Peter the Great
Saint-Petersburg Polytechnic University

Язык программирования Rust. Параллелизм. ООП.

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

POLYTECH

План презентации

- Подробно:
  - Параллелизм (Основные подходы)
  - ООП
- Кратко:
  - Обработка ошибок
  - Управление памятью

Параллелизм. Основные подходы (1/6)

Rust предоставляет безопасные инструменты для параллелизма благодаря системе владения, предотвращающей гонки данных на этапе компиляции.

1. Потоки через **std::thread**

   Функция thread::spawn в Rust создаёт новый поток.

   Метод join() в Rust ожидает завершения потока, блокируя основной поток до завершения порождённого потока. Метод join() возвращает объект Result, к которому далее применяется метод unwrap().

   Метод unwrap() используется для обработки любых ошибок, которые могут возникнуть во время присоединения

Параллелизм. Основные подходы (2/6)

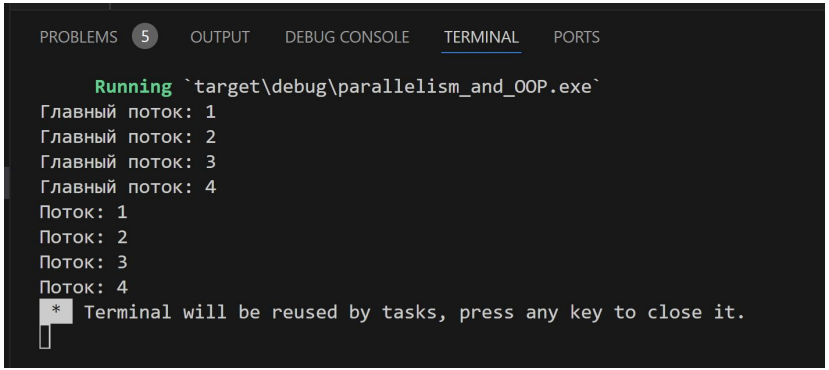
1. Потоки через **std::thread**. Пример.

```rust
//использование потоков
use std::thread;

pub fn using_threads() {
    let handle: JoinHandle<()> = thread::spawn(||
        {
        for i: i32 in 1..5 {
            println!("Поток: {}", i);
        }
    });

    for i: i32 in 1..5 {
        println!("Главный поток: {}", i);
    }

    handle.join().unwrap();
}
```



POLYTECH

Параллелизм. Основные подходы (3/6)

2. **Обмен данными между потоками через каналы:**

   Метод send() в каналах на языке Rust позволяет отправить сообщение. **Метод recv**, что является сокращением от receive, который блокирует выполнение основного потока и ждёт, пока данные не будут переданы по каналу.

```rust
//обмен с помощью каналов
use std::sync::mpsc;

pub fn using_channel() {
    let (tx: Sender<String>, rx: Receiver<String>) = mpsc::channel();

    thread::spawn(move || {
        let message: String = String::from("Привет из потока!");
        tx.send(message).unwrap();
    });

    let received: String = rx.recv().unwrap();
    println!("Получено: {}", received);
}

```

```
     Running `target\debug\parallelism_and_OOP.exe`
Получено: Привет из потока!
```

Параллелизм. Основные подходы (4/6)

3. **Мьютексы для совместного доступа**.

- **Arc** в языке Rust **позволяет безопасно обмениваться данными между потоками**.

  Он оборачивает значение, которым пытаются поделиться, и является указателем на него. Этим Arc отслеживаются все копии указателя, и по выходе последнего указателя из области видимости безопасно освобождается память.

- **Mutex** в языке программирования Rust **помогает защитить общие данные в нескольких потоках**. Он гарантирует, что **только один поток может одновременно получить доступ к защищенным данным**, предотвращая гонку данных.

Параллелизм. Основные подходы (5/6)

3. **Мьютексы для совместного доступа** (пример)

Функция Mutex::new в Rust создаёт новый мьютекс (Mutex). В качестве аргумента ей передаётся начальное значение, которое представляет общие данные, которые будет защищать мьютекс.

Функция Arc::clone в Rust создаёт новую управляемую ручку, которую можно переместить в новый поток. При этом клонируется не значение, а только ссылка.

Метод Arc::lock в Rust блокирует доступ к общему ресурсу, защищённому мьютексом, для текущего потока.

```rust
use std::sync::{Arc, Mutex};

pub fn using_mutex() {
    let counter: Arc<Mutex<i32>> = Arc::new(data: Mutex::new(0));
    let mut handles: Vec<JoinHandle<()>> = vec![];

    for _ in 0..10 {
        let counter: Arc<Mutex<i32>> = Arc::clone(self: &counter);
        let handle: JoinHandle<()> = thread::spawn(move || {
            let mut num: MutexGuard<'_, i32> = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }

    for handle: JoinHandle<()> in handles
    {
        handle.join().unwrap();
    }

    println!("Результат: {}", *counter.lock().unwrap());
}
```

```
    Running `target\debug\parallelism_and_OOP.exe`
Результат: 10
```

Параллелизм. Основные подходы (6/6)

4. **Высокоуровневые библиотеки**, такие как **rayon**:
Для подключения этой библиотеки нужно её добавить в Cargo.toml:

parallelism_and_OOP > Cargo.toml

```
1  [package]
2  name = "parallelism_and_OOP"
3  version = "0.1.0"
4  edition = "2021"
5
6  [dependencies]
7  rayon = "1.5.1"
```

```
62  use rayon::prelude::*;
63
64  pub fn high_level_library()
65  {
66      let numbers: Vec<i32> = (1..10).collect();
67      let squares: Vec<i32> = numbers.par_iter() Iter<'_, i32>
68                                     .map(map_op: |x: &i32| x * x)
69                                     .collect();
70
71      println!("Квадраты: {:?}", squares);
72  }
```

Подробнее с этой библиотекой можно ознакомиться по ссылке:
[Implementing data parallelism with Rayon Rust](https://docs.rs/rayon/latest/rayon/iter/trait.ParallelIterator.html)

```
    Running `target\debug\parallelism_and_OOP.exe`
Квадраты: [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

ООП (1/4)

Rust не поддерживает ООП в классическом виде, но позволяет реализовать его принципы через структуры, трейты и инкапсуляцию.

1. **Инкапсуляция.**

Достигается с помощью структур и модификаторов доступа (pub).

```rust
// инкапсуляция
2 implementations
struct Rectangle {
    width: f64,
    height: f64,
}

impl Rectangle {
    fn new(width: f64, height: f64) -> Self {
        Self { width, height }
    }

    fn perimeter(&self) -> f64 {
        2.0 * (self.width + self.height)
    }
}
```

```rust
// инкапсуляция
let rect: Rectangle = Rectangle::new(width: 10.0, height: 20.0);
println!("Периметр: {}", rect.perimeter());
```

ООП (2/4)

```rust
// наследование
2 implementations
trait Shape {
    fn area(&self) -> f64;
}

1 implementation
struct Circle {
    radius: f64,
}

impl Shape for Circle {
    fn area(&self) -> f64 {
        3.14 * self.radius * self.radius
    }
}

impl Shape for Rectangle{
    fn area(&self) -> f64 {
        self.width * self.height
    }
}
```

2. **Наследование** (через композицию и трейты).

Rust использует трейты вместо традиционного наследования.

```rust
// наследование
let circle: Circle = Circle { radius: 5.0 };
println!("Площадь круга: {}", circle.area());
println!("Площадь периметра: {}", rect.area());
```

ООП (3/4)

```
2 implementations
44  trait Shape {
45      fn area(&self) -> f64;
46  }
```

```
65  // полиморфизм
66  fn print_area(shape: &dyn Shape) {
67      println!("Площадь: {}", shape.area());
68  }
```

```
21      // полиморфизм
22      print_area(shape: &circle);
23      print_area(shape: &rect);
```

3. **Полиморфизм**.

Обеспечивается через объекты-диспатчеры (Box<dyn Trait>).

ООП (4/4)

```rust
// абстракция
2 implementations
trait Animal {
    fn speak(&self);
}

1 implementation
struct Dog;

impl Animal for Dog {
    fn speak(&self) {
        println!("Гав!");
    }
}

1 implementation
struct Cat;

impl Animal for Cat {
    fn speak(&self) {
        println!("Мяу!");
    }
}
```

4. Абстракция.

Реализуется через интерфейсы в виде трейтов.

```rust
// абстракция
let dog: Dog = Dog;
let cat: Cat = Cat;

dog.speak();
cat.speak();
```

Обработка ошибок (1/3)

Для более подробного ознакомления с этой темой можно воспользоваться информацией по ссылке:
[Error handling in Rust](https://doc.rust-lang.org/book/ch09-00-error-handling.html)

В Rust используется два типа ошибок: паника и обрабатываемые ошибки.

- **Паника** (panic!) используется для критических ситуаций.

```
3   pub fn using_panic()
4   {
5       panic!("Что-то пошло не так!");
6   }
```

Обработка ошибок (2/3)

- Обрабатываемые ошибки реализуются через типы **`Result<T, E>`** и **`Option<T>`**.

```rust
pub fn divide(a: i32, b: i32) -> Result<i32, String> {
    if b == 0 {
        Err(String::from("Деление на ноль"))
    } else {
        Ok(a / b)
    }
}
```

```rust
    match error_handling::divide(a: 10, b: 2) {
        Ok(result: i32) => println!("Результат: {}", result),
        Err(e: String) => println!("Ошибка: {}", e),
    }
```

Обработка ошибок (3/3)

- Для сокращения кода можно использовать оператор **?**

```rust
pub fn read_file() -> Result<String, std::io::Error> {
    std::fs::read_to_string(path: "example.txt")
}

pub fn call_read() -> Result<(), std::io::Error>
{
    let content: String = read_file()?;
    println!("{}", content);
    Ok(())
}
```

Управление памятью (1/3)

Rust управляет памятью с помощью системы владения (ownership), ссылок и заимствования (borrowing). Основные правила:

- У каждого значения есть владелец.
- Может быть только один владелец.
- Данные освобождаются автоматически, когда владелец выходит из области видимости.

Подробнее с темой можно ознакомиться по ссылке: [Ownership and Memory Management in Rust](https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html)

Управление памятью (2/3)

- Пример

```rust
pub fn example_memory_1() {
    let s: String = String::from("Привет");
    takes_ownership(some_string: s);
    // println!("{}", s); // Ошибка: владение передано

    let x: i32 = 5;
    makes_copy(some_integer: x);
    println!("{}", x); // x доступен, так как i32 копируется
}

fn takes_ownership(some_string: String) {
    println!("{}", some_string);
}

fn makes_copy(some_integer: i32) {
    println!("{}", some_integer);
}
```

Управление памятью (3/3)

- Для доступа без передачи владения используются ссылки:

```rust
pub fn example_memory_2() {
    let s: String = String::from("Привет");
    let len: usize = calculate_length(&s);
    println!("Длина '{}': {}", s, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

Специфические конструкции Rust:

- Ownership: управление памятью через владение.
- Borrowing: ссылки с явной аннотацией заимствования.
- Match: мощный инструмент сопоставления с шаблоном.

```
53      // пример для специфической конструкции match
54      let num: Option<i32> = Some(5);
55      match num {
56          Some(x: i32) => println!("Число: {}", x),
57          None => println!("Нет числа"),
58      }
```

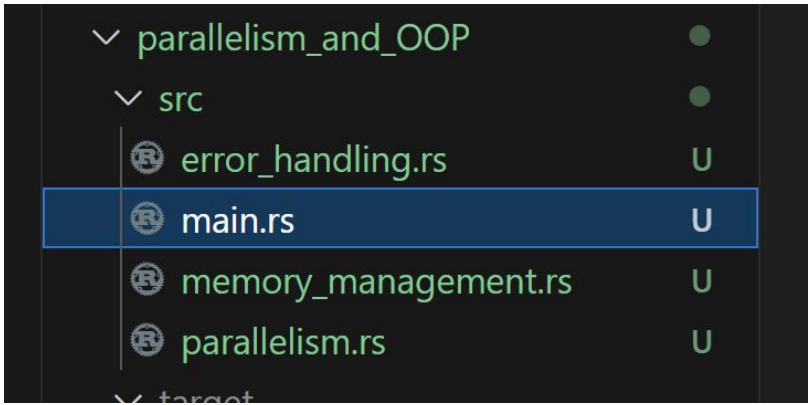
Подробнее со специфическими конструкциями в Rust можно ознакомиться в официальной документации: [The Rust Programming Language Book] (https://doc.rust-lang.org/book/)

Виртуальная машина

- Rust **не использует** виртуальную машину. Код компилируется в машинные инструкции с помощью LLVM, что обеспечивает высокую производительность без накладных расходов на интерпретацию.

Как в Rust вызывать функции из других файлов .rs

- Для доступа к функциям из других файлов в Rust используется объявление: **mod**, что компилятор Rust автоматически искать соответствующие .rs файлы.
- Те функции, которые будут вызываться извне, должны быть типа public, что объявляется с помощью ключевого слова **pub**.




```rust
mod parallelism;
mod error_handling;
mod memory_management;
```

```rust
// параллелизм
parallelism::using_threads();
parallelism::using_channel();
parallelism::using_mutex();
parallelism::high_level_library();
```

Список литературы

- **The Rust Programming Language** - Steve Klabnik, Carol Nichols. [Читать онлайн](https://doc.rust-lang.org/book/)
- **Programming Rust** - Jim Blandy, Jason Orendorff.
- **[Asynchronous Programming in Rust]** (https://doc.rust-lang.ru/async-book/01_getting_started/01_chapter.html)
- **[Implementing data parallelism with Rayon Rust]** (https://docs.rs/rayon/latest/rayon/iter/trait.ParallelIterator.html)
- **[Error handling in Rust]** (https://doc.rust-lang.org/book/ch09-00-error-handling.html)
- **[Ownership and Memory Management in Rust]** (https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html)