

# Rust 2019

[compscicenter.ru](http://compscicenter.ru)

[aleksey.kladov@gmail.com](mailto:aleksey.kladov@gmail.com)



# Лекция 8

## Время Жизни II

# Lifetime Elision



У каждой ссылки есть время жизни, хотя мы его пишем не всегда.

Явный синтаксис для `&i32` это `&'_ i32`, где `'_` — какой-то (выведенный компилятором) `lifetime`.



Ссылка — конструктор типа, с одним параметром времени жизни

При определении типов данных вж надо указывать явно:

```
struct Ref<'a, T> {  
    r: &'a T  
}
```

Для локальных переменных вж всегда выводится:

```
fn main() {  
    let x: &'_ i32 = &92;  
}
```

В параметрах функции вж можно не указывать:

```
fn substring(text: &str, start_index: usize) -> &str
```

# Lifetime Elision

Почти всегда `вж` результата совпадает с `вж` аргумента

Если `вж` не указаны явно, то для каждого `'_'` в параметрах создаётся свой уникальный `вж`, а возвращаемое значение получает `вж` **`self`** или уникальное `вж` параметра.



Отличие `elision` от `inference` в том, что компилятор не смотрит на тело функции, только на заголовок.

# Примеры

```
fn f1(x: &i32, y: &i32)
```

# Примеры

```
fn f1(x: &i32, y: &i32)
```

```
fn f1<'a, 'b>(x: &'a i32, y: &'b i32)
```

```
fn f2(x: &mut i32) -> &i32
```



# Примеры

```
fn f1(x: &i32, y: &i32)
```

```
fn f1<'a, 'b>(x: &'a i32, y: &'b i32)
```

```
fn f2(x: &mut i32) -> &i32
```

```
fn f2<'a>(x: &'a mut i32) -> &'a i32
```

```
fn f3(x: Ref<'_, i32>) -> &i32
```

# Примеры

```
fn f1(x: &i32, y: &i32)
```

```
fn f1<'a, 'b>(x: &'a i32, y: &'b i32)
```

```
fn f2(x: &mut i32) -> &i32
```

```
fn f2<'a>(x: &'a mut i32) -> &'a i32
```

```
fn f3(x: Ref<'_, i32>) -> &i32
```

```
fn f3<'a>(x: Ref<'a, i32>) -> &'a i32
```

```
fn f4(x: &i32, y: &i32) -> &i32
```

# Примеры

```
fn f1(x: &i32, y: &i32)
```

```
fn f1<'a, 'b>(x: &'a i32, y: &'b i32)
```

```
fn f2(x: &mut i32) -> &i32
```

```
fn f2<'a>(x: &'a mut i32) -> &'a i32
```

```
fn f3(x: Ref<'_, i32>) -> &i32
```

```
fn f3<'a>(x: Ref<'a, i32>) -> &'a i32
```

```
fn f4(x: &i32, y: &i32) -> &i32
```

```
// Ошибка компиляции, надо указать вж явно
```

```
impl Foo {
```

```
    fn f5(&self, x: &i32, y: &i32) -> &i32
```

```
}
```

# Примеры

```
fn f1(x: &i32, y: &i32)
```

```
fn f1<'a, 'b>(x: &'a i32, y: &'b i32)
```

```
fn f2(x: &mut i32) -> &i32
```

```
fn f2<'a>(x: &'a mut i32) -> &'a i32
```

```
fn f3(x: Ref<'_, i32>) -> &i32
```

```
fn f3<'a>(x: Ref<'a, i32>) -> &'a i32
```

```
fn f4(x: &i32, y: &i32) -> &i32
```

```
// Ошибка компиляции, надо указать вж явно
```

```
impl Foo {
```

```
    fn f5(&self, x: &i32, y: &i32) -> &i32
```

```
    fn f5<'a, 'b, 'c>(&'a self, x: &'b i32, y: &'c i32) -> &'a i32
```

```
}
```


# Примеры

```
struct Foo {  
    x: i32  
}
```

```
impl Foo {  
    fn f(&self, y: &i32) -> &i32 {  
        y  
    }  
}
```

# Примеры

```
struct Foo {  
    x: i32  
}
```

```
impl Foo {  
    fn f<'a, 'b>(&'a self, y: &'b i32) -> &'a i32 {  
          
    }  
}
```

# Примеры

```
struct Foo {  
    x: i32  
}
```

```
impl Foo {  
    fn f<'a, 'b>(&'a self, y: &'b i32) -> &'b i32 {  
        y  
    }  
}
```



Неправильный lifetime elision может приводить к ошибкам вж!

# Примеры

```
struct Ref<'a, T> {  
    r: &'a T  
}
```

```
impl<'a, T> Ref<'a, T> {  
    fn get(&self) -> &T {  
        self.r  
    }  
}
```

```
fn unwrap_ref(r: Ref<'_, i32>) -> &i32 {  
    r.get()  
}
```



# Примеры

```
struct Ref<'a, T> {  
    r: &'a T  
}
```

```
impl<'a, T> Ref<'a, T> {  
    fn get<'b>(&'b self) -> &'b T {  
        self.r // сократили lifetime от 'a до 'b  
    }  
}
```

```
fn unwrap_ref<'a>(r: Ref<'a, i32>) -> &'a i32 {  
    let r: &Ref<'a, i32> = &r;  
    r.get() // вж временной переменной  
}
```

# Примеры

```
struct Ref<'a, T> {  
    r: &'a T  
}
```

```
impl<'a, T> Ref<'a, T> {  
    fn get(&self) -> &'a T {  
        self.r  
    }  
}
```

```
fn unwrap_ref<'a>(r: Ref<'a, i32>) -> &'a i32 {  
    r.get() // ok  
}
```



неправильный elision может привести к ошибкам на call site!

Reborrowing

# Reborrowing

`&'_ T` всегда Copy

```
struct Ref<'a, T> {  
    r: &'a T,  
}
```

```
impl<'a, T> Clone for Ref<'a, T> {  
    fn clone(&self) -> Self { *self }  
}
```

```
impl<'a, T> Copy for Ref<'a, T> {  
}
```

`&'_ mut T` не может быть Copy: получили бы две `&mut` ссылки

# Reborrowing

Почем это код работает?

```
fn reverse(xs: &mut [i32]) {  
    xs.reverse()  
}
```

```
fn do_nothing(xs: &mut [i32]) {  
    reverse(xs);  
    reverse(xs);  
}
```

Если **&mut** не Copy, то вызов `reverse` тратит ссылку...

# Reborrowing

Перепишем (компилятор делает это за нас):

```
fn reverse(xs: &mut [i32]) {  
    xs.reverse()  
}
```

```
fn do_nothing(xs: &mut [i32]) {  
    {  
        let tmp = &mut *xs; // временная ссылка с коротким вж  
        reverse(tmp);      // xs не сдвинут, но заморожен  
    }  
    {  
        let tmp = &mut *xs;  
        reverse(xs);  
    }  
}
```

## reborrowing

автоматическая замена `r` на `&mut *r / & *r` для `&mut` ссылок  
(`&` ссылки копируются)

Алиасинг



# mut

На самом деле в Rust нет const-correctness и неизменяемости

Можно убрать **mut** с локальной переменной:

```
fn sneaky(xs: Vec<i32>) {  
    let mut xs = xs;  
    xs.sort()  
}
```

**mut** нельзя поставить у поля

Можно убрать **mut** для локальных переменных из языка, и ничего не сломается.

# &mut

**mut** в **&mut** не означает изменяемый

Можно придумать более логичный синтаксис:

- **&unique** — уникальная ссылка
- **&** — разделяемая (общая) ссылка

**shared ^ mutable**

Существует единственная уникальная, или произвольное количество разделяемых ссылок

# Alias analysis

## Алиасинг

Наличие более одного указателя (пути) до объекта

Компилятор проводит **точный** alias analysis:

- **&mut** — алиасинга точно нет
- **&** — возможен алиасинг

Алиасинг: фундаментальное явление

# Оптимизации

```
#include <stdio.h>
```

```
void check(int *x, int *y) {  
    *x = 5;  
    *y = 6;  
    printf("%d\n", *x);  
}
```



Можно ли заменить на `printf("%d\n", 5);`?

# Оптимизации

```
#include <stdio.h>
```

```
void check(int *x, int *y) {  
    *x = 5;  
    *y = 6;  
    printf("%d\n", *x);  
}
```



Можно ли заменить на `printf("%d\n", 6);` ?

Да, если `x` и `y` это разные указатели.

# Алиасинг в C

В C используется Type Based Alias Analysis: компилятор в общем случае считает, что `Foo *` и `Bar *` не пересекаются

Следствие: приводить указатели в C — в общем случае UB

Для `check(int*, int*)` не работает (одинаковые типы) нужен **`restrict`**

<https://blog.regehr.org/archives/1307>

## Каст к "префиксу" — UB

```
typedef struct { int i1; int i2; } base;  
typedef struct { int i1; int i2; int i3; } derived;
```

```
base* upcast(derived* d) {  
    return (base*)d; // UB  
}
```

Адрес объекта совпадает с адресом первого поля:

```
typedef struct { int i1; int i2; } base;  
typedef struct { base b; int i3 } derived;
```

```
base* upcast(derived* d) {  
    return &d->b; // OK  
}
```

```
derived* downcast(base* b) {  
    return (derived*)b; // Maybe OK  
}
```

# Rust

В Rust в TBAА нет необходимости, `&mut` даёт строго больше информации

```
fn check(x: &mut i32, y: &mut i32) {  
    *x = 5;  
    *y = 6;  
    println!("{}", *y)  
}
```



Сигнатура гарантирует, что `x` и `y` — разные указатели



# memcpy

```
void * memcpy(void * dst, const void * src, size_t num);
```

The **memcpy()** function copies *n* bytes from memory area *src* to memory area *dst*. The memory areas **must not overlap**. Use **memmove(3)** if the memory areas do overlap.

Если *dst* и *src* не пересекаются, то можно написать более эффективную реализацию.

# Rust

```
impl<T> [T] {  
    pub fn copy_from_slice(&mut self, src: &[T])  
    where  
        T: Copy,  
    { /* вызов memcpy */ }  
  
    pub fn copy_within<R>(&mut self, src: R, dest: usize)  
    where  
        R: RangeBounds<usize>,  
        T: Copy,  
    { /* вызов memmove */ }  
}
```

Аналогично, **self** и `src` не пересекаются

# &mut и memory safety

Если ссылка на объект уникальна, то никакая операция с ним не может сломать указатели снаружи

То, что **&mut T** можно менять следствие уникальности **&mut T** ссылки

# Interior Mutability



Можно ли как-то ослабить условие, при котором можно изменять объект?

Текущая версия: если ссылка на объект **&mut**



Можно ли как-то ослабить условие, при котором можно изменять объект?

Текущая версия: если ссылка на объект **&mut**

Ослабленная версия: если нет ссылок на внутренности объекта

# Cell

Cell это модельный тип с так называемой interior mutability

```
impl<T: Copy> Cell<T> {  
    fn new(value: T) -> Cell<T>;  
    fn get(&self) -> T;  
    fn set(&self, value: T);  
}
```

Можно изменять по & ссылке

Аналогия с **mutable** из C++ не совсем корректа: В Rust нет const-корректности, только анализ алиасов.

Memory Safe: нельзя получить ссылку "внутри" Cell, только копию значения

Cell<T> нужен, если T: Copy и нельзя/сложно корректно использовать &mut (логирование)

```
pub(crate) struct Parser<'t> {
    tokens: &'t[Token],
    pos: usize,
    steps: Cell<u32>,
}

impl<'t> Parser<'t> {
    pub(crate) fn current(&self) -> &'t Token {
        let steps = self.steps.get();
        assert!(steps <= 10_000_000, "the parser seems stuck");
        self.steps.set(steps + 1);
        &self.tokens[self.token_pos]
    }
}
```





Как работает **cell** ?

# UnsafeCell

UnsafeCell — примитив, на котором основана **вся** interior mutability

```
#[lang = "unsafe_cell"]  
#[repr(transparent)] // маркер  
pub struct UnsafeCell<T: ?Sized> {  
    value: T,  
}  
  
impl<T: ?Sized> UnsafeCell<T> {  
    pub fn new(value: T) -> UnsafeCell<T>;  
    pub fn get(&self) -> *mut T;  
}
```

Компилятор считает, что данные за &T , не обёрнутые в UnsafeCell , меняться не могут.

# Unsafe 101

`UnsafeCell` даёт доступ к значению внутри через сырой указатель `*mut T`

`&mut T`

- не null
- `T` валидный живой объект
- нет алиасинга

`*mut T`

- адрес без гарантий

# Unsafe 101

Ссылка приводится к указателю, это безопасно:

```
let p: &mut T = ...;  
let p: *mut T = p;
```

Разыменовывание указателя требует **unsafe** :

```
let p: *mut T = ...  
let p: T = unsafe { *p };
```

Каст указателя к ссылке требует **unsafe** :

```
let p: *mut T = ...  
let p: &mut T = unsafe { &mut *p };
```

# Unsafe 101

```
unsafe fn cast_ptr_to_ref<'a, T>(ptr: *mut T) -> &'a mut T {  
    &mut ptr  
}
```



Что странного в этой сигнатуре?

# Unsafe 101

```
unsafe fn cast_ptr_to_ref<'a, T>(ptr: *mut T) -> &'a mut T {  
    &mut ptr  
}
```



Что странного в этой сигнатуре?

'a встречается только в результате, функция может вернуть любой lifetime

# Cell

```
pub struct Cell<T: ?Sized> {  
    value: UnsafeCell<T>,  
}  
  
impl<T: Copy> Cell<T> {  
    pub fn new(value: T) -> Cell<T> {  
        Cell { value: UnsafeCell::new(value) }  
    }  
    pub fn get(&self) -> T {  
        unsafe { *self.value.get() }  
    }  
    pub fn set(&self, val: T) {  
        let old = self.replace(val);  
        drop(old);  
    }  
    pub fn replace(&self, val: T) -> T {  
        mem::replace(unsafe { &mut *self.value.get() }, val)  
    }  
}
```

# RefCell

Cell работает только с Copy типами. Что если хотим изменять Vec<i32> по & ссылке?

Идея: будем временно выдавать &mut ссылку, поддерживая количество ссылок

```
let cell = RefCell::new(Vec::new());
cell.borrow_mut().push(1); // +1, push, -1
cell.borrow_mut().push(2); // +1, push, -1
let b1 = cell.borrow_mut(); // +1
let b2 = cell.borrow_mut(); // паника, != 0
```



```

pub struct RefCell<T: ?Sized> {
    /// Количество Ref ссылок, если >= 0
    /// -1 -- есть RefMut ссылка
    borrow: Cell<isize>,
    value: UnsafeCell<T>,
}

impl<T> RefCell<T> {
    /// увеличить счётчик и вернуть обёртку над `&T`
    pub fn borrow(&self) -> Ref<'_, T>
    /// поставить -1 и вернуть обёртку над `&mut T`
    pub fn borrow_mut(&self) -> RefMut<'_, T>
}

impl<'a, T> Deref for Ref<'a, T> {
    type Target = T;
    ...
}

impl<'a, T> DerefMut for RefMut<'a, T> { ... }

```

RefCell похожа на однопоточный read/write lock

```

pub struct RefMut<'a, T: ?Sized + 'a> {
    value: &mut 'a T,
    borrow: &'a Cell<isize>,
}

impl<'a, T: ?Sized> RefMut<'a, T> {
    fn new(cell: &'a RefMutCell<T>) -> RefMut<'a, T> {
        let n_readers = cell.borrow().get();
        if n_readers != 0 {
            panic!() // важно для memory safety
        }
        cell.borrow().set(-1);
        RefMut {
            // safe, потому что нет других Ref/RefMut
            value: unsafe { &mut *cell.value.get() },
            borrow: &cell.borrow,
        }
    }
}

```

```
pub struct RefMut<'a, T: ?Sized + 'a> {  
    value: &mut 'a T,  
    borrow: &'a Cell<isize>,  
}
```

```
impl<'a, T: ?Sized> Drop for RefMut<'a, T> {  
    fn drop(&mut self) {  
        self.borrow.set(0)  
    }  
}
```

```
impl<'a, T: ?Sized> DerefMut for RefMut<'a, T> {  
    fn deref_mut(&mut self) -> &mut T { // какой тут lifetime?  
        self.value // reborrowing!  
    }  
}
```

# Guard Pattern

Ref обёртка над &T

В new увеличивает количество ссылок, в drop уменьшает

Из deref возвращает &T, привязанную вж к Ref

⇒

Пока &T жива, Ref<T> тоже жива, количество ссылок больше нуля, и

&mut появится не может

# RefCell

В отличие от `Cell`, можно использовать для любого `T`

Цена:

- можно получить панику, если сделать `borrow_mut` два раза
- в API будут `Ref` и `RefMut` вместо `&` / `&mut`
- маленькая, но не нулевая цена — в runtime поддерживается количество ссылок

# OnceCell



RefCell и Cell конструкции стандартной библиотеки,  
без языковой поддержки

Давайте самостоятельно напишем interior mutability примитив

## OnceCell

- работает с не Copy типами, как RefCell
- get возвращает & ссылки, а не Ref
- можно вызвать .set только один раз (поэтому get safe)

Примитив для написания ленивых значений

```

pub struct OnceCell<T> {
    // Invariant: written to at most once.
    inner: UnsafeCell<Option<T>>,
}

impl<T> OnceCell<T> {
    pub fn new() -> OnceCell<T> {
        OnceCell { inner: UnsafeCell::new(None) }
    }

    pub fn get(&self) -> Option<&T> {
        unsafe { &*self.inner.get() }.as_ref()
    }

    pub fn set(&self, value: T) -> Result<(), T> {
        let slot = unsafe { &mut *self.inner.get() };
        if slot.is_some() { return Err(value); }
        *slot = Some(value);
        Ok(())
    }
}

```

Заведём функцию, удобную в контексте ленивости:

```
impl<T> OnceCell<T> {  
    pub fn get_or_init(&self, f: impl FnOnce() -> T) -> &T {  
        let slot = unsafe { &mut *self.inner.get() };  
        match slot {  
            None => {  
                *slot = Some(f());  
                slot.as_ref().unwrap()  
            }  
            Some(value) => value,  
        }  
    }  
}
```



Заведём функцию, удобную в контексте ленивости:

```
impl<T> OnceCell<T> {  
    pub fn get_or_init(&self, f: impl FnOnce() -> T) -> &T {  
        let slot = unsafe { &mut *self.inner.get() };  
        match slot {  
            None => {  
                *slot = Some(f());  
                slot.as_ref().unwrap()  
            }  
            Some(value) => value,  
        }  
    }  
}
```



В данном случае `unsafe` содержит баг, и может привести к UB

# Реентерабельность

Частый случай неожиданного алиасинга в однопоточном коде — реентерабельность

```
fn main() {  
    let cell: OnceCell<Box<i32>> = OnceCell::new();  
    let mut r1: Option<&i32> = None;  
    let r2: &i32 = cell.get_or_init(|| {  
        r1 = Some(&*cell.get_or_init(|| Box::new(1)));  
        Box::new(2)  
    });  
    let r1: &i32 = r1.unwrap();  
    println!("{}", r1, r2);  
}
```

```
$ ./main  
599433320 2
```

```
impl<T> OnceCell<T> {
    pub fn get_or_init(&self, f: impl FnOnce() -> T) -> &T {
        let slot = unsafe { &mut *self.inner.get() };
        match slot {
            None => {
                // smell: вызов callback в unsafe
                *slot = Some(f());
                slot.as_ref().unwrap()
            }
            Some(value) => value,
        }
    }
}
```

```
impl<T> OnceCell<T> {  
    pub fn get_or_init(&self, f: impl FnOnce() -> T) -> &T {  
        self.get().unwrap_or_else(|| {  
            let inserted = self.set(f());  
            assert!(inserted.is_ok(), "reentrancy");  
            self.get().unwrap()  
        })  
    }  
}
```

```
$ ./main
```

```
thread 'main' panicked at 'reentrancy', main.rs:39:13
```

# Контракт `unsafe`

**unsafe** даёт только четыре возможности:

- позвать **unsafe** функцию
- разыменовать сырой указатель
- реализовать **unsafe trait**
- обратиться к **static mut**

Если функция использует внутри **unsafe**, но сама не помечена **unsafe**, то её можно вызвать с **любыми** аргументами без боязни вызывать UB.

Если сама функция **unsafe**, то у неё есть safety-invariant для аргументов. Проверка инварианта — задача вызывающего кода.

# Контракт unsafe

Что именно нельзя делать в **unsafe** ?

- пока нет чёткого определения (нужна memory model)
- нельзя нарушать правила алиасинга &
- каст & к &**mut** немедленное UB
- нельзя писать через &T в не-UnsafeCell данные
- нельзя создать не-utf8 строку

# Разное

<http://smallcultfollowing.com/babysteps/blog/2014/05/13/focusing-on-ownership/>

Реентерабельность приводит к багам в криптовалютах:

```
transferMoney(  
    src: Addr,  
    dst: Addr,  
    amount: u64,  
    willTransferEvent: Fn(),  
) {  
    if balance[src] >= amount {  
        willTransferEvent();  
        balance[src] -= amount;  
        balance[dst] += amount;  
    }  
}
```