Peter the Great
Saint-Petersburg Polytechnic University

Язык программирования Rust. Коллекции. Обобщённые типы

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

POLYTECH

План презентации

- Коллекции
  - Массивы
  - Векторы
  - Хэш-карты
  - Множества
- Обобщённые типы

Массивы (1/2)

Массивы в Rust представляют собой последовательный блок памяти фиксированного размера. Все элементы массива должны быть одного типа, и их количество задается при создании массива.

- Массивы не используют сложных алгоритмов для вставки или удаления, поскольку это невозможно для фиксированных массивов. Для работы с элементами используются прямые вычисления адресов (вычисление смещения от начала массива).

- Массивы **нельзя** **изменять по размеру**, но **можно** **изменять элементы**.

Массивы (2/2)

```
 9      let arr_1: [i32; 5] = [1, 2, 3, 4, 5];
10      println!("First element: {}", arr_1[0]);
11
12      let mut arr_2: [i32; 3] = [9, 8, 7];
13      //arr_1[0] = 10
14      arr_2[0] = 10;
15      println!("{:?}", arr_2);
16
```

Векторы(1/2)

Vec<T> — это динамический массив, который может изменять свой размер в процессе выполнения программы. Вектора используют динамическое выделение памяти. При необходимости, когда вектору требуется больше памяти, происходит выделение нового блока памяти большего размера, а существующие элементы копируются в новый блок.

- При добавлении элементов в вектор происходит динамическое увеличение его размера. Внутренний алгоритм управления памятью основан на стратегии удвоения емкости при исчерпании свободного места, что позволяет сократить количество операций выделения памяти.
- Доступ к элементам осуществляется через прямые адресации, как и в массивах.

Векторы(2/2)

```rust
    let mut vec: Vec<i32> = Vec::new();
    vec.push(1);
    vec.push(2);
    vec.push(3);
    println!("{:?}", vec);


    let vec: Vec<i32> = vec![10, 20, 30];
    let first: i32 = vec[0];
    println!("First element: {}", first);
```

Хэш-карты(1/2)

HashMap<K, V> — это реализация хэш-таблицы, где данные хранятся в виде пар ключ-значение. Для быстрого доступа к элементам используется хэширование ключей.

- **Алгоритмы внутри**:
  Хэширование: В основе хэш-карты лежит хэш-функция, которая преобразует ключ в индекс в массиве. В Rust используется хэш-функция по умолчанию, но она может быть переопределена.
  Коллизии: Rust использует алгоритм, основанный на открытой адресации или цепочках (в зависимости от реализации) для разрешения коллизий. В случае коллизии элемент помещается в связанную структуру или смежную ячейку.

- Для работы с хэш-картами необходимо подключить модуль **std::collections**.

Хэш-карты(2/2)

```rust
use std::collections::HashMap;
fn hashmap()
{
    let mut scores: HashMap<String, i32> = HashMap::new();
    scores.insert(k: String::from("Blue"), v: 10);
    scores.insert(k: String::from("Yellow"), v: 50);

    println!("{:?}", scores);

    // доступ по ключу
    if let Some(score: &i32) = scores.get("Blue") {
        println!("Score for Blue: {}", score);
    }

}
```

Множества

HashSet<T> — это реализация множества, которая внутри использует хэш-карту (HashMap), где ключи представляют собой элементы множества, а значения опущены.

- **Алгоритмы внутри**:
  Так как HashSet основан на хэш-карте, все алгоритмы, описанные для HashMap, применимы и здесь. Хэширование и разрешение коллизий также происходят аналогично.

- Также требует подключения модуля **std::collections**.

```rust
use std::collections::HashSet;
fn set()
{
    let mut set: HashSet<i32> = HashSet::new();
    set.insert(1);
    set.insert(2);
    set.insert(2);  // Второй раз не добавится

    println!("{:?}", set);

    // проверка наличия элемента в множестве
    if set.contains(&5) {
        println!("Set contains 5");
    }
    else {
        println!("Set not contains 5");
    }

    // длина множества
    println!("Length of set {}", set.len());

    // цикл по элементам множества
    for item: &i32 in &set {
        println!("{item}");
    }

} fn set
```

Коллекции. Итог

Структура данных	Доступ к элементу	Вставка	Удаление	Размер
Массив	O(1)	N/A	N/A	Фиксированный
Вектор	O(1)	O(1) / O(n)	O(1)	Динамический
Хэш-карта	O(1) в среднем	O(1) в среднем	O(1) в среднем	Динамический
Множество	O(1) в среднем	O(1) в среднем	O(1) в среднем	Динамический

Обобщённые типы (1/5)

- **Обобщённые типы** — это мощный инструмент, который позволяет писать универсальный и безопасный код в Rust.
- **Обобщённые типы** (или дженерики) в языке Rust позволяют писать **функции, структуры**, которые могут работать с разными типами данных. Это помогает избежать дублирования кода и делает его гибче.
- В языке Rust **нет привычного понятия классов**, как в языках, поддерживающих ООП. Вместо этого используются структуры (struct) и трейты (trait), которые в комбинации с обобщёнными типами позволяют создавать гибкие и переиспользуемые конструкции, похожие на обобщённые классы в других языках.

Обобщённые типы (2/5)

```rust
// обобщённые типы
fn max<T: PartialOrd>(a: T, b: T) -> T {
    if a > b {
        a
    } else {
        b
    }
}
```

```rust
    // пример 1
    let int_max: i32 = max(a: 10, b: 20);
    let float_max: f64 = max(a: 10.5, b: 7.3);
    println!("Max int: {}", int_max);
    println!("Max float: {}", float_max);
```

Здесь T — обобщённый тип. Ограничение T: PartialOrd означает, что тип T должен поддерживать операции сравнения.

Обобщённые типы (3/5)

Создадим структуру Point, которая может хранить координаты разных типов данных.

В этом примере Point<T> — структура с обобщённым типом T, который позволяет использовать любые типы для x и y.

```
133  struct Point<T> {
134      x: T,
135      y: T,
136  }
```

```
90   // пример 2
91   let int_point: Point<i32> = Point { x: 5, y: 10 };
92   let float_point: Point<f64> = Point { x: 1.0, y: 4.5 };
93   println!("Int Point: ({}, {})", int_point.x, int_point.y);
94   println!("Float Point: ({}, {})", float_point.x, float_point.y);
95
```

Обобщённые типы (4/5)

Обобщённые методы в структурах

Методы могут быть также обобщёнными, что позволяет методам структуры принимать или возвращать значения различных типов.

```
136  struct Point<T> {
137      x: T,
138      y: T,
139  }
140
141  impl<T> Point<T> {
142      fn get_x(&self) -> &T {
143          &self.x
144      }
145  }
```

```
96      let point: Point<i32> = Point { x: 5, y: 10 };
97      println!("x: {}", point.get_x());
```

Обобщённые типы (5/5)

Перечисление Option используется для указания наличия или отсутствия значения. В Rust Option — это стандартный тип, но его можно представить как обобщённый:

```
138  enum Option<T> {
139      Some(T),
140      None,
141  }
```

Option<T> может принимать любое значение T, что делает его универсальным для различных типов данных.

```
96   // пример 3
97   let some_number: Option<i32> = Option::Some(42);
98   let some_string: Option<&str> = Option::Some("Hello");
99   let none_value: Option<i32> = Option::None;
100
```

Ограничения на обобщённые типы (Trait Bounds)

Чтобы наложить ограничения на обобщённые типы, можно использовать трейты.

```rust
use std::fmt::Display;
fn print_value<T: Display>(value: T) {
    println!("{}", value);
}
```

Здесь T: Display позволяет передавать в функцию только значения, поддерживающие вывод на экран.

```rust
// пример 4
print_value(10); // Для типа i32
print_value("Hello, Rust!"); // Для типа &str
```

Несколько ограничений (1/2)

Можно задать несколько ограничений для одного обобщённого типа:

```rust
fn compare_and_display<T: PartialOrd + Display>(a: T, b: T) {
    if a > b {
        println!("{} is greater", a);
    } else {
        println!("{} is greater", b);
    }
}
```

```rust
    // пример 5
    compare_and_display(a: 10, b: 20);
    compare_and_display(a: 3.14, b: 2.71);

```

Несколько ограничений (2/2)

```
166  fn compare_and_display_2<T>(a: T, b: T)
167  where
168      T: PartialOrd + Display
169  {
170      if a > b {
171          println!("{} is greater", a);
172      } else {
173          println!("{} is greater", b);
174      }
175  }
176
```

Пример обобщённого типа с несколькими параметрами

Также можно использовать несколько обобщённых типов для одной структуры или функции:

```
156  struct Pair<T, U> {
157      first: T,
158      second: U,
159  }
160
```

Здесь Pair<T, U> может принимать два разных типа T и U.

```
113      // пример 6
114      let pair: Pair<i32, &str> = Pair { first: 10, second: "Rust" };
115      println!("Pair: ({}, {})", pair.first, pair.second);
116
```

Список литературы

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