

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 методы

Функции

Multiple generics

Multiple generics 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

Circular references

Утечка памяти из-за `Rc``Rc::downgrade()``Rc::upgrade()`

23-03-14 \todo

23-03-21

Модули или Как компануется приложение раста

Package

Crates

Rustc entity

Теперь про модули

Создание
 pub
 pub(super) \TODO
 use
 Импорт крейтов
 Линковка в стиле C

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 = "aboba");
```

`print!` и `println!`

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

`eprint!` и `eprintln!`

То же, что и `format!`, но печатают 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

```

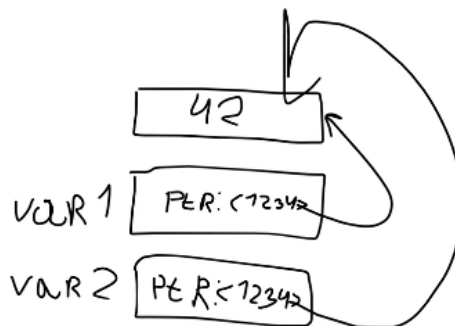
1 struct MyStruct {
2     val1: i32,
3     val2: bool,
4 }
5
6 let struct = MyStruct {
7     val1: 42,
8     val2: true,
9 }

```

```

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: возвращение из функции

```
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
```

Пример 2: разные лайфтаймы

```

1 let mut s_ptr: &String;
2 {
3     let s = String::from("aboba");
4     s_ptr = &s
5 }
6 println!("{}", *s_ptr)

```

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

```

1 error[E0597]: `s` does not live long enough
2 s_ptr = &s
3     ^^ borrowed value does not live long enough
4
5 `s` dropped here while still borrowed
6 println!("{}", *s_ptr)
7     ----- borrow later used here

```

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

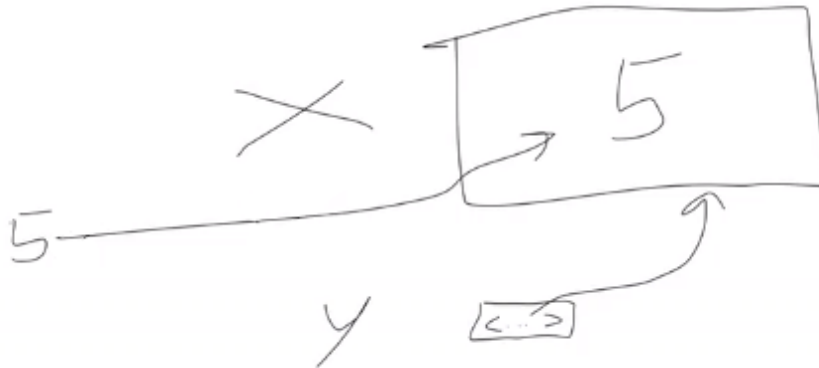
Нельзя создать mutable ссылку, если есть хотя бы одна shared ссылка, и наоборот.

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

23-02-21

mut переменная или mut значение?

```
1 fn f(mut mutable: &i32, immutable: &i32, mut_ref: &mut i32) {
2     mutable = &666;
3     println!("{}", *immutable);
4     *mut_ref = 666;
5 }
```

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

Если в переменной 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
<code>let mut str = MyStruct { int: 42 };</code>	True	True	True	True
<code>let shared_ref = &str;</code>	False	True	False	False
<code>let mut_ref = &mut str;</code>	False	False	False	False
<code>drop(str);</code>	False	False	False	True

Указатель vs ссылка

Можно создать raw pointer. Это указатель, который привязывается к определенной области памяти. Rust 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:

```

1 fn main() {
2     let str1: String = String::from("aboba");
3     let str2: String = String::from("abober");
4
5     println!("longest string = '{}'", find_longest_string(&str1, &str2))
6 }
7
8 fn find_longest_string<'a>(x: &'a String, y: &'a String) -> &'a String {
9     if x.len() > y.len() { x } else { y }
10 }
```

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

Lifetime bounds

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

```
1 fn get_reference_tuple<'a, 'b: 'a>(left: &'a MyStruct, right: &'b MyStruct)
2     -> (&'a i32, &'a i32)
3 {
4     (&left.value1, &right.value1)
5 }
```

23-02-28

Struct Update Syntax

```
1 struct MyBigStruct {
2     value1: i32,
3     value2: i32,
4     value3: i32,
5     value4: i32,
6     value5: i32,
7     value6: i32,
8     value7: String,
9 }
10
11 fn main() {
12     let var1 = MyBigStruct{
13         value1: 1, value2: 2, value3: 3,
14         value4: 4, value5: 5, value6: 6,
15         value7: String::from("Hello World"),
16     };
17     let var2 = MyBigStruct{
18         value1: 42,
19         ..var1 // all elems except `value1` are moved
20                // from `var1` to `var2`
21     };
22
23     println!("{}", var1.value1);
24     println!("{}", var2.value1);
25     println!("{}", var1.value7); // does not compile
26 }
```

```

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

```

Tuples

Синтаксис

```

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

```

Проблема тупла и Borrowing

```

1 struct StrWithTuple {
2     field: (i32, bool, String),
3 }
4
5 fn main() {
6     let tuple: (i32, bool, String) = (42, true, String::from("Hi"));
7
8     let (num, bool, str) = tuple;
9     let newStruct = StrWithTuple { field: tuple };
10    // does not compile. A string `Hi` was moved into `str`.
11 }

```

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

Корректный код:

```

1 fn main() {
2     let tuple: (i32, bool, String) = (42, true, String::from("Hi"));
3
4     let (num, bool, ref str) = tuple; // keyword `ref`
5     let newStruct = StrWithTuple { field: tuple };
6
7
8     println!("{}", str); // does not compile.
9     // Data was moved into `newStruct.field`,
10    // and reference stored in `str` is not valid anymore.
11 }

```

TupleStruct

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

```

1 struct TupleStruct(i32, i32);
2 let point = TupleStruct(42, 666);

```

Unit

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

```

1 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 атрибутов.

```

1 struct UnitStruct;
2
3 let unit_struct = UnitStruct;
4
5 let unit: () = while false {};
6 // в расте все возвращает значение, просто иногда это `Unit`.
7 // например, цикл `while`.

```

Bang (!)

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

```

1 let bang_loop: ! = loop {};
2 let bang_panic: ! = panic!();
3
4 let bang_unimpl: ! = unimplemented!();
5 // 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`:

```

1 struct MyStruct {
2     val1: i32,
3 }
4
5 impl MyStruct {
6     fn print_me(&self) {
7         println!("val1 = {}", self.val1);
8     }
9 }
10
11 fn main() {
12     let aboba = MyStruct {
13         val1: 42,
14     };
15

```

```

16     MyStruct::print_me(&aboba);
17     aboba.print_me();
18 }

```

`impl` блоков может быть несколько, но методы не должны повторяться. `impl` блок должен быть в том же кейсе, 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.

```

1  impl MyStruct {
2      fn new(value1: i32, value2: bool) -> Self {
3          Self { value1, value2 }
4      }
5
6      // если на объект кто-то в коде уже ссылается,
7      // метод вызвать не удастся.
8      fn update_me(&mut self, new_value1: i32) {
9          self.value1 = new_value1;
10     }
11
12     fn move_me(self, new_owner: &mut (i32, MyStruct)) {
13         new_owner.1 = self;
14     }
15
16     // деструктор. Параметр передается без ссылки (с помощью move), поэтому,
17     // когда выходим из тела функции, переданный параметр умирает, память
18     // чистится.
19     fn kill(self) {}
20 }

```

```

1  impl MyStruct {

```

```

2     fn any_static_method() {}
3
4     fn new(value1: i32, value2: bool) -> Self {
5         Self { value1, value2 }
6     }
7
8     fn print_me(&self) {
9         println!("{}", self.value1, self.value2);
10    }
11
12    fn update_me(&mut self, new_value1: i32) {
13        self.value1 = new_value1;
14    }
15
16    fn move_me(self, new_owner: &mut (i32, MyStruct)) {
17        new_owner.1 = self;
18    }
19
20    fn kill(self) {}
21 }
22

```

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.

```

1 trait MyTrait {
2     fn print_me(&self);
3     // здесь же может быть дефолтная реализация
4 }
5
6 // реализация для структуры
7 impl MyTrait for MyStruct {
8     fn print_me(&self) {
9         println!("{}", self.value1, self.value2);
10    }
11 }

```

```

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

```

1  impl MyTrait for i32 {
2      fn get_value1(&self) -> String {
3          self.to_string()
4      }
5
6      fn get_value2(&self) -> String {
7          self.to_string()
8      }
9  }
10
11 let str: String = 42.get_value1();

```

Default

```

1  #[derive(Default)]
2  struct MyLuckyStruct {
3      value1: i32,
4      value2: bool,
5  }
6
7  let default_str = MyLuckyStruct::default();

```

Можно определить вручную:

```

1  impl Default for MyStruct {
2      fn default() -> Self {
3          Self { value1: 0, value2: false }
4      }
5  }

```

Clone

```

1  #[derive(Clone, Default)]
2  struct MyLuckyStruct {
3      value1: i32,
4      value2: bool
5  }
6
7  let default_str = MyLuckyStruct::default();
8  let another_str = default_str.clone();
9  println!("{}", default_str.value1);
10 println!("{}", another_str.value1);

```


Copy

Требует наличие `clone`.

```
1  #[derive(Copy, Clone, Default)]
2  struct MyLuckyStruct {
3      value1: i32,
4      value2: bool
5  }
6  let default_str = MyLuckyStruct::default();
7  let another_str = default_str;
8  println!("{}", default_str.value1);
9  println!("{}", another_str.value1);
```

Array

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

```
1  let array: [i32; 5] = [1, 2, 3, 4, 5];
2
3  let var1 = array[4];
4
5  let var2 = array[42];
6  // error: this operation will panic at runtime
7
8  struct MyStructWithArray {
9      array: [i32; 5],
10 }
```

Vector

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

```

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

```

Generics

Структуры и методы

```

1 struct MyPair<T> {
2     left: T,
3     right: T,
4 }
5
6 impl<T> MyPair<T> {
7     fn do_something_with_type(&self) {
8         println!("{}", self.left + self.right)
9     }
10 }
11
12 let my_small = MyPair::<i8> { left: 42, right: 1 };
13 let my_big = MyPair::<i128> {
14     left: 42424212123123123123123312312312,
15     right: 4121234236423648726348263874621,
16 };

```

Функции

```

1 fn generic_function<T>(p1: T, p2: T) -> MyPair<T> {
2     MyPair::<T>{
3         left: p1,
4         right: p2
5     }
6 }
7 generic_function::<i32>(42, 1);
8 generic_function(42, 1);

```

Multiple generics

```

1 fn generic_function<T, V>(p1: T, p2: T, other: V) -> MyPair<T> {
2     MyPair::<T>{
3         left: p1,
4         right: p2
5     }
6 }
7 generic_function::<i32, bool>(42, 1, true);
8 generic_function(42, 1, true);

```

Multiple generics in impl

```

1 impl<T> MyPair<T> {
2     fn new<V>(p1: T, p2: T, other: V) -> Self {
3         Self{
4             left: p1,
5             right: p2
6         }
7     }
8 }
9 let pair = MyPair::<i32>::new::<bool>(42, 1, true);
10 let pair = MyPair::new(42, 1, true);

```

Generic bounds

Add

```

1 impl<T: Add> MyPair<T> {
2     fn do_summ(self) {
3         let sum = self.left + self.right;
4     }
5 }

```

ToString

```

1 trait ToString {
2     fn to_string(&self) -> String;
3 }
4
5 impl<T: ToString> MyPair<T> {
6     fn print(&self) {
7         println!("{}", self.left.to_string(), self.right.to_string())
8     }
9 }

```

Display

```

1 impl<T: Display> MyPair<T> {
2     fn print(&self) {
3         println!("{}", self.left, self.right)
4     }
5 }
6
7 pub trait Display {
8     #[stable(feature = "rust1", since = "1.0.0")]
9     fn fmt(&self, f: &mut Formatter<'_>) -> Result;
10 }

```

Multiple Bounds

```

1 fn trace<T: Display + Debug>(to_trace: T, verbose: bool) {
2     println!("Display: `{}`", to_trace);
3     if verbose {
4         println!("Debug: `{:?}`", to_trace);
5     }
6 }
7
8 #[derive(Debug)]
9 struct MyPair<T> { left: T, right: T, }

```

Вывод получается такой:

```

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

```

Where clause

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

```

1  fn trace_multiple<T: Display + Debug, V: Display + Debug>(to_trace1: T,
2    to_trace2: V) {...
3
4  //      ^
5  //      |||
6  //      v
7
7  fn trace_multiple<T, V>(to_trace1: T, to_trace2: V)
8    where
9      T: Display + Debug,
10     V: Display + Debug {

```

23-03-07

Option

Аналог `null` в расте.

```

1  fn main() {
2    let mut var: Option<i32> = Option::None;
3    assert_eq!(true, var.is_none());
4
5    var = Option::Some(42);
6    assert_eq!(true, var.is_some());
7
8    print!("{}", var.unwrap());
9    // в данном случае, происходит move
10
11    let reference: &i32 = var.as_ref().unwrap();
12    // `as_ref` возвращает ссылку на объект
13 }

```

How to move data to heap?

Так никто не делает:

```

1  unsafe {
2      let layout = Layout::new::<u16>();
3      let ptr = alloc(layout);
4
5      if ptr.is_null() { // не получилось выделить память для кучи
6          handle_alloc_error(layout);
7      }
8
9      // кладем значение на кучу
10     *(ptr as *mut u16) = 42;
11     assert_eq!(*(ptr as *mut u16), 42);
12
13     dealloc(ptr, layout);
14 }

```

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 вызовами 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

```

1  struct Node<'a> {
2      value: i32,
3      next: Option<&'a Node<'a>>
4  }
5
6
7  fn main() {
8      let mut node1 = Node { value: 1, next: Option::None };
9      {
10         let mut node2 = Node { value: 2, next: Option::None };
11         {
12             let node3 = Node { value: 3, next: Option::None };
13
14             node2.next = Some(&node3);
15             node1.next = Some(&node2);
16

```

```

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

```

```

1 error[E0597]: `node3` does not live long enough
2 |
3 14 |         node2.next = Some(&node3);
4 |                                ^^^^^^^ borrowed value does not live long
   |                                enough
5 ...
6 |         - `node3` dropped here while still borrowed
7 20 |         println!("{}", node1.value) // does not compile
8 |                                ----- 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.

```

1 struct Node {
2     value: i32,
3     next: Option<Box<Node>>,
4 }

```

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:

```

1 struct Node {
2     value: i32,
3     next: Option<Box<Node>>,
4 }
5

```

```

6
7 fn main() {
8     let mut root = Box::new(Node {
9         value: 1,
10        next: Some(
11            Box::new(
12                Node {
13                    value: 2,
14                    next: Some(Box::new(Node {
15                        value: 3,
16                        next: None,
17                    })),
18                })),
19        });
20
21    println!("{}", root.next.unwrap().next.unwrap().value);
22
23    root.next = None;
24    println!("{}", root.value);
25 }

```

Взятие ссылки на значение в боксе

Функции `borrow()` и `borrow_mut()`.

```

1 use std::borrow::Borrow;
2 use std::borrow::BorrowMut;
3
4 struct MyStruct {
5     value1: i32,
6 }
7
8 fn main() {
9     let mut boxed = Box::new(MyStruct { value1: 42 });
10
11     //let reference = boxed.borrow(); // does not compile
12                                     // type must be known at this point
13     let reference: &MyStruct = boxed.borrow();
14     println!("{}", reference.value1);
15
16     let mut_reference: &mut MyStruct = boxed.borrow_mut();
17     mut_reference.value1 = 123;
18 }

```


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`:

```

1 use std::rc::Rc;
2
3 struct MyStruct {
4     value: i32,
5 }
6
7
8 fn main() {
9     let owner1 = Rc::new(MyStruct { value: 42 });
10    let owner2: Rc<MyStruct> = owner1.clone();
11    let owner3: Rc<MyStruct> = owner2.clone();
12
13    println!("{}", owner1.value, owner2.value, owner3.value);
14 }
```

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 мутать значение в контейнер и из него.

```

1 use std::cell::Cell;
2 use std::borrow::Borrow;
3
4 fn main() {
5     let cell = Cell::new(42);
6     let ref1: &Cell<i32> = cell.borrow();
7     let ref2: &Cell<i32> = cell.borrow();
8     println!("{}", ref1.get(), ref2.get());
9     ref1.replace(123);
10    println!("{}", ref1.get(), ref2.get());
11 }

```

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

```

1 use std::cell::Ref;
2 use std::cell::RefMut;
3 use std::cell::RefCell;
4
5 struct MyStruct {
6     value: i32,
7 }
8
9
10 fn main() {
11     let cell = RefCell::new(MyStruct { value: 42 });
12     {
13         // немутабельная ссылка на `cell`
14         let ref1: Ref<MyStruct> = cell.borrow();
15         println!("{}", ref1.value); // > 42
16     }
17     {
18         // мутабельная ссылка на `cell`
19         let mut ref_mut: RefMut<MyStruct> = cell.borrow_mut();
20         ref_mut.value = 123;

```

```

21     }
22     let ref2: Ref<MyStruct> = cell.borrow();
23     println!("{}", ref2.value); // > 123
24 }

```

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

Circular references

```

1  use std::cell::Cell;
2  use std::rc::Rc;
3
4  struct NodeLeft {
5      right: Cell<Option<Rc<NodeRight>>>,
6  }
7
8  struct NodeRight {
9      left: Cell<Option<Rc<NodeLeft>>>,
10 }
11
12
13 fn main() {
14     let node_left = Rc::new(NodeLeft { right: Cell::new(None) });
15     let node_right = Rc::new(NodeRight {
16         left: Cell::new(Some(node_left.clone()))
17     });
18     let e = node_left.right.replace(Some(node_right.clone()));
19 }

```

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()`.

```

1  use core::cell::Cell;
2  use std::rc::Rc;
3  use std::rc::Weak;
4
5  struct NodeLeft {
6      right: Cell<Option<Rc<NodeRight>>>,
7  }
8  struct NodeRight {
9      left: Cell<Option<Weak<NodeLeft>>>,
10 }
11
12
13 fn main() {
14     let node_left = Rc::new(NodeLeft { right: Cell::new(None) });
15     let node_right = Rc::new(NodeRight {
16         left: Cell::new(Some(Rc::downgrade(&node_left)))
17     });
18     let e = node_left.right.replace(Some(node_right.clone()));
19 }
```

Rc::upgrade()

Получить `Rc` ссылку из `Weak` ссылки:

```

1  use std::rc::Rc;
2
3  fn main() {
4      let five = Rc::new(5);
5      let weak_five = Rc::downgrade(&five);
6
7      // create a new Rc reference
8      let strong_five: Option<Rc<_>> = weak_five.upgrade();
9
10     // check if modification succeeded
11     assert!(strong_five.is_some());
12
13     drop(strong_five);
14     drop(five);
15
16     assert!(weak_five.upgrade().is_none());
17 }
```

23-03-14 \todo

23-03-21

Модули или Как компануется приложение раста

Есть package. Внутри него живут crates. Crates делятся на модули.

Package

Описывается в файле `Cargo.toml`:

```
1 [package]
2 name = "hello-world"
3 version = "0.1.0"
4 edition = "2021"
```

Crates

Crate = единица компиляции. Крейты могут быть

- Binary (скомпилированный файл - `exe`)

```
1 [[bin]]
2 name = "create_name"
3 path = "src/main.rs"
```

- Library (скомпилированный файл - `rlib`)

```

1  [lib]
2  name = "create_name"
3  path = "src/lib.rs"

```

Symbol table -- символьная таблица -- хранит соответствие между символом и строкой.
Уменьшает время компиляции

`rlib` -- набор `.o` файлов, метаданных, символьных таблиц. Хранится в виде архива.

Команда `rustc foo.rs --crate-type rlib` компилирует крейт как либу.

Rustc entity

Изначально у крейта нет собственного имени. Во время компиляции мы можем задать имя импортируемому крейту.

```

1  // foo.rs
2  pub fn aboba() {
3      println!("aboba")
4  }
5
6  // bar.rs
7  fn main() {
8      custom_name::aboba();
9  }

```

```

1  $ rustc foo.rs --crate-type rlib
2  > libfoo.rlib
3
4  $ rustc bar.rs --extern custom_name=./libfoo.rlib
5  > bar

```

Теперь про модули

Рутовым крейтом считается файл `main.rs`. В него можно импортировать другие исходные файлы:

```

1 //features.rs
2 pub struct MyFeature;
3 impl MyFeature {
4     pub fn print(&self) {
5         println!("I am a feature!")
6     }
7 }

```

```

1 // main.rs
2 mod features; // импорт из файла `features.rs`
3
4 fn main() {
5     let feature = crate::features::MyFeature;
6     feature.print();
7 }

```

Создание

```

1 struct MyStruct0;
2 mod level1 {
3     struct MyStruct1;
4     mod level2 {
5         struct MyStruct2;
6     }
7 }

```

pub

У модулей есть уровни доступа.

По дефолту, элемент из текущего модуля может использовать элементы родительских модулей, но не может использовать элементы дочерних модулей:

```

1 struct MyStruct0;
2 mod level1 {
3     struct MyStruct1;
4     mod level2 {
5         struct MyStruct2;
6     }
7     fn test() {
8         let parent = crate::level1::MyStruct1;
9         let child = crate::level1::level2::MyStruct2;
10    }
11 }

```

```

1 error[E0603]: unit struct `MyStruct2` is private
2     let child = crate::level1::level2::MyStruct2;
3         ^^^^^^^^^ private unit struct

```

Поменять такое поведение можно с помощью ключевого слова `pub`:

```

1 struct MyStruct0;
2 mod level1 {
3     struct MyStruct1;
4     mod level2 {
5         pub struct MyStruct2;
6     }
7     pub fn test() {
8         let parent = crate::level1::MyStruct1; // полный путь
9         let child = level2::MyStruct2; // относительный путь
10    }
11 }
12
13 fn main() {
14     crate::level1::test();
15 }

```

`pub(super)` \TODO

use

```

1 // использование `MyStruct` без дальнейшего прописывания полного пути
2 use crate::level1::level2::level3::MyStruct;
3 use crate::level1::level2::level3::{MyStruct1, MyStruct2};
4
5 // с помощью алиасинга
6 use crate::level1::level2::level3::MyStruct as Aboba;
7
8 // использование всего, что живет в выбранном модуле,
9 // без прописывания полного пути
10 use crate::level1::level2::level3::*;

```

Импорт крейтов

Простой пример [здесь](#).

Если внутри `src` есть еще директории с исходными файлами.

Для директории `<name>` создаем файл `<name>.rs`, внутри него импортируем `rs` файлы, которые живут в директории `<name>`:

- | | |
|-------------------|--|
| • Cargo.toml | • sub_folder.rs |
| • Cargo.lock | <code>mod sub_feature;</code> |
| • src | • main.rs |
| • sub_folder.rs | <code>mod sub_folder;</code> |
| • sub_folder | <code>use sub_folder::sub_feature::*;</code> |
| • sub_features.rs | <code>fn main() {</code> |
| • main.rs | <code>let feature = MySubFeature;</code> |
| • features.rs | <code>}</code> |
| • magic.rs | |

Линковка в стиле C

Раст часто используется для взаимодействия с каким-то куском кода на другом языке.

