

# Rust. Функции

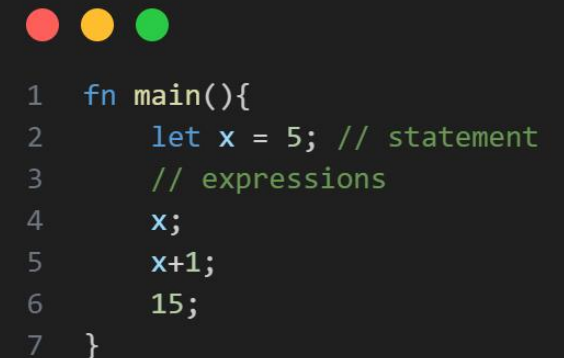
Выполнили: учащиеся группы: 5030102/00201

# Операторы и выражения (1/2)

- Тело функции состоит из серии операторов, которые могут заканчиваться выражением.
- **Оператор(statement)** - инструкция, которая выполняет действие и ничего не возвращают. Заканчиваются ( ; )
- **Выражения(expression)** – результирующее значение.



```
1  fn main(){  
2      //statement 1  
3      //statement 2  
4      //statement 3  
5  }  
6
```



```
1  fn main(){  
2      let x = 5; // statement  
3      // expressions  
4      x;  
5      x+1;  
6      15;  
7  }
```

# Операторы и выражения (2/2)

- Выражения могут быть частью операторов.



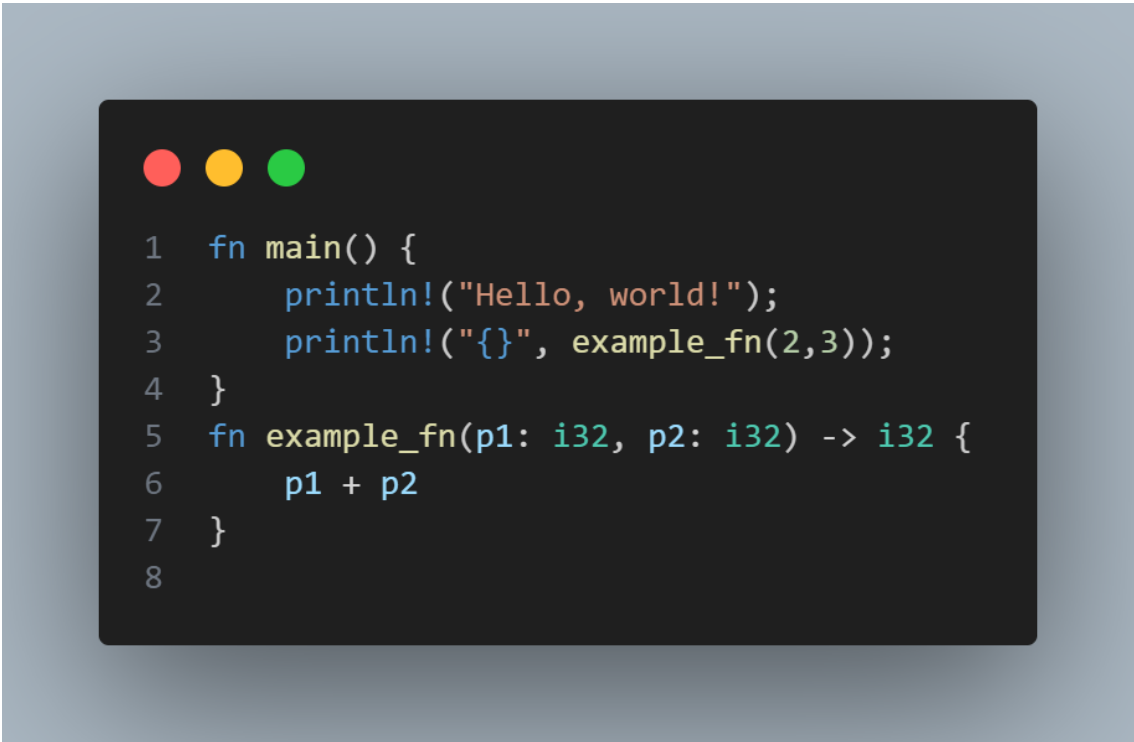
```
1 fn main() {  
2     let y = {  
3         let mut x = 3;  
4         x += 1;  
5         x // expression  
6     };  
7     println!("The value of y is: {y}");  
8 }
```



```
1 //error: the operator returns nothing  
2 fn main() {  
3     let x = (let y = 6);  
4 }
```

# Функции

- **Функции** в Rust определяются с помощью ключевого слова **fn**, за которым следуют имя функции, параметры и тип возвращаемого значения после ( **->** ).
- Не важно где определяются функции; важно, чтобы они были определены в области видимости, которую может видеть вызывающая процедура.



```
1 fn main() {  
2     println!("Hello, world!");  
3     println!("{}", example_fn(2,3));  
4 }  
5 fn example_fn(p1: i32, p2: i32) -> i32 {  
6     p1 + p2  
7 }  
8
```

## Console:

Hello, world!

5

# Входные данные (1/2)

- Явное объявление типов параметров **необходимо**.
- Передача осуществляется либо по изменяемой/не изменяемой ссылке, либо по значению, либо по скопированному значению.
- Целочисленные типы(**i32**, **u32**,...) реализуют свойства копирования.

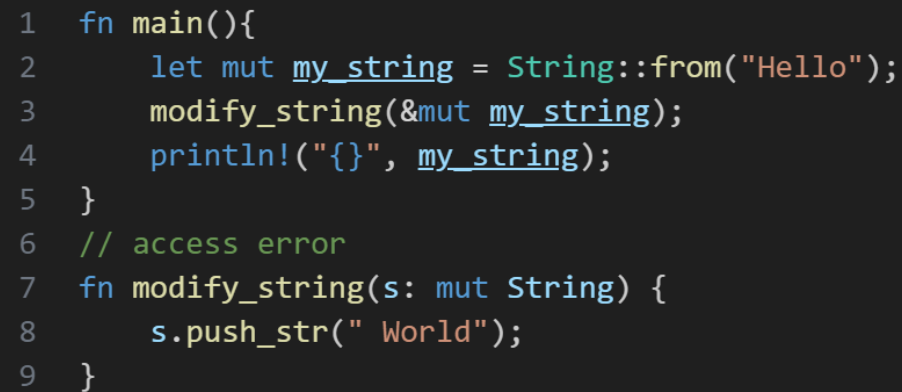
## Console:

```
take_by_value: x = 5
Main: num = 5
modify_by_value: x = 6
Main: num = 5
modify_by_reference: x = 6
Main: num = 6
```

```
1 fn main(){
2     let mut num = 5;
3     take_by_value(num);
4     println!("Main: num = {}", num);
5     modify_int(num);
6     println!("Main: num = {}", num);
7     modify_int_by_reference(&mut num);
8     println!("Main: num = {}", num);
9 }
10 fn take_by_value(x: i32) {
11     println!("take_by_reference: x = {}", x);
12 }
13 fn modify_int(mut x: i32) {
14     x += 1;
15     println!("take_by_reference: x = {}", x);
16 }
17 fn modify_int_by_reference(x: &mut i32){
18     *x += 1;
19     println!("modify_by_reference: x = {}", x);
20 }
```

# Входные данные (2/2)

- Другие типы (в том числе пользовательские) требуют передавать по ссылке.
- Для того чтобы изменять исходное значение нужно использовать (**&mut**)



```
1 fn main(){
2     let mut my_string = String::from("Hello");
3     modify_string(&mut my_string);
4     println!("{}", my_string);
5 }
6 // access error
7 fn modify_string(s: mut String) {
8     s.push_str(" World");
9 }
```

Console:

take\_by\_value: x = 6

Main: num = 5

take\_by\_reference: x = 6

Main: num = 6

# () – «единица измерения»(unit) (1/2)

- В Rust () – особый тип и значение, которое означает
  - Пустой кортеж
  - Отсутствие значения – функция/выражение не возвращает результата. Например, служит для определения

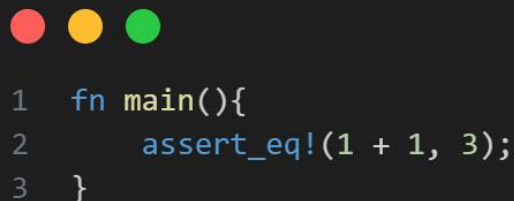
```
1 fn log_message(messaage: &str) -> (){
2     println!("{}", messaage);
3 }
4
5 fn log_message2(messaage: &str){
6     println!("{}", messaage);
7 }
8 fn main(){
9     let empty_tuple: () = ();
10    println!("Empty tuple: {:?}", empty_tuple);
11    log_message("Hello, world!");
12    log_message2("Hello, world!");
13 }
```

Console:

```
Empty tuple: ()
Hello, world!
Hello, world!
```

# () – «единица измерения»(unit) (2/2)

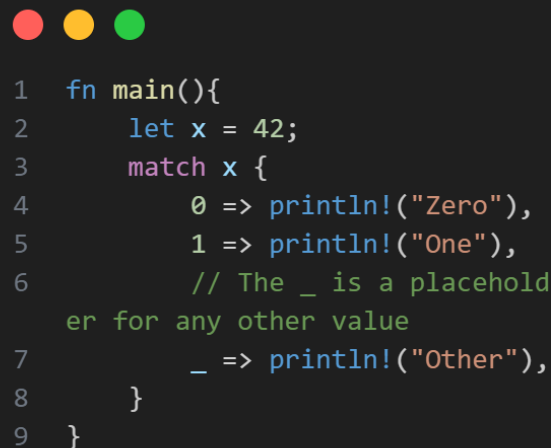
- В Rust () – особый тип и значение, которое означает
  - Распространённый тип макросов (например, генерация кода)
  - Заполнитель шаблонов



```
1 fn main(){
2     assert_eq!(1 + 1, 3);
3 }
```

Console:

```
thread 'main' panicked at
'assertion failed: `(left ==
right)`
  left: `2`,
 right: `3`', src/main.rs:52:5
```



```
1 fn main(){
2     let x = 42;
3     match x {
4         0 => println!("Zero"),
5         1 => println!("One"),
6         // The _ is a placeholder for any other value
7         _ => println!("Other"),
8     }
9 }
```

Console:

Other



# Выходные данные

- Последнее выражение в функции используется, как возвращаемое значение.
- Также используется оператор **return** (чтобы вернуть значение раньше: из цикла или оператора if).
- Функции, которые «не» возвращают значение — возвращают единичный тип **()**

```
1 fn main() {
2     let x = plus_one(5);
3     println!("The value of x is: {x}");
4     let y = plus_one2(5);
5     println!("The value of x is: {y}");
6 }
7
8 fn plus_one(x: i32) -> i32 {
9     x + 1
10 }
11
12 // error: must return i32
13 // but statement doesn't return anything
14 fn plus_one2(x: i32) -> i32 {
15     x + 1;
16 }
17
```

# Некоторые примеры...

```
1 fn main(){
2     let mut a:i32 = 10;
3     f2(&mut a);
4
5     fn f1(a: &mut i32){
6         println!("{}", *a);
7     }
8
9     fn f2(a: &mut i32){
10         *a = 20;
11         f1(a);
12     }
13 }
```

Console:

20

```
1 fn main(){
2     let mut a:i32 = 10;
3     f2(&mut a);
4
5     fn f1(a: &mut i32){
6         println!("{}", *a);
7     }
8
9     fn f2(a: &mut i32){
10         *a = 20;
11         return f1(a);
12     }
13     fn f(a: &mut i32){
14         f2(a);
15     }
16     f(&mut a);
17 }
```


Console:

20

20

# Рекурсия

- Рекурсия в Rust так же, как и в других в других ЯП, является функцией вызывающей саму себя.



```
1  fn factorial(n: u64) -> u64 {
2      if n == 0 {
3          1
4      } else {
5          n * factorial(n - 1)
6      }
7  }
8
9  fn main() {
10     let n = 5;
11     let result = factorial(n);
12     println!("Factorial of {} is: {}", n, result);
13 }
```

## Console:

Factorial of 5 is: 120

# Замыкания

- Анонимные функции, которые могут захватывать переменные
- В отличие от функции: тип входных и выходных данных указывать необязательно, а название аргумента обязательно.
- Используется `||` вместо `()`
- Ограничение тела функции `{}` - опционально
- Захват переменных за пределами окружения

```
1 fn main() {
2     fn function      (i: i32) -> i32 { i + 1 }
3     let closure_annotated = |i: i32| -> i32 { i + 1
4 };
5     let closure_inferred  = |i      |          i + 1
6     ;
7     let i = 1;
8     println!("Function: {}", function(i));
9     println!("Closure with type indication: {}", closure_annotated(i));
10    println!("Closure with type output: {}", closure_inferred(i));
11
12    // no arguments, but returns `i32`
13    // type is identified automatically
14    let one = || 1;
15    println!("Closure returning one: {}", one());
16 }
```

## Console:

Function: 2

Closure with type indication: 2

Closure with type output: 2

Closure returning one: 1

# Захват переменных в замыканиях (1/2)

```
fn distance(a: Point, b: Point) -> f64 {  
    let dist = | p : Point | -> f64 {((a.x - p.x).powi(2) + (a.y - p.y).powi(2)).sqrt()};  
    dist(b)  
}
```

- Замыкание «захватывает» переменную “a” по ссылке
- Замыкание «захватывает» переменную “p” полностью
- После выполнения замыкания “dist(b)” переменная “a” будет доступна, переменная “b” будет уничтожена.
- Для переменных, передаваемых в замыкание работают все правила «Владения»
- Такой захват называется «заимствование»

## Захват переменных в замыканиях (2/2)

```
fn distance(a: Vec<f64>, b: Vec<f64>) -> f64 {  
    let dist = move || -> f64 {((a[0] - b[0]).powi(2) + (a[1] - b[1]).powi(2)).sqrt()};  
    dist()  
}
```

- С помощью ключевого слова “move” перед пайпом, владение полностью передается в замыкание
- Согласно концепции владения, после выполнения “dist()”, переменные “a”, “b” уничтожаются
- Для простых типов данных выполняется копирование
- Такой захват называется захватом «кражей», потокобезопасно.

# Типы функций и замыканий (1/2)

- Функции, как и любые другие переменные могут иметь тип
- Это делает возможным передачу функции, как параметр переменной
- Функции можно хранить в структурах данных
- Для создания переменных с типом функции используется ключевое слово “fn”

```
fn dist_sqr(dist: fn(Vec<f64>, Vec<f64>) -> f64){...}
```

```
fn add(a: i32, b: i32) -> i32 {  
    a + b  
}
```

```
let add_func = add;
```

```
let add: fn(i32, i32) -> i32 = |a, b| a + b;
```

# Типы функций и замыканий (2/2)

- Для создания и использования переменной типа замыкание, используется ключевое слово “Fn”
- “Fn” – вызываемый тип

```
fn(&Point) -> bool // тип fn (только функции)  
Fn(&Point) -> bool // характеристика Fn  
(функции и замыкания)
```

- Тип замыкания всегда уникален, он зависит от возвращаемых данных, передаваемых и захватываемых переменных, создается компилятором
- Но каждое замыкание реализует тип “Fn”



# Производительность замыканий

- В Rust память для замыканий выделяется не в куче
- Замыкания могут подставляться компилятором в код автоматически
- Замыкания в Rust часто используют в циклах, в отличие от других ЯП

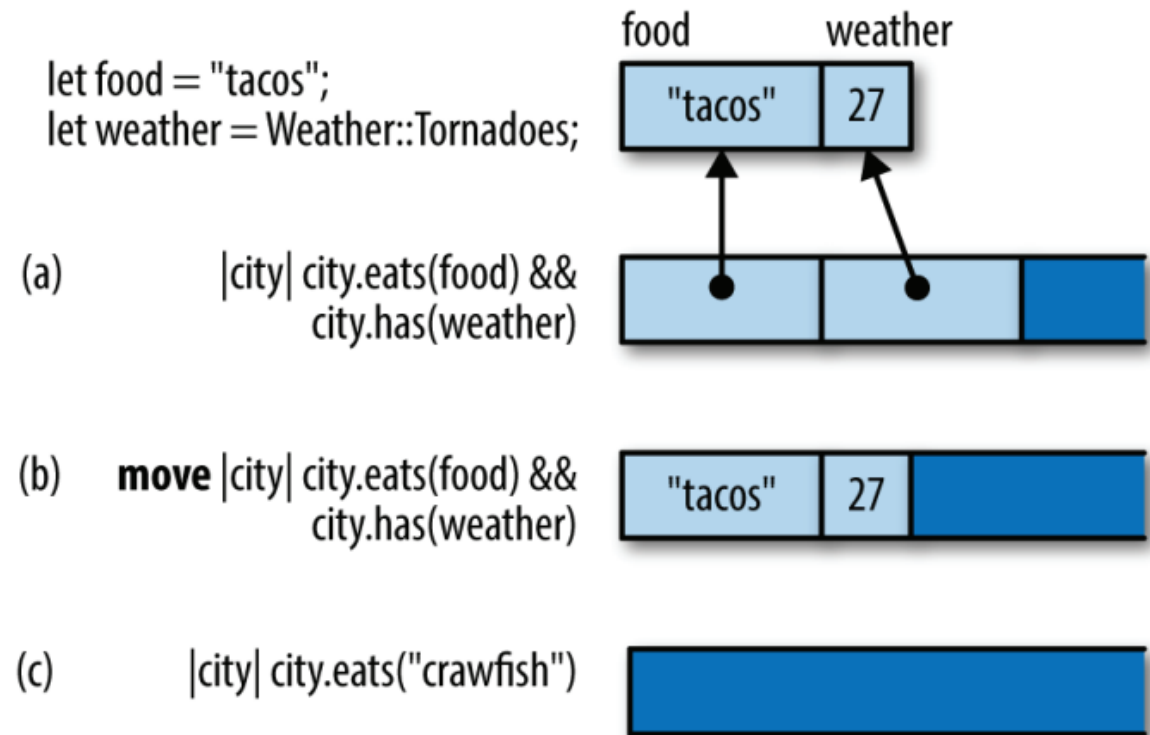


Рис. 14.1 ❖ Размещение замыканий в памяти

# Материалы

- Официальная документация Rust(en, ru)
  - <https://doc.rust-lang.org/book/ch03-03-how-functions-work.html>
  - <https://doc.rust-lang.ru/stable/rust-by-example/fn.html>
  - <https://doc.rust-lang.ru/stable/rust-by-example/expression.html>
- Программирование на языке Rust
  - [https://vk.com/topic-51126445\\_36552642](https://vk.com/topic-51126445_36552642)

Спасибо за внимание!

