Peter the Great
Saint-Petersburg Polytechnic University

Язык программирования Rust. Данные и переменные.

Исполнитель: Команда №2
Казакевич Анна, Лапина Ольга, Марченко Елизавета

POLYTECH

План презентации

- Объявление переменных
  - Переменные и изменяемость
  - Константы
- Типы данных
  - Примитивные типы
  - Пользовательские типы
- Операторы
  - Сравнение типов
  - Операции над типами
- Преобразование/приведение типов
- Области видимости
- Владение и передача владения

Объявление переменных. Переменные и изменяемость

Переменные в Rust по умолчанию неизменяемы. Чтобы создать изменяемую переменную, используется ключевое слово **mut**.

Неправильно

```
▶ Run | Debug
fn main() {
    let x: i32 = 5;
    //let mut x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

Правильно

```
▶ Run | Debug
fn main() {
    //let x = 5;
    let mut x: i32 = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

Объявление переменных. Константы

- Правила объявления констант:
  1. **Нельзя** использовать mut с константами.
  2. Тип значения *должен быть* указан в аннотации.
  3. Соглашение Rust для именования констант требует использования всех заглавных букв с подчёркиванием между словами.

- Константы можно объявлять в любой области видимости, включая глобальную, благодаря этому они полезны для значений, которые нужны во многих частях кода.
- Константы существуют в течение всего времени работы программы в пределах области, в которой они были объявлены.

```
1   //Объявление переменных
2   const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
3
```

Типы данных.

Rust является ***статически типизированным*** (statically typed) языком. Это означает, что он должен **знать типы всех переменных** во время компиляции.

Обычно компилятор может предположить, какой тип используется (вывести его), основываясь на значении и на том, как мы с ним работаем. В случаях, когда может быть выведено несколько типов, необходимо добавлять аннотацию типа вручную. Например, когда мы конвертировали **String** в число с помощью вызова **parse**.

Типы данных. Примитивные типы (1/3)

Rust предоставляет основные примитивные типы, такие как:

Целые числа:	i8, i16, i32, i64, i128
Беззнаковые варианты целых чисел:	u8, u16, u32, u64, u128
Числа с плавающей точкой:	f32, f64
Булевы значения:	bool (значения true и false)
Символы:	char, представляющий один Unicode символ

Типы данных. Примитивные (скалярные) типы (2/3)

Если вы не уверены какой тип использовать, то типы, принятые в Rust по умолчанию, достаточно хороши.

Целочисленные типы: имеют тип **i32** — обычно он самый быстрый, даже в 64-битных системах.

Числа с плавающей точкой: по умолчанию используется тип **f64**, потому что в современных процессорах он имеет примерно такую же скорость, как и f32, но при этом обладает большей точностью.

Типы данных. Примитивные (скалярные) типы (3/3)

Пример объявления переменных примитивного типа:

```rust
// Примитивные типы
▶ Run | Debug
fn main() {
    let x: i32 = 42;          // 32-битное целое число
    let pi: f64 = 3.1415;     // 64-битное число с плавающей запятой
    let is_active: bool = true;  // Логический тип
    let letter: char = 'A';    // Один символ
    println!(" x = {x}\n pi = {pi}\n is_active = {is_active}\n letter= {letter}\n");

    //Пример, когда нужна анотация типа
    let guess: u32 = "42".parse().expect(msg: "Not a number!");
    //let guess = "42".parse().expect("Not a number!");
    println!(" guess {guess}");
}
```

Типы данных. Пользовательские (составные) типы (1/2)

Rust позволяет создавать собственные типы данных, такие как структуры и перечисления.

- Структуры (struct):

  Структуры используются для группировки данных.

```rust
// Пользовательские типы. Структуры
0 implementations
struct Rectangle {
    width: u32,
    height: u32,
}

▶ Run | Debug
fn main() {
    let rect: Rectangle = Rectangle { width: 30, height: 50 };
    println!("Rectangle: {} x {}", rect.width, rect.height);
}
```

Типы данных. Пользовательские (составные) типы (2/2)

- Перечисления (enum):

  Перечисления позволяют создать тип, который может принимать несколько различных значений.

```rust
// Польсовательские типы. Перечисления
0 implementations
enum Direction {
    Up,
    Down,
    Left,
    Right,
}

▶ Run | Debug
fn main() {
    let dir: Direction = Direction::Up;
    match dir {
        Direction::Up => println!("Going up"),
        Direction::Down => println!("Going down"),
        Direction::Left => println!("Going left"),
        Direction::Right => println!("Going right"),
    }
}
```

Операторы. Сравнение типов

Для сравнения типов в Rust используются операторы:

- **==** - равенства
- **!=** - не равно
- **<** - меньше
- **>** - больше
- **<=** - меньше или равно
- **>=** - больше или равно

```rust
//Операторы
▶ Run | Debug
fn main() {
    // Сравнение типов
    let a: i32 = 1;
    let b: i32 = 2;

    let c: bool = a == b; // false
    let d: bool = a != b; // true
    let e: bool = a < b; // true
    let f: bool = a > b; // false
    let g: bool = a <= a; // true
    let h: bool = a >= a; // true

    let i: bool = true > false; // true
    let j: bool = 'a' > 'A'; // true

    print!(" c = {}\n d = {}\n e = {}\n f = {}\n g = {}\n h = {}\n i = {}\n j = {}",
            c, d, e, f, g, h, i, j);
```

Операторы. Операции над типами (1/2)

Rust поддерживает стандартные арифметические и логические операции:

- **Арифметические:** +, –, *, /, %
- **Логические:** &&, | |, !

```rust
    // Числовые операции:
    let sum: i32 = 5 + 10;

    let difference: f64 = 95.5 - 4.3;

    let product: i32 = 4 * 30;

    let quotient: f64 = 56.7 / 32.2;
    let truncated: i32 = -5 / 3; // Results in -1

    let remainder: i32 = 43 % 5;

    println!("\n addition={}\n substraction={}\n product={}\n quotient={}\n truncated={}\n remainder={}",
            sum, difference, product, quotient, truncated, remainder);


    //Логические операции:
    let x: bool = true;
    let y: bool = false;
    let mut result: bool = !x;  // result = false
    println!("\nresult !x = {}", result);
    result = !(5>3);
    println!("result !(5>3) = {}", result);

    result = x && y;  // result = false
    println!("result x && y = {}", result);
    result = 10 > 2 && 4 < 5;  // result = true (10> 2 равно true и 4 < 5 равно true)
    println!("result 10 > 2 && 4 < 5 = {}", result);

    result = x || y;  // result = true
    println!("result x || y = {}", result);
    result = 2 > 10 || 5 < 4;  // result = false (2 > 10 равно false и 5 < 4 равно false)
    println!("result 2 > 10 || 5 < 4 = {}", result);
} fn main
```

Операторы. Операции над типами (2/2)

А также:

- **Поразрядные:**
  - <<, >> - сдвига
  - <<=, >>= - присваивания и сдвига
  - & - конъюнкция
  - | - дизъюнкция
  - ^ - исключающее ИЛИ

```rust
//Поразрядные
//сдвига
let a: i32 = 2 << 2;          // 10  на два разрядов влево = 1000 - 8
let b: i32 = 16 >> 3;         // 10000 на три разряда вправо = 10 - 2

println!("a = {a}, b = {b}");

//сдвиг и присваивание
let mut x: i32 = 8;
println!("x = {}", x);

x <<= 2;  // 8 в двоичной системе 1000,
        // после сдвига на 2 разряда вправо 10000 или 32 в десятичной системе
println!("x = {}", x);  // 32

x >>= 3;  // 32 в двоичной системе 100000,
        // после сдвига на 3 разряда вправо 100 или 4 в десятичной системе
println!("x = {}", x);  // 4

//конъюнкция, дизъюнкция исключающее ИЛИ
let d: i32 = 5 | 2;          // 101 | 010 = 111  - 7
let e: i32 = 6 & 2;          // 110 & 010 = 10  - 2
let f: i32 = 5 ^ 2;          // 101 ^ 010 = 111 - 7

println!("d = {d}, e = {e}, f = {f}");
} fn main
```

Преобразование/приведение типов

В Rust отсутствует автоматическое преобразование типов, чтобы избежать потери данных. Преобразование происходит явно с помощью оператора **as** или с помощью вызова функции **into()**.

Неверно

```
117  // Преобразование/приведение типов
     ▶ Run | Debug
118  fn main() {
119      let x: i32 = 42;
120      let y: f64 = x;  // Неправильно
121      //let y: f64 = x as f64;  // Преобразование i32 в f64
122      //let y: f64 = x.into();  // Преобразование i32 в f64
123      println!("x as f64: {}", y);
124  }
```

Верно

```
117  // Преобразование/приведение типов
     ▶ Run | Debug
118  fn main() {
119      let x: i32 = 42;
120      //let y: f64 = x;  // Неправильно
121      let y: f64 = x as f64;  // Преобразование i32 в f64
122      //let y: f64 = x.into();  // Преобразование i32 в f6
123      println!("x as f64: {}", y);
124  }
```

Верно

```
117  // Преобразование/приведение типов
     ▶ Run | Debug
118  fn main() {
119      let x: i32 = 42;
120      //let y: f64 = x;  // Неправильно
121      //let y: f64 = x as f64;  // Преобразование i32 в f64
122      let y: f64 = x.into();  // Преобразование i32 в f64
123      println!("x as f64: {}", y);
124  }
```

Области видимости (1/2)

Переменные имеют область видимости в пределах блока, где они объявлены. Блоки создаются с помощью фигурных скобок.

```
127  // Области видимости
128  const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
129
     ▶ Run | Debug
130  fn main() {
131      let x: i32 = 5;
132      println!("x={}", x);
133
134      {
135          const SIZE: u32 = 60;
136          let y: i32 = 10;
137          println!("Inside block: {}", y);  // y доступна внутри блока
138          println!("const SIZE in Inside block = {}", SIZE);  // SIZE доступна внутри блока
139          println!("x in Inside block = {}", x); // доступна так как этот блок находится внутри основного
140      }
141
142      println!("const THREE_HOURS_IN_SECONDS = {}", THREE_HOURS_IN_SECONDS);  // верно, так как константа объявлена в глобалтной области видимости
143
144      println!("const SIZE = {}", SIZE);  // Ошибка: константа вне области видимости
145      println!("{}", y);  // Ошибка: y вне области видимости
146  }
147
```

Неправильно

Области видимости (2/2)

Переменные имеют область видимости в пределах блока, где они объявлены. Блоки создаются с помощью фигурных скобок.

```rust
// Области видимости
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;

▶ Run | Debug
fn main() {
    let x: i32 = 5;
    println!("x={}", x);

    {
        const SIZE: u32 = 60;
        let y: i32 = 10;
        println!("Inside block: {}", y);  // y доступна внутри блока
        println!("const SIZE in Inside block = {}", SIZE);  // SIZE доступна внутри блока
        println!("x in Inside block = {}", x); // доступна так как этот блок находится внутри основного
    }

    println!("const THREE_HOURS_IN_SECONDS = {}", THREE_HOURS_IN_SECONDS);  // верно, так как константа объявлена в глобалтной области видимости

    // println!("const SIZE = {}", SIZE);  // Ошибка: константа вне области видимости
    // println!("{}", y);  // Ошибка: y вне области видимости
}

```

Правильно

Владение и передача владения (1/2)

Rust использует концепцию владения, чтобы управлять памятью. Когда переменная **передается** другой, она **"теряет"** владение.

```rust
// Передача владения
▶ Run | Debug
fn main() {
    let s1: String = String::from("Hello");
    let s2: String = s1;  // s1 передает владение
    //println!("{}", s1);  // Ошибка: s1 больше не доступна
    println!("{}", s2);  // s2 владеет строкой

```

Владение и передача владения (2/2)

Чтобы избежать передачи владения, можно использовать **ссылки** или функцию **clone**:

```rust
// Передача владения
▶ Run | Debug
fn main() {
    let s1: String = String::from("Hello");
    //let s2 = s1;  // s1 передает владение
    //println!("{}", s1);  // Ошибка: s1 больше не доступна
    //println!("{}", s2);  // s2 владеет строкой

    let s2: &String = &s1;  // s2 заимствует s1
    let s3: String = s1.clone();
    println!("s1: {}, s2: {}, s3: {}", s1, s2, s3);  // Оба доступны
}
```

Список литературы

- **The Rust Programming Language** - Steve Klabnik, Carol Nichols. [Читать онлайн](https://doc.rust-lang.org/book/)
- **Programming Rust** - Jim Blandy, Jason Orendorff.