RUST

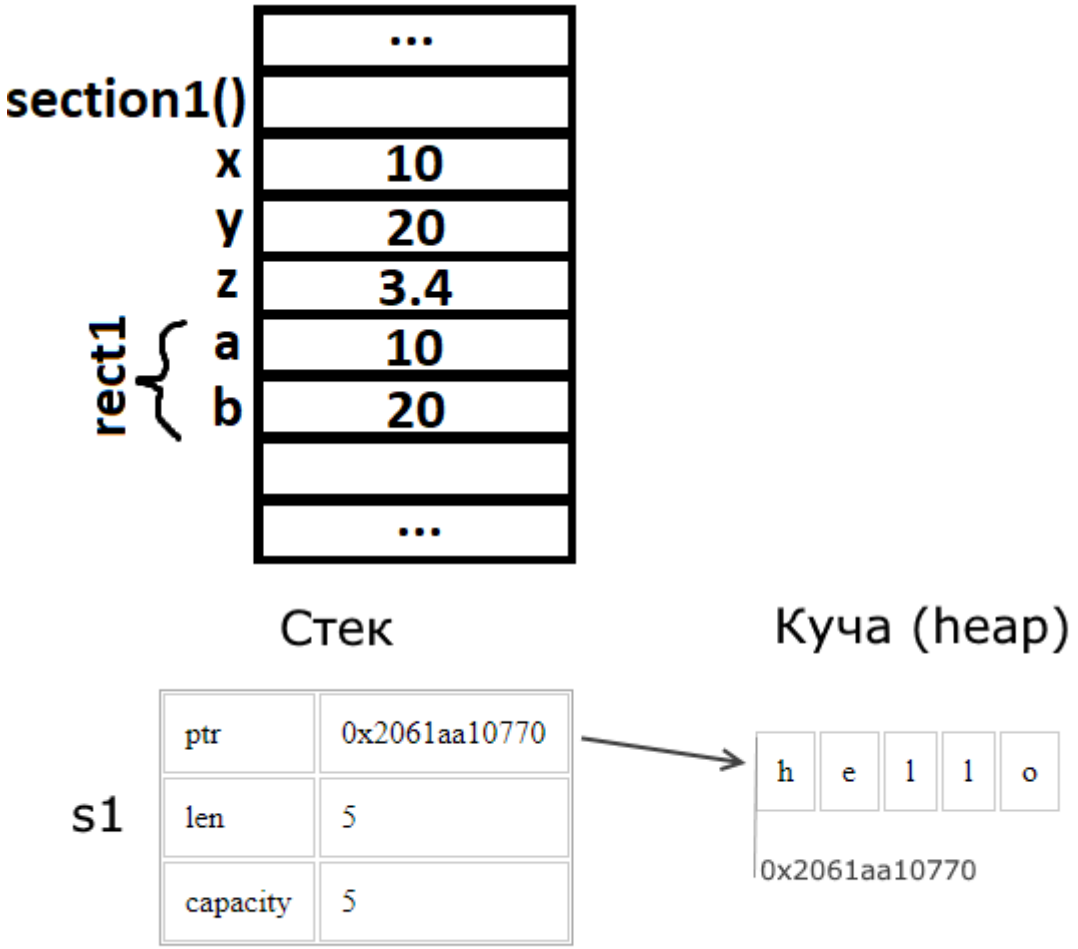
Пользовательские типы данных.

1. Стек и куча

- Стек – структура данных, хранящая локальные переменные и вызовы функций
- При вызове функции, её локальные переменные и адрес возврата помещаются на вершину стека
- Стек хранит данные с заранее известным размером памяти, в том числе: указатели, структуры, адресные переменные и т.п.

- Куча – область памяти, использующаяся для динамически выделяемых данных
- Возможны утечки памяти, не безопасна
- Медленнее, чем стек

2. Стек и куча




```rust
fn section1() {
    struct Rectangle{
        a: f32,
        b: f32
    }

    let x = 10;
    let y = 20;
    let z = 3.4;

    let rect1 = Rectangle{
        a: 10.0,
        b: 20.0,
    };

    let s1: String = "hello".to_string();
}
```

2. Концепция Владения

1) У каждого значения может быть только один «владелец»
2) Когда «владелец» выходит из области видимости, выделенная память высвобождается
3) Для передачи только значения необходимо выполнять копирование
4) Для передачи владения необходимо использовать либо оператор “=“, либо методы или функции
5) Для передачи объекта в функцию без владения, необходимо использовать ссылки
6) Для изменения объекта необходимо использовать immutable ссылки («mut»)
7) Нельзя иметь одновременно больше одной изменяемой ссылки на одно и то же значение
8) Можно иметь сколько угодно неизменяемых ссылок на объект

2. Концепция Владения

Замечания:

- Владение над примитивами не теряется при передаче в другую область видимости
- Объекты и примитивы уничтожаются при выходе из области видимости
- Передача осуществляется либо по изменяемой/не изменяемой ссылке, либо по значению, либо по скопированному значению
- Оператор “=“ для объектов выполняет перемещение

3. Пользовательские типы данных

В Rust имеется два ключевых слова для создания пользовательских типов данных – «struct» и «enum»

```
struct SomeStructure{
    some_integer: i32,
    some_double: f64,
    some_symbol: char
}

enum Numbers{
    ZERO,
    TEN = 10,
    HUNDRED = 100
}
```

В зависимости от способа определения пользовательские типы данных могут храниться как в куче, так и на стеке

4. Структуры

- Структуры в Rust имеют синтаксис, похожий на Java, Kotlin
- Поскольку структуры в Rust являются объектами, следует помнить про концепцию Владения при работе с ними
- Создание структуры в Rust обычно происходит на стеке
- Для динамического создания структур используется умный указатель Box::New()

5. Перечисления

- Перечисления в Rust ничем не отличаются от перечислений в других C-подобных языках
- Перечисления – тип данных, поэтому их можно использовать не только в качестве констант, как в языке Си, а ещё и как переменные с внятным названием.
- В основном перечисления используются для частоты кода, но встречаются и такие реализации
  https://github.com/rustkas/rust-by-example-ru/blob/master/src/custom_types/enum/testcase_linked_list.md

6. Имплементация структур

- Для поддержания парадигмы ООП в Rust имеется имплементация.
- С помощью ключевого слова «impl» можно создать методы для типов данных, как для структур, так и для перечислений
- «&self» в параметрах методов структуры указывает на то, что метод работает с объектом
- Отсутствие «&self» говорит о том, что метод ассоциированный, иными словами, статический метод

```
struct Rectangle{
    width: f64,
    height: f64
}
impl Rectangle{
    fn get_width(&self) -> f64{
        self.width
    }
    fn get_height(&self) -> f64{
        self.height
    }
    fn calc_area(&self) -> f64{
        self.height*self.width
    }
    fn calc_len_diagonal(&self) -> f64{
        (self.width*self.width + self.height*self.height).sqrt()
    }
}
```

7. Имплементация перечислений

- Создание методов для перечислений ничем не отличается от создания методов структур

```
enum Comment {
    Excellent(String),
    Good(String),
    Normal(String),
    Awful(String)
}
impl Comment {
    fn express_correctly(&self){ // вариант для обработчика ошибок
        match self{
            Comment::Excellent(e) => println!("excellent {}", e),
            Comment::Good(g) => println!("good {}", g),
            Comment::Normal(n) => println!("normal {}", n),
            Comment::Awful(a) => println!("awful {}", a)
        }
    }
}
```