

Rust

Author: Darua Shutina

Rust

23-02-07

Crate

Cargo

Cargo.toml

Переменные

Функции

Смысл точки с запятой

Макросы

`format!`

`print!` и `println!`

`eprint!` и `eprintln!`

`panic!`

Data types

If

Loop

While/for

23-02-14

Structs

String: общие черты

Ownership

Copy

Move

Пример 1: присвоение данных переменной

Пример 2: переменная как аргумент функции

Drop

Drop on move out of scope

Borrowing

Dangling references

Пример

Mutable references

Mutable vs shared references

Dereference via `*`

Меняем значение по ссылке или крадем значение?

Тут кража

Тут замена

Dereference via `.`

Copy trait

Выводы \todo

23-02-07

rustlings 🦀❤️ : <https://github.com/rust-lang/rustlings>

Crate

A crate is a compilation unit in Rust. Whenever `rustc some_file.rs` is called, `some_file.rs` is treated as the *crate file*.

A crate can be compiled into a binary or into a library. By default, `rustc` will produce a binary from a crate. This behavior can be overridden by passing the flag `--crate-type=lib`.

Cargo

Cargo is **Rust's build system and package manager**. With this tool, you'll get a repeatable build because it allows Rust packages to declare their dependencies in the file `Cargo.toml`.

Это инструмент, который позволяет билдить, запускать, тестить и фиксировать проект.

Создать новый проект:

```
1 $ cargo new new_project
2 $cd new_project
3
4 $cargo init
```

Cargo.toml

В этом файле объявляются имя, версия, сурс, чексумм и зависимости. В начале файла также добавляется версия проекта (?).

```
1 version = 3
2 [[package]]
3 name = "time"
4 version = "0.1.45"
5 source =
6 "registry+https://github.com/rust-lang/crates.ioindex"
7 checksum =
8 "1b797afad3f312d1c66a56d11d0316f916356d11158fbc6ca6389ff6bf805a"
9 dependencies = [
10 "libc",
11 "wasi",
12 "winapi 0.3.9",
13 ]
```

`checksum` - это хеш для цифровой подписи. Когда заливаешь свой пакет в репозиторий, от него формируется Криптографический Хеш и прописывается локально.

Если злоумышленник получит доступ к пакетному менеджеру и попытается подменить пакет, то не пройдет билд, так как сохраненный хеш и хеш пакета не совпадут.

Переменные

```
1 let par1: String = "aboba"; // cannot be modified
2 let mut par2: String = "abober"; // can be modified
3 let par3 = par1;
```

Функции

```
1 fn f(par1: String) -> String {
2     return format!("{}", par1);
3 }
4 fn f(par1: String) { // ==> void function
5     println!("{}", par1);
6 }
7
8 fn f(mut par1: String) {
9     // mut => переменную можно изменять внутри функции
10    println!("{}", par1);
11 }
```

Смысл точки с запятой

Если в конце строки стоит `;`, то строка превращается в statement и ничего не возвращает. Если в такой строке дописать в начале `return`, то тогда она будет что-то возвращать.

Если оставить строку без точки с запятой и без слова `return`, то строка будет что-то возвращать:

```

1 fn f(par1: String) -> String {
2     return format!("{}", par1);
3 }
4
5 fn f(par1: String) -> String {
6     format!("{}", par1)
7 }

```

Макросы

`format!`

Возвращает отформатированный текст в виде строки.

```
1 format!("the value is {var}", var = "aboba");
```

`print!` и `println!`

То же, что и `format!`, но печатает вывод в `io::stdout`.

`eprint!` и `eprintln!`

То же, что и `format!`, но печатает вывод в `io::stderr`.

`panic!`

Аналог выкидывания исключений. Мы можем кидать панику и перехватывать панику, вот класс!

```
1 panic!("this is my message");
```

Data types

- Numeric -- всевозможные числа и операции над ними

Возможные типы:

- `i8` (int 8 bit), `u8` (unsigned int 8 bit), ..., `i128`, `u128`;
- `f32`, `f64`, `0xff`;

- etc.

Если происходит переполнение, то получим `panic!` в режиме дебага и `overflow` в обычном режиме.

- `bool`
- `char 32bit`: `'a'`

Строка в расте -- это не массив чаров, а какая-то более сложная вещь

- `array`
- `tuple`
- etc

If

Фигурные скобочки обязательные

```
1 if a > 0 {
2     // do smth
3 } else {
4     // do smth
5 }
6
7 let b: i32 = if a > 0 { 1 } else { 2 };
```

Loop

```
1 loop {
2     counter += 1;
3     if counter > 42 {
4         break
5     }
6 }
7
8
9
10 let b = loop {
11     counter += 1;
12     if counter > 42 {
13         break counter * 2
14     }
15 };
16 // `loop` вернет 84, и это значение положится в `b`
17 // после второй ``}`` ставится точка с запятой, потому что присваивание -- это
    всегда statement и требует точку с запятой.
18
19
```

```

20
21 let b = 'main_loop: loop {
22     loop {
23         counter += 1;
24         if counter > 42 {
25             break 'main_loop counter * 2
26         }
27     }
28 };
29 // используем лейбл для цикла

```

While/for

```

1 let mut counter = 0;
2
3 while counter < 42 {
4     counter += 1;
5 }
6
7 for _ in 0..42 {
8     println!("we are in a while loop");
9 }

```

`while` -- это `loop` с условием. Но, в отличие от `loop`, он не может возвращать значение (`loop` может, пример выше).

23-02-14

Structs

```

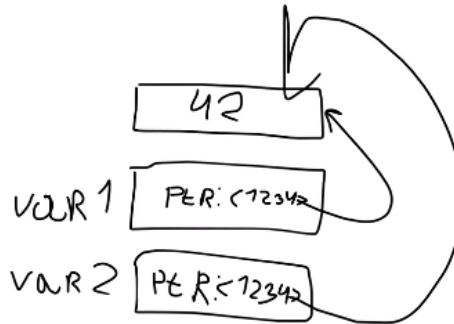
1 struct MyStruct {
2     val1: i32,
3     val2: bool,
4 }
5
6 let struct = MyStruct {
7     val1: 42,
8     val2: true,
9 }

```

```

1 struct MyStructRef {
2     ref: &i32, // ссылочная переменная
3 }
4
5 let var1 = MyStructRef {
6     ref: &42 // ссылка на значение `42`
7 }
8 let var2 = var1

```



String: общие черты

```

1 pub struct String {
2     vec: Vec<u8>,
3 }

```

`Vec<u8>` -- это не просто вектор чаров. Один чар занимает 32 бита. В строке один символ может занимать две ячейки из-за кодирования. Если будем обращаться к одной ячейке, не факт, что получим символ, который хотим.

Ownership

- [Copy](#) -- копирование
- [Move](#) -- перемещение
- [Drop](#) -- удаление данных
- [Borrowing](#) -- обращение по ссылке
 - [Dangling refs](#)
 - [Mutable refs](#)
 - [Mutable vs shared refs](#)
 - [Dereference via `*`](#)

- [Кража или замена?](#)
- [Dereference via `.`](#)
- [Copy trait](#)
- [Выводы](#)

Copy

Классическая операция `copy`. В расте не является поведением по умолчанию.

Для более сложных структур нужно явно указывать, что структура копируется:

```
1 pub trait Copy: Clone {
2     // Empty. Need to be implemented.
3 }
```

Move

В расте операция `move` является поведением по умолчанию.

Пример 1: присвоение данных переменной

```
1 let var1 = myStruct;
2 let var2 = var1;
3 println!("{}", var1); // error
```

Происходит перемещение структуры из `var1` в `var2`. Теперь данные в `var1` перестают быть доступными. Если захотим напечатать что-то из `var1`, получим ошибку:

```
1 error[E0382]: borrow of moved value: `var1`
2 let var2 = var1;
3     ---- value moved here
4 println!("{}", var1);
5             ^^^^^ value borrowed here after move
```

Чтобы жили обе переменные, можно сделать ссылочную переменную:

```
1 let var1 = myStruct;
2 let var2 = &var1;
```


Пример 2: переменная как аргумент функции

```

1 | let var1 = myStruct;
2 | myFunction(var1);
3 | println!("{}", var1);

```

```

1 | error[E0382]:
2 | my_fn(var1);
3 |     ---- value moved here
4 | println!("{}", var1);
5 |           ^^^^^ value borrowed here after moveborrow of moved value: `var1`

```

Drop

`drop` затирает данные в переменной.

```

1 | pub fn drop<T>(_x: T) {}

```

Drop on move out of scope

```

1 | let var1 = my_struct;
2 | {
3 |     let var2 = var1;
4 | }

```

При выходе из скоупа данные уже не хранятся внутри `var1`. И данные уже стерты из `var2`, потому что мы вышли из скоупа и произошел `drop`.

Borrowing

Это обычное обращение по ссылке. Отличие в том, что для `borrowing` есть `compile-check` проверки, чтобы гарантировать, что ссылка валидная.

После создания ссылки нельзя организовать перемещение, получим ошибку компилятора:

```

1 | let var1 = myStruct;
2 | let var2 = &var1;
3 | my_move(var1);
4 | println!("{}", var2);

```

```

1 error[E0505]: cannot move out of `var` because it is borrowed
2   my_move(var1);
3       ^^^^^

```

Замечание: если прямо передавать переменную `var` в `println!`?, то переменную больше нельзя использовать. Если передавать ее как `&var`, то переменную можно использовать потом.

Dangling references

Пример

```

1 fn createAndReturnRef() -> &String {
2     let s = String::from("aboba");
3     &s
4 }

```

Ошибка компиляции. При выходе из функции переменная `s` стирается. Ссылка ведет на ту часть фрейма, которая уже уничтожена:

```

1 error[E0515]: cannot return reference to local variable `s`
2   &s
3   ^^ returns reference to data owned by the current function

```

Mutable references

Вот мы в функцию передаем переменную по ссылке. Внутри функции хотим поменять переменную:

```

1 fn main() {
2     let mut var = String::from("Hello");
3     append_world(&var);
4     println!("{}", var)
5 }
6
7 fn append_world(str: &String){
8     str.push_str(" World!")
9 }

```

Получаем ошибку:

```

1 error[E0596]: cannot borrow `*str` as mutable, as it is behind a `&`
2 reference
3 str.push_str(" World!")
4 `str` is a `&` reference, so the data it refers to cannot be borrowed as
5 mutable

```

Надо явно прописать, что переданный аргумент -- это мутабельная ссылка:

```

1 fn append_world(str: &mut String){
2     str.push_str(" World!")
3 }

```

Теперь получаем другую ошибку:

```

1 error[E0308]: mismatched types
2     append_world(&var);
3     ----- ^^^^ types differ in mutability
4     |
5     arguments to this function are incorrect
6 = note: expected mutable reference `&mut String`
7          found reference `&String`

```

Проблема в том, что при вызове функции мы передаем немутабельную ссылку.

Finally, корректный код:

```

1 fn main() {
2     let mut var = String::from("Hello");
3     append_world(&mut var); // изменения тут
4     println!("{}", var)
5 }
6
7 fn append_world(str: &mut String){
8     str.push_str(" World!")
9 }

```

Mutable vs shared references

&mut – Mutable (Unique)

- Single
- No other Shared References
- Allow using in &mut parameters
- Allow using in & parameters

& - Shared

- Multiple
- No another Unique Reference
- Allow using in & parameters

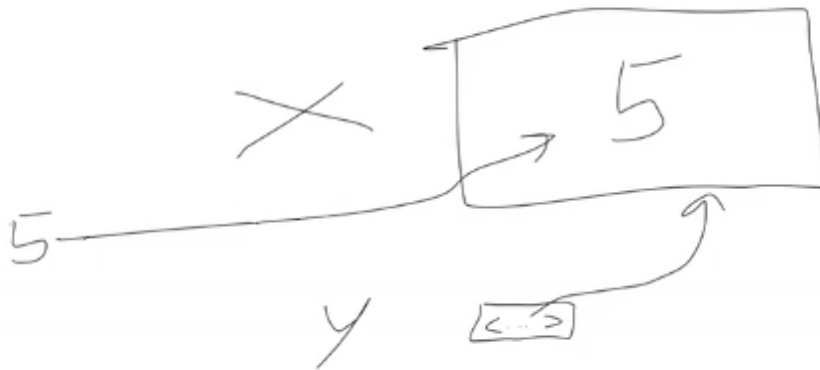
Нельзя создать mutable ссылку, если есть хотя бы одна shared ссылка, и наоборот.

Dereference via `*`

```

1 let x = 5;
2 let y = &x;
3
4 assert_eq!(5, x); // true
5 assert_eq!(5, *y); // true. переходим по ссылке `y`
6 assert_eq!(5, y); // false. `y` -- это ссылка на `5`

```



Меняем значение по ссылке или крадем значение?

Тут кража

```

1 let mut var = String::from("aboba");
2 let reference = &mut var;
3 let moved = *reference;

```

Пытаемся из `var` переместить значение в `moved`. Это кража! Так в расте делать нельзя:

```

1 error[E0507]: cannot move out of `*reference` which is behind a mutable reference
2
3 let moved = *reference;
4           ^^^^^^^^^^^
5           |
6           move occurs because `*reference` has type `String`, which does not
           implement the `Copy` trait

```

Тут замена

```

1 let mut var = String::from("aboba");
2 let reference = &mut var;
3 *reference = String::from("abober");

```

Поменяли значение в переменной `var`. Так в расте делать можно.

Dereference via `*`

```

1 let mut var = String::from("aboba");
2 let ref = &mut var;
3
4 (*ref).push_str("aboba");
5 ref.push_str("aboba");

```

Если есть ссылка на ссылку на ссылку на переменную, то, поставив одну точку, мы пройдем по всему этому пути сразу к значению переменной. Во приколу.

Copy trait

- обычное копирование
- не overloadable
- всё что `Copy`, является еще и `Clone`. Но не наоборот
- Нельзя реализовать `Copy` для `&mut` переменных (потому что `&mut` -- это уникальная ссылка)
- Не стоит реализовывать `Copy` для мутабельных структур

Выводы \todo

В плюсах очень легко выстрелить себе в ногу, и её там оторвёт нахрен.

Чтобы выстрелить в ногу в расте, надо очень хорошо прицелиться.