

Rust 2019

compscicenter.ru

aleksey.kladov@gmail.com



Лекция 2

Время Жизни, ADT

ССЫЛКИ

- <https://www.rust-lang.org/>
- <https://rustup.rs/>
- <https://doc.rust-lang.org/book/>
- <https://doc.rust-lang.org/rust-by-example/>
- <https://doc.rust-lang.org/std/>

У Rust превосходная документация, можно выучить язык по книжке.

Move

- объекты уничтожаются при выходе из области видимости
- присваивание, передача аргумента, возврат значения передают владение
- move это memcpu
- объекты образуют дерево:

```
struct Function {  
    parameteres: Vec<Parameter>,  
    return_type: Box<Type>,  
}
```

```
fn main() {  
    let fns: Vec<Function> = ...;  
    drop(fns); // рекурсивно освобождает память  
}
```

Copy

Объекты, не владеющие ресурсами, остаются доступны после move:

```
let x = 1;  
let y = x;  
let z = x;  
assert_eq!(y, z);
```

Copy

Агрегаты из Copy объектов тоже Copy:

```
#[derive(Clone, Copy)]
```

```
struct Point { x: f64, y: f64 }
```

```
(Point, Point)
```

```
[Point; 1024] // копировать можно, но не стоит
```

- Box<T> и Vec<T> не Copy — должны освобождать память

Ссылки

Жизнь без ссылок

```
fn print_vec(xs: Vec<i32>) {  
    for x in xs {  
        println!("{}", x);  
    }  
}  
  
fn main() {  
    let xs = vec![1, 2, 3];  
    print_vec(xs);    // ok  
    print_vec(xs);    // value used after move  
}
```


ЖИЗНЬ С ССЫЛКАМИ

```
fn print_vec(xs: &Vec<i32>) {  
    for x in xs {  
        println!("{}", x);  
    }  
}
```

```
fn main() {  
    let xs = vec![1, 2, 3];  
    print_vec(&xs); // ok  
    print_vec(&xs); // ok  
}
```

& позволяет использовать значение, не меняя структуры владения

ВЖ ССЫЛКИ

- ссылка (&T или &**mut** T) — указатель на объект
- время жизни **ССЫЛКИ** — время, когда ссылка используется

```
fn main() {  
    let x = 1;  
    let r = &x;           // ^  
    foo(r);               // | время жизни &r  
    println!("{}", *r);  // v  
}
```

```
fn foo(r: &i32) { // lifetime больше чем функция  
}
```

Ссылки на значения

- ВЖ объекта должно быть больше, чем ВЖ любой ссылки на него

```
let x = 1;
let r: &i32;
{
    let y = 2;
    r = &x;    // ok
}
println!("{}", *r);
```

Ссылки на значения

- ВЖ объекта должно быть больше, чем ВЖ любой ссылки на него

```
let x = 1;
let r: &i32;
{
    let y = 2;
    r = &y; // borrowed value does not live long enough
}
println!("{}", *r);
```

Именованные ВЖ

```
fn main() {  
    let x = 1;  
    let r: &i32;  
    {  
        let y = 2;  
        r = f(&x, &y); // ???  
    }  
    println!("{}", *r);  
}  
  
fn f(x: &i32, y: &i32) -> &i32 {  
    ...  
}
```

Не посмотрев в тело `f`, не понять, корректен ли код.

Именованные ВЖ

```
fn main() {  
    let x = 1;  
    let r: &i32;  
    {  
        let y = 2;  
        r = f(&x, &y); // ok  
    }  
    println!("{}", *r);  
}  
  
fn f<'a, 'b>(x: &'a i32, y: &'b i32) -> &'a i32 {  
    y // parameter and the return type are declared  
        // with different lifetimes  
}
```

Можем проверить тело и вызов f по отдельности!

Ссылки на подобыекты

```
struct S { value: i32 }
```

```
fn main() {  
    let r: &i32;  
    {  
        let s = S { value: 92 };  
        let rs: &S = &s; // does not live long enough  
        r = f(rs);  
    }  
    println!("{}", *r); // borrowed value needs to live until here  
}  
  
fn f<'a>(s: &'a S) -> &'a i32 {  
    &s.value // ok!  
}
```

Что такое 'a?

```
fn f<'a>(s: &'a S) -> &'a i32 {  
    &s.value  
}
```

- времена жизни ссылок назначаются компилятором
- 'a это ограничение (ВЖ &S и &i32 равны)
- borrow checking — решить систему уравнений на lifetime:
 - заданные именованными ВЖ (равенство или подмножество)
 - lifetime ссылки содержит все её использования
 - lifetime ссылки меньше, чем lifetime объекта
 - ...

Заморозка

```
struct Wrapper {  
    value: Box<i32>,  
}  
  
fn main() {  
    let w = Wrapper { value: Box::new(92) };  
    let r: &i32 = &*w.value;  
    w.value = Box::new(62);  
    println!("{}", *r); // dangling reference  
}
```

Заморозка

```
struct Wrapper {  
    value: Box<i32>,  
}
```

```
fn main() {  
    let w = Wrapper { value: Box::new(92) };  
    let r: &i32 = &*w.value;  
    w = Wrapper { value: Box::new(62) }  
    println!("{}", *r); // dangling reference  
}
```

Заморозка

```
struct Wrapper {  
    value: Box<i32>,  
}  
  
fn main() {  
    let w = Wrapper { value: Box::new(92) };  
    let r: &i32 = &*w.value;  
    w = Wrapper { value: Box::new(62) }  
    println!("{}", *r); // dangling reference  
}
```

Активная ссылка замораживает объект и все родительские объекты: их нельзя менять.

Заморозка

```
struct Wrapper {  
    value: Box<i32>,  
}  
  
fn main() {  
    let w = Wrapper { value: Box::new(92) };  
    let r: &i32 = &*w.value;  
    if w.value > 640 { println!("enough"); }  
    println!("{}", *r); // ok  
}
```

Смотреть можно!

mut

```
struct Wrapper {  
    value: Box<i32>,  
}  
  
fn main() {  
    let w = Wrapper { value: Box::new(92) };  
    let r: &i32 = &*w.value;  
    f(&w);  
    println!("{}", *r);  
}  
  
fn f(w: &Wrapper) {  
    ...  
}
```

Хотим не смотреть в тело `f` — нужен способ понять, изменяет ли `f` аргумент.

mut

```
struct Wrapper {  
    value: Box<i32>,  
}  
  
fn main() {  
    let mut w = Wrapper { value: Box::new(92) };  
    // cannot borrow w as mutable  
    // because *w.value is also borrowed as immutable  
    let r: &i32 = &*w.value;  
    f(&mut w);  
    println!("{}", *r);  
}  
  
fn f(w: &mut Wrapper) {  
    w.value = Box::new(62); // ok  
}
```

Shared ^ Mutable

- для изменения объекта нужна **&mut** ссылка
- & ссылки замораживают объект



Правило "shared XOR mutable"

Либо единственная **&mut** ссылка, либо произвольное количество & ссылок

Пример

main.cpp

```
std::vector<int> xs = {1, 2, 3};  
auto& x = xs[0];  
xs.push_back(4);  
std::cout << x; // UB!
```


Пример

main.rs

```
let mut xs = vec![1, 2, 3];  
let x = &xs[0];  
xs.push(4);  
println!("{}", x);
```

error[E0502]: cannot borrow `xs` as mutable because it is also borrowed as immutable

```
--> main.rs:4:1  
  |  
3 | let x = &xs[0];  
  |           -- immutable borrow occurs here  
4 | xs.push(4);  
  | ^^ mutable borrow occurs here  
...  
7 | }  
  | - immutable borrow ends here
```

Ссылки в C++ и Rust

C++

- создаются неявно
- не являются первоклассными объектами
(`std::reference_wrapper`)
- не всегда валидны

Rust

- требуют явных `&`/`&mut` и `*`
- обычные объекты

```
let x = 1;  
let y = 2;  
let mut r: &i32 = &x;  
r = &y;
```

- всегда валидны

Итого

- у каждого объекта есть один владелец (\Rightarrow дерево владения)
- $\&$ и $\&\text{mut}$ ссылки позволяют использовать объект без владения
- mut позволяет менять объект
- $\&\text{mut}$ ссылка всегда одна
- именованные ВЖ устанавливают отношения между ссылками
- ВЖ для ссылок выбираются компилятором
- исполняемый код **не зависит** от конкретных ВЖ (parametricity)
- 80% правды

Выражения

Выражения

- С с ароматом ML
- почти все конструкции — выражения

Блоки

`{ }` — выражение

```
1 + { let x = 2; x * 2 }
```

Блок состоит из инструкций (statement), завершённых ;

Значение блока — значение хвостового выражения.

```
let i: i32 = { 1 };  
let i: () = { 1; };
```

Точки с запятой имеют значение!

Инициализация блоком

```
// Лишние переменные не видны снаружи
let omelet = {
    let eggs = get_eggs(&mut refrigerator, 3);
    let bacon = open_bacon(&mut refrigerator);
    fry(eggs, bacon)
};
```

if

```
let res = if condition1 {  
    expr1;  
    expr2  
} else if condition2 {  
    expr3  
} else {  
    expr4  
};
```

- нет () вокруг условия
- {} обязательны
- `else if` — особый синтаксис, нет dangling else problem
- если нет блока `else`, значение — ()

if

```
fn main() {  
    if true {  
        92 // expected (), found integral variable  
    }  
}
```

if

```
fn main() {  
    if true {  
        92; // ok!  
    }  
}
```

while

```
while condition {  
    body // <- должно быть типа ()  
}
```

```
let x: () = while false {};
```

break и continue

```
while true {  
    if cond1 {  
        continue;  
    }  
    if cond2 {  
        break;  
    }  
}
```

```
'outer: while cond1 {  
    while cond2 {  
        break 'outer;  
    }  
}
```

loop

Специальная конструкция для бесконечного цикла

```
loop {  
    body  
}
```



loop {} и **while true {}** отличаются информацией
про поток управления

loop

```
let uninit;  
while true {  
    if condition {  
        uninit = 92;  
        break;  
    }  
}  
pritnln!("{}", uninit);
```

```
let init;  
loop {  
    if condition {  
        init = 92;  
        break;  
    }  
}  
pritnln!("{}", init); // ok
```

loop

```
let init;
```

```
if condition {  
    init = 92;  
} else {  
    loop {}  
}
```

```
println!("{}", init); // ok!
```

Гарантированная инициализация

- в C/C++ доступ к неинициализированной переменной — UB
- инициализация значением по умолчанию прячет баги

!

```
let x: ! = loop {};
```

- ненаселённый тип
- может выступать в роли любого другого типа

```
let x: u32 = loop {};
```

- пока ещё не настоящий тип

panic!()

Семейство макросов, возвращающих !:

- `panic!("something went wrong")` — для сигнализации о багах
- `unimplemented!()` — плейсхолдер для ещё не написанного кода

```
if complex_condition {  
    complex_logic  
} else {  
    unimplemented!()  
}
```

- `unreachable!()` — маркер для "невозможных" условий

break со значением

```
let init: i32 = loop {  
    if condition {  
        break 92;  
    }  
};
```

for

```
for x in vec![1, 2, 3] {  
    println!("x = {}", x);  
}
```

```
let xs = vec![1, 2, 3];  
for i in 0..xs.len() {  
    let x = xs[i];  
    println!("x = ", x);  
}
```

Протокол итераторов — дальше в курсе

ranges

```
let bounded: std::ops::Range<i32> = 0..10;
let from = 0..;
let to = ..10;
let full = ..;
let inclusive = 0..=9;

for i in (0..10).step_by(2) {
    println!("i = {}", i);
}
```

По lo..hi и lo.. можно итерироваться

Ещё раз о ;

; превращает expression в statement

После выражений-блоков ; не нужна:

```
{  
    if x == 0 {  
        println!("zero");  
    }           // statement  
  
    { 0; } // statement  
  
    if true { 92 } else { 62 } // expression!  
}
```

Ещё раз о ;

После **let** ; обязательна:

```
let s = if x > 0 {  
    "positive"  
} else {  
    "negative"  
};
```

Два слова о функциях

```
fn hypot(x: f64, y: f64) -> f64 {  
    let x_squared = x * x;  
    let y_squared = y * y;  
    (x_squared + y_squared).sqrt()  
}
```

- типы параметров и результата обязательны
- нет перегрузки
- **fn** main {} == **fn** main() -> ()
- тело функции — блок
- **return** опционален
- **fn** diverge() -> ! { **loop** {} }

Quiz

```
fn foo() {  
    let x = return;  
}
```



Какой тип у x?

Алгебраические типы данных

Структуры

```
#[derive(Clone, Copy)]
```

```
struct Point {
```

```
    x: f64,
```

```
    y: f64,
```

```
}
```

```
let p1 = Point { x: 1.0, y: 2.0 };
```

```
let p2 = Point {
```

```
    x: 2.0,
```

```
    .. p1
```

```
};
```

```
assert_eq!(p2.x, p2.y);
```

Методы

```
struct Point { x: f64, y: f64 }
```

```
impl Point {  
    fn distance_from_origin(&self) -> f64 {  
        (self.x * self.x + self.y * self.y).sqrt()  
    }  
}
```

- **self** — явная версия `this`
- **&self** — передача по ссылке

Методы

```
struct Point { x: f64, y: f64 }
```

```
impl Point {  
    fn scale(&mut self, factor: f64) {  
        self.x *= factor;  
        self.y *= factor;  
    }  
}
```

- **&mut self** — передача по уникальной ссылке

Методы

```
struct Point { x: f64, y: f64 }
```

```
impl Point {  
    fn consume(self) {  
    }  
}
```

- **self** — передача владения

Ассоциированные функции

```
struct Point { x: f64, y: f64 }

impl Point {
    fn origin() -> Point {
        Point { x: 0.0, y: 0.0 }
    }
}

let p = Point::origin();
assert_eq!(
    p.distance_from_origin(),
    0.0,
);
```

Deref

```
fn foo(mut p: Point) {  
    p.scale(2.0);  
    let d1 = p.distance_from_origin();  
    let boxed = Box::new(p);  
    let d2 = boxed.distance_from_origin();  
    assert_eq!(d1, d2);  
}
```



Вызов метода автоматически добавляет **&**, **&mut** и *****

Deref

```
fn foo(mut p: Point) {  
  
    let boxed = Box::new(p);  
    let d2 = (&*boxed).distance_from_origin();  
  
}
```



Вызов метода автоматически добавляет **&**, **&mut** и *****

Структуры-кортежи

```
struct Point(f64, f64);
```

```
impl Point {  
    fn origin() -> Point {  
        Point(0.0, 0.0)  
    }  
    fn dist(self, other: Point) -> f64 {  
        let Point(x1, y1) = self;  
        let Point(x2, y2) = other;  
        ((x1 - x2).powi(2) + (y1 - y2).powi(2)).sqrt()  
    }  
}
```

```
let p = Point(0.0, 1.0);  
assert_eq!(p.0, 0.0);
```

Паттерн newtype

```
struct Kilometers(f64);
```

```
struct Miles(f64);
```

- Представление в памяти такое же, как и у внутреннего типа
- Нет необходимости в аннотациях
- Нет автоматической конверсии/автоматических методов

Zero Sized Types

```
struct Tag;
```

```
let t = Tag;
```

```
assert!(std::mem::size_of::<Tag>() == 0);
```

```
assert!(std::mem::size_of::<(Tag, Tag)>() == 0);
```

```
assert!(std::mem::size_of::<[Tag; 1024]>() == 0);
```

```
assert!(std::mem::size_of::<()>() == 0);
```



ZST существуют только во время компиляции

👉 zero cost abstraction!

Type Tags

```
struct Kilometers;
```

```
struct Miles;
```

```
struct Distance<M> {  
    amount: f64,  
    metric: M,  
}
```

```
let d1: Distance<Kilometers> = Distance {  
    amount: 92.0,  
    metric: Kilometers,  
};
```

```
let d2: Distance<Miles> = Distance {  
    amount: 92.0,  
    metric: Miles,  
};
```

Виды структур

struct

```
struct Point { x: f64, y: f64 }
```

tuple struct

```
struct Point(f64, f64);
```

newtype (tuple) struct

```
struct Point1D(f64);
```

unit struct

```
struct ThePoint; // ZST
```

Dynamically Sized Types

- `[i32; 4]` — четыре числа
- `&[i32; 4]` — адресс в памяти, где лежат четыре числа
- `[i32]` — n чисел

```
mem::size_of::<[i32]>();
```

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ doesn't have a size known at compile-time

- `&[i32]` — указатель + количество элементов, fat pointer

```
assert_eq!(  
    mem::size_of::<&[i32]>(),  
    mem::size_of::<usize>() * 2,  
)
```

Slices

- `&[T]` — слайс

```
fn print_slice(xs: &[i32]) {  
    for idx in 0..xs.len() {  
        println!("{}", xs[i]);  
    }  
}
```

- всегда знает свою длину
- доступ по индексу проверяет выход за границу
- нельзя отключить флагом компилятора
- `&[T]` можно получить из `&[T; N]` или `Vec<N>`

Enums

```
enum Shape {  
    Circle {  
        center: Point,  
        radius: f64,  
    },  
    Square {  
        bottom_left: Point,  
        top_right: Point,  
    },  
}
```

Enums

```
impl Shape {  
    fn circle(center: Point, radius: f64) -> Shape {  
        Shape::Circle { center, radius }  
    }  
    fn area(&self) -> f64 {  
        match self {  
            Shape::Circle { radius, .. } => {  
                std::f64::consts::PI * radius * radius  
            }  
            Shape::Square { bottom_left, top_right } => {  
                unimplemented!()  
            }  
        }  
    }  
}
```

Enums

```
enum Expr {  
    Negation(Box<Expr>),  
    BinOp { lhs: Box<Expr>, rhs: Box<Expr> },  
    Unit,  
}
```

- варианты **enum** бывают такие же, как и структуры
- квалификация обязательна **Expr::BinOp { lhs, rhs }**,
но можно импортировать вариант: `use Expr::BinOp`
- `mem::size_of` — размер самого большого варианта + дискриминант
- размер объекта не может быть бесконечным

Полезные enum'y

```
// use std::cmp::Ordering;
enum Ordering {
    Less,
    Equal,
    Greater,
}

fn binary_search(xs: &[i32], x: i32) -> bool {
    if xs.is_empty() { return false; }
    let mid = xs.len() / 2;
    let subslice = match xs[mid].cmp(&x) {
        Ordering::Less => &xs[mid + 1..],
        Ordering::Equal => return true,
        Ordering::Greater => &xs[..mid],
    };
    binary_search(subslice, x)
}
```

Полезные еnumы

```
enum Option<T> {  
    Some(T),  
    None,  
}
```

```
fn foo(xs: &[i32]) {  
    match xs.get(92) {  
        Some(value) => ...,  
        None => panic!("out of bounds access")  
    }  
}
```



Имена **None** и **Some** доступны по умолчанию

Полезные enum

```
enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```

```
impl<T> [T] {  
    pub fn binary_search(&self, x: &T) -> Result<usize, usize>  
    where  
        T: Ord  
    {  
        self.binary_search_by(|p| p.cmp(x))  
    }  
}
```

Newtype Variant

```
enum Expr {  
  BinOp {  
    lhs: Box<Expr>,  
    rhs: Box<Expr>,  
    op: Op,  
  },  
  If {  
    cond: Box<Expr>,  
    then_branch: Box<Expr>,  
    else_branch: Box<Expr>,  
  },  
}
```



BinOp и **If** типами не являются

Newtype Variant

```
enum Expr {  
    BinOp(BinOp),  
    If(If)  
}
```

```
struct BinOp {  
    lhs: Box<Expr>,  
    rhs: Box<Expr>,  
    op: Op,  
}
```

```
struct If {  
    cond: Box<Expr>,  
    then_branch: Box<Expr>,  
    else_branch: Box<Expr>,  
}
```


Void

```
enum Void {}
```

Void

```
enum Void {}
```

```
fn foo(void: Void) -> Vec<Point> {  
    match void {  
        }  
    }  
}
```

- эnum без вариантов — аналог !
- гарантия, что код не достижим
- `size_of::<Void>() == 0` 😊

Result

- `Result<T, Void> == T`

```
fn extract(result: Result<Spam, Void>) -> Spam {  
    match result {  
        Ok(spam) => spam,  
        Err(void) => match void {},  
    }  
}
```



На что похож `Result<(), ()>`?

Представление в памяти

- представление в памяти не специфицировано
- компилятор минимизирует паддинг (есть `#[repr]` атрибуты)
- дискриминант может прятаться в неиспользованных битах
- гарантировано что

```
mem::size_of::<Option<&T>>() == mem::size_of::<&T>()
```

```
mem::size_of::<Option<Box<T>>>() == mem::size_of::<Box<T>>()
```

- `bool` занимает 1 байт, чтобы `&bool` работал всегда
- Newtype Variant может быть больше обычного `enum` из-за паддинга :o)
- бывают типы "странных" размеров: ZST, DST, uninhabited