Основы синтаксиса Rust

Андрей Ситников

21-09-2023

Очень краткий пересказ вводной лекции

Очень краткий пересказ вводной лекции

Rust — круто.

Развёрнутый пересказ вводной лекции

Rust — венец творения дизайна языков программирования: вобрал в себя лучшее из опыта разработки ПО всего человечества.

Развёрнутый пересказ вводной лекции

Rust:

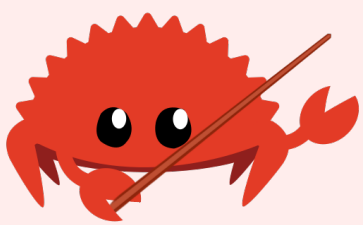
- Быстрый (до 300+ раз относительно кода на питоне)
- Безопасный (в отличие от плюсов, $\sim$ всегда будет давать ожидаемый результат)
- Компилируется $\rightarrow$ работает
- Экосистема: много библиотек и инструментов
- Работает в браузерах и голых железках

Часть I

Ключевые выражения, мутабельность, макросы.

Часть I

Ключевые выражения, мутабельность, макросы.



Поначалу будет много *магии*!

`cargo init` создаст файл `main.rs` следующего содержания:

```
1 fn main() {
2   println!("Hello, world!");
3 }
```

`cargo init` создаст файл `main.rs` следующего содержания:

```
1 fn main() {
2   println!("Hello, world!");
3 }
```

Это — функция

`cargo init` создаст файл `main.rs` следующего содержания:

```
1 fn main() {
2   println!("Hello, world!");
3 }
```

А это — магия!

Восклицательный знак — дело нечисто.

Примитивы для примеров

- `bool` — boolean, `true/false`

Примитивы для примеров

- `bool` — boolean, `true/false`
- `i32` — 32-битный integer, от -2147483648 до 2147483647.

Примитивы для примеров

- `bool` — boolean, `true/false`
- `i32` — 32-битный integer, от -2147483648 до 2147483647.
- `u32` — 32-битный *unsigned* integer, от 0 до 4294967295.

let

```
1 let x = 3;
```

let

```
1 let x: i32 = 3;
```

let

```
1 let x = 3;
2 let y: u32 = 2;
3 y = x;
```

Оно компилируется.

Что здесь происходит?

`let`

```
1 let x: u32 = 3;
2 let y: u32 = 2;
3 y = x;
```

В **Rust** есть вывод типов!

Все типы *статически заданы*, но писать типы явно обязательно далеко не всегда

`mut`

По умолчанию переменные *неизменяемые* (для **Rust** это важно!).

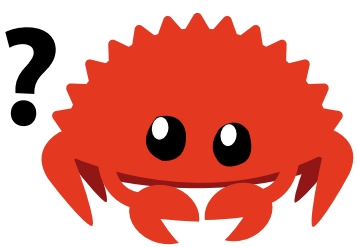
```
1 let mut x = 3;
2 x += 1;
```

```
1 let x;
2 x = 3;
```

Можно сначала объявить переменную, потом присвоить значение!

```
1 let x: i64;
2 x += 1;
```

Что если *не присвоить* значение и попробовать воспользоваться?



```
1 let x: i64;
2 x += 1;
```

```
     error[E0381]: used binding `x` isn't initialized
 --> src/main.rs:9:5
  |
8 |     let x: i64;
  |         - binding declared here but left uninitialized
9 |     x += 1;
  |     ^^^^^^ `x` used here but it isn't initialized
  |
help: consider assigning a value
  |
8 |     let x: i64 = 0;
  |                ++++
```

(в C++ это Undefined Behaviour)

Область видимости (≈ время жизни)

```
{
  // x создался здесь
  let x = 0;

  x += 1;

  // Здесь Rust автоматически "уничтожит" x.
}

println!("{x}"); // Ошибка компиляции, x уже будет уничтожен.
```

В **Rust** можно *переиспользовать* переменные под новые типы:

```
1 let x: u32 = 5;
2 let x: i32 = x - 10;
```

Условный оператор

```
1 if <условие> {
2   <код>
3 }
4 else if <второе условие> {
5   <код>
6 }
7 else { // (опционально)
8   <код>
9 }
```

Операции с `bool`

Стандартные

- not: `!`
- and: `a && b` (битовое, оно же *не* ленивое — `&`)
- or: `a || b` (битовое, оно же *не* ленивое — `|`)
- xor: `^`
- `==`, `>`, `<`

Выражения в Rust

`if` возвращает значение в скобках!

```
1 let x = if 3 > 2 {5} else {6}
```

Выражения в Rust

`if` возвращает значение в скобках!

```
let x = if 3 > 2 {5} else {6}
```

```
let x = if 3 > 2 {
  println!("Hello world!"); // ← ничего не возвращает!
  5 // ← то же самое, что и return 5;
} else {
  6 // Возвращаемые типы должны быть одинаковыми, иначе компилятор будет зол.
}
```

Точка с запятой *съедает* возвращаемое значение.

Циклы

Циклов в **Rust** есть три вида:

- `for i in <что-то итерируемое> {<тело цикла>}`
- `while <условие> {<тело цикла>}`
- `loop {<тело цикла>}` — ∼ то же самое, что и `while true`.

`break` и `continue` в комплект входят.

По чему можно итерироваться в `for`? Проще всего — по `range`. В **Rust** есть специальный синтаксис для них:

- `0..n` — от нуля включительно до n *не включительно*.

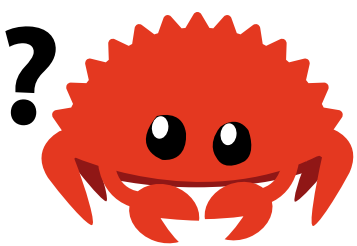
По чему можно итерироваться в `for`? Проще всего — по `range`. В **Rust** есть специальный синтаксис для них:

- `0..n` — от нуля включительно до n *не включительно*.
- `0..=n` — от нуля включительно до n *включительно*

Инициализация внутри цикла

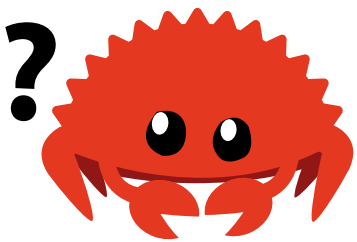
```
let mut x;

for i in 0..3 {
    if i == 1 {
        x += 1;
    }
    else {
        x = 3;
    }
}
```



Инициализация внутри цикла

```
error[E0381]: used binding `x` is possibly-uninitialized
-→ src/main.rs:15:13
   |
11 |     let x;
   |         - binding declared here but left uninitialized
12 |
13 |     for i in 0..3 {
   |              ——— if the `for` loop runs 0 times, `x` is not initialized
14 |         if i == 1 {
15 |             x += 1;
   |             ^^^^^^ `x` used here but it is possibly-uninitialized
```



Анализ **Rust** не всегда идеальный, но вполне понятный. В общем случае невозможно проанализировать всё.

```rust
fn get_num() -> u32 {
  5
}

fn main() {
    let x: u32 = 3;

    /*
    error: this arithmetic operation will overflow
 --> src/main.rs:7:18
  |
7 |     let y: u32 = x - 5;
  |                  ^^^^^ attempt to compute `3_u32 - 5_u32`, which would overflow
  |
  = note: `#[deny(arithmetic_overflow)]` on by default
  */
    let y: u32 = x - 5;

    //thread 'main' panicked at 'attempt to subtract with overflow', src/main.rs:8:17
    let y: u32 = x - get_num();
  }
```

Сигнатуры функций

Функции принимают значения и что-то дают на выходе. То, что делает функция, определяется её сигнатурой — нужно, чтобы программист её указал сам, исходя из неё компилятор и будет выводить типы.

```
fn f(x: u32, y: bool) → i32 {
  if y {
    x as i32 // работает так же, как в let, return не обязателен
  }
  else{
    0
  }
}
```

Сигнатуры функций

Функции принимают значения и что-то дают на выходе. То, что делает функция, определяется её сигнатурой — нужно, чтобы программист её указал сам, исходя из неё компилятор и будет выводить типы.

```
fn f(x: u32, y: bool) → i32 {
  if y {
    x as i32 // работает так же, как в let, return не обязателен
  }
  else{
    0
  }
}
```

Важный момент: функции принимают *фиксированное* количество аргументов

Макросы

Макросы — правила, которые *во время компиляции* раскрываются в обычный код.

Важные макросы:

- `println!("x = {x}, y = {}", y)`: первый аргумент — строка для форматирования, остальные — переменные для подстановки.

Макросы

Макросы — правила, которые *во время компиляции* раскрываются в обычный код.

Важные макросы:

- `println!("x = {x}, y = {}", y)`: первый аргумент — строка для форматирования, остальные — переменные для подстановки.
- `dbg!(x, y, z)`: печатает строчку, на которой вызовется, и имя и значение каждой переменной.

Макросы

Макросы — правила, которые *во время компиляции* раскрываются в обычный код.

Важные макросы:

- `println!("x = {x}, y = {}", y)`: первый аргумент — строка для форматирования, остальные — переменные для подстановки.
- `dbg!(x, y, z)`: печатает строчку, на которой вызовется, и имя и значение каждой переменной.
- `panic!("Ошибка")`: прерывает выполнение программы с сообщением об ошибке.

Макросы

Макросы — правила, которые *во время компиляции* раскрываются в обычный код.

Важные макросы:

- `println!("x = {x}, y = {}", y)`: первый аргумент — строка для форматирования, остальные — переменные для подстановки.
- `dbg!(x, y, z)`: печатает строчку, на которой вызовется, и имя и значение каждой переменной.
- `panic!("Ошибка")`: прерывает выполнение программы с сообщением об ошибке.
- `assert!()` — паникует, если неправда. `assert_eq!()` паникует, если значения не совпадают, и печатает оба.

Часть II

Примитивы. Составные типы. Сложные объекты.

Примитивы

- знаковые целочисленные: `i8`, `i16`, `i32`, `i64` и `isize` (pointer-sized)
- беззнаковые целочисленные: `u8`, `u16`, `u32`, `u64` и `usize` (pointer-sized)
- вещественные: `f32`, `f64`
- `char`: «Unicode scalar value», например: `'a'`, `'α'` и `'∞'` (железно 4 байта каждый)
- `bool`: `true` или `false`
- единичный тип: `()` — не занимает места вообще

- Десятичный: `98_222`
- Шестнадцатеричный: `0xff`
- Восьмеричный: `0o77`
- Двоичный: `0b1111_0000`
- Байт (только u8): `b'A'`

- Десятичный: `98_222`
- Шестнадцатеричный: `0×ff`
- Восьмеричный: `0o77`
- Двоичный: `0b1111_0000`
- Байт (только u8): `b'A'`

Можно указывать тип числа напрямую:

```
1 let x = 5u32; // x будет типа u32
```

`as`

Ключевое слово `as` превращает один тип в другой. Если не получается — использует насыщение для знаков, обрезку для понижения байтов целых и приближение для `float`:

```
1 let x = 5u32 as f32;
2 assert_eq!(x, 5.0);
3
4 let x = -5f32 as u32;
5 assert_eq!(x, 0);
6
7 let x = 1000 as u8;
8 assert_eq(x, 232);
```

as

Ключевое слово `as` превращает один тип в другой. Если не получается — использует насыщение для знаков, обрезку для понижения байтов целых и приближение для `float`:

```
let x = 5u32 as f32;
assert_eq!(x, 5.0);

let x = -5f32 as u32;
assert_eq!(x, 0);

let x = 1000 as u8;
assert_eq(x, 232);
```

На самом деле ещё есть `.into()` и `.try_into()` или `.from()` и `.try_from()`.

`try_into` возвращает `Result` с возможной ошибкой преобразования.

Составные типы

- Tuples. (T_1, T_2, T_3). К каждому можно обращаться отдельно.

```rust
let x = (5, true);
// x имеет тип (usize, bool)
assert_eq!(x.0, 5);

let (y, z) = x;
assert_eq!(z, true);

println!("{x}"); // Ошибка компиляции, `x` уже разобрали!
```

Составные типы

- Tuples. (T_1, T_2, T_3). К каждому можно обращаться отдельно.

```rust
let x = (5, true);
// x имеет тип (usize, bool)
assert_eq!(x.0, 5);

let (y, z) = x;
assert_eq!(z, true);

println!("{x}"); // Ошибка компиляции, `x` уже разобрали!
```

- Arrays. `[T;n]`.

```rust
let y = [true;5];
```

Arrays

```
1  use std::mem;
2
3  fn main() {
4      // Fixed-size array (type signature is superfluous).
5      let xs: [i32; 5] = [1, 2, 3, 4, 5];
6
7      // All elements can be initialized to the same value.
8      let ys: [i32; 500] = [0; 500];
9
10     // Indexing starts at 0.
11     assert_eq!(xs[0], 1);
12
13     // `len` returns the count of elements in the array.
14     assert_eq!(xs.len(), 5);
15
16     // Arrays are stack allocated.
17     // 32 = 4 bytes, 4×5 = 20
18     assert_eq!(mem::size_of_val(&xs), 20);
19 }
```

Non-copy объекты

Все объекты выше были `Copy`, то есть когда мы пишем

```
1 let x = …;
2 let y = x;
```

Значение в `y` становится *в точности* равным значению x.

Так логично делать не всегда.

Вектор

Объекты, которые нельзя копировать, могут находиться только в одном месте:

```
1 let a = Vec::new();
2 let b = a; // Теперь вектор лежит в b
3 println!("{}", a.len()); // Ошибка компиляции, значение из a уже использовано!
```

Можно вызвать методы структуры через `::`. Если есть конкретный объект, поля и методы вызываются через точку.

```
let x = Vec::new()
x.push(3);
assert_eq!(x[1].pow(3), 27);
let y = x.clone(); // Копирует все объекты внутри
```

Конструкторы *как правило* называются через `::new()`. Иногда они могут иметь вид `::from_<smth>()`, `::try_new` и так далее, может их и не быть вообще.

Ссылки

Иногда хочется использовать структуру больше, чем в одном месте.

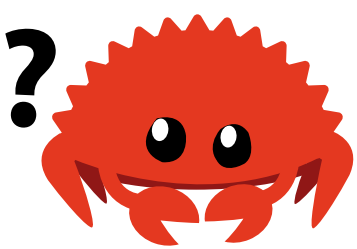
```
1 let x = vec![1, 2, 3];
2 let y = &x; // ← это ссылка на чтение для x
3 println!("{}", y.len()); // Метод взятия длины работает по ссылке на чтение!
4 y[0] += 1; // А вот изменить число по этой ссылке не выйдет :(
```

Мутабельные ссылки

```rust
let x = vec![1, 2, 3];
let y = &mut x; // ← это ссылка на редактирование
println!("{}", y.len()); // Метод взятия длины работает по изменяемой ссылке
*y[0] += 3; // И изменять можно!
// * отвечает за разыменовывание, взятие объекта по ссылке
```

Пример из вводной лекции

```
1 let mut nums = vec![1, 2, 3];
2 let r = &nums[0];
3 nums.push(5);
4 println!("{r}");
```

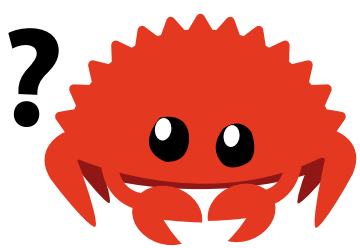


Что здесь было не так?

Пример из вводной лекции

```
1 let mut nums = vec![1, 2, 3];
2 let r = &nums[0];
3 nums.push(5);
4 println!("{r}");
```

Изменение объекта
делало ссылку *некорректной*



Что здесь было не так?

Обобщение

```
1 let x = 3;
2 let y = ComplexThing::new(&mut x); // ComplexThing как-то обработало ссылку и
  оставило её в себе, чтобы менять x по мере изменения y
3 x += 3;
4 println!("{y}"); // ???
```

Как должен себя вести объект? Ловить как-то изменение ссылки?

Пытаться его обработать и привести в соответствие со старыми данными?

Обобщение

```
1 let x = 3;
2 let y = ComplexThing::new(&mut x); // ComplexThing как-то обработало ссылку и
  оставило её в себе, чтобы менять x по мере изменения y
3 x += 3;
4 println!("{y}"); // ???
```

Как должен себя вести объект? Ловить как-то изменение ссылки?

Пытаться его обработать и привести в соответствие со старыми данными?

Надо *запретить* мутирование и чтение одновременно!

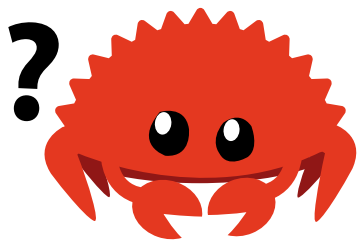
Borrow-rules (кратко)

- Каждый объект привязан к переменной, которая им «владеет».
- Владение можно передавать.
- На объект можно создавать ссылки на чтение и изменение.

- Нельзя редактировать объект, если для него активна ссылка.
- Нельзя иметь ссылку на изменение и другую ссылку (на чтение/изменение) *одновременно*.
- Ссылок на чтение можно иметь сколько угодно, пока объект не меняется.

Попытки нарушить эти правила ловятся *во время компиляции*.

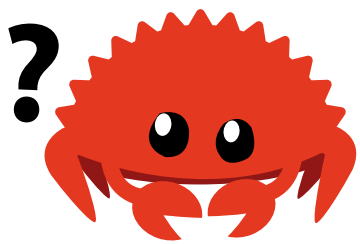
```
1 let y;
2 {
3   let x = 3;
4   y = &x;
5 }
6
7 println!("{y}");
```



Скомпилируется?

Создается ссылка, потом читается вне «области видимости» **x**.

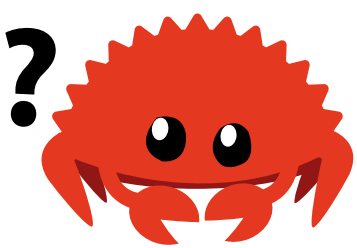
```
1 let y;
2 {
3     let x = 3;
4     y = &x;
5 }
6
7 println!("{y}");
```



Скомпилируется?

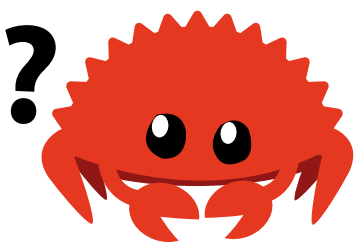
Здесь **x** разрушается — то есть
меняется, когда на него есть ссылка,
а это *запрещено*.

```rust
fn get_ref() → &i32 {
    let x = 3;
    &x
}
```



Корректная ли функция?

```rust
// Ошибка компиляции
fn get_ref() → &i32 {
    let x = 3;
    &x
    // Здесь x уничтожается.
    // Возвращать нельзя!
}
```



А вообще можно вернуть ссылку из функции?

```rust
fn get_ref() → &str {
    // Эта строчка — ссылка на данные,
    // Хранящиеся всё время исполнения
    "My string"
}
```

На самом деле да, если мы возвращаем ссылку на объект, который «живёт» все время выполнения программы

Ссылки и времена жизни

К моменту, когда объект выходит из своего `scope`, он *удаляется*. Это — редактирование (на самом деле даже потребление) объекта.

Ссылки невозможно использовать за пределами *времени жизни* (в нашем случае — области видимости) объекта.

Slices

Бывают и более интересные ссылки. Например, `slice`-ы Array/Vec-ов или строк.

Это ссылка, хранящая в себе *начало* и *длину* выбранного диапазона.

```rust
// This function borrows a slice.
fn analyze_slice(slice: &[i32]) -> (i32, usize) {
    // Returs first element of the slice and its length.
    (slice[0], slice.len())
}

fn main() {
  let xs: [i32; 5] = [1, 2, 3, 4, 5];

  // Arrays can be automatically borrowed as slices.
  analyze_slice(&xs);

  // Slices can point to a section of an array.
  // They are of the form [starting_index..ending_index].
  // `starting_index` is the first position in the slice.
  // `ending_index` is one more than the last position in the slice.
  analyze_slice(&xs[1 .. 4]);
}
```

```rust
fn main() {
  let xs: [i32; 5] = [1, 2, 3, 4, 5];
  // Example of empty slice `&[]`:
  let empty_array: [u32; 0] = [];
  assert_eq!(&empty_array, &[]);
  assert_eq!(&empty_array, &[][..]); // Same but more verbose

  // Out of bound indexing causes compile time error.
  println!("{}", xs[5]); // ОШИБКА!
}
```

`mut` slice

Та же ссылка на «диапазон» в массиве, но с правом изменения.

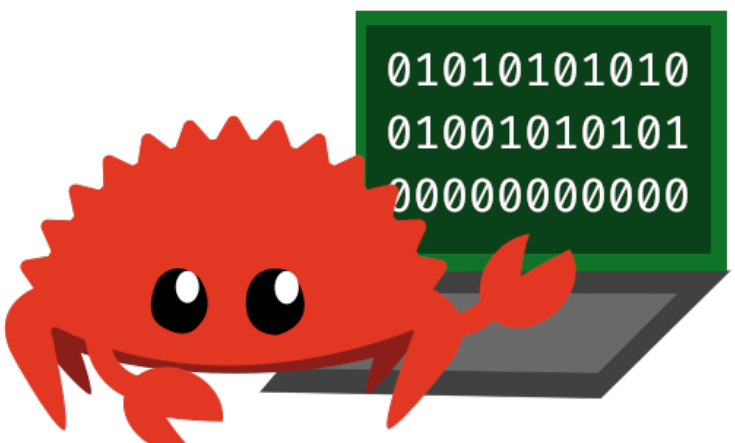
```
1 let mut x = [1, 2, 3];
2 let x = &mut x[..]; // Take a full slice of `x`.
3 x[1] = 7;
4 assert_eq!(x, &[1, 7, 3]);
```

Универсальная шпаргалка

```
$rustup doc
```

Books:

- **Rust API**
- **The Rust programming language**
- **Rust by Example**


01010101010
01001010101
00000000000