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 1: возвращение из функции
      - Пример 2: разные лайфтаймы
    - Mutable references
    - Mutable vs shared references
    - Dereference via `*`
    - Меняем значение по ссылке или крадем значение?
      - Тут кража
      - Тут замена
    - Dereference via `.`
  - Copy trait
- 23-02-21
  - mut переменная или mut значение?
  - Указатель vs ссылка
  - Lifetime specifiers

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 фиксить проект.

Создать новый проект:

```
$ cargo new new_project
$cd new_project

$cargo init
```

Cargo.toml

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

```
version = 3
[[package]]
name = "time"
version = "0.1.45"
source =
"registry+https://github.com/rust-lang/crates.ioindex"
checksum =
"1b797afad3f312d1c66a56d11d0316f916356d11158fbc6ca6389ff6bf805a"
dependencies = [
"libc",
"wasi",
"winapi 0.3.9",
]
```

`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 попробует подменить пакет, то не пройдет билд, так как сохраненный хеш и хеш пакета не совпадут.

Переменные

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

Функции

```rust
fn f(par1: String) -> String {
    return format!{"{}", par1};
}
fn f(par1: String) {   // <=> void function
    println!{"{}", par1};
}

fn f(mut par1: String) {
// mut => переменную можно изменять внутри функции
    println!{"{}", par1};
}
```

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:

```rust
fn f(par1: String) -> String {
    return format!("{}", par1);
}

fn f(par1: String) -> String {
    format!("{}", par1)
}
```

Макросы

`format!`

Возвращает отформатированный текст в виде строки.

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

`print!` и `println!`

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

`eprint!` и `eprintln!`

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

`panic!`

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

```rust
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

```
if a > 0 {
    // do smth
} else {
    // do smth
}

let b: i32 = if a > 0 { 1 } else { 2 };
```

Loop

```
loop {
    counter += 1;
    if counter > 42 {
        break
    }
}



let b = loop {
    counter += 1;
    if counter > 42 {
```

```
        break counter * 2
    }
};
// `loop` вернет 84, и это значение положится в `b`
// после второй `}` ставится точка с запятой, потому что присваивание -- это всегда statement и требует точку с запятой.



let b = 'main_loop: loop {
    loop {
        counter += 1;
        if counter > 42 {
            break 'main_loop counter * 2
        }
    }
};
// используем лейбл для цикла
```

While/for

```
let mut counter = 0;

while counter < 42 {
    counter += 1;
}

for _ in 0..42 {
    println!("we are in a while loop");
}
```

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

23-02-14

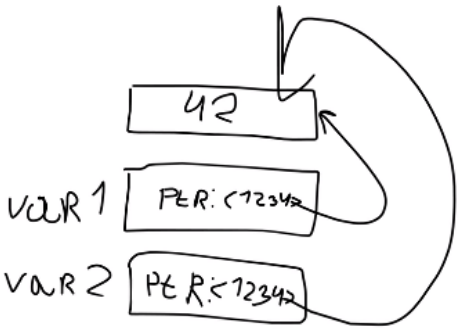
Structs

```rust
struct MyStruct {
    val1: i32,
    val2: bool,
}

let struct = MyStruct {
    val1: 42,
    val2: true,
}
```

```rust
struct MyStructRef {
    ref: &i32,  // ссылочная переменная
}

let var1 = MyStructRef {
    ref: &42  // ссылка на значение `42`
}
let var2 = var1
```



String: общие черты

```rust
pub struct String {
    vec: Vec<u8>,
}
```

`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

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

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

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

Move

В расте операция `move` является поведением по умолчанию.

Пример 1: присвоение данных переменной

```
let var1 = myStruct;
let var2 = var1;
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:

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

Чтобы жили обе переменные, можно сделать ссылочную переменную:

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

Пример 2: переменная как аргумент функции

```
let var1 = myStruct;
myFunction(var1);
println!("{}", var1);
```

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

Drop

`drop` затирает данные в переменной.

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

Drop on move out of scope

```
let var1 = my_struct;
{
   let var2 = var1;
}
```

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:

```
let var1 = myStruct;
let var2 = &var1;
my_move(var1);
println!("{}", var2);
```

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

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: возвращение из функции

```
fn createAndReturnRef() -> &String {
    let s = String::from("aboba");
    &s
}
```

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

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

Пример 2: разные лайфтаймы

```
let mut s_ptr: &String;
{
    let s = String::from("aboba");
    s_ptr = &s
}
println!("{}", *s_ptr)
```

Ошибка компиляции. Обращаемся к данным, которые жили в другом скоупе и к моменту обращения уже умерли:

```
error[E0597]: `s` does not live long enough
s_ptr = &s
        ^^ borrowed value does not live long enough

`s` dropped here while still borrowed
println!("{}", *s_ptr)
               ------ borrow later used here
```

Mutable references

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

```
fn main() {
    let mut var = String::from("Hello");
    append_world(&var);
    println!("{}", var)
}

fn append_world(str: &String){
   str.push_str(" World!")
}
```

Получаем ошибку:

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

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

```rust
fn append_world(str: &mut String){
   str.push_str(" World!")
}
```

Теперь получаем другую ошибку:

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

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

Finally, корректный код:

```rust
fn main() {
        let mut var = String::from("Hello");
    append_world(&mut var);  // изменения тут
    println!("{}", var)
}

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

Mutable vs shared references

&mut – Mutable (Unique)

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

& - Shared

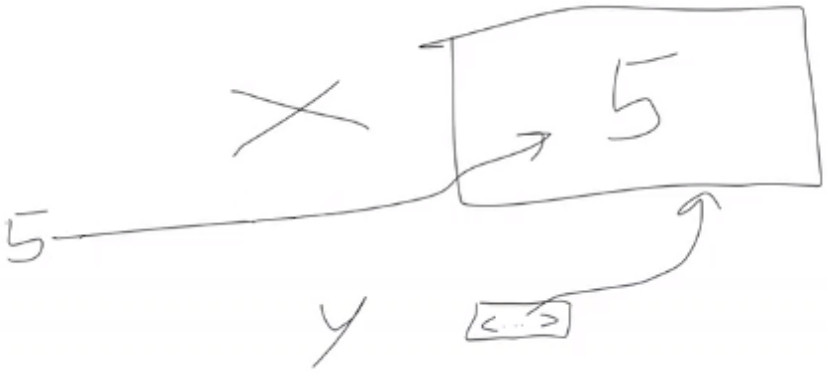
- Multiple
- No another Unique Reference
- Allow using in & parameters

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

Dereference via `*`

```
let x = 5;
let y = &x;

assert_eq!(5, x);  // true
assert_eq!(5, *y); // true. переходим по ссылке `y`
assert_eq!(5, y);  // false. `y` -- это ссылка на `5`
```



Меняем значение по ссылке или крадем значение?

Тут кража

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

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

```
error[E0507]: cannot move out of `*reference` which is behind a mutable reference

let moved = *reference;
            ^^^^^^^^^^
            |
            move occurs because `*reference` has type `String`, which does not
implement the `Copy` trait
```

Тут замена

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

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

Dereference via `.`

```
let mut var = String::from("aboba");
let ref = &mut var;

(*ref).push_str("aboba");
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 прикол.

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

23-02-21

mut переменная или mut значение?

```
fn f(mut mutable: &i32, immutable: &i32, mut_ref: &mut i32) {
    mutable = &666;
    println!("{}", *immutable);
    *mut_ref = 666;
}
```

Если переменная `mut`, то в ней можно поменять значение. Например, присвоить ссылку на другой объект.

Если в перменной хранится ссылка на мутабельный объект (`&mut`), по этой ссылке можно поменять значение объекта.

Если в переменной хранится ссылка на обычный объект (`&`), то, грубо говоря, данные объекта открыты только для чтения.

	Can Move	Can Borrow	Can Borrow Mut	Can Reassign
let mut str = MyStruct { int: 42 };	True	True	True	True
let shared_ref = &str;	False	True	False	False
let mut_ref = &mut str;	False	False	False	False
drop(str);	False	False	False	True

Указатель vs ссылка

Можно создать raw pointer. Это указатель, который привязывается к определенной области памяти. Раст не проверяет, валиден ли указатель.

В отличие от указателя, ссылка привязывается к объекту, а не к области памяти.

Lifetime specifiers

Есть как минимум два способа получить dangling reference. Чтобы такого не было, можно явно указать, из какого скоупа переменная. Синтаксис: `'x`.

Пример:

```
fn main() {
    let str1: String = String::from("aboba");
    let str2: String = String::from("abober");

    println!("longest string = '{}'", find_longest_string(&str1, &str2))
}

fn find_longest_string<'a>(x: &'a String, y: &'a String) -> &'a String {
    if x.len() > y.len() { x } else { y }
}
```

Если убрать лайфтаймы, то код не скомпилится, потому что функция возвращает ссылку на умирающий объект.