

Rust 2019

compscicenter.ru

aleksey.kladov@gmail.com



Лекция 9

Умные Указатели, static

const

const

```
#[derive(Debug)]  
struct Color {  
    r: u8, g: u8, b: u8  
}  
  
const BLACK: Color = Color { r: !0, g: !0, b: !0 };  
  
fn main() {  
    println!("men in {:?}", BLACK)  
}
```

const — конструкция верхнего уровня

Тип нужно указывать явно

const



Байты **const** находятся в **.text** (неизменяемой) секции исполняемого файла

Константа вычисляется во время компиляции

Можно использовать арифметику, литералы, простые выражения и **const fn** функции

```
impl<T> Cell<T> {  
    pub const fn new(value: T) -> Cell<T> {  
        Cell {  
            value: UnsafeCell::new(value),  
        }  
    }  
}
```

match

Константы можно использовать в **match** :

```
fn foo(x: i32) {  
    const ZERO: i32 = 0;  
  
    match x {  
        ZERO => println!("zero"), ①  
        other => println!("other: {}", other), ②  
    }  
}
```

① сравнили x с нулём

② записали x в other



Семантика зависит от наличия константы в области видимости

const МОЖНО ИСПОЛЬЗОВАТЬ В **impl** :

```
trait Tagged {  
    const TAG: &'static str;  
}
```

```
struct Foo;  
impl Tagged for Foo { const TAG: &'static str = "Foo"; }
```

```
struct Bar;  
impl Tagged for Bar { const TAG: &'static str = "Bar"; }
```

```
fn by_tag(tag: &str) {  
    match tag {  
        Foo::TAG => println!("foo"),  
        Bar::TAG => println!("bar"),  
        _ => panic!("unknown tag: {:?}", tag)  
    }  
}
```

Области Видимости

Видимость

Константы (как и любые конструкции верхнего уровня) можно объявлять внутри блока

Для них создаётся невидимый модуль, порядок объявления не важен

```
fn main() {  
    println!("π/2 = {}", FRAC_PI_2);  
  
    const PI: f32 = 3.1415926;  
    const FRAC_PI_2: f32 = PI / 2.0;  
}
```

Локальные Функции

Функцию тоже можно объявить внутри блока, но она не будет замыканием:

```
fn fib(n: usize) -> usize {  
    let mut cache = vec![0; n + 1];  
    return fib_memo(&mut cache, n);  
  
    fn fib_memo(cache: &mut Vec<usize>, n: usize) -> usize {  
        if n == 0 || n == 1 {  
            return 1;  
        }  
        if cache[n] == 0 {  
            cache[n] =  
                fib_memo(cache, n - 1) + fib_memo(cache, n - 2);  
        }  
        cache[n]  
    }  
}
```

Локальный Мир

Модули тоже можно объявлять локально:

```
fn main() {  
    mod m {  
        pub(super) fn hello() {  
            println!("Hello, world!");  
        }  
    }  
    m::hello();  
}
```

Пространства имён

В Rust есть два непересекающихся пространства имён (a-la Lips-2):
"типы" и "значения":

```
const F00: i32 = 92;    // value
type F00 = ();         // type

struct Bar { f: u32 }   // type
fn Bar(f: u32) -> Bar { // value
    Bar { f }
}

mod m {}               // type
fn m() {}              // value
```

На практике коллизий мало, из-за разных соглашений об именовании

Пространства имён

Unit и tuple структуры рессахариваются в определение типа и функции/константы

```
struct S; ❶
```

```
struct T(u32, i32); ❷
```

❶ создали тип S и **const** S: S

❷ создали тип T и **fn** T(_0: u32, _1: i32) -> T

None — константа типа Option<T>

Some — функция типа **fn**(T) -> Option<T>

'static

'static

На константы можно брать ссылку

ВЖ — 'static , больше любого другого времени жизни

```
const X: i32 = 92;  
const R: &'static i32 = &X;
```

```
fn foo(x: &i32) {  
    println!("{}", x);  
}
```

```
fn main() {  
    foo(R);  
}
```

Lifetime Elision

В константах работает lifetime elision, результат — `'static`

```
const WORDS: &[&str] = &[  
    "hello",  
    "world",  
];
```

Тип строкового литерала — `&'static str`

Тип байтового литерала — `&'static [u8; _]`

```
const HELLO: [u8; 5] = *b"hello";  
const HELLO_2: [u8; 5] = [104, 101, 108, 108, 111];  
  
fn main() {  
    assert_eq!(HELLO, HELLO_2);  
}
```


Строковые литералы

```
"hello": &'static str      // байты в utf-8  
b"hello": &'static [u8; 5] // байты в ASCII
```

```
'a': char  // 32 бита  
b'a': u8
```

```
"hello\nworld"      // обычное экранирование через \  
"hello  
world"         // многострочный литерал
```

```
"hello \  
world"         // \ убирает \n и пробелы после него
```

```
r"hello\nworld"    // сырой литерал, \ это \
```

```
r###"raw literal with "## inside!"###
```

[]

У [] тип &'static [], пустые слайсы можно извлекать из воздуха

```
fn as_slice<'a>(xs: Option<&'a Vec<i32>>) -> &'a [i32] {  
    match xs {  
        Some(xs) => xs.as_slice(),  
        None => &[],  
    }  
}
```

Box::leak

Ещё один способ получить вж `'static` это пообещать, что память никогда не будет освобождена

```
impl<T> Box<T> {  
    pub fn into_raw(b: Box<T>) -> *mut T { ... }  
    pub unsafe fn from_raw(raw: *mut T) -> Self { ... }  
  
    pub fn leak<'a>(b: Box<T>) -> &'a mut T  
    {  
        unsafe { &mut *Box::into_raw(b) }  
    }  
}
```

- после вызова `into_raw` память не будет освобождена
- можно получить **любое** вж (но `'static` — самый интересный случай)

dyn T + 'static

У **dyn** T ТИПОВ ТОЖЕ ЕСТЬ lifetime:

```
type Callback<'a> = Box<dyn Fn() + 'a>;
```

```
fn main() {  
    let f: Callback<'static> =  
        Box::new(|| println!("hello"));  
  
    let x = 92;  
    let g: Callback<'_> =  
        Box::new(|| println!("x = {}", x));  
}
```

- Box<**dyn** T> == Box<**dyn** T + 'static>
- &'a **dyn** T == &'a (**dyn** T + 'a)
- **dyn** T == **dyn** T + 'static

static

static

const пишет в .text секцию, **static** — в изменяемую .data секцию

```
static COUNTER: usize = 0;
```

```
fn main() {  
    println!("{}", COUNTER);  
}
```

static mut

```
static mut COUNTER: usize = 0;
```

```
fn main() {  
    COUNTER += 1;           // use of mutable static is unsafe  
    println!("{}", COUNTER); // and requires unsafe function  
}                           // or block
```

Использовать **static mut** так просто нельзя...

static mut

```
static mut X: usize = 0;

pub fn sneaky() -> &'static mut usize {
    &mut X
}

fn main() {
    let x1: &mut usize = sneaky();
    let x2: &mut usize = sneaky(); // алиасинг!
}
```

При операциях со **static mut**, вызывающий код должен гарантировать, что не создаётся алиасинг



Глобальные переменные доступны всем (&)

static + interior mutability

```
use std::cell::Cell;

static COUNTER: Cell<usize> = Cell::new(0);

fn main() {
    COUNTER.set(COUNTER.get() + 1);
    println!("{}", COUNTER.get());
}
```

Казалось бы, должно работать: не врем про **mut**

Не работает из-за многопоточности



В Rust глобальные переменные имеют *объективные* недостатки

Life Before Main

Так же, как и **const** , **static** это просто байты в бинарном файле



static можно инициализировать только константой

В C++, Java, Swift etc глобальную переменную можно инициализировать чем угодно, инициализация происходит при первом обращении или до main

В Rust нет жизни до main , ленивую инициализацию надо писать явно (a-la OnceCell).

Умные Указатели

Owning Smart Pointers

Умный указатель управляет доступом к ресурсу

`Box<T>` — главный умный указатель, управляет памятью, вызывает `free` в `drop`

`Vec<T>` — тоже умный указатель, `Box` для нескольких значений



`Vec<T>` это не просто коллекция, а стратегия управления памятью

Borrowing Smart Pointers

`cell::Ref<'a, T>` и `cell::RefMut<'a, T>` — умные
заимствующие указатели

Оборачивают `&T` / `&mut T` и в `Drop` уменьшают счётчик ссылок
на `RefCell<T>`

Deref

Умные указатели реализуют Deref / DerefMut :

- **impl**<T> Deref<Target = [T]> **for** Vec<T>
- **impl**<T> DerefMut **for** Vec<T>
- **impl**<'a, T> Deref<Target = T> **for** Ref<'a, T>
- ...

Паттерн: ассоциированная функция лучше метода

```
impl Box<T> {  
    fn leak<'a>(b: Box<T>) -> &'a T { ... }  
}  
  
fn uppercase(data: &str) -> &'static str {  
    // unicode, размер новой строки может быть больше,  
    // in-place uppercase не работает  
    let data: String = data.to_uppercase();  
  
    // убрали поле capacity  
    let data: Box<str> = data.into_boxed_str();  
  
    // Вызвали "метод"  
    Box::leak(data)  
}
```

Метод мог бы конфликтовать с методом на типе (pipe.leak())

```

impl <'a, T> Ref<'a, T> {
    pub fn map<U, F>(orig: Ref<'a, T>, f: F) -> Ref<'a, U>
    where
        F: FnOnce(&T) -> &U,
        U: ?Sized,
}

```

Ref::map — проекция:

```

use std::cell::{RefCell, Ref};

#[derive(Default)]
struct Person { first_name: String, last_name: String }

fn main() {
    let cell = RefCell::new(Person::default());
    let p: Ref<Person> = cell.borrow();
    let first_name: Ref<str> =
        Ref::map(p, |it| it.first_name.as_str());
}

```


std::rc::Rc

Rc<T>

Reference Counting owning smart pointer

Rc это Box с $O(1)$ клонированием (**impl**<T> Clone **for** Rc<T>)

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

```
fn main() {  
    let hello1: Rc<String> = Rc::new("hello".to_string());  
    let hello2 = Rc::clone(&hello);  
}
```

Rc<T> реализует Deref<Target = T>, но не DerefMut: Rc разделяется между несколькими владельцами, уникальной ссылки быть не может

std::rc::Rc

Внутри лежит RcBox

```
struct RcBox<T: ?Sized> {  
    strong: Cell<usize>,  
    weak: Cell<usize>,  
    value: T,  
}
```

- Нужен `Cell::clone` получает & ссылку, но мы должны увеличить счётчик ссылок.
- Значение и счётчик ссылок всегда аллоцированы вместе \Rightarrow нет необходимости в `std::make_shared`

std::rc::Rc::make_mut

```
impl<T: Clone> Rc<T> {  
    pub fn make_mut(this: &mut Self) -> &mut T { ... }  
}
```

`make_mut` — магический метод, если `rc == 1`, то возвращаем ссылку на данные, иначе клонируем их

Так как `this: &mut`, то не может быть других `&Rc<T>`

Можно писать неизменяемые структуры данных, поддерживающие in-place модификацию (clone-on-write)

В функциональных языках:

```
insert :: set -> a -> set
```

В Rust:

```
#[derive(Clone)] // O(1)
```

```
struct Set<T> { /* Rc */ }
```

```
impl<T: Clone> Set<T> {
```

```
    fn insert(&mut self, value: T) { ... }
```

```
}
```

- из-за &**mut** Vec так же надёжен, как и cons-list
- МОЖНО МЕНЯТЬ std::Vec на im::Vector без изменения API!
- <http://smallcultfollowing.com/babysteps/blog/2018/02/01/in-rust-ordinary-vectors-are-values/>
- <https://docs.rs/im/>

std::borrow::Cow

Cow хранит либо ссылку с вж 'a, либо значение:

```
pub enum Cow<'a, B>
where
    B: ToOwned + ?Sized + 'a,
{
    Borrowed(&'a B),
    Owned(<B as ToOwned>::Owned),
}

pub trait ToOwned {
    type Owned: Borrow<Self>;

    fn to_owned(&self) -> Self::Owned;
}

pub trait Borrow<Borrowed: ?Sized> {
    fn borrow(&self) -> &Borrowed;
}
```

ToOwned

`X: Borrow<Y>` — из `X` можно получить `&Y`, согласованную по `Eq`, `Hash`, `Ord`

`ToOwned` — противоположность `Borrow`, из ссылки получаем значение

```
impl ToOwned for str {  
    type Owned = String;  
}
```

```
impl<T: Clone> ToOwned for [T] {  
    type Owned = Vec<T>;  
}
```

Cow

```
use std::borrow::Cow;
```

```
fn to_lowercase<'a>(s: &'a str) -> Cow<'a, str> {  
    if s.chars().all(char::is_lowercase) {  
        Cow::Borrowed(s)  
    } else {  
        Cow::Owned(s.to_lowercase())  
    }  
}
```

Если `s` и так в нижнем регистре — экономим аллокацию

```
pub fn search_case_insensitive(s: &str) -> bool {  
    let s: String = s.to_lowercase(); // аллоцируем всегда  
    search_lowercased(&s);  
}  
  
fn search_lowercased(s: &str) -> bool {  
    ...  
}
```

Хотим написать поиск без учёта регистра. Для этого приводим и данные, и паттерн в нижний регистр


```
pub fn search_case_insensitive(s: &str) -> bool {  
    let s: &str = if is_lowercase(s) {  
        s  
    } else {  
        &s.to_lowercase() // ссылка на локальную переменную  
    };  
    search_lowercased(&s);  
}  
  
fn search_lowercased(s: &str) -> bool {  
    ...  
}
```

```
pub fn search_case_insensitive(s: &str) -> bool {  
    if is_lowercase(s) {  
        search_lowercased(s) // немного дублирования  
    } else {  
        search_lowercased(&s.to_lowercase())  
    }  
}  
  
fn search_lowercased(s: &str) -> bool {  
    ...  
}
```

```
pub fn search_case_insensitive(s: &str) -> bool {  
    let s: Cow<str> = to_lowercase(s); // нет аллокации  
    search_lowercased(&*s); // runtime проверка варианта при Deref  
}
```

```
fn search_lowercased(s: &str) -> bool {  
    ...  
}
```

```

pub fn search_case_insensitive(s: &str) -> bool {
    let lowercased: String;
    let s: &str = if is_lowercase(s) {
        s
    } else {
        lowercased = s.to_lowercase();
        &lowercased
    };
    search_lowercased(&s);
}

fn search_lowercased(s: &str) -> bool {
    ...
}

```



Всегда можно расширить время жизни локальной переменной до всей функции, не обязательно её инициализировать