Peter the Great
Saint-Petersburg Polytechnic University

Язык программирования Rust. Функции. Управляющие конструкции

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

POLYTECH

План презентации

- Функции
  - Объявление функции и её параметры
  - С возвращаемыми аргументами
  - Рекурсивные
- Управляющие конструкции
  - Условия
  - Конструкция потока управления match
  - Циклы
- Стандартная библиотека ввода/вывода

Объявление функции (1/2)

- Код Rust использует *змеиный регистр (snake case)* как основной стиль для имён функций и переменных, в котором все буквы строчные, а символ подчёркивания разделяет слова.

- Rust **не важно**, где вы определяете свои функции, главное, чтобы они были определены где-то в той области видимости, которую может видеть вызывающий их код.

Объявление функции (2/2)

- Пример объявления функции

```
36  fn main()
37  {
38      println!("Hello, world!");
39
40      // объявление и параметры функции
41      another_function();
42
```

```
58  fn another_function() {
59      println!("Another function.");
60  }
61
```

Параметры функции

- В сигнатурах функций **необходимо указывать тип** каждого параметра.
- При определении нескольких параметров, разделяйте объявления параметров запятыми.

```
1   fn print_sum_integer(x: i32, y: i32)
2   {
3       println!("Sum of x and y is: {}", x+y);
4   }
5
```

```
43      print_sum_integer(x: 52, y: 86);
44
```

Функции с возвращаемыми значениями (1/3)

- Функции могут возвращать значения коду, который их вызывает. Не называем возвращаемые значения, но **должны объявить** их **тип** после стрелки (->).

- В Rust **возвращаемое значение** функции является **синонимом** значения **конечного выражения в блоке тела** функции. Вы можете раньше выйти из функции и вернуть значение, используя ключевое слово **return** и указав значение, но большинство функций неявно возвращают последнее выражение.

Функции с возвращаемыми значениями (2/3)

- Пример функции с возвращаемыми значениями

```
6   fn five() -> i32
7   {
8       5
9   }
10
11  fn div_integer(x: i32, y: i32) -> i32
12  {
13      x-y
14  }
15
16  fn sum_integer_return(x: i32, y:i32) -> i32
17  {
18      return x+y;
19  }
```

```
// с возвращаемыми значениями
let mut result: i32 = five();
println!("result of five function = {}", result);

result = div_integer(x: 45, y: 52);
println!("result of div_integer function = {result}");

result = sum_integer_return(x: 54, y: 106);
println!("result of sum_integer_return finction = {result}");
```

Функции с возвращаемыми значениями (3/3)

- Пример функции с двумя возвращаемыми значениями

```
76  fn func_with_two_return_arg(mut x: i32) -> (i32, i32)
77  {
78      let mut y: i32 = 4;
79      x += 3;
80      y += x;
81
82      return (x, y);
83  }
```

```
62      let (x_: i32, y_: i32) = func_with_two_return_arg(5);
63      println!("{x_}, {y_}");
64
```

Рекурсивные функции

- **Рекурсивные функции в Rust** можно определить, указав **fn**, за которым следует имя функции и набор круглых скобок. Фигурные скобки указывают компилятору, где начинается и заканчивается тело функции.

```
23  fn rec_fibonacci(n: u64) -> u64
24  {
25      match n {
26          0|1 => n,
27          _=> rec_fibonacci(n-1) + rec_fibonacci(n-2),
28      }
29  }
```

```
66      //рекурсивные функции
67      let n: u64 = 4;
68      let result_: u64 = rec_fibonacci(n);
69      println!("{n}-th value fibonacci = {result_}");
70
71
```

lambda- функции (1/2)

В Rust лямбда-функции, или замыкания, — это анонимные функции, которые могут захватывать переменные из окружающего контекста.

Основные моменты:

- Синтаксис: **let closure = |params| expression**;
  Захват окружения: Замыкания могут захватывать переменные по ссылке, по значению или по изменяемой ссылке.
- Типы параметров: Компилятор может выводить типы параметров, но их можно указывать явно.
- Использование с высшими функциями: Замыкания часто используются в функциях высшего порядка и при обработке коллекций.

lambda- функции (2/2)

```
108     // lambda-функции
109     let add: impl Fn(i32, i32) -> i32 = |a: i32, b: i32| a + b;
110     result = add(a: 2, b: 3);
111     println!("result of lambda = {result}");
112
113     println!("result of add = {}", apply_function(5, 6, add));
114
```

```
74  fn apply_function(a: i32, b: i32, func: impl Fn(i32, i32) -> i32) -> i32  {
75      // применяем переданную функцию к аргументам a и b
76      func(a, b)
77  }
```

Управляющие конструкции. Условия (1/3)

- **Выражения if**

  Позволяют выполнять части кода в зависимости от условий.

  Условие в коде **должно быть** логического **типа bool**. Если условие не является bool, возникнет ошибка.

```
97      if number {
98          println!("number was three");
99      }
```

НЕВЕРНО

```
89      let number: i32 = 3;
90
91      if number < 5 {
92          println!("condition was true");
93      } else {
94          println!("condition was false");
95      }
96
```

ВЕРНО

Управляющие конструкции. Условия (2/3)

- **Обработка нескольких условий с помощью else if**

  Можно использовать несколько условий, комбинируя if и else в выражении **else if**.

```
100
101     if number % 4 == 0 {
102         println!("number is divisible by 4");
103     }
104     else if number % 3 == 0 {
105         println!("number is divisible by 3");
106     }
107     else if number % 2 == 0 {
108         println!("number is divisible by 2");
109     }
110     else {
111         println!("number is not divisible by 4, 3, or 2");
112     }
113
```

Управляющие конструкции. Условия (3/3)

- **Использование if в инструкции let**

  Поскольку **if** является **выражением**, его можно использовать **в правой части** инструкции **let** для присвоения результата переменной

```
120     let number: i32 = if condition { 5 } else { "six" };
121
```

НЕВЕРНО

```
116     // Пример на присвоение let значение if
117     let condition: bool = true;
118     let number: i32 = if condition { 5 } else { 6 };
```

ВЕРНО

Управляющие конструкции. Match (1/3)

Конструкция **match** оценивает некоторое выражение и сравнивает его с набором значений. И совпадении значений выполняет определенный код.

После каждого значения после оператора **=>** идут действия, которые выполняются, если это значение соответствует сравниваемому выражению.

В конце после всех значений указывается универсальное значение **_**, действия которого выполняются, если ни одно из значений не соответствуют сравниваемому выражению.

```
124     // Примеры на match
125     let num: i32 = 2;
126     match num
127     {
128         1=>println! ("один"),
129         2=>println! ("два"),
130         3=>println! ("три"),
131         _=>println! ("непонятно")
132     }
133
```

Управляющие конструкции. Match (2/3)

Конструкция match может **возвращать значение**. В этом случае мы можем присвоить его переменной:

```
133
134     let num: i32 = 4;
135     let result: &str = match num
136     {
137         1=> "один",
138         2=> "два",
139         3=> "три",
140         _=> "непонятно"
141     };
142
143     println!("result = {}", result);
144
```

Управляющие конструкции. Match (3/3)

```
145     let num: i32 = 3;
146     match num
147     {
148         1|2=>println! ("один или два"),
149         3=>{
150             println!("вторая ветка");
151             println!("три");
152         }
153         _=>println! ("непонятно")
154     }
```

Если хотим выполнить одно действие **для нескольких значений** (то есть в одной ветви): значения разделяются с помощью |

Если хотим выполнить **несколько действий** в одной ветви, вы должны использовать **фигурные скобки**, а запятая после этой ветви не обязательна.

Управляющие конструкции. Циклы (1/5)

В Rust есть три вида циклов: **loop**, **while** и **for**

- **Повторение выполнения кода с помощью loop**

  Ключевое слово **loop** говорит Rust выполнять блок кода снова и снова до бесконечности или пока не будет явно приказано остановиться.

  Ключевое слово **break** нужно поместить в цикл, чтобы указать программе, когда следует прекратить выполнение цикла.

  Использовать **continue**, которое указывает программе в цикле пропустить весь оставшийся код в данной итерации цикла и перейти к следующей итерации.

```
177     // Пример для loop
178     let mut count_1: i32 = 0;
179
180     loop {
181         match count_1{
182             1 =>
183             {
184                 count_1 += 1;
185                 continue;
186             }
187             3 => break,
188             _ => (),
189         }
190         println!("count = {}", count_1);
191         count_1 += 1;
192     };
193
194
```

Управляющие конструкции. Циклы (2/5)

- **Повторение выполнения кода с помощью loop**

  Может понадобиться передать из цикла результат операции в остальную часть кода. Для этого можно добавить **возвращаемое значение** после выражения break, которое используется для остановки цикла.

```
176     let mut counter: i32 = 0;
177
178     let result: i32 = loop {
179         counter += 1;
180
181         if counter == 10 {
182             break counter * 2;
183         }
184     };
185
186     println!("The result loop is {result}");
187
```

Управляющие конструкции. Циклы (3/5)

- **Повторение выполнения кода с помощью loop**

  Если у вас есть циклы внутри циклов, **break** и **continue** применяются к самому **внутреннему циклу** в этой цепочке. При желании вы можете **создать *метку цикла***, которую вы затем сможете использовать с break или continue для указания, что эти ключевые слова применяются к помеченному циклу, а не к самому внутреннему циклу. **Метки цикла** должны начинаться **с одинарной кавычки**.

```
189    // Пример остановки с внутренним циклом
190    let mut count: i32 = 0;
191    'counting_up: loop {
192        println!("count = {count}");
193        let mut remaining: i32 = 10;
194
195        loop {
196            println!("remaining = {remaining}");
197            if remaining == 9 {
198                break;
199            }
200            if count == 2 {
201                break 'counting_up;
202            }
203            remaining -= 1;
204        }
205
206        count += 1;
207    }
208    println!("End count = {count}");
209
```

Управляющие конструкции. Циклы (4/5)

- **Циклы с условием while**

Пока условие истинно, цикл выполняется. Когда условие перестаёт быть истинным, программа вызывает break, останавливая цикл.

```
211    // Пример для цикла while
212    let mut number: i32 = 3;
213
214    while number != 0 {
215        println!("number = {number}!");
216
217        number -= 1;
218    }
219
220    println!("end of while, number = {number}");
221
```

Управляющие конструкции. Циклы (5/5)

- **Цикл по элементам диапазона с помощью for**

Для этого можно использовать Range, предоставляемый стандартной библиотекой, который генерирует последовательность всех чисел, начиная с первого числа и заканчивая вторым числом, но не включая его (т.е. (start..end) эквивалентно [start, start+1, start+2, ... , end-2, end-1])

```
221    // Пример для цикла for
222    for number: i32 in 1..4
223    {
224        println!("number = {number}");
225    }
226
227    println!("end of for");
228
```

Стандартная библиотека ввода/вывода (1/3)

В Rust стандартная библиотека ввода/вывода (I/O) предоставляет несколько ключевых модулей для работы с вводом и выводом данных, среди которых наиболее важные — это **std::io**, **std::fs**, и **std::env**.

- **Ввод и вывод**

  std::io: Этот модуль содержит функциональность для работы с вводом и выводом, включая функции для чтения и записи данных.

```rust
use std::io;

fn input_output_text() {
    let mut input: String = String::new();
    println!("Введите текст: ");

    io::stdin().read_line(buf: &mut input).expect(msg: "Не удалось прочитать строку");
    println!("Вы ввели: {}", input.trim());
}
```

Стандартная библиотека ввода/вывода (2/3)

- **Аргументы командной строки**

  std::env: Этот модуль позволяет получать аргументы командной строки.

  Аргументы хранятся в векторе **Vec<String>** и доступны через функцию **args()**.

```
cargo run -- test, hi
```

```rust
use std::env;

fn args_command_line() {
    let args: Vec<String> = env::args().collect();
    for arg: &String in args.iter() {
        println!("Аргумент: {}", arg);
    }

    if args.len() >= 2
    {
        let arg_1: &String = &args[1];
        let arg_2: &String = &args[2];

        println!("Arg_1 = {arg_1}");
        println!("Arg_2 = {arg_2}");
    }

}
```

Стандартная библиотека ввода/вывода (3/3)

- **Чтение файлов и запись в них**

std::fs: Этот модуль используется для работы с файловой системой, включая чтение и запись файлов.

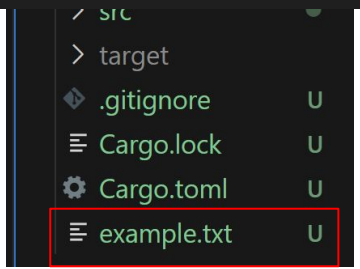
Для чтения файла можно использовать **File::open()** и **read_to_string()**.

Для записи в файл можно использовать **File::create()** и **write_all()**.

```rust
use std::fs::File;
use std::io::Read;
use std::io::Write;

fn work_with_file() -> io::Result<()> {
    let mut file: File = File::open(path: "example.txt")?;
    let mut contents: String = String::new();
    file.read_to_string(buf: &mut contents)?;
    println!("Содержимое файла:\n{}", contents);

    let mut file: File = File::create(path: "foo.txt")?;
    file.write_all(buf: contents.as_bytes())?;
    Ok(())
}
```



POLYTECH

Список литературы

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