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
    - Lifetime bounds

- 23-02-28
  - Struct Update Syntax
  - Tuples
    - Синтаксис
    - Проблема тупла и Borrowing
  - TupleStruct
  - Unit
    - StructUnit
  - Bang (!)
  - Реализация методов у структур
    - Associated functions
    - new
  - Traits
    - Пример использования `trait`
    - Трейты для (простых) типов
    - Default
    - Clone
    - Copy
  - Array
  - Vector
  - Generics
    - Структуры и метоы
    - Функции
      - Multiple generics
      - Multiple gerenics in impl
    - Generic bounds
      - Add
      - ToString
      - Display
      - Multiple Bounds
        - Where clause
- 23-03-07
  - Option
  - How to move data to heap?
  - Linked List: пример плохого кода
  - Box (unique ownership)
    - Взятие ссылки на значение в боксе
  - RC (shared ownership)
  - Модуль `std::cell`
    - Cell
    - RefCell
    - Circuler references
    - Утечка памяти из-за `Rc`
      - `Rc::downgrade()`
      - `Rc::upgrade()`

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

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

Функции

```
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.


```

```
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).

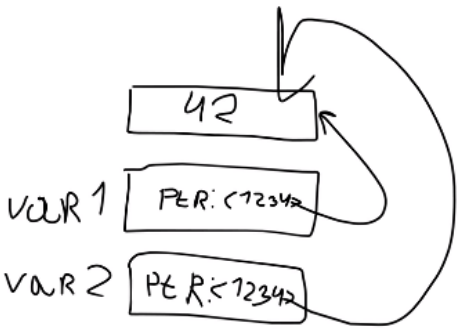
23-02-14

Structs

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

```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

```rust
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

```rust
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:

```
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

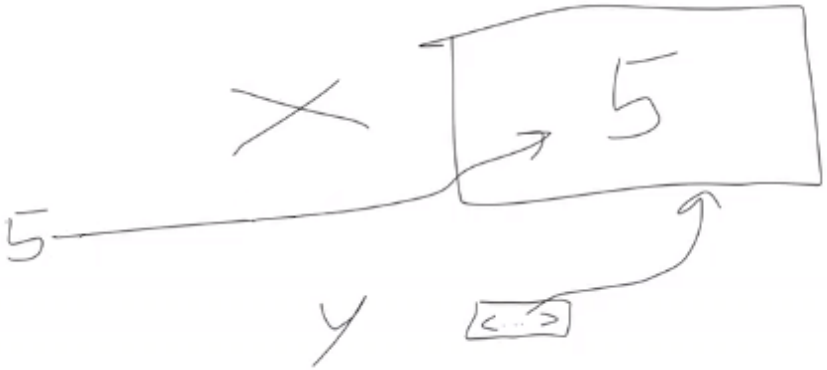
& - Shared
- Multiple
- No another Unique Reference
- Allow using in & parameters

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

Dereference via `*`

```rust
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.

Lifetime bounds

`b: a` означает, что лайфтайм `b` входит в лайфтайм `a`. Пример:

```rust
fn get_reference_tuple<'a, 'b: 'a>(left: &'a MyStruct, right: &'b MyStruct)
    -> (&'a i32, &'a i32)
{
    (&left.value1, &right.value1)
}
```

23-02-28

Struct Update Syntax

```rust
struct MyBigStruct {
    value1: i32,
    value2: i32,
    value3: i32,
    value4: i32,
    value5: i32,
    value6: i32,
    value7: String,
}

fn main() {
    let var1 = MyBigStruct{
        value1: 1, value2: 2, value3: 3,
        value4: 4, value5: 5, value6: 6,
        value7: String::from("Hello World"),
    };
    let var2 = MyBigStruct{
        value1: 42,
        ..var1 // all elems except `value1` are moved
               // from `var1` to `var2`
    };

    println!("{}", var1.value1);
    println!("{}", var2.value1);
    println!("{}", var1.value7); // does not compile
}
```

```
error[E0382]: borrow of moved value: `var1.value7`
    println!("{}", var1.value7);
                   ^^^^^^^^^^^ value borrowed here after move
```

Tuples

Синтаксис

```rust
let tuple: (i32, bool, String) = (42, true, String::from("Hi"));
let number = tuple.0;
let (num, bool, str) = tuple; // move for string, copy for others
```

Проблема тупла и Borrowing

```rust
struct StrWithTuple {
    field: (i32, bool, String),
}

fn main() {
    let tuple: (i32, bool, String) = (42, true, String::from("Hi"));

    let (num, bool, str) = tuple;
    let newStruct = StrWithTuple { field: tuple };
    // does not compile. A string `Hi` was moved into `str`.
}
```

Не скомпилится из-за строки 9. Мы пытаемся создать структуру из `tuple`, но `Hi` была перемещена из `tuple` в переменную `str`.

Корректный код:

```rust
fn main() {
    let tuple: (i32, bool, String) = (42, true, String::from("Hi"));

    let (num, bool, ref str) = tuple; // keyword `ref`
    let newStruct = StrWithTuple { field: tuple };


    println!("{}", str); // does not compile.
    // Data was moved into `newStruct.field`,
    // and reference stored in `str` is not valid anymore.
}
```

TupleStruct

Как обычная структура, но поля без имен. Имеет смысл использовать, если поля разных типов (тогда можно догадаться, какое поле что означает).

```rust
struct TupleStruct(i32, i32);
let point = TupleStruct(42, 666);
```

Unit

По конструкции, `Unit` -- это пустой тупл. Штука, которая не хранит никаких данных.

```rust
let unit = ();
```

Если не нужно создавать структуру для хранения данных (i.e. данных нет), можно использовать `Unit`, который занимает 0 байтов памяти.

StructUnit

Частный случай `Unit`. Пустая структура, на которую можно потом навесить метаданные с помощью аттрибутов.

```rust
struct UnitStruct;

let unit_struct = UnitStruct;

let unit: () = while false {};
// в расте все возвращает значение, просто иногда это `Unit`.
// например, цикл `while`.
```

Bang (!)

Тип `Bang`. Оозначается как `!`. Означает, что текущий код никогда не будет достижим. Пример -- бесконечный `loop`, который как раз вернет `!`:

```rust
let bang_loop: ! = loop {};
let bang_panic: ! = panic!();

let bang_unimpl: ! = unimplemented!();
// макрос; используется, если еще нет реализации для метода.
```

Реализация методов у структур

Методы реализуются в блоке с ключевым словом `impl <struct_name>`. В качестве аргумента передается ссылка на сам объект -- `&self`:

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

impl MyStruct {
    fn print_me(&self) {
        println!("val1 = {}", self.val1);
    }
}

fn main() {
    let aboba = MyStruct {
        val1: 42,
    };

```

```
    MyStruct::print_me(&aboba);
    aboba.print_me();
}
```

`impl` блоков может быть несколько, но методы не должны повторяться. `impl` блок должен быть в том же кейте, что и определение структуры.

Associated functions

Ассоциированная функция -- любая функция в блоке `impl`. Её можно вызвать только через синтаксис `<struct_name>::`.

Метод -- это ассоциированная функция, в которую в качестве аргумента передается `&self`. Метод можно вызвать любым способом.

new

Общая договоренность, что так называется конструктор.

```rust
impl MyStruct {
    fn new(value1: i32, value2: bool) -> Self {
        Self { value1, value2 }
    }

    // если на объект кто-то в коде уже ссылается,
    // метод вызвать не удастся.
    fn update_me(&mut self, new_value1: i32) {
        self.value1 = new_value1;
    }

    fn move_me(self, new_owner: &mut (i32, MyStruct)) {
        new_owner.1 = self;
    }

    // деструктор. Параметр передается без ссылки (с помощью move), поэтому,
    // когда выходим из тела функции, переданный параметр умирает, память чистится.
    fn kill(self) {}
}
```

```rust
impl MyStruct {
```

```rust
    fn any_static_method() {}

    fn new(value1: i32, value2: bool) -> Self {
        Self { value1, value2 }
    }

    fn print_me(&self) {
        println!("{}{}", self.value1, self.value2);
    }

    fn update_me(&mut self, new_value1: i32) {
        self.value1 = new_value1;
    }

    fn move_me(self, new_owner: &mut (i32, MyStruct)) {
        new_owner.1 = self;
    }

    fn kill(self) {}
}

```

Traits

Что-то похожее на интерфейсы, но с возможностью дефолтной реализации. Трейты используются для:

- дефолтной реализации методов,
- статического полиморфизма,
- динамического полиморфизма.

Блок `trait` -- это объявление того, что способен сделать объект.

Блок `impl` -- реализация объявленных методов для какой-то определенной структуры.

```rust
trait MyTrait {
    fn print_me(&self);
    // здесь же может быть дефолтная реализация
}

// реализация для структуры
impl MyTrait for MyStruct {
    fn print_me(&self) {
        println!("{}{}", self.value1, self.value2);
    }
}
```

```
12
13 // реализация для ссылки на структуру
14 impl MyTrait for &MyStruct {
15     fn do (self) {
16         let it_is_reference: &MyStruct = self;
17     }
18 }
```

Реализацию для ссылки на структуру необходимо вызывать явно, через `::`. Иначе будет вызываться реализация для структуры (по дефолту).

Пример использования `trait`

```
1  struct MyStruct { val1: i32, }
2
3  trait MyTrait {
4      fn print_me(&self) {
5          println!("{}", self.get_val1());
6      }
7      fn get_val1(&self) -> i32;
8  }
9
10 impl MyTrait for MyStruct {
11     fn get_val1(&self) -> i32 { self.val1 }
12 }
13
14 fn main() {
15     let my_struct = MyStruct { val1: 42, };
16     my_struct.print_me();
17 }
```

Трейты для (простых) типов

```rust
impl MyTrait for i32 {
    fn get_value1(&self) -> String {
        self.to_string()
    }

    fn get_value2(&self) -> String {
        self.to_string()
    }
}

let str: String = 42.get_value1();
```

Default

```rust
#[derive(Default)]
struct MyLuckyStruct {
    value1: i32,
    value2: bool,
}

let default_str = MyLuckyStruct::default();
```

Можно определить вручную:

```rust
impl Default for MyStruct {
    fn default() -> Self {
        Self { value1: 0, value2: false }
    }
}
```

Clone

```rust
#[derive(Clone, Default)]
struct MyLuckyStruct {
    value1: i32,
    value2: bool
}

let default_str = MyLuckyStruct::default();
let another_str = default_str.clone();
println!("{}", default_str.value1);
println!("{}", another_str.value1);
```

Copy

Требует наличие `clone`.

```
#[derive(Copy, Clone, Default)]
struct MyLuckyStruct {
    value1: i32,
    value2: bool
}
let default_str = MyLuckyStruct::default();
let another_str = default_str;
println!("{}", default_str.value1);
println!("{}", another_str.value1);
```

Array

Не лежит на куче, если это явно не прописано. Живет на стеке или в структуре.

```
let array: [i32; 5] = [1, 2, 3, 4, 5];

let var1 = array[4];

let var2 = array[42];
// error: this operation will panic at runtime

struct MyStructWithArray {
    array: [i32; 5],
}
```

Vector

Динамический массив. Реализуется через макрос `vec!` -- синтаксический сахар.

```rust
let mut vec: Vec<i32> = vec![1, 2, 3, 4, 5];
vec.push(42);
let len = vec.len();
println!("{}", vec[3]);
```

Generics

Структуры и метоы

```rust
struct MyPair<T> {
    left: T,
    right: T,
}

impl<T> MyPair<T> {
    fn do_something_with_type(&self) {
        println!("{}", self.left + self.right)
    }
}

let my_small = MyPair::<i8> { left: 42, right: 1 };
let my_big = MyPair::<i128> {
    left: 424242121231231231231233123123 12,
    right: 41212342364236487263482638746 21,
};
```

Функции

```rust
fn generic_function<T>(p1: T, p2: T) -> MyPair<T> {
    MyPair::<T>{
        left: p1,
        right: p2
    }
}
generic_function::<i32>(42, 1);
generic_function(42, 1);
```

Multiple generics

```rust
fn generic_function<T, V>(p1: T, p2: T, other: V) -> MyPair<T> {
    MyPair::<T>{
        left: p1,
        right: p2
    }
}
generic_function::<i32, bool>(42, 1, true);
generic_function(42, 1, true);
```

Multiple gerenics in impl

```rust
impl<T> MyPair<T> {
    fn new<V>(p1: T, p2: T, other: V) -> Self {
        Self{
            left: p1,
            right: p2
        }
    }
}
let pair = MyPair::<i32>::new::<bool>(42, 1, true);
let pair = MyPair::new(42, 1, true);
```

Generic bounds

Add

```rust
impl<T: Add> MyPair<T> {
    fn do_summ(self)  {
        let sum = self.left + self.right;
    }
}
```

ToString

```rust
trait ToString {
    fn to_string(&self) -> String;
}

impl<T: ToString> MyPair<T> {
    fn print(&self) {
        println!("{}{}", self.left.to_string(), self.right.to_string())
    }
}
```

Display

```rust
impl<T: Display> MyPair<T> {
    fn print(&self) {
        println!("{}{}", self.left, self.right)
    }
}

pub trait Display {
    #[stable(feature = "rust1", since = "1.0.0")]
    fn fmt(&self, f: &mut Formatter<'_>) -> Result;
}
```

Multiple Bounds

```rust
fn trace<T: Display + Debug>(to_trace: T, verbose: bool) {
    println!("Display: `{}`", to_trace);
    if verbose {
        println!("Debug: `{:?}`", to_trace);
    }
}

#[derive(Debug)]
struct MyPair<T> { left: T, right: T, }
```

Вывод получается такой:

```
> Display: `(42, 1)`
> Debug: `MyPair { left: 42, right: 1 }`
```

Where clause

Для лучшей читаемости можно параметризированный тип определить после `where`:

```
fn trace_multiple<T: Display + Debug, V: Display + Debug>(to_trace1: T,
to_trace2: V) {…

//     ^
//    |||
//     v

fn trace_multiple<T, V>(to_trace1: T, to_trace2: V)
    where
        T: Display + Debug,
        V: Display + Debug {
```

23-03-07

Option

Аналог `null` в расте.

```
fn main() {
    let mut var: Option<i32> = Option::None;
    assert_eq!(true, var.is_none());

    var = Option::Some(42);
    assert_eq!(true, var.is_some());

    print!("{}", var.unwrap());
    // в данном случае, происходит move

    let reference: &i32 = var.as_ref().unwrap();
    // `as_ref` возвращает ссылку на объект
}
```

How to move data to heap?

Так никто не делает:

```
unsafe {
    let layout = Layout::new::<u16>();
    let ptr = alloc(layout);

    if ptr.is_null() { // не  получилось выделить память для кучи
        handle_alloc_error(layout);
    }

    // кладем значение на кучу
    *(ptr as *mut u16) = 42;
    assert_eq!(*(ptr as *mut u16), 42);

    dealloc(ptr, layout);
}
```

Ну, на самом деле, делает, но такой способ не самый безопасный: нужно не забывать про `dealloc`, следить за указателем `ptr`.

В расте есть концепция *OBRM -- Ownership Based Resource Management* -- аналог RAII. Общая идея в том, что объект живет между вызывами конструктора и деструктора.

Вот объект создается на куче. Используется определенное количество ресурсов, на стек кладется сырой указатель. Когда объект умирает, сырой указатель убирается со стека, а занятые ресурсы освобождаются.

Linked List: пример плохого кода

```
struct Node<'a> {
    value: i32,
    next: Option<&'a Node<'a>>
}


fn main() {
    let mut node1 = Node { value: 1, next: Option::None };
    {
        let mut node2 = Node { value: 2, next: Option::None };
        {
            let node3 = Node { value: 3, next: Option::None };

            node2.next = Some(&node3);
            node1.next = Some(&node2);

```

```
17              println!("{}", node1.next.unwrap().next.unwrap().value);
18              node1.next = None;
19          }
20
21      println!("{}", node1.value) // does not compile
22      }
23  }
```

```
error[E0597]: `node3` does not live long enough
   |
14 |              node2.next = Some(&node3);
   |                                ^^^^^^ borrowed value does not live long
enough
...
   |          - `node3` dropped here while still borrowed
20 |      println!("{}", node1.value) // does not compile
   |                     ----------- borrow later used here
```

Все ноды получают одинаковый лайфтайм, причем наименьший, поэтому все три ноды умирают внутри третьего скоупа. В строке 21 мы обращаемся к мертвой ноде.

Box (unique ownership)

Структура вида `Box<type>`. Внутри бокса хранится ссылка объект, живущий на куче. Создается вызовом функции `Box::new(<object>)`.

`Box` -- это как `smart_ptr`. Кто владеет боксом, тот владеет объектом.

```
struct Node {
    value: i32,
    next: Option<Box<Node>>,
}
```

Теперь нода владеет следующей нодой. Лайфтаймы у нод получаются вложенные, а не равные.

Удаление последнего элемента не влияет на остальные ноды. Но удаление ноды N в середине листа влечет за собой удаление последующих нод, потому что N (скажем, косвенно) владеет всеми последующими нодами.

Полный пример кода:

```
struct Node {
    value: i32,
    next: Option<Box<Node>>,
}

```

```rust

fn main() {
    let mut root = Box::new(Node {
        value: 1,
        next: Some(
            Box::new(
                Node {
                    value: 2,
                    next: Some(Box::new(Node {
                        value: 3,
                        next: None,
                    })),
                })),
    });

    println!("{}", root.next.unwrap().next.unwrap().value);

    root.next = None;
    println!("{}", root.value);
}
```

Взятие ссылки на значение в боксе

Функции `borrow()` и `borrow_mut()`.

```rust
use std::borrow::Borrow;
use std::borrow::BorrowMut;

struct MyStruct {
    value1: i32,
}

fn main() {
    let mut boxed = Box::new(MyStruct { value1: 42 });

    //let reference = boxed.borrow(); // does not compile
                                      // type must be known at this point
    let reference: &MyStruct = boxed.borrow();
    println!("{}", reference.value1);

    let mut_reference: &mut MyStruct = boxed.borrow_mut();
    mut_reference.value1 = 123;
}
```

RC (shared ownership)

`Rc` = reference counter. Можно сделать несколько указателей на один и тот же объект (живущий на куче), и каждый из указателей владеет этим объектом.

`Rc` -- это как `shared_ptr`. Объект живет, пока есть хотя бы один `Rc`.

Модифицировать данные внутри `Rc` нельзя.

Пример использования `Rc`:

```rust
use std::rc::Rc;

struct MyStruct {
    value: i32,
}


fn main() {
    let owner1 = Rc::new(MyStruct { value: 42 });
    let owner2: Rc<MyStruct> = owner1.clone();
    let owner3: Rc<MyStruct> = owner2.clone();

    println!("{} {} {}", owner1.value, owner2.value, owner3.value);
}
```

Модуль `std::cell`

Компилятор позволяет иметь одновременно либо несколько немутабельных ссылок (`&`) на объект, либо только одну мутабельную ссылку (`&mut`). Такая мутабельность называется `inherited mutability`. Но это не всегда удобно.

Контейнеры `Cell` и `RefCell` позволяют изменять объект, даже если на него ссылаются несколько немутабельных ссылок. Такая мутабельность называется `interior mutability`. Мутабельные ссылки получают с помощью метода `borrow()`.

Cell

Внутри контейнера хранится сам объект. Умеет мувать значение в контейнер и из него.

```rust
use std::cell::Cell;
use std::borrow::Borrow;

fn main() {
    let cell = Cell::new(42);
    let ref1: &Cell<i32> = cell.borrow();
    let ref2: &Cell<i32> = cell.borrow();
    println!("{} {}", ref1.get(), ref2.get());
    ref1.replace(123);
    println!("{} {}", ref1.get(), ref2.get());
}
```

Функция `Cell::get()` возвращает копию значения. Работает только на `Copy` объектах.

Функция `Cell::take()` меняет старое значение на дефолтное и возвращает старое значение. Работает только на `Default` объектах, внутри вызывает функцию `Default::default()`.

Функция `Cell::replace()` меняет старое значение на новое и возвращает старое значение.

RefCell

В отличие от простого `Cell`, внутри `RefCell` хранится именно ссылка на структуру.

```rust
use std::cell::Ref;
use std::cell::RefMut;
use std::cell::RefCell;

struct MyStruct {
    value: i32,
}


fn main() {
    let cell = RefCell::new(MyStruct { value: 42 });
    {
        // немутабельная ссылка на `cell`
        let ref1: Ref<MyStruct> = cell.borrow();
        println!("{}", ref1.value); // > 42
    }
    {
        // мутабельная ссылка на `cell`
        let mut ref_mut: RefMut<MyStruct> = cell.borrow_mut();
        ref_mut.value = 123;
```

```
    }
    let ref2: Ref<MyStruct> = cell.borrow();
    println!("{}", ref2.value); // > 123
}
```

Компилятор проверяет, существуют ли одновременно и мутабельные, и немутабельные ссылки. Если убрать вложенные скоупы, код не скомпилится из-за проверок компилятора.

Circuler references

```
use std::cell::Cell;
use std::rc::Rc;

struct NodeLeft {
    right: Cell<Option<Rc<NodeRight>>>,
}

struct NodeRight {
    left: Cell<Option<Rc<NodeLeft>>>,
}


fn main() {
    let node_left = Rc::new(NodeLeft { right: Cell::new(None) });
    let node_right = Rc::new(NodeRight {
        left: Cell::new(Some(node_left.clone()))
    });
    let e = node_left.right.replace(Some(node_right.clone()));
}
```

Создаем левую ноду. Создаем правую ноду с ссылкой на левую. Потом через метод `replace()` сохраняем внутри левой ноды ссылку на правую.

Без `Cell` обойтись нельзя, потому что нам нужны мутабельные ссылки.

Утечка памяти из-за `Rc`

В коде выше два раза использовалась функция `clone()`. Когда `node_left` и `node_right` будут умирать, сами объекты не умрут, потому что все еще существуют ссылки на них.

`Rc::downgrade()`

Чтобы избежать эту проблему, используется `Weak` ссылка. Получить ее из `Rc` можно с помощью `Rc::downgrade()`.

```rust
use core::cell::Cell;
use std::rc::Rc;
use std::rc::Weak;

struct NodeLeft {
    right: Cell<Option<Rc<NodeRight>>>,
}
struct NodeRight {
    left: Cell<Option<Weak<NodeLeft>>>,
}


fn main() {
    let node_left = Rc::new(NodeLeft { right: Cell::new(None) });
    let node_right = Rc::new(NodeRight {
        left: Cell::new(Some(Rc::downgrade(&node_left)))
    });
    let e = node_left.right.replace(Some(node_right.clone()));
}
```

`Rc::upgrade()`

Получить `Rc` ссылку из `Weak` ссылки:

```rust
use std::rc::Rc;

fn main() {
    let five = Rc::new(5);
    let weak_five = Rc::downgrade(&five);

    // create a new Rc reference
    let strong_five: Option<Rc<_>> = weak_five.upgrade();

    // check if modification succeeded
    assert!(strong_five.is_some());

    drop(strong_five);
    drop(five);

    assert!(weak_five.upgrade().is_none());
}
```