Rust. Основы. Часть 1

Нгуен Тхи Хань Хуен
Королева Дарья

Типы данных: примитивные

- `i32`, `i64` — знаковые числа,
- `u32`, `u64` — беззнаковые.
- `f32`, `f64` — ч**исла с плавающей точкой**.
- `true`, `false` — **булевы значения**.
- **Символы: char**

```
1    let a: i32 = -5;
2    let b: u32 = 10;
3
4    let pi: f64 = 3.1415;
5
6    let flag: bool = true;
7
8    let letter: char = 'A';
```

Типы данных: пользовательские

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

let p = Point { x: 3, y: 4 };
```

```
enum Shape {
    Circle(f64),
    Rectangle(f64, f64),
}

let c = Shape::Circle(2.5);
```

Операции над типами

```
let x = 5 + 3;    // сложение
let y = 10 - 2;   // вычитание
let z = 4 * 2;    // умножение
let t = 7 % 3;    // остаток от деления

let cond = true && false; // логическое И
let cond2 = true || false; // логическое ИЛИ
```

Преобразование и приведение типов

Rust требует **явного приведения типов**.

```rust
let a: i32 = 10;
let b: f64 = a as f64 + 0.5;
```

```rust
struct Meters(u32);
struct Kilometers(u32);

impl From<Meters> for Kilometers {
    fn from(m: Meters) -> Self {
        Kilometers(m.0 / 1000)
    }
}

let m = Meters(1500);
let km: Kilometers = Kilometers::from(m);
let km: Kilometers = m.into();
```

Сравнение типов и значений (1/2)

Rust умеет сравнивать значения стандартными операторами: ==, !=, <, >, <=, >=

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

println!("{}", x < y);  // true
println!("{}", x == y); // false
```

Для пользовательских типов можно автоматически добавить сравнение

```rust
#[derive(PartialEq, Eq, PartialOrd, Ord)]
struct Point {
    x: i32,
    y: i32,
}

let p1 = Point { x: 1, y: 2 };
let p2 = Point { x: 2, y: 1 };

println!("{}", p1 < p2);  // true
```

Сравнение пользовательских объектов (2/2)

```rust
use std::cmp::Ordering;

impl PartialEq for Point {
    fn eq(&self, other: &Self) -> bool {
        self.x == other.x && self.y == other.y
    }
}

impl Eq for Point {}

impl PartialOrd for Point {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.x.cmp(&other.x)) // сравнение только по X
    }
}
```

Области видимости

```rust
fn main() {
    let x = 5;
    {
        let y = 10;
        println!("{}", x); // доступно
    }
    // println!("{}", y); // ошибка: y вне области
}
```

Владение

```rust
fn main() {
    let s = String::from("Hello");
    let t = s; // владение передано в t
    // println!("{}", s); // ошибка: s больше не владеет строкой
}
```

Заимствование

```rust
fn main() {
    let mut s = String::from("Hi");
    let r1 = &s;           // можно читать
    let r2 = &mut s;       // можно изменять
    println!("{}", r1);    // ошибка: нельзя совмещать
}
```

Передача владения в функции

```rust
fn takes_ownership(s: String) { // будет сделана копия
    println!("{}", s);
}

fn borrow_string(s: &String) { // передача по ссылке
    println!("{}", s);
}

fn main() {
    let s = String::from("Rust");
    takes_ownership(s);
    borrow_string(&s);
}
```

Изменяемые ссылки

```rust
fn append_world(s: &mut String) {
    s.push_str(" world!");
}

fn main() {
    let mut s = String::from("Hello");
    append_world(&mut s); // передаём изменяемую ссылку
    println!("{}", s); // Hello world!
}
```

Возвращаемые значения

```rust
fn square(x: i32) -> i32 {
    x * x
}

fn main() {
    let result = square(6);
    println!("Квадрат = {}", result);
}
```

Рекурсия

```rust
fn factorial(n: u32) -> u32 {
    if n == 0 { 1 } else { n * factorial(n - 1) }
}

fn main() {
    println!("5! = {}", factorial(5)); // 120
}
```

Замыкания

```rust
fn main() {
    let factor = 10;
    let multiply = |x: i32| x * factor;
    println!("{}", multiply(3)); // 30
}
```

Функции высшего порядка

```rust
fn apply<F>(f: F, x: i32) -> i32
where F: Fn(i32) -> i32 {
    f(x)
}

fn main() {
    let double = |x| x * 2;
    println!("{}", apply(double, 5)); // 10
}
```