_string(), 42); assert_eq!(map.get("Foo"), Some(&42)); }

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

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.

5.11. Каналы сборок6.1. Связывание имён

Связывание имён

Любая реальная программа на Rust посложнее, чем «Hello World», использует *связывание имён*. Это выглядит так:

fn main() { let x = 5; }

```
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:

fn main() { let (x, y) = (1, 2); }

```
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

Функции

Каждая программа на Rust имеет по крайней мере одну функцию — `main`:

fn 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 main() { fn foo() { } }

```
fn foo() {
}
```

Итак, что насчёт аргументов, принимаемых функцией? Вот функция, печатающая число:

fn main() { fn print_number(x: i32) { println!("x равен: {}", x); } }

```
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); }

```
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

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`:

fn main() { let x = true; let y: bool = 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: (')

fn main() { let x = 'x'; let two_hearts = ''; }

```
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

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*.

fn main() { // 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. }

```
// Строчные комментарии — это всё что угодно

let x = 5; // это тоже строчный комментарий.

// Если у вас длинное объяснение для чего-ли
// комментарии один за другим. Поместите про
// так как это более читаемо.
```

Другое применение комментария — это doc-комментарий. Doc-комментарий использует `///` вместо `//`, и поддерживает Markdown-разметку внутри:

fn main() { /// Прибавляем единицу к заданному числу. /// /// # Examples /// /// ``` /// let five = 5; /// /// assert_eq!(6, add_one(5)); /// ``` fn add_one(x: i32) -> i32 { x + 1 } }

```
/// Прибавляем единицу к заданному числу.
///
/// # Examples
///
/// ```
/// let five = 5;
///
/// assert_eq!(6, add_one(5));
/// ```
```

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:

fn main() { let x = 5; if x == 5 { println!("x равняется пяти!"); } }

```
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`:

fn main() { let x = 5; if x == 5 { println!("x равняется пяти!"); } else { println!("x это не пять :("); } }

```
let x = 5;

if x == 5 {
    println!("x равняется пяти!");
} else {
    println!("x это не пять :(");
}
```

Когда необходимо больше одного выбора,

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:

fn main() { loop { println!("Зациклились!"); } }

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

Циклы `while`

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

fn main() { 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; } } }

```
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.

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

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.

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

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:

fn main() { let x = 5; x = 6; // ошибка! }

```
let x = 5;
x = 6; // ошибка!
```

Изменяемость можно добавить с помощью ключевого слова `mut`:

fn main() { let mut x = 5; x = 6; // нет проблем! }

```
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:

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

```
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:

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

```
let mut x = 5;
let mut y = &mut x;
```

Структуры

Структуры (`struct`) — это один из способов создания более сложных типов данных. Например, если мы рассчитываем что-то с использованием координат 2D пространства, то нам понадобятся оба значения — `x` и `y`:

fn main() { let origin_x = 0; let origin_y = 0; }

```
let origin_x = 0;
let origin_y = 0;
```

Структура позволяет нам объединить эти два значения в один тип с `x` и `y` в качестве имен полей:

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); }

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let origin = Point { x: 0, y: 0 }; // or

    println!("Начало координат находится в (
}
```

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

Перечисления

В Rust *перечисление* (`enum`) — это тип данных, который представляет собой один из нескольких возможных вариантов. Каждый вариант в перечислении может быть также связан с другими данными:

fn main() { enum Message { Quit, ChangeColor(i32, i32, i32), Move { x: i32, y: i32 }, Write(String), } }

```
enum Message {
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 },
    Write(String),
}
```

Синтаксис для объявления вариантов схож с синтаксисом для объявления структур: у вас могут быть варианты без данных (как unit-подобные структуры), варианты с именованными данными и варианты с безымянными данными (подобно кортежным структурам). Варианты перечисления имеют один и тот же тип, и в отличии от структур не являются определением отдельных типов. Значение перечисления может соответствовать любому из вариантов. Из-за этого перечисления иногда называют *тип-сумма* (*sum-type*): множество возможных значений перечисления — это сумма множеств возможных значений каждого варианта.

Мы используем синтаксис `::` чтобы использовать имя каждого из вариантов. Их область видимости ограничена именем самого перечисления. Это позволяет использовать оба варианта из примера ниже совместно:

fn main() { enum Message { Move { x: i32, y: i32 }, } let x: Message = Message::Move { x: 3, y: 4 }; enum BoardGameTurn { Move { squares: i32 }, Pass, } let y: BoardGameTurn = BoardGameTurn::Move { squares: 1 }; }

```
let x: Message = Message::Move { x: 3, y: 4
```

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:

fn main() { let x = 5; match x { 1 => println! ("один"), 2 => println!("два"), 3 => println! ("три"), 4 => println!("четыре"), 5 => println! ("пять"), _ => println!("что-то ещё"), } }

```
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 cove
```

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:

fn main() { let x = 1; match x { 1 => println!("один"), 2 => println!("два"), 3 => println!("три"), _ => println!("что угодно"), } }

```
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 `|`:

fn main() { let x = 1; match x { 1 | 2 => println!("один или два"), 3 => println!("три"), _ => println!("что угодно"), } }

```
let x = 1;

match x {
    1 | 2 => println!("один или два"),
    3 => println!("три"),
    _ => println!("что угодно"),
```

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:

fn main() { baz(bar(foo())); }

```
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?

fn main() { foo.bar().baz(); }

```
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()); }

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius
    }
}

fn main() {
```

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!`:

fn main() { let v = vec![1, 2, 3, 4, 5]; // v: Vec<i32> }

```
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!`:

fn main() { let v = vec![0; 10]; // десять нулей }

```
let v = vec![0; 10]; // десять нулей
```

Доступ к элементам

Чтобы получить значение по определенному индексу в векторе, мы используем `[]`:

fn main() { let v = vec![1, 2, 3, 4, 5]; println! ("Третий элемент вектора v равен {}", v[2]); }

```
let v = vec![1, 2, 3, 4, 5];

println!("Третий элемент вектора v равен {}"
```

Индексы отсчитываются от `0`, так что третьим элементом является `v[2]`.

Обход

Вы можете обойти элементы вектора с

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:

fn main() { let greeting = "Всем привет."; // greeting: &'static str }

```
let greeting = "Всем привет."; // greeting:
```

`"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.

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:

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

```
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:

fn main() { let x: Option<i32> = Some(5); }

```
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`,

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?

#![feature(core)] fn main() { struct Circle { x: f64, y: f64, radius: f64, } impl Circle { fn area(&self) -> f64 { std::f64::consts::PI * (self.radius * self.radius) } } }

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (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:

#![feature(core)] fn main() { 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) } } }

```
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
    }
```

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

```
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 }; }

```
struct Firework {
    strength: i32,
}

impl Drop for Firework {
```

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:

fn main() { let option = Some(5); fn foo(x: i32) { } match option { Some(x) => { foo(x) }, None => {}, } }

```
match option {
    Some(x) => { foo(x) },
    None => {},
}
```

Здесь необязательно использовать `match`. `if` тоже подойдёт:

fn main() { let option = Some(5); fn foo(x: i32) { } if option.is_some() { let x = option.unwrap(); foo(x); } }

```
if option.is_some() {
    let x = option.unwrap();
    foo(x);
}
```

Но оба этих варианта выглядят странно. Мы можем исправить это с помощью `if let`:

fn main() { let option = Some(5); fn foo(x: i32) { } if let Some(x) = option { foo(x); } }

```
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.

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`.

fn main() { trait Foo { fn method(&self) -> String; } }

```
trait Foo {
    fn method(&self) -> String;
}
```

Также мы реализуем этот типаж для `u8` и `String`:

fn main() { trait Foo { fn method(&self) -> String; } impl Foo for u8 { fn method(&self) -> String { format!("u8: {}", *self) } } impl Foo for String { fn method(&self) -> String { format!("string: {}", *self) } } }

```
impl Foo for u8 {
    fn method(&self) -> String { format!("u8
}

impl Foo for String {
    fn method(&self) -> String { format!("st
}
```

Статическая диспетчеризация

Мы можем использовать этот типаж для

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:

fn main() { let plus_one = |x: i32| x + 1; assert_eq!(2, plus_one(1)); }

```
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:

fn main() { let plus_two = |x| { let mut result: i32 = x; result += 1; result += 1; result }; assert_eq!(4, plus_two(2)); }

```
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

Универсальный синтаксис вызова функций (universal function call syntax)

Иногда, функции могут иметь одинаковые имена. Рассмотрим этот код:

fn main() { 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; }

```
trait Foo {
    fn f(&self);
}

trait Bar {
    fn f(&self);
}

struct Baz;

impl Foo for Baz {
    fn f(&self) { println!("Baz’s impl of Fo
}

impl Bar for Baz {
    fn f(&self) { println!("Baz’s impl of Ba
}

let b = Baz;
```

Если мы попытаемся вызвать `b.f()`, то получим ошибку:

```
error: multiple applicable methods in scope
b.f();
  ^~~
note: candidate #1 is defined in an impl of
`main::Baz`
    fn f(&self) { println!("Baz’s impl of Fo
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
note: candidate #2 is defined in an impl of
`main::Baz`
    fn f(&self) { println!("Baz’s impl of Ba
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Нам нужен способ указать, какой конкретно метод нужен, чтобы устранить неоднозначность. Эта возможность называется

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:

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:

`const` и `static`

В Rust можно определить постоянную с помощью ключевого слова `const`:

fn main() { const N: i32 = 5; }

```
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:

fn main() { static N: i32 = 5; }

```
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):

fn main() { static NAME: &'static str = "Steve"; }

Атрибуты

В Rust объявления могут быть аннотированы с помощью «атрибутов». Они выглядят так:

fn main() { #[test] fn foo() {} }

```
#[test]
```

или так:

fn main() { mod foo { #![test] } }

```
#![test]
```

Разница между ними состоит в символе `!`, который изменяет его поведение, определяющее к какому элементу применяется атрибут:

fn main() { #[foo] struct Foo; mod bar { #![bar] } }

```
#[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:

fn main() { #[test] fn check() { assert_eq!(2, 1 + 1); } }

```
#[test]
fn check() {
    assert_eq!(2, 1 + 1);
}
```

Функция помечена как `#[test]`. Это означает, что она особенная: эта функция будет

Псевдонимы типов

Ключевое слово `type` позволяет объявить псевдоним другого типа:

fn main() { type Name = String; }

```
type Name = String;
```

Затем вы можете использовать этот псевдоним вместо реального типа:

fn main() { type Name = String; let x: Name = "Hello".to_string(); }

```
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:

fn main() { let x: i32 = 5; let y: i64 = 5; if x == y { // ... } }

```
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:

fn main() { type Num = i32; let x: i32 = 5; let y: Num = 5; if x == y { // ... } }

```
type Num = i32;

let x: i32 = 5;
```

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:

fn main() { let x: i32 = 5; let y = x as i64; }

```
let x: i32 = 5;

let y = x as i64;
```

Оно допускает только определенные виды приведения типов:

fn main() { let a = [0u8, 0u8, 0u8, 0u8]; let b = a as u32; // four eights makes 32 }

```
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

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:

fn main() { trait Graph<N, E> { fn has_edge(&self, &N, &N) -> bool; fn edges(&self, &N) -> Vec<E>; // etc } }

```
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 main() { fn distance<N, E, G: Graph<N, E>> (graph: &G, start: &N, end: &N) -> u32 { ... } }

```
fn distance<N, E, G: Graph<N, E>>(graph: &G,
```

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:

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:

fn main() { impl Foo for str { }

```
impl Foo for str {
```

или

fn main() { impl<T> Foo for [T] { }

```
impl<T> Foo for [T] {
```

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); }

```
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
    }
}

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

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); }

```
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 main() { fn foo(s: &str) { // позаимствуем строку на секунду } // String реализует Deref<Target=str> let owned = "Hello".to_string(); // Поэтому, такой код работает: foo(&owned); }

```
fn foo(s: &str) {
    // позаимствуем строку на секунду
}
```

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.

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

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:

fn main() { unsafe fn beregis_avtomobilya() { // страшные вещи } }

```
unsafe fn beregis_avtomobilya() {
    // страшные вещи
}
```

Например, все функции, вызываемые через FFI, должны быть помечены как небезопасные. Другое использование `unsafe` — это отметка небезопасного блока:

fn main() { unsafe { // страшные вещи } }

```
unsafe {
    // страшные вещи
}
```

Третье — небезопасные типажи:

fn main() { unsafe trait Scary { } }

```
unsafe trait Scary { }
```

И четвёртое — реализация (`impl`) таких типажей:

fn main() { unsafe trait Scary { } unsafe impl Scary for i32 {} }

```
unsafe impl Scary for i32 {}
```

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:

fn main() { unsafe fn beregis_avtomobilya() { // страшные вещи } }

```
unsafe fn beregis_avtomobilya() {
    // страшные вещи
}
```

Например, все функции, вызываемые через FFI, должны быть помечены как небезопасные. Другое использование `unsafe` — это отметка небезопасного блока:

fn main() { unsafe { // страшные вещи } }

```
unsafe {
    // страшные вещи
}
```

Третье — небезопасные типажи:

fn main() { unsafe trait Scary { } }

```
unsafe trait Scary { }
```

И четвёртое — реализация (`impl`) таких типажей:

fn main() { unsafe trait Scary { } unsafe impl Scary for i32 {} }

```
unsafe impl Scary for i32 {}
```

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/rustu
```

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/ru
$ 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,

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

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

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:

fn main() { asm!(assembly template : output operands : input operands : clobbers : options); }

```
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(); // ... }

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] // fn main() {} tricked you, rustdoc!

```
// 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 {} } // fn main() {} tricked you, rustdoc!

```
#![feature(lang_items, start, no_std, libc)]
#![no_std]

// Pull in the system libc library for what
extern crate libc;
```

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

#![feature(intrinsics)] fn main() {} extern "rust-intrinsic" { fn transmute<T, U>(x: T) -> U; fn offset<T>(dst: *const T, offset: isize) -> *const T; }

```
extern "rust-intrinsic" {
    fn transmute<T, U>(x: T) -> U;

    fn offset<T>(dst: *const T, offset: isiz
}
```

Как и с любыми другими FFI функциями, их вызов всегда небезопасен и помечен как `unsafe`.

7.3. Без stdlib7.5. Элементы языка (lang items)

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 {} }

```
#![feature(lang_items, box_syntax, start, nc
```

Продвинутое руководство 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 {} fn main() {}

```
#![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

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)] fn main() { 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)); } } }

```
#![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`

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!() } }

```
#![feature(box_syntax, box_patterns)]

fn main() {
    let b = Some(box 5);
    match b {
        Some(box n) if n < 0 => {
            println!("Box contains negative
        },
        Some(box n) if n >= 0 => {
            println!("Box contains non-negat
        },
        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

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), _ => {}, } }

```
#![feature(slice_patterns)]

fn main() {
    let v = vec!["match_this", "1"];

    match &v[..] {
        ["match_this", second] => println!("
        _ => {},
    }
}
```

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)); }

```
#![feature(advanced_slice_patterns, slice_pa

fn is_symmetric(list: &[u32]) -> bool {
    match list {
        [] | [_] => true,
        [x, inside.., y] if x == y => is_sym
        _ => false
    }
}

fn main() {
```

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); }

```
#![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)] fn main() { trait Foo { const ID: i32; } impl Foo for i32 { } }

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32;
}

impl Foo for i32 {
}
```

выдаст ошибку

```
error: not all trait items implemented, miss
     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); }

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

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.

fn main() { let x = (2, 3); let y = (4, 6); let z = (8, 2, 6); }

```
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

Библиография

Это — список материалов, имеющих отношние к Rust. Он включает в себя предварительные исследования, которые в тот или иной момент оказали влияние на структуру Rust'a, а также публикации о 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