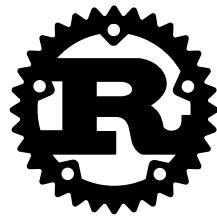


Rust 2019

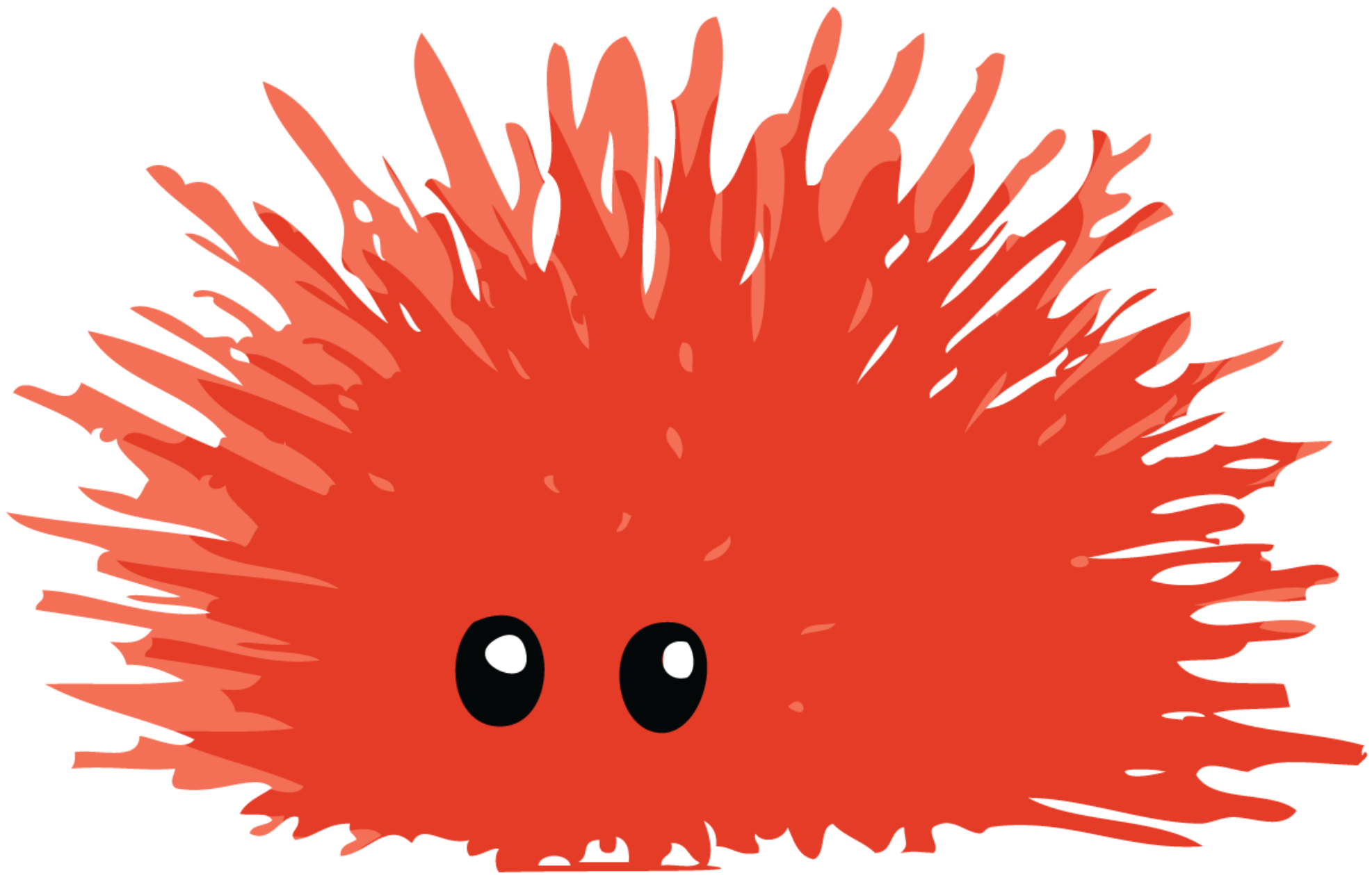
compscicenter.ru

aleksey.kladov@gmail.com



Лекция 11

unsafe



В любом языке нужны низкоуровневые, "системные" операции:

- взаимодействие с OS
- управление памятью
- быстрые коллекции

Где Живёт Интересный Код?

- в runtime языка (Go) / C-расширениях (Python)

Нужно убедиться, что расширение:

1. даёт правильный результат при правильном использовании
2. даёт ошибку при неправильном использовании (нет UB)

-
- написан на самом языке (C/C++)

Язык должен быть достаточно низкоуровневым, сложно отделить "runtime" от бизнес-логики

Rust

В Rust **unsafe** выполняет роль C-расширений

Ядро языка:

- ***const** T, ***mut** T (разыменовывание — **unsafe**)
- **extern "C" fn** malloc(usize); (вызов — **unsafe**)
- &T, &**mut** T, move semantics, alias analysis

Стандартная библиотека:

- Box, Vec, HashMap
- stdio, println!

Rust

unsafe это "эффект": **unsafe** операцию можно выполнить только из **unsafe** функции

Внутри safe функции можно написать **unsafe** блок, но нужно гарантировать, что не возникнет UB

<https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/>

unsafe операции

1. разыменовывание сырого указателя
2. вызов **unsafe** функции
3. **impl** для **unsafe trait**
4. доступ к **static mut**
5. доступ к полям **union**

UB

1. разыменование висящего/нулевого указателя
2. чтение не инициализированной памяти
3. нарушение правил алиасинга (создание перекрывающихся **&mut**)
4. создание невалидных примитивных значений: нулевые/висящие ссылки, нулевые указатели на функцию, `bool` отличный от 0 или 1, невалидный вариант **enum**, невалидный `char` (например, одиночный суррогат), не UTF-8 `str`
5. разматывание стэка через FFI
6. гонка данных

Центральное Обещание Rust

Используя систему типов, можно писать `safe` функции, вызывающие внутри **`unsafe`** операции, но **гарантирующие** отсутствие UB для любых значений параметров

`unsafe` это не инструмент выхода за пределы языка, это основание!

Пример

```
fn get_pair_mut<T>(
    xs: &mut [T],
    idx1: usize,
    idx2: usize,
) -> (&mut T, &mut T)
```

Функция `get_pair_mut` принимает массив, пару индексов и возвращает уникальные ссылки на соответствующие элементы

Пример

```
fn get_pair_mut<T>(
    xs: &mut [T],
    idx1: usize,
    idx2: usize,
) -> (&mut T, &mut T) {

    let x1: *mut T = &mut xs[idx1];
    let x2: *mut T = &mut xs[idx2];

    unsafe {
        assert!(idx1 != idx2);
        (&mut *x1, &mut *x2)
    }
}
```

Вызов `get_pair_mut` с **любыми** `idx1` и `idx2` не может вызвать UB (но может привести к панике)

Пример

```
/// Safety:
/// idx1 and idx2 must not be equal
unsafe fn get_pair_mut<T>(
    xs: &mut [T],
    idx1: usize,
    idx2: usize,
) -> (&mut T, &mut T) {

    let x1: *mut T = &mut xs[idx1];
    let x2: *mut T = &mut xs[idx2];

    (&mut *x1, &mut *x2)
}
```

assert убрать можно, но тогда мы **обязаны** пометить функцию как **unsafe**

Пример

```
fn get_pair_mut<T>(
    xs: &mut [T],
    idx1: usize,
    idx2: usize,
) -> (&mut T, &mut T) {
    let (idx1, idx2) = (idx1.min(idx2), idx1.max(idx2));
    let (lo, hi) = xs.split_at_mut(idx2);
    (&mut lo[idx1], &mut hi[0])
}
```



Написав одну функцию с **unsafe** (`split_at_mut`),
другие можно выразить через неё, и получить
memory safety бесплатно

Параметризованные функции

```
impl<T> [T] {  
    fn sort(&mut self)  
    where  
        T: Ord,  
}
```

Функция `sort` может использовать **unsafe** внутри (например, чтобы убрать проверку выхода за границу массива)

Для любых `T`, `sort` должна гарантировать отсутствие UB (потому что не помечена **unsafe**)



Даже если реализация **T: Ord** не корректна (например, нет транзитивности), UB возникнуть не должно!

Параметризованные функции

Для сравнения, В C++ вызов `std::sort` для типа с не-транзитивным `<` это UB (и может привести к выходу за границу массива на практике)

Баг в `<=>` в modern C++ может привести к UB: весь код `unsafe`

В Rust **unsafe** позволяет установить границы возможного UB

Параметризованные функции

Возможные альтернативы:

```
impl<T> [T] {  
    /// Safety:  
    /// Ord must be a correct total order for T  
    unsafe fn sort_faster(&mut self)  
    where  
        T: Ord,  
}
```

Проверка контракта — задача вызывающего `sort_faster`

Параметризованные функции

Возможные альтернативы:

```
/// Safety:  
/// Implementors guarantee that Ord is a correct total order  
unsafe trait TrustedOrd: Ord {}  
  
impl<T> [T] {  
    fn sort_faster(&mut self)  
    where  
        T: TrustedOrd,  
}
```

Проверка контракта — задача автора типа T

Send & Sync

Send и Sync — интересные **unsafe trait**

Автор **unsafe impl** Sync **for** T обязан гарантировать отсутствие гонок данных

В большинстве случаев автор — компилятор

Если все компоненты типа Sync, то сам тип тривиально Sync

Напишем std

Show Me the Code

[https://github.com/rust-
lang/rust/blob/bfb443eb1de484fde141fa9090a9f4291cbe60a5/](https://github.com/rust-lang/rust/blob/bfb443eb1de484fde141fa9090a9f4291cbe60a5/)

Нас будет интересовать `src/libcore`

[T]

libcore/slice/mod.rs:

```
#[repr(C)]  
struct FatPtr<T> {  
    data: *const T,  
    len: usize,  
}
```

FatPtr — представление `&[T]` в runtime: пара из указателя и длины

[T]

libcore/slice/mod.rs:

```
#[repr(C)]  
union Repr<'a, T: 'a> {  
    rust: &'a [T],  
    rust_mut: &'a mut [T],  
    raw: FatPtr<T>,  
}
```

C-style, **unsafe** union: содержит одно из полей, без тэга

Доступ к полю **union** — **unsafe** операция

Изначально исключительно для FFI, но нашёл применение в **unsafe** коде

В данном случае: способ получить FatPtr<T> из &T

[T]

```
#[repr(C)]
union Repr<'a, T: 'a> {
    rust: &'a [T],
    rust_mut: &'a mut [T],
    raw: FatPtr<T>,
}

impl<T> [T] {
    pub const fn len(&self) -> usize {
        unsafe {
            Repr { rust: self } ①
                .raw             ②
                .len
        }
    }
}
```

① скоструировали Repr из [T]

② получили FatPtr<T>


```

use std::mem::{size_of, align_of};

pub unsafe fn from_raw_parts<'a, T>(
    data: *const T,
    len: usize,
) -> &'a [T] {
    debug_assert!(
        data as usize % align_of::<T>() == 0,
        "attempt to create unaligned slice",
    );
    debug_assert!(
        size_of::<T>().saturating_mul(len) <= isize::MAX as usize,
        "attempt to create slice covering half the address space",
    );
    Repr { raw: FatPtr { data, len } }.rust
}

```

Конструктор слайсов: сделали `FatPtr<T>`, посмотрели как на `&[T]`

[T]

Немного компиляторной магии для синтаксиса [T], в остальном библиотечный код

Представление — repr(C) структура

Каст между &[T] и FatPtr<T> через **union**

Знаем, что слайсы представлены как FatPtr<T>, потому что так работают from_raw_parts и from_raw_parts_mut

[T]

```
impl<T> [T] {  
    pub const fn as_ptr(&self) -> *const T {  
        self as *const [T] as *const T ①  
    }  
}
```

- ① получить указатель можно без **unsafe** (тоже магия компилятора)!

[T]

```
pub fn get<I: SliceIndex<Self>>(&self, index: I)
    -> Option<&I::Output>
{
    index.get(self)
}
```

```
pub fn get_mut<I: SliceIndex<Self>>(&mut self, index: I)
    -> Option<&mut I::Output>
{
    index.get_mut(self)
}
```

Индексация — через вспомогательный трейт `SliceIndex`, чтобы работали `.get(0)` и `.get(0..10)`

```

impl<T> SliceIndex<[T]> for usize {
    type Output = T;
    fn get(self, slice: &[T]) -> Option<&T> {
        if self < slice.len() {
            unsafe { Some(self.get_unchecked(slice)) }
        } else {
            None
        }
    }
    unsafe fn get_unchecked(self, slice: &[T]) -> &T {
        &*slice.as_ptr().add(self)
    }
}

impl<T: ?Sized> *const T {
    pub unsafe fn add(self, count: usize) -> Self where T: Sized
}

```

<***const** T>::add **unsafe** — указатель должен быть in-bounds

```

impl<T> SliceIndex<[T]> for ops::Range<usize> {
    type Output = [T];

    fn get(self, slice: &[T]) -> Option<&[T]> {
        if self.start > self.end || self.end > slice.len() {
            None
        } else {
            unsafe { Some(self.get_unchecked(slice)) }
        }
    }

    unsafe fn get_unchecked(self, slice: &[T]) -> &[T] {
        from_raw_parts(
            slice.as_ptr().add(self.start),
            self.end - self.start,
        )
    }
}

```

split_at_mut

```
pub fn split_at_mut(&mut self, mid: usize) -> (&mut [T], &mut [T]) {  
    let len = self.len();  
    let ptr = self.as_mut_ptr();  
  
    unsafe {  
        assert!(mid <= len);  
  
        (from_raw_parts_mut(ptr, mid),  
         from_raw_parts_mut(ptr.add(mid), len - mid))  
    }  
}
```

assert необходим!

IterMut

```
pub struct IterMut<'a, T: 'a>(&'a mut [T]);

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T; ❶

    fn next(&mut self) -> Option<&'a mut T> {

        if self.0.is_empty() { return None; }

        let (l, r) = self.0.split_at_mut(1);
        self.0 = r;
        l.get_mut(0)
    }
}
```

- ❶ Магия! Повторные вызовы `next` гарантируют
непересекающиеся `&mut` ссылки

IterMut

```
pub struct IterMut<'a, T: 'a>(&'a mut [T]);

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;

    fn next(&mut self) -> Option<&'a mut T> {

        if self.0.is_empty() { return None; }

        let (l, r) = self.0.split_at_mut(1); ❶
        self.0 = r;
        l.get_mut(0)
    }
}
```

- ❶ reborrowing, потому что не можем сделать move из поля `&mut` значения :-)

swap trick

```
pub struct IterMut<'a, T: 'a>(&'a mut [T]);

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;

    fn next(&mut self) -> Option<&'a mut T> {
        let slice = std::mem::replace(&mut self.0, &mut []);
        if slice.is_empty() { return None; }

        let (l, r) = slice.split_at_mut(1);
        self.0 = r;
        l.get_mut(0)
    }
}
```

swap trick

В реальности `IterMut` устроен по другому — как пара указателей

```
struct IterMut<'a, T> { // unused type parameter
    begin: *mut T,
    end: *mut T,
}
```

Variance

Subtyping

В Rust есть отношение "быть подтипом" на временах жизни:

`'a: 'b`

Читается как `'a` outlives `'b`

`&'a T` это подтип `&'b T` — сокращение вж не нарушает memory safety

```

fn min<'a>(xs: &Vec<'a str>, default: &'a str) -> &'a str {
    ...
}

fn main() {
    let xs: Vec<'static str> = vec!["hello", "world"];
    let local: String = "spam".to_string()
    let m = min(&xs, local.as_str())
}

```

Из `Vec<&'static str>` и `&'a str` можно выбрать `&'a str`

Так как `&'static str <: &'a str`
 то `Vec<'static str> <: Vec<'a str>`



Vec<T> ковариантен по T

```
fn assign_ref<'a>(r: &mut &'a str, s: &'a str) {  
    *r = s  
}
```

```
fn evil(r: &mut &'static str) {  
    let local: String = "spam".to_string();  
    let local: &str = local.as_str();  
    assign_ref(r, local)  
}
```

```
fn main() {  
    let mut hello: &'static str = "hello";  
    evil(&mut hello);  
    println!("{}", hello);  
}
```



```
fn assign_ref<'a>(r: &mut &'a str, s: &'a str) {  
    *r = s  
}
```

```
fn evil(r: &mut &'static str) {  
    let local: String = "spam".to_string();  
    let local: &str = local.as_str();  
    assign_ref(r, local)  
}
```

```
fn main() {  
    let mut hello: &'static str = "hello";  
    evil(&mut hello);  
    println!("{}", hello);  
}
```



&'a mut T инвариантна по T

<code>T</code>	ковариантность
<code>&'a T</code>	ковариантность по <code>'a</code> и <code>T</code>
<code>&'a mut T</code>	ковариантность по <code>'a</code> , инвариантность по <code>T</code>
<code>*const T</code>	ковариантность
<code>*mut T</code>	инвариантность
<code>fn(T)</code>	контрвариантность
<code>fn() -> T</code>	ковариантность
<code>fn(T) -> T</code>	инвариантность
<code>Cell<&'a T></code>	инвариантность

Единственная разница между `*const T` и `*mut T` — variance

И `&'a T`, и `&'a mut T` ковариантны по `'a`

PhantomData

```
struct S<'a, T> {  
    ...  
}
```



Как определить вариантность по 'a и T?

PhantomData

```
struct S<'a, T> {  
    xs: &'a mut Vec<T>,  
}
```



Как определить вариантность по 'a и T?

Автоматически, из определения S

PhantomData

Если параметр типа не используется, то нельзя определить
вариантность \Rightarrow все параметры должны использоваться

PhantomData<T>

Магический ZST тип, который ведёт себя как T с точки зрения
variance и dropcheck

```
struct A<'a, T> {  
    value: T,  
    r: &'a ()  
}
```

```
struct B<'a, T> {  
    value: &'a T,  
}
```

А вызывает деструктор T, B нет

```
use std::marker::PhantomData;
```

```
struct IterMut<'a, T> {  
    begin: *mut T,  
    end: *mut T,  
    slice: PhantomData<&'a mut [T]>,  
}
```

```
impl<'a, T> IterMut<'a, T> {  
    fn new(slice: &'a mut [T]) -> Self {  
        assert!(std::mem::size_of::<T>() > 0);  
        let begin = slice as *mut [T] as *mut T;  
        let end = unsafe { begin.add(slice.len()) };  
        IterMut {  
            begin,  
            end,  
            slice: PhantomData,  
        }  
    }  
}
```

```

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;

    fn next(&mut self) -> Option<&'a mut T> {
        if self.begin == self.end {
            return None;
        }
        let curr = self.begin;
        unsafe {
            self.begin = self.begin.add(1);
            Some(&mut *curr)
        }
    }
}

```


Nonnull<T>

Для написания структур данных часто нужен ковариантный указатель, из которого удобно получать **&mut T**

`std::ptr::Nonnull<T>` — как раз такой тип

```
size_of::<Option<Nonnull<T>>>()  
== size_of::<Nonnull<T>>()
```

```

#[rustc_layout_scalar_valid_range_start(1)]
pub struct NonNull<T: ?Sized> {
    pointer: *const T,
}
impl<T: ?Sized> NonNull<T> {
    pub const unsafe fn new_unchecked(ptr: *mut T) -> Self {
        NonNull { pointer: ptr as _ }
    }
    pub fn new(ptr: *mut T) -> Option<Self> {
        if !ptr.is_null() {
            Some(unsafe { Self::new_unchecked(ptr) })
        } else {
            None
        }
    }
    pub const fn as_ptr(self) -> *mut T {
        self.pointer as *mut T
    }
    pub unsafe fn as_mut(&mut self) -> &mut T {
        &mut *self.as_ptr()
    }
}

```

