



Rust and what's this thing for



Abc Xyz

@dura_lex

https://gitlab.com/saruman9/rust_pres

<https://bit.ly/2SrcofT>



Agenda

1. Foreword

2. What is Rust?

3. (Un)safe

4. Syntax

5. Ecosystem

6. Pitfalls

7. Experience

8. Summary

Foreword



В последнее время **всё чаще** в IT новостях можно услышать, что был разработан тот или иной проект на **Rust**, будь то Web (WebAssembly) или Embedded (TrustZone, IoT) или GameDev или консольная утилита или новая ОС или файловая система или кодек или декодировщик и т. д. Интерес к языку среди программистов растёт всё больше и больше. Вот уже и **среди безопасников** интерес также подогревается, не смотря на то, что многие относятся скептически. Я даже **в исследовательском центре** всё чаще сталкиваюсь с проектами на Rust: при изучении ChromeBook (система виртуализации, песочница), блокчейн проектов (Rust здесь практически лидирует), изучение новых фаззеров (на Rust), в IoT устройствах.



Я отчасти Rust евангелист (в принципе как и евангелист Linux и KISS принципа), поэтому буду призывать попробовать данный язык по возможности. Статистика показывает, что **люди**, которые перешагнули определённый порог при изучении, получают удовольствие от того, что разрабатывают на Rust.

Motivation



Мне интересны языки программирования, поэтому я знаю, что существует ещё множество других интересных и полезных языков (Racket, Haskell, Ada, OCaml, SPARK, Coq, Nim и др.), поэтому выбирайте язык исходя из задачи, а не задачу, исходя из известного вам языка. В своём докладе я сделаю упор на сам язык, а не на его аудит, т. к. на сколько мне известно, до сих пор небольшой процент людей нашей компании знаком с данным языком. Это недоразумение я и хочу исправить.

- Since 1.0.0
- Scope (by time)
 - Bindings (FFI – foreign function interface)
 - Analyzers
 - CLI (TUI) tools for PC and IoT
 - GUI for fun
 - Libraries
 - RE
- Nim, Crystal, Zig, Pony



Начал более-менее изучать Rust в то время, когда его версия стала **стабильной**, до этого только смотрел примеры кода. В то время занимался «железом», поэтому интересно было изучать такие языки, как C, Ada (Spark), Rust.

Начинал с того, с чего обычно не начинают — с байндингов (FFI) для нужных мне инструментов. Затем стал разрабатывать **анализатор НДВ**. Между делом для себя писал консольные тулы (**слез с Python**). Затем помогал в разработке различных библиотек. Знаком и писал на других **новых языках**, также писал и немного знаю концепции C, Ada, Haskell, Lisp, Python.

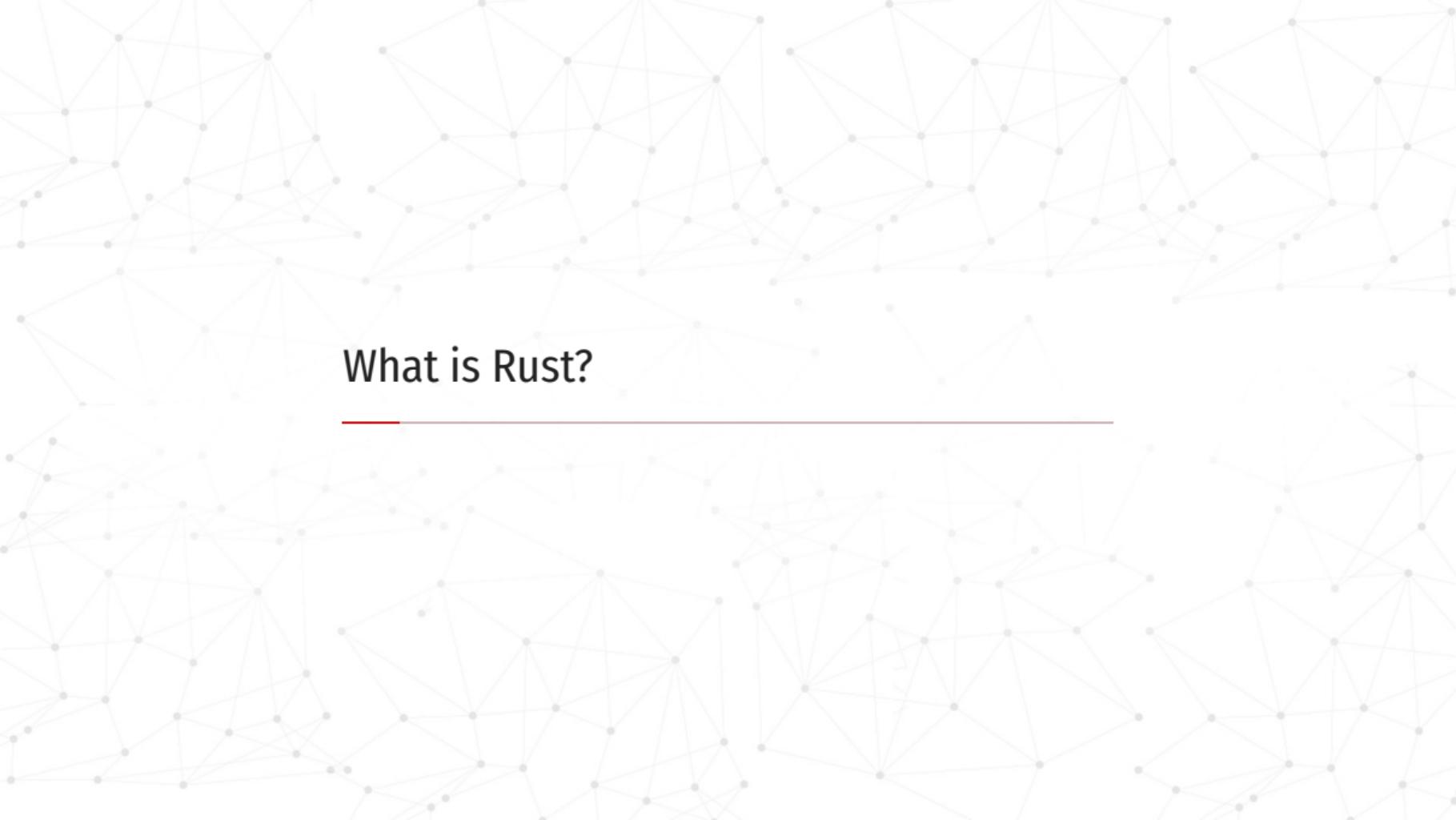
I'm not true programmer



Я не программист и не реверсер. На программиста **не учился**, читал книги, поэтому много пробелов в матчасти.

Буду рассказывать, исходя только из **собственного опыта**, моё мнение может не совпадать с мнением многих других людей (постоянные холивары, некоторые уже вчера смогли убедиться в этом).

Я не буду учить вас Rust, я просто расскажу основные концепции и фишки.



What is Rust?

What is Rust?

“Rust is a multi-paradigm systems programming language focused on safety, especially safe concurrency”.

— Wikipedia

Википедия говорит, что Rust – это мультипарадигменный системный язык программирования, нацеленный на безопасность, особенно на безопасность при параллелизме.

What is Rust?

“Rust is a systems programming language that runs blazingly fast, prevents nearly all segfaults, and guarantees thread safety”.

— www.rust-lang.org (2015)

На официальном сайте раньше подчёркивались такие черты языка, как **быстрая скорость выполнения, предотвращение ошибок сегментации и гарантия безопасности при работе с потоками**.

What is Rust?

“Empowering everyone to build reliable and efficient software”.

— www.rust-lang.org

На данный момент язык находится в процессе расширения и смены ориентации, отсюда появляются **другие цели**.

What is Rust?

Quick facts about Rust

Quick facts about Rust

- Started by Mozilla (sponsorship & support) employee Graydon Hoare
- Influenced by C++ & Haskell and others
- First announced by Mozilla in 2010
- Community driven development
- 88,281 commits on GitHub
- First stable release: 1.0 in May 2015
- Latest stable release: 1.32

- Был написан на OCaml, начат в 2006.
- Смесь C++ (синтаксис на вид), Cyclone, функциональных языков, в частности — Haskell (внутренности) и OCaml.
- Mozilla поддержала инициативу Грейдона для написания своего движка для браузера — Servo.
- Любой может виться в разработку, сделать своё предложение в виде RFC — Request for Comments, в отличие от C++ и других.
- Является одним из самых популярных репозиториев на GitHub, самым любимым языком на StackOverflow 2016, 2017, 2018 годов.
- Недавно произошло радикальное изменение языка (2018 edition), но обратная совместимость от этого не пострадала.

What is Rust?

Why Rust?

Why Rust?



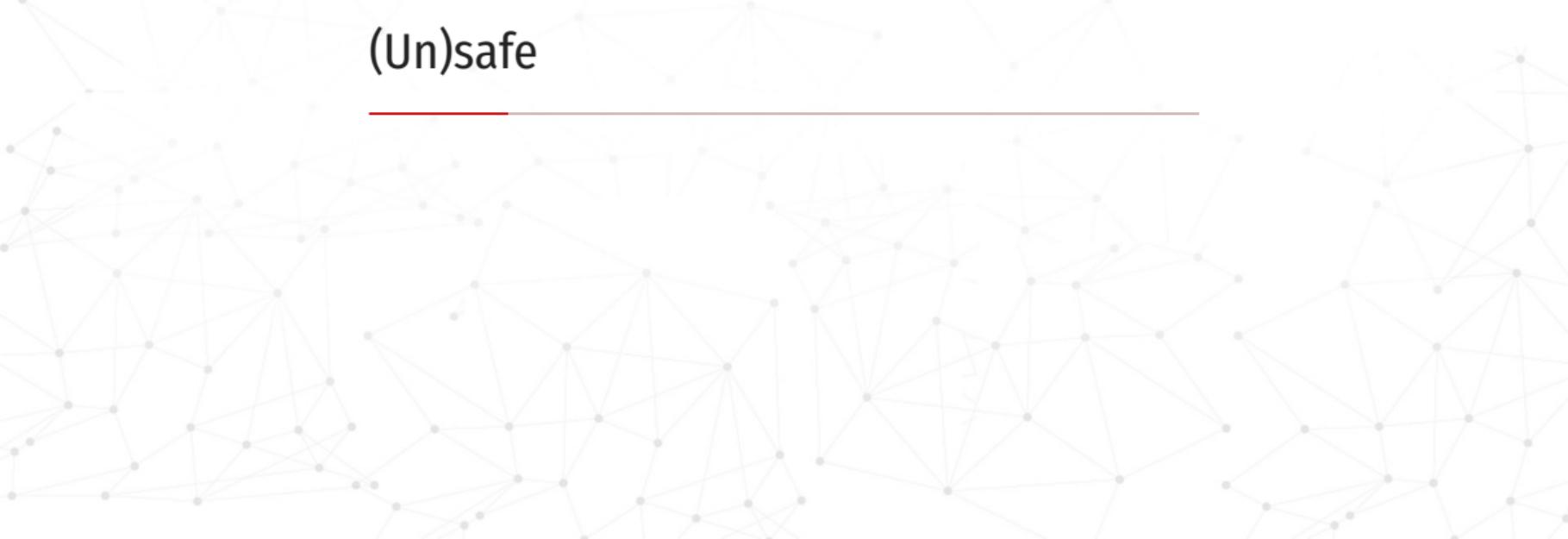
- Performance
 - Fast, memory-efficient
 - No runtime or garbage collector
 - Zero-cost abstractions
- Reliability
 - Rich type system
 - Ownership model
- Productivity
 - Documentation
 - Friendly compiler
 - Top-notch tooling

Какие характерные черты данного языка можно выделить?

- Быстрый за счёт **нулевой стоимости абстракций**, без GC, поэтому можно встраивать в другие языки, в критические сервисы, запускать на встраиваемых системах.
- Богатая система типов и модель владения, отсюда — **безопасная работа с памятью** и потоками, многие ошибки отлавливаются **во время компилирования**.
- Благодаря **дружелюбному сообществу** и **open-source разработке** у Rust много преимуществ.



(Un)safe

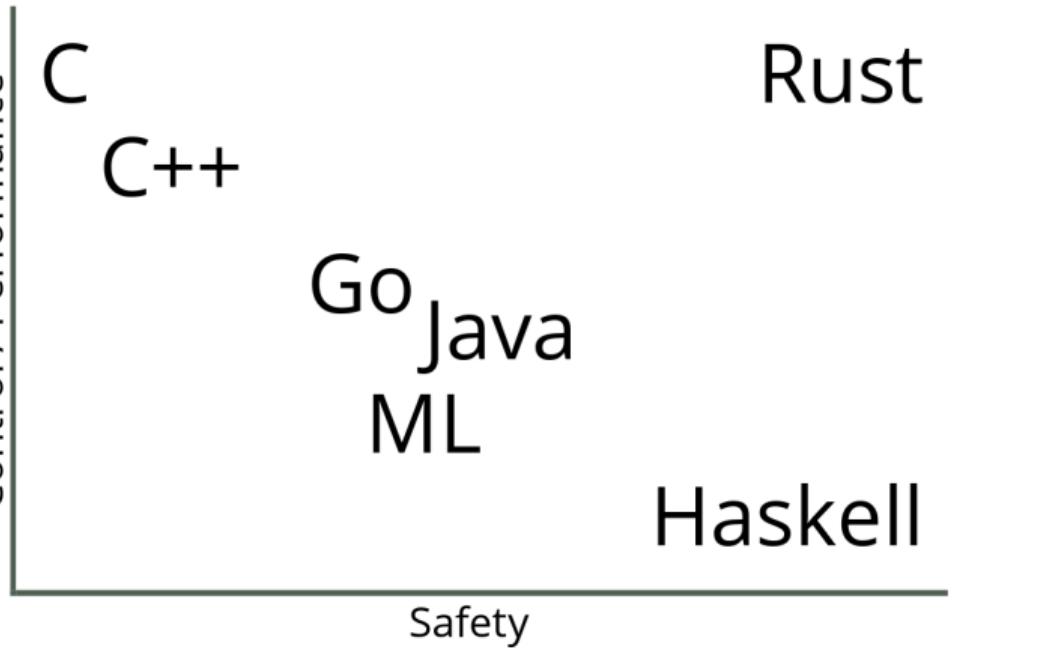




(Un)safe

Control vs Safety

Control vs Safety



Есть низкоуровневые языки, которые позволяют контролировать всё и вся, жертвуя при этом безопасностью. Есть высокоуровневые языки, которые менее производительны, зато более безопасные. Также есть языки, с богатой системой типов, которые отличаются своей безопасностью, но не производительностью (IDRIS). А есть Rust!

(Un)safe

What's wrong with systems languages?

What's wrong with systems languages?

- It's difficult to write secure code
- It's very difficult to write multithreaded code

Freedom to shoot yourself in the foot is not a rust marketing point © Rust

Программируя на системных ЯП, сталкиваешься с определёнными проблемами (на слайде).
Rust был задуман как раз **для решения этих проблем**. Конечно же, это не серебряная пуля.



(Un)safe

Problems



Problems

Memory corruption

- Using uninitialized memory
- Using non-owned memory (null pointer, dangling pointer dereference, out of bounds error)
- Using memory beyond the memory that was allocated (buffer overflow)
- Faulty heap memory management (memory leaks, freeing non-heap or un-allocated memory)



В системных (и не только: Java, C#) языках **существуют проблемы с безопасностью при работе с памятью**. Rust нацелен на решение этих самых проблем.

Примеры проблем:

- использование не инициализированной памяти;
- использование не принадлежащей программе памяти;
- использование памяти за пределами выделенной;
- ошибки управления памятью на куче.

(Un)safe

Ownership and Borrowing



Ownership and Borrowing

Nicholas Matsakis

Для того, чтобы понять, почему Rust считается безопасным языком, нужно понимать, как он работает, какие концепции используются.

Сейчас будет большой блок, он может показаться долгим и скучным, но советую внясть ему, потому что это ядро Rust.

Мне было лень, поэтому я взял хорошо оформленные и понятные слайды презентации Николаса Матсакиса (один из главных разработчиков Rust), за что ему большое спасибо.

Владение и заимствование — вот одни из главных концепций ЯП Rust.
На следующих слайдах я расскажу, что это такое.

Ownership

n. The act, state, or right of possessing something.

Borrow

v. To receive something with the promise of returning it.

И так, концепция владения.
У нас есть книга, мы ею владеем.



Ownership

Как только мы передаём книгу, мы перестаём ей владеть. У книги появляется новый хозяин.



Ownership

```
fn main() {  
    let name = format!("...");  
    helper(name);  
    helper(name);  
}
```



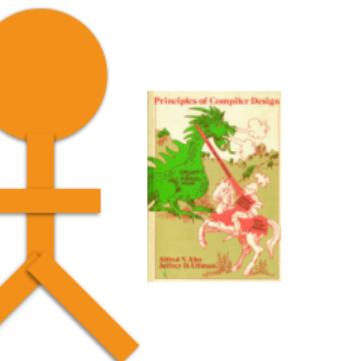
Ownership

```
fn helper(name: String) {  
    println!(...);  
}
```

Рассмотрим на примере кода.

У нас есть переменная `name` (наша книга) типа `String`.

```
fn main() {  
    let name = format!("...");  
    → helper(name);  
    helper(name);  
}
```



Ownership

```
fn helper(name: String) {  
    println!(...);  
}
```



Также у нас есть функция `helper` – будущий хозяин нашей строки (книги). Что же происходит при вызове функции?

```
fn main() {  
    let name = format!("...");  
    helper(name);  
    helper(name);  
}
```

```
fn helper(name: String) {  
    println!(...);  
}
```

Take ownership
of a String



Ownership



Посмотрим на определение функции `helper` — в качестве параметра она принимает объект с типом `String`. Она принимает **не ссылку и не само значение**, а именно возможность владения данным **объектом**, если можно так сказать.

```
fn main() {  
    let name = format!("...");  
    helper(name);  
    helper(name);  
}
```



Ownership

```
fn helper(name: String) {  
    println!(...);  
}
```



Возвращаемся обратно, вызываем функцию.

```
fn main() {  
    let name = format!("...");  
    helper(name);  
    helper(name);  
}
```

```
fn helper(name: String) {  
    println!(...);  
}
```



Ownership

При вызове функции у нас **происходит смена владельца** нашей книги.

```
fn main() {  
    let name = format!("...");  
    helper(name);  
    helper(name);  
}
```

```
fn helper(name: String) {  
    println!(...);  
}
```



Ownership

При вызове функции у нас **происходит смена владельца** нашей книги.

```
fn main() {  
    let name = format!("...");  
    helper(name);  
    helper(name);  
}
```

```
fn helper(name: String) {  
    println!(...);  
}
```



Ownership



После того, как функция отработала — она, то бишь хозяин, исчезает.

```
fn main() {  
    let name = format!("...");  
    helper(name);  
    helper(name);  
}
```

```
fn helper(name: String) {  
    println!(...);  
}
```



Ownership

А если исчезает хозяин, то вместе и с ним объект, которым он владел. Это называется RAII
(Resource acquisition is initialization) – получение ресурса есть инициализация.

```
fn main() {  
    let name = format!("...");  
    helper(name);  
     helper(name);  
}
```

```
fn helper(name: String) {  
    println!(...);  
}
```

Если мы попытаемся **ещё раз вызвать функцию**, которая в качестве аргумента принимает объект, которого **у нас уже нет**, т. к. мы его уже отдали, **что же тогда произойдёт?**



Ownership

Правильно, ошибка — использование уже перемещённого значения.

```
fn main() {  
    let name = format!("...");  
    helper(name);  
    helper(name);  
}
```

```
fn helper(name: String) {  
    println!(...);  
}
```

Error: use of moved value: `name`



Ownership

```
void main() {  
    Vector name = ...;  
    helper(name);  
    helper(name);  
}
```

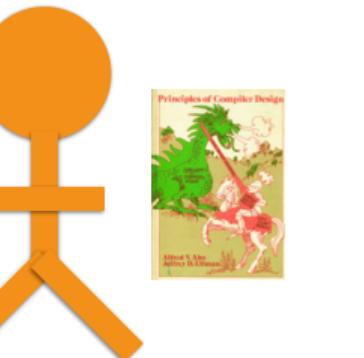


“Ownership” in Java

```
void helper(Vector name) {  
    ...  
}
```

Посмотрим как данная концепция (или её подобие) реализована в других языках.
У нас также есть объект, но типа `Vector` (что по сути также может быть строкой, вектором символов).

```
void main() {  
    Vector name = ...;  
     helper(name);  
    helper(name);  
}
```



“Ownership” in Java

```
void helper(Vector name) {  
    ...  
}
```

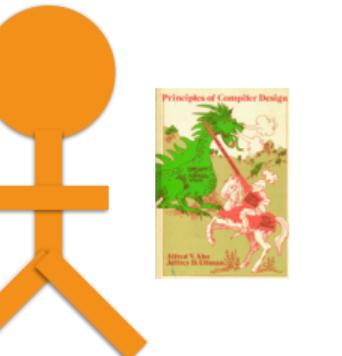


У нас есть функция, которая также в качестве параметра принимает объект типа Vector.

```
void main() {  
    Vector name = ...;  
    → helper(name);  
    helper(name);  
}
```

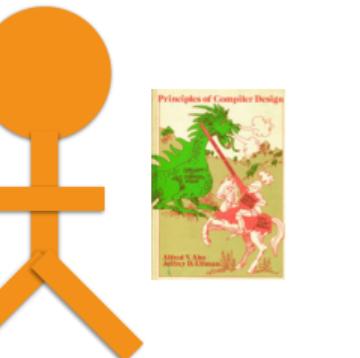
```
void helper(Vector name) {  
    ...  
}  
↑  
Take reference  
to Vector
```

Но в данном случае параметр функции передаётся в качестве ссылки.



“Ownership” in Java

```
void main() {  
    Vector name = ...;  
    → helper(name);  
    helper(name);  
}
```



“Ownership” in Java

```
void helper(Vector name) {  
    ...  
}
```



Снова возвращаемся и вызываем нашу функцию.

```
void main() {  
    Vector name = ...;  
    helper(name);  
    helper(name);  
}
```

```
→ void helper(Vector name) {  
    ...  
}
```



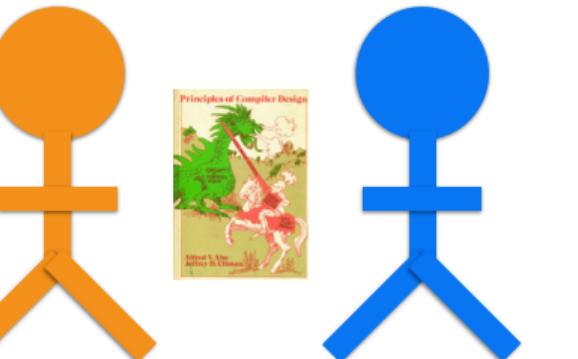
“Ownership” in Java

В Java при вызове функции у нас оказывается два хозяина одного объекта.

```
void main() {  
    Vector name = ...;  
    helper(name);  
    helper(name);  
}
```

```
→ void helper(Vector name) {  
    ...  
}
```

Хорошо. Продолжаем выполнение функции.

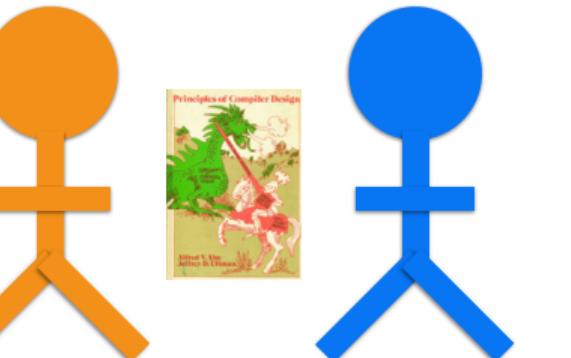


“Ownership” in Java

```
void main() {  
    Vector name = ...;  
    helper(name);  
    helper(name);  
}
```

```
void helper(Vector name) {  
    ...  
}
```

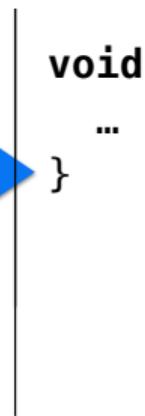
Функция закончила своё выполнение.



“Ownership” in Java

```
void main() {  
    Vector name = ...;  
    helper(name);  
    helper(name);  
}
```

```
void helper(Vector name) {  
    ...  
}
```



Функция исчезает, как и в Rust. Но что же происходит с объектом?

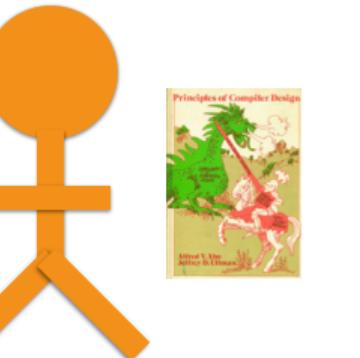


“Ownership” in Java

```
void main() {  
    Vector name = ...;  
    helper(name);  
  
    helper(name);  
}
```

```
void helper(Vector name) {  
    ...  
}
```

Объект никуда не пропадает, он остаётся у своего первоначального хозяина.
Также возможен повторный вызов функции helper.



“Ownership” in Java

```
void main() {  
    Vector name = ...;  
    helper(name);  
    helper(name);
```



```
void helper(Vector name) {  
    ...  
}
```

Главная функция заканчивает свою работу и также исчезает.



“Ownership” in Java

```
void main() {  
    Vector name = ...;  
    helper(name);  
    helper(name);  
}
```

```
void helper(Vector name) {  
    ...  
}
```

При этом созданный функцией объект остаётся. Для чего это нужно?



```
void main() {  
    Vector name = ...;  
    helper(name);  
    helper(name);  
}
```

```
void helper(Vector name) {  
    new Thread(...);  
}
```

Представим, что функция `helper` внутри себя создаёт поток.



```
void main() {  
    Vector name = ...;  
    helper(name);  
    helper(name);  
}
```

```
void helper(Vector name) {  
    new Thread(...);  
}
```

Тогда хозяином объекта, даже после исчезновения главной функции, будет функция `helper`.



“Ownership” in Java

```
void main() {  
    Vector name = ...;  
    helper(name);  
    helper(name);  
}
```

```
void helper(Vector name) {  
    new Thread(...);  
}
```

Когда поток закончит свою работу, то текущий хозяин объекта также перестанет существовать.



```
void main() {  
    Vector name = ...;  
    helper(name);  
    helper(name);  
}
```

```
void helper(Vector name) {  
    new Thread(...);  
}
```

Ну а сборщик мусора сделает своё дело...



```
void main() {  
    Vector name = ...;  
    helper(name);  
    helper(name);  
}
```

```
void helper(Vector name) {  
    new Thread(...);  
}
```

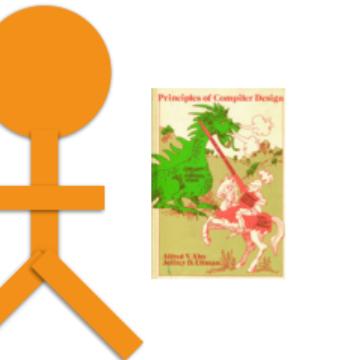
... и удалит уже никому ненужный объект.

Рассмотрим одну интересную особенность в Rust при передаче права владения объектом, а именно концепцию **клонирования**.

Clone

```
fn main() {  
    let name = format!("...");  
    helper(name.clone());  
    helper(name);  
}
```

```
fn helper(name: String) {  
    println!(...);  
}
```

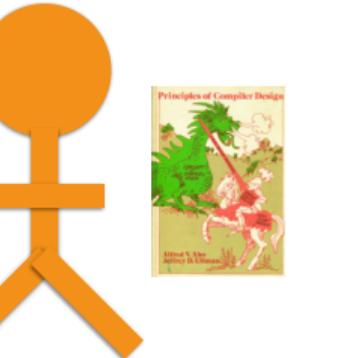


Всё также вызываем нашу функцию `helper`.

Clone

```
fn main() {  
    let name = format!("...");  
    helper(name.clone());  
    helper(name);  
}
```

```
fn helper(name: String) {  
    println!(...);  
}
```

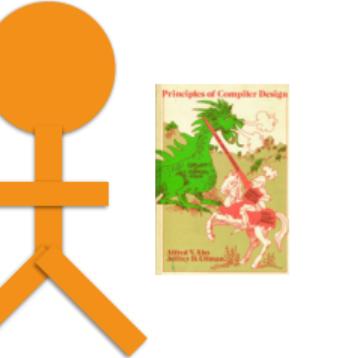


Но при передаче прав владения объектом **мы делаем клон** этого самого объекта!

Clone

```
fn main() {  
    let name = format!("...");  
    helper(name.clone());  
    helper(name);  
}
```

Copy the String



```
fn helper(name: String) {  
    println!(...);  
}
```

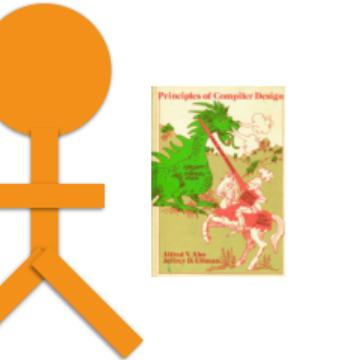


То есть в функцию мы отдаём не сам объект, а его клон.

Clone

```
fn main() {  
    let name = format!("...");  
    helper(name.clone());  
    helper(name);  
}
```

```
fn helper(name: String) {  
    println!(...);  
}
```

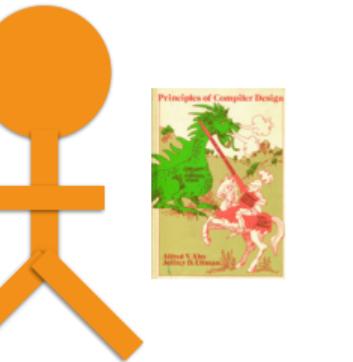


Вызываем функцию.

Clone

```
fn main() {  
    let name = format!("...");  
    helper(name.clone());  
    helper(name);  
}
```

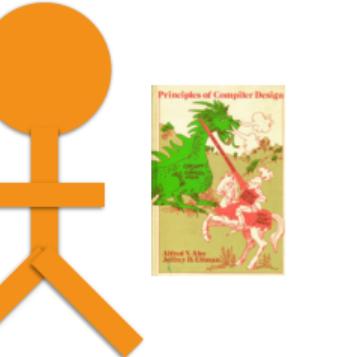
fn helper(name: String) {
 println!(...);
}



После того, как функция отработает, **она исчезает** и исчезает аргумент, как было показано ранее, но при этом **исчезает клон объекта**, а не сам объект!

Clone

```
fn main() {  
    let name = format!("...");  
    helper(name.clone());  ➔ }  
    helper(name);  
}
```

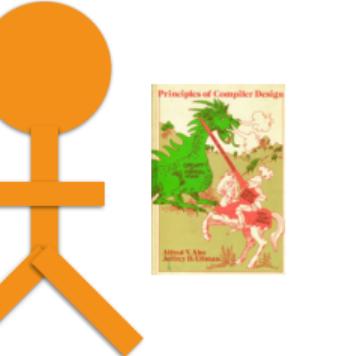


После этого мы повторно можем вызвать функцию `helper`, но передать уже оригинальный объект.

Clone

```
fn main() {  
    let name = format!("...");  
    helper(name.clone());  
    helper(name);  
}
```

```
fn helper(name: String) {  
    println!(...);  
}
```



В этом случае, мы, конечно же, его уже теряем.

Clone

```
fn main() {  
    let name = format!("...");  
    helper(name.clone());  
     helper(name);  
}
```

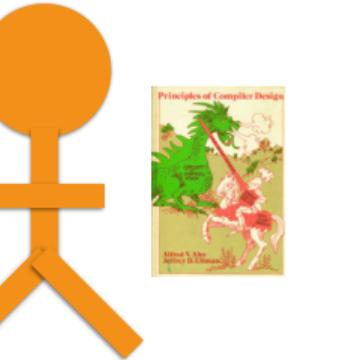


Есть один небольшой нюанс в концепции клонирования — **копирование** или по-другому **авто-клонирование**.

Copy (auto-Clone)

```
fn main() {  
    let count = 22;  
    helper(count);  
    helper(count);  
}
```

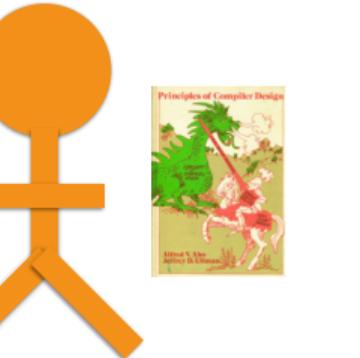
```
fn helper(count: i32) {  
    println!(..);  
}
```



Если тип объекта обладает свойством копирования (реализован типаж), то при передаче прав владения клонирование происходит автоматически.

Copy (auto-Clone)

```
fn main() {  
    let count = 22;  
    helper(count);  
    helper(count);  
}
```

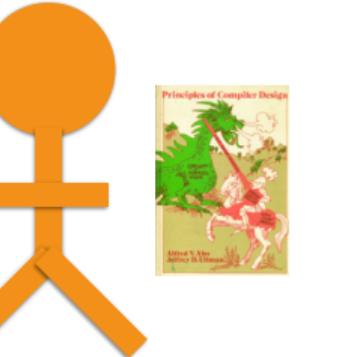


```
fn helper(count: i32) {  
    println!(..);  
}  
i32 is a Copy type
```



Copy (auto-Clone)

```
fn main() {  
    let count = 22;  
    → helper(count);  
    helper(count);  
}
```



```
fn helper(count: i32) {  
    println!(..);  
}  
i32 is a Copy type
```



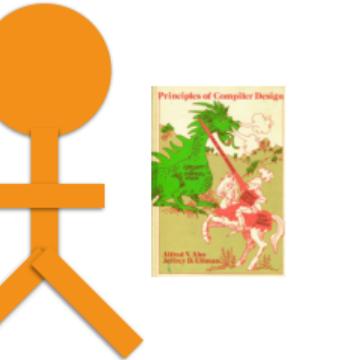
Вызовем функцию `helper` в первый раз, передав при этом целочисленный тип `i32`, который обладает свойством `Copy`.

Произойдёт автоматическое клонирование при передаче прав владения.

Copy (auto-Clone)

```
fn main() {  
    let count = 22;  
    → helper(count);  
    helper(count);  
}
```

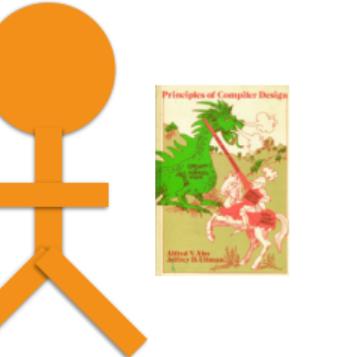
```
fn helper(count: i32) {  
    println!(..);  
}  
i32 is a Copy type
```



После исполнения функции объект исчезает.

Copy (auto-Clone)

```
fn main() {  
    let count = 22;  
    → helper(count);  
    helper(count);  
}
```

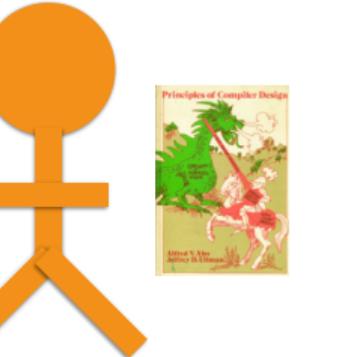


```
fn helper(count: i32) {  
    println!(..);  
}  
i32 is a Copy type
```

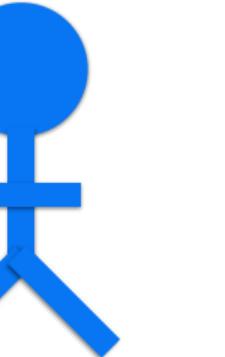


Copy (auto-Clone)

```
fn main() {  
    let count = 22;  
    helper(count);  
    helper(count);  
}
```



```
fn helper(count: i32) {  
    println!(..);  
}  
i32 is a Copy type
```



При втором вызове функции `helper` происходит то же самое, то есть **объект остаётся у функции-создателя**.

Non-copyable: Values **move** from place to place.

Example: *money*

Clone: Run custom code to make a copy.

Example: *strings*

Copy: Type is implicitly copied when referenced.

Example: *integers or floating-point numbers*

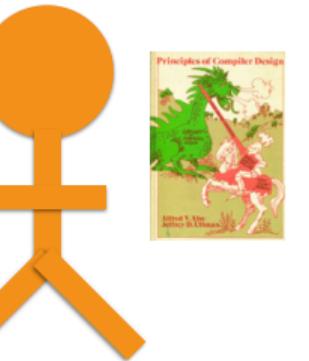
Подытожим.

У нас есть **некопируемые объекты**, при передаче прав владения которых, **передаётся сам объект**.

У нас есть **объекты с возможностью клонирования**, для которых мы вручную можем задать, чтобы передаче прав владения **создавался клон объекта**.

И у нас есть объекты, которые имеют **свойство Copy**. Данные объекты **прозрачно для разработчика** копируются при передаче в качестве аргумента.

Концепция заимствования.
Разделяемые заимствования.
У нас как всегда **есть книга дракона**.



Borrowing: Shared Borrows

Мы отдаём эту книгу, но с условием, что нам её вернут.



Borrowing: Shared Borrows

После того, как нашу книгу прочитали, нам её возвращают.
В отличие от передачи прав владения, мы даём книгу только на время.



Borrowing: Shared Borrows

```
fn main() {  
    let name = format!("...");  
    let reference = &name;  
    helper(reference);  
    helper(reference);  
}
```



Shared borrow

```
fn helper(name: &String) {  
    println!(...);  
}
```

Теперь рассмотрим пример кода.

У нас всё так же есть переменная `name` типа `String`.

```
fn main() {  
    let name = format!("...");  
    let reference = &name;  
    helper(reference);  
    helper(reference);  
}
```

```
fn helper(name: &String) {  
    println!(...);  
}
```

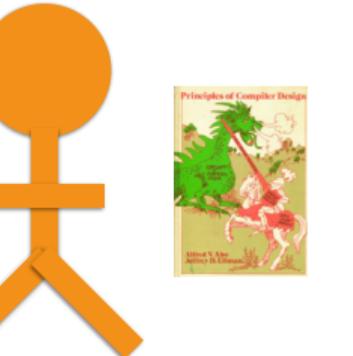
Change type to a
reference to a String



Shared borrow

Обратите внимание, как поменялся тип параметра — теперь передаётся не объект, а ссылка на объект, то бишь ссылка на переменную типа `String`.

```
fn main() {  
    let name = format!("...");  
    let reference = &name;  
    helper(reference);  
    helper(reference);  
}  
  
Lend the string,  
creating a reference
```



Shared borrow

```
fn helper(name: &String) {  
    println!(...);  
}
```

Change type to a
reference to a String

Чтобы передать нашу строку функции, нам нужно получить ссылку, делаем это.

```
fn main() {  
    let name = format!("...");  
    let reference = &name;  
    helper(reference);  
    helper(reference);  
}
```



Shared borrow

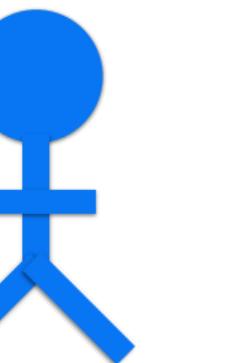
```
fn helper(name: &String) {  
    println!(...);  
}
```

Вызываем функцию `helper` и передаём ей ссылку на строку в качестве аргумента.
Стоить заметить, что ссылка на объект также остаётся у нас в пользовании.

```
fn main() {  
    let name = format!("...");  
    let reference = &name;  
    helper(reference);  
    helper(reference);  
}
```

```
fn helper(name: &String) {  
    println!(...);  
}
```

ФУНКЦИЯ ВЫПОЛНЯЕТСЯ.



Shared borrow

```
fn main() {  
    let name = format!("...");  
    let reference = &name; ➔  
    helper(reference);  
    helper(reference);  
}
```

```
fn helper(name: &String) {  
    println!(...);  
}
```

Функция заканчивает своё выполнение.



Shared borrow

```
fn main() {  
    let name = format!("...");  
    let reference = &name; ➔  
    helper(reference);  
    helper(reference);  
}
```

```
fn helper(name: &String) {  
    println!(...);  
}
```

Функция исчезает, а с ней исчезает и ссылка.



Shared borrow

```
fn main() {  
    let name = format!("...");  
    let reference = &name;  
    helper(reference);  
    helper(reference);  
}
```



Shared borrow

```
fn helper(name: &String) {  
    println!(...);  
}
```

Мы снова можем вызвать функцию `helper`, передав ссылку на объект.

Когда главная функция прекращает своё выполнение, то и ссылка, и объект уничтожаются.

```
fn main() {  
    let name = format!("...");  
    let reference = &name;  
    helper(reference);  
    helper(reference);  
    }  
  
    fn helper(name: &String) {  
        println!(...);  
    }
```



Shared borrow

Стоит сказать, что в Rust **все объекты**, будь то значение или ссылки — **неизменяемые по умолчанию**.

Shared == Immutable

```
fn helper(name: &String) {  
    println!("{}", name);  
}
```

```
fn helper(name: &String) {  
    name.push_str("foo");  
}
```

Поэтому, если мы просто читаем данные, переданные нам по ссылке, то всё будет хорошо.

Shared == Immutable

```
fn helper(name: &String) {  
    println!("{}", name); ← OK. Just reads.  
}
```

```
fn helper(name: &String) {  
    name.push_str("foo");  
}
```

А если попробуем изменить данные, то получим ошибку.

Shared == Immutable

```
fn helper(name: &String) {  
    println!("{}", name); ← OK. Just reads.  
}
```

```
fn helper(name: &String) {  
    name.push_str("foo"); ← Error. Writes.  
}
```

Shared == Immutable

```
fn helper(name: &String) {  
    println!("{}", name); ← OK. Just reads.  
}
```

```
fn helper(name: &String) {  
    name.push_str("foo"); ← Error. Writes.  
}
```

```
error: cannot borrow immutable borrowed content `*name`  
      as mutable  
      name.push_str("s");  
      ^~~~
```

На самом деле, данные **можно изменять**, но **только контролируемо**, об этом я не буду рассказывать подробно, считайте, что по умолчанию всё **неизменяемое**.

Shared == Immutable^{*}

```
fn helper(name: &String) {  
    println!("{}", name); ← OK. Just reads.  
}
```

```
fn helper(name: &String) {  
    name.push_str("foo"); ← Error. Writes.  
}
```

```
error: cannot borrow immutable borrowed content `*name`  
      as mutable  
      name.push_str("s");  
      ^~~~
```

* **Actually:** mutation only in **controlled circumstances**.

Побольше рассмотрим ссылки.

Play time



Waterloo, Cassius Coolidge, c. 1906

Всё тот же пример.

```
fn main() {  
    let name = format!("...");  
    helper(&name[1..]);  
    helper(&name);  
}
```

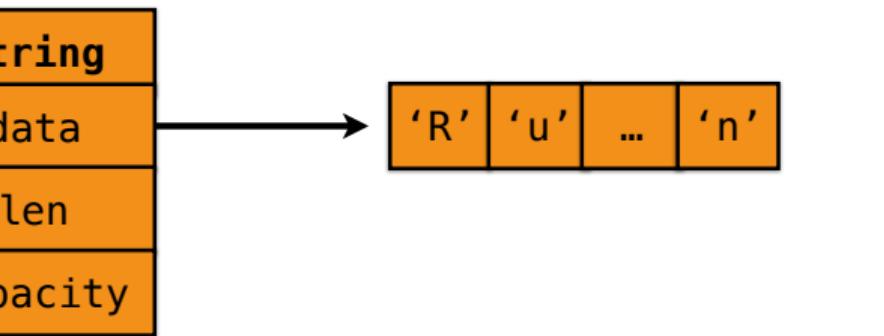
```
fn helper(name: &str) {  
    println!(...);  
}
```

Looks like other languages:
• Python: name[1:]
• Ruby: name[1..-1]
But no copying at runtime.

```
fn main() {  
    let name = format!("...");  
    helper(&name[1..]);  
    helper(&name);  
}
```

```
fn helper(name: &str) {  
    println!(...);  
}
```

Looks like other languages:
• Python: name[1:]
• Ruby: name[1..-1]
But no copying at runtime.

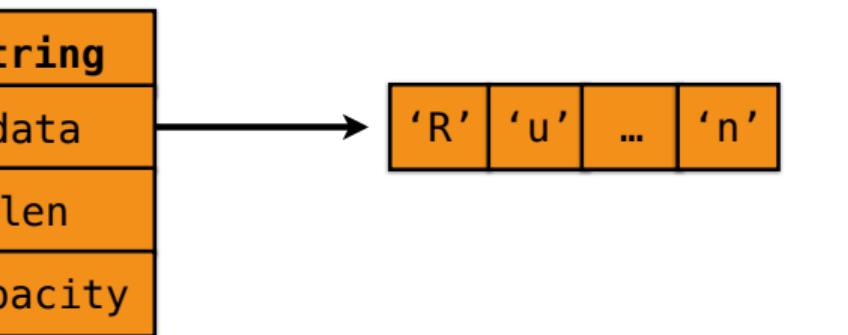


На самом деле тип `String` представляет из себя структуру, которая состоит из нескольких полей.

```
fn main() {  
    let name = format!("...");  
    helper(&name[1..]);  
    helper(&name);  
}
```

```
fn helper(name: &str) {  
    println!(...);  
}
```

Looks like other languages:
• Python: name[1:]
• Ruby: name[1..-1]
But no copying at runtime.



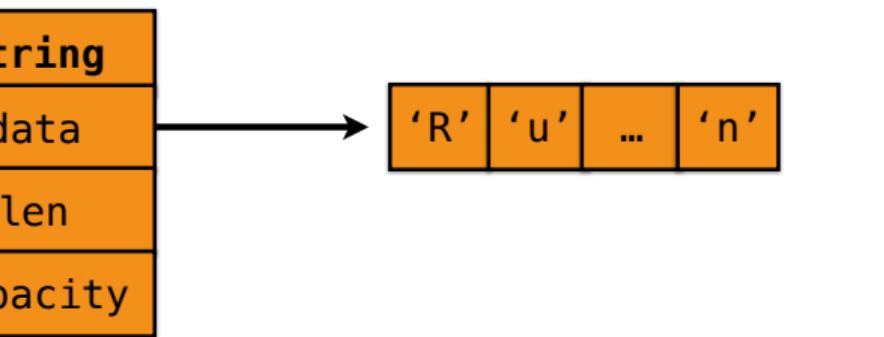
В функцию `helper` мы можем передать часть строки, как и в любых других ЯП.
Но стоит обратить внимание на то, что в отличие от других ЯП, копирование объекта не происходит, мы передаём данные по ссылке (zero-cost abstraction).

```
fn main() {  
    let name = format!("...");  
    helper(&name[1..]);  
    helper(&name);  
}
```

```
fn helper(name: &str) {  
    println!(...);  
}  
  
Change type from `&String`  
to a string slice, `&str`
```

Если вы внимательны, то могли заметить, что тип параметра у функции поменялся на `&str`.

Looks like other languages:
• Python: `name[1:]`
• Ruby: `name[1..-1]`
But no copying at runtime.

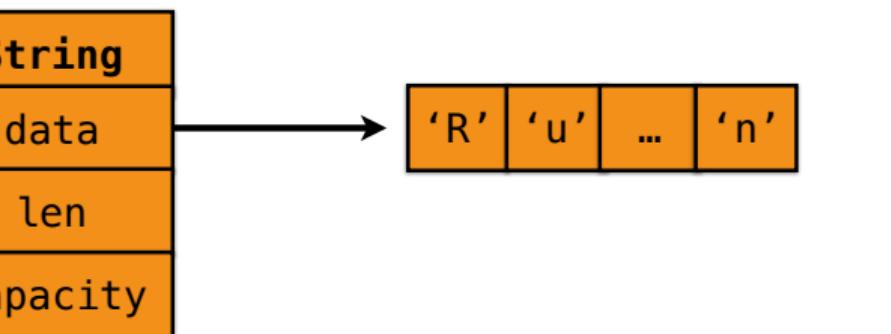


```
fn main() {  
    let name = format!("...");  
    helper(&name[1..]);  
    helper(&name);  
}
```

Lend some of
the string

```
fn helper(name: &str) {  
    println!(...);  
}
```

Change type from `&String`
to a **string slice**, `&str`



Looks like other languages:
• Python: `name[1:]`
• Ruby: `name[1..-1]`
But no copying at runtime.

Так произошло по причине того, что мы передаём ссылку не на саму строку типа `String`, а только на **часть строки**.

```
fn main() {  
    let name = format!("...");  
    helper(&name[1..]);  
    helper(&name);  
}
```

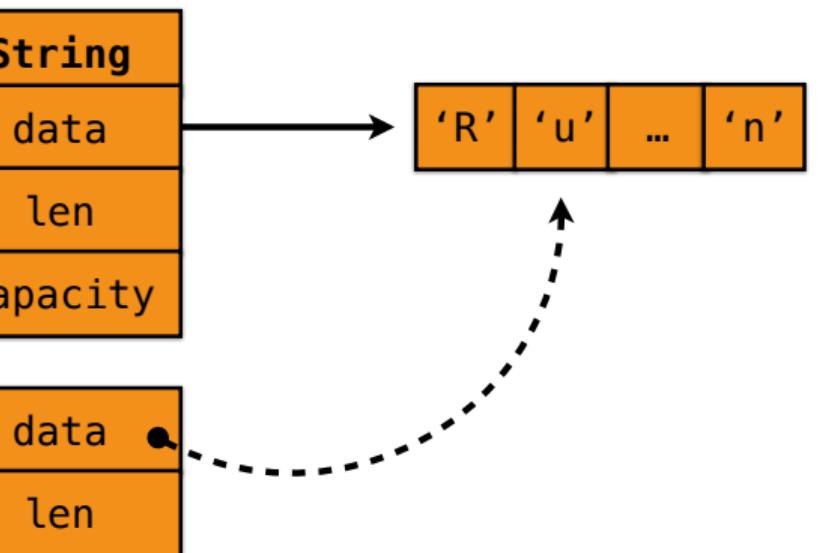
Lend some of
the string

```
fn helper(name: &str) {  
    println!(...);  
}
```

Change type from `&String`
to a **string slice**, `&str`

Looks like other languages:
• Python: name[1:]
• Ruby: name[1..-1]
But no copying at runtime.

Таким образом у нас появляется новый объект типа str.

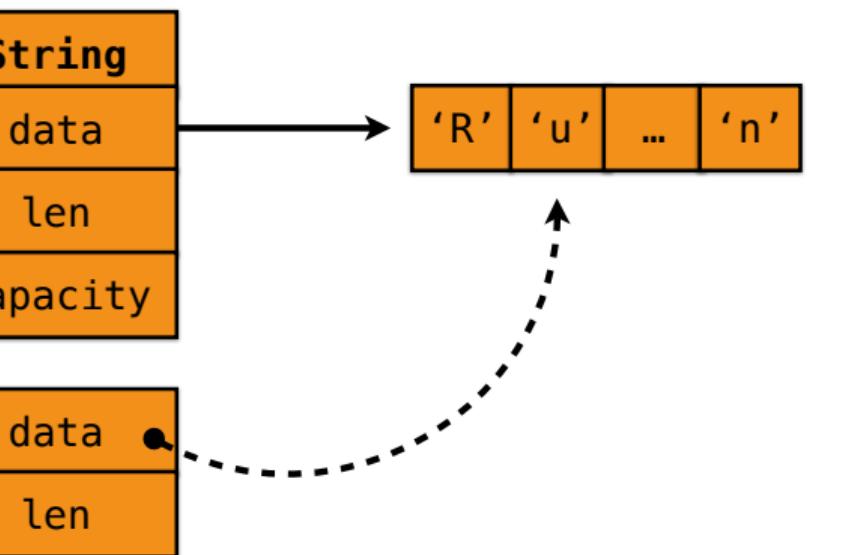


```
fn main() {  
    let name = format!("...");  
    helper(&name[1..]);  
    helper(&name);  
}
```

```
fn helper(name: &str) {  
    println!(...);  
}
```

Итак выполняем нашу функцию.

Looks like other languages:
• Python: name[1:]
• Ruby: name[1..-1]
But no copying at runtime.

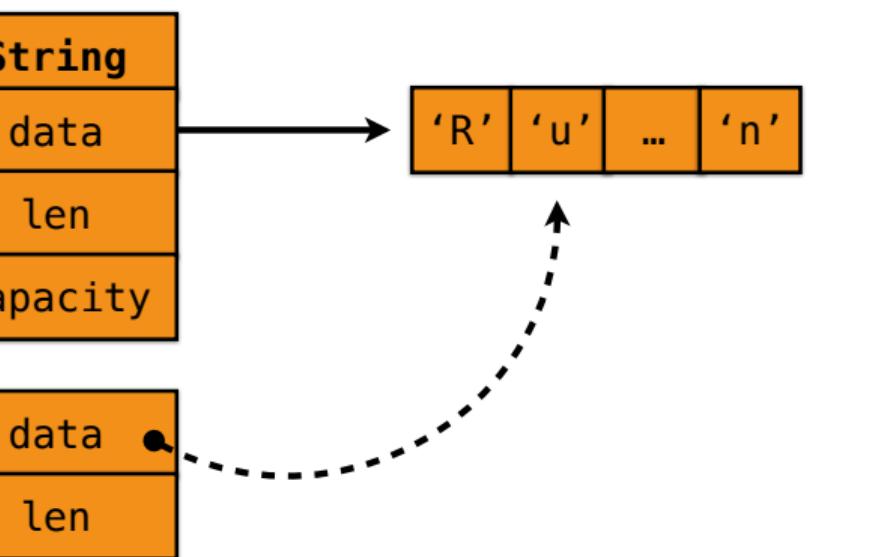


```
fn main() {  
    let name = format!("...");  
    helper(&name[1..]);  
    helper(&name);  
}
```

```
fn helper(name: &str) {  
    println!(...);  
}  
}
```

Итак выполняем нашу функцию.

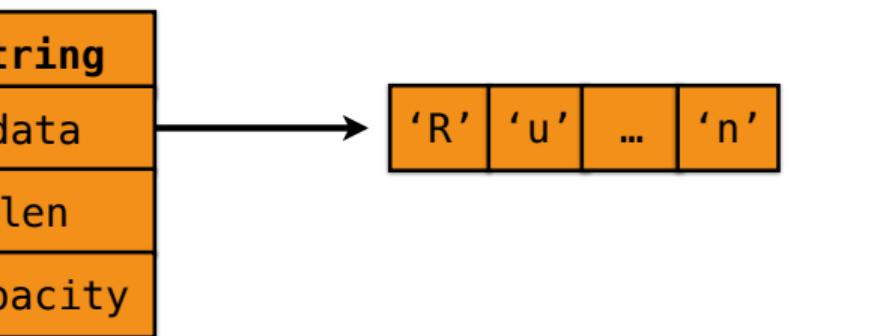
Looks like other languages:
• Python: name[1:]
• Ruby: name[1..-1]
But no copying at runtime.



```
fn main() {  
    let name = format!("...");  
    helper(&name[1..]);  
    helper(&name);  
}
```

```
fn helper(name: &str) {  
    println!(...);  
}  
}
```

Looks like other languages:
• Python: name[1:]
• Ruby: name[1..-1]
But no copying at runtime.

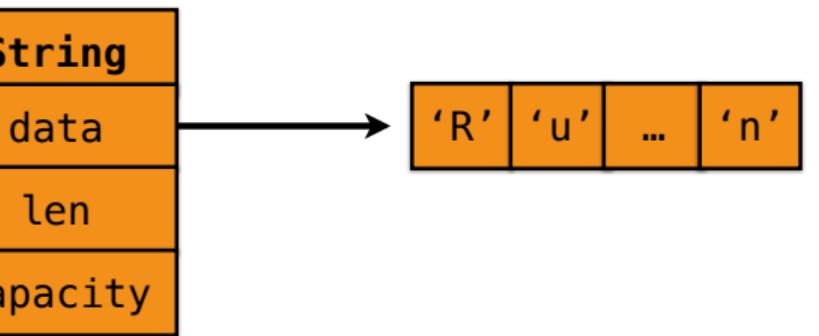


После окончания выполнения функции, ссылка на объект удаляется, так же, как и объект типа str.
Вызываем повторно нашу функцию, но уже со ссылкой на String. Стоит заметить, что тип функции нам менять не нужно.

```
fn main() {  
    let name = format!("...");  
    helper(&name[1..]);  
    helper(&name);  
}
```

```
fn helper(name: &str) {  
    println!(...);  
}
```

После выполнения все данные очищаются.



Looks like other languages:
• Python: name[1:]
• Ruby: name[1..-1]
But no copying at runtime.

Таким образом, мы можем выполнять высокоуровневый код с нулевой стоимостью.

High-level code, low-level efficiency

```
for word in line.split(' ') {  
    sum += word.len();  
}
```

No copying, no allocations.

High-level code, low-level efficiency

```
for word in line.split(' ') {  
    sum += word.len();  
}
```

Iterator over slices
borrowed from **line**.

Допустим, мы разбиваем строку на слова, между которыми стоят пробелы.

Обратите внимание, что код выглядит так, как будто написан на высокоуровневом языке программирования, т. е. также, как в Python или Ruby, например, вызывается метод, создаётся итератор, но при этом объект не копируется (мы передаём ссылку), а следовательно — нет дополнительной аллокации.

No copying, no allocations.

High-level code, low-level efficiency

```
for word in line.split(' ') {  
    sum += word.len();  
}
```

Iterator over slices
borrowed from **line**.

String
data
len
capacity

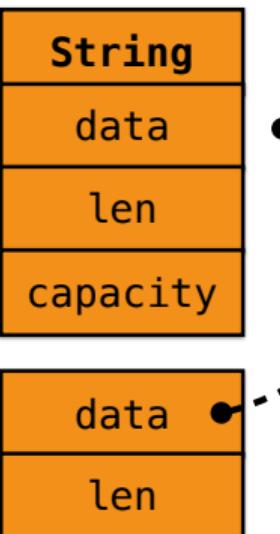
→ “Sing, Goddess, of Achilles’ rage, black and murderous...

No copying, no allocations.

High-level code, low-level efficiency

```
for word in line.split(' ') {  
    sum += word.len();  
}
```

Iterator over slices
borrowed from **line**.

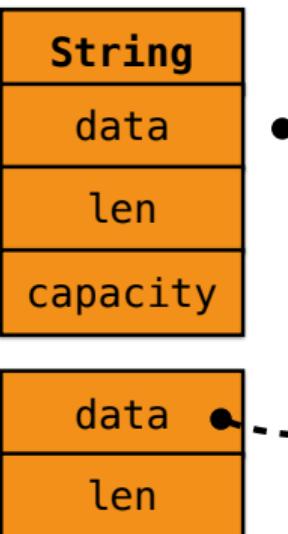


No copying, no allocations.

High-level code, low-level efficiency

```
for word in line.split(' ') {  
    sum += word.len();  
}
```

Iterator over slices
borrowed from **line**.



No copying, no allocations.

High-level code, low-level efficiency

```
for word in line.split(' ') {  
    sum += word.len();  
}
```

Iterator over slices
borrowed from **line**.

String
data
len
capacity

→ “Sing, Goddess, of Achilles’ rage, black and murderous...

data
len

No copying, no allocations.

Следующая концепция — изменяемые заимствования.



Borrowing: Mutable Borrows

Следующая концепция — изменяемые заимствования.



Borrowing: Mutable Borrows

Всё то же самое, что и в обычных заимствованиях, только функция **может изменять** данные.

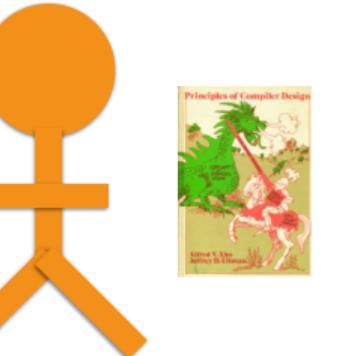


Borrowing: Mutable Borrows

Всё то же самое. Только объект мы делаем таким, чтобы можно было его **изменять**.

```
fn main() {  
    let mut name = ...;  
    update(&mut name);  
    println!("{}", name);  
}
```

```
fn update(name: &mut String) {  
    name.push_str("...");  
}
```

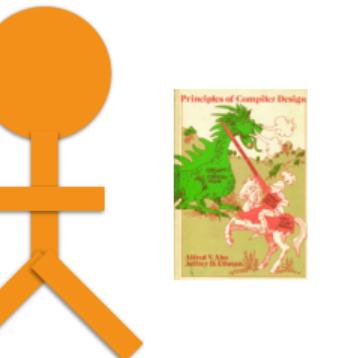


Mutable borrow

```
fn main() {  
    let mut name = ...;  
    update(&mut name);  
    println!("{}", name);  
}
```

```
fn update(name: &mut String) {  
    name.push_str("...");  
}  
  
Take a mutable  
reference to a String
```

Тип параметра также следует поменять на изменяемую ссылку.



Mutable borrow



```
fn main() {  
    let mut name = ...;  
    update(&mut name);  
    println!("{}", name);  
}
```

Lend the string
mutably



Mutable borrow

```
fn update(name: &mut String) {  
    name.push_str("...");  
}
```

Take a **mutable**
reference to a String



Для передачи изменяемой ссылки, мы также должны явно указать, что данная ссылка является изменяемой.

```
fn main() {  
    let mut name = ...;  
    update(&mut name);  
    println!("{}", name);  
}
```

Lend the string
mutably



```
fn update(name: &mut String) {  
    name.push_str("...");  
}
```

Take a **mutable**
reference to a String



Так же, как и раньше мы передаём ссылку, но в Rust при передачи изменяемой ссылки у хозяина пропадает возможность читать и писать **данные** по этой ссылке.

Начинаем выполнения функции.

```
fn main() {           → fn update(name: &mut String) {  
    let mut name = ...;  
    update(&mut name);  
    println!("{}", name);  
}
```

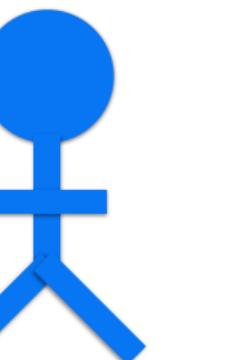


Mutable borrow

Изменяем переданный нам объект.

```
fn main() {  
    let mut name = ...; →  
    update(&mut name);  
    println!("{}", name);  
}
```

Mutate string
in place



Mutable borrow

Заканчиваем выполнение функции.

```
fn main() {  
    let mut name = ...;  
    update(&mut name);  
    println!("{}", name);  
}
```

```
fn update(name: &mut String) {  
    name.push_str("...");  
}
```



Mutable borrow

Эксклюзивные права на запись пропадают, так как изменяемая ссылка уничтожается.

```
fn main() {  
    let mut name = ...;  
    update(&mut name);  
    println!("{}", name);  
}
```

```
fn update(name: &mut String) {  
    name.push_str("...");  
}
```

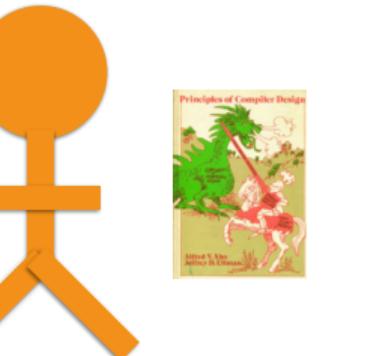


Mutable borrow

```
fn main() {  
    let mut name = ...;  
    update(&mut name);  
    println!("{}", name);  
}
```

```
fn update(name: &mut String) {  
    name.push_str("...");  
}
```

Дальше можем оперировать строкой как обычно.



Mutable borrow

```
fn main() {  
    let mut name = ...;  
    update(&mut name);  
    println!("{}", name);  
}
```



```
fn update(name: &mut String) {  
    name.push_str("...");  
}
```

Prints the
updated string.

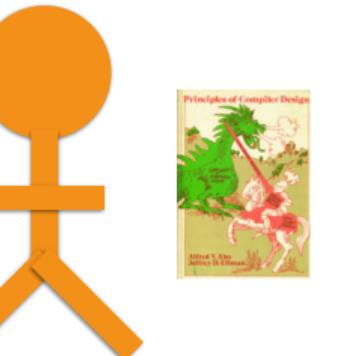


Mutable borrow

Например распечатать её (данные передаются по ссылке).

Главная функция заканчивает своё выполнение.

```
fn main() {  
    let mut name = ...;  
    update(&mut name);  
    println!("{}", name);  
}  
  
fn update(name: &mut String) {  
    name.push_str("...");  
}
```



Mutable borrow

Данные как всегда – удаляются.

```
fn main() {  
    let mut name = ...;  
    update(&mut name);  
    println!("{}", name);  
}  
  
fn update(name: &mut String) {  
    name.push_str("...");  
}
```



Mutable borrow

`name: String`

Ownership:

control all access, will free when done

`name: &String`

Shared reference:

many readers, no writers

`name: &mut String`

Mutable reference:

no readers, one writer



`name: String`

Ownership:

control all access, will free when done

`name: &String`

Shared reference:

many readers, no writers

`name: &mut String`

Mutable reference:

no readers, one writer

Владение.

Мы имеем полный доступ к данным, которые удалятся, когда будут не нужны.

`name: String`

`name: &String`

`name: &mut String`

Ownership:

control all access, will free when done

Shared reference:

many readers, no writers

Mutable reference:

no readers, one writer

Заемствоvание.

По ссылке мы можем **многократно читать, но не писать**.

`name: String`

Ownership:

control all access, will free when done

`name: &String`

Shared reference:

many readers, no writers

Mutable reference:

no readers, one writer



`name: &mut String`

Изменяемое заимствование.

По изменяемой ссылке мы можем **только писать**.

(Un)safe

How do we get safety?

Как же в итоге все эти концепции влияют на безопасность?

How do we get safety?



```
fn main() {
    let r;
    {
        let name = format!("...");
        r = &name;
    }
    println!("{}", r);
}
```

Итак, у нас есть программа.

```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```

У нас создаётся **неинициализированная** переменная *r*.

```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```

r

Данная переменная, соответственно, создаётся на стеке.

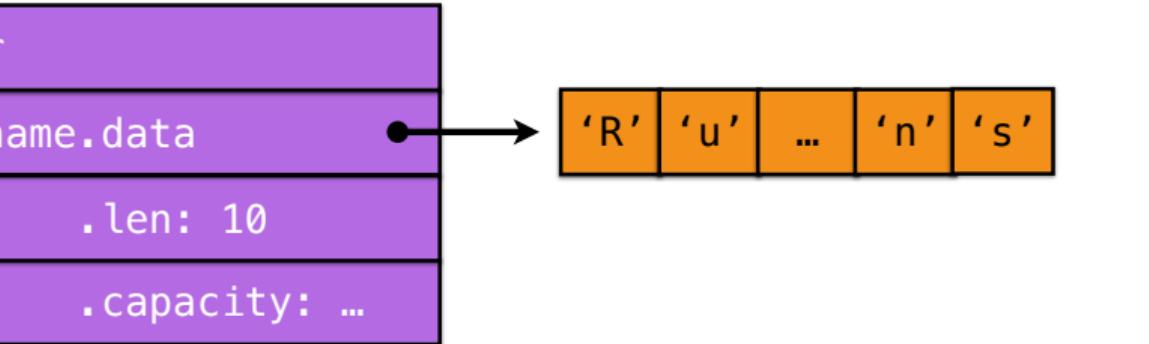
```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```

r

Создаём ещё одну переменную `name` типа `String`. Переменная создаётся в **своей области видимости**, позже я про это расскажу.

Стоит также заметить, что в Rust **есть type inference** (вывод типов), который ранее был характерен в основном для функциональных ЯП.

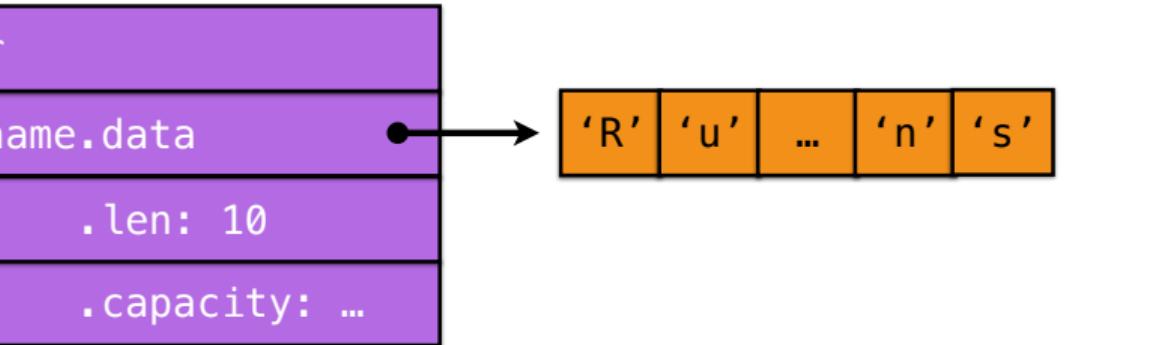
```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```



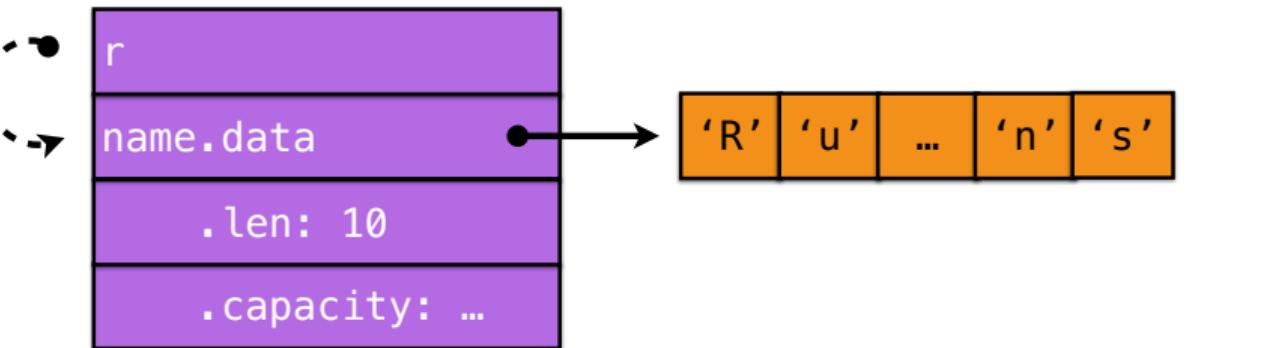
Как мы видим, новая переменная также помещается на стек.

```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```

Производим присвоение переменной `r` ссылки на объект `name`.



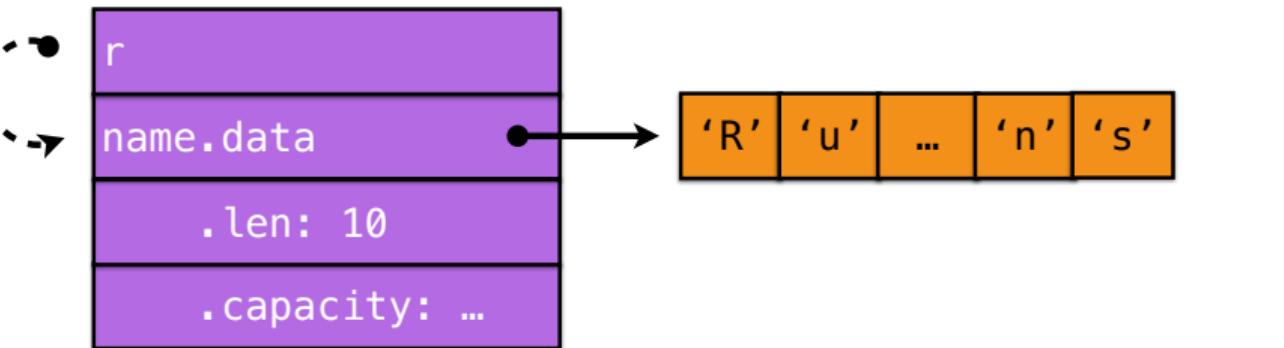
```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```



На стеке это будет выглядеть примерно так.

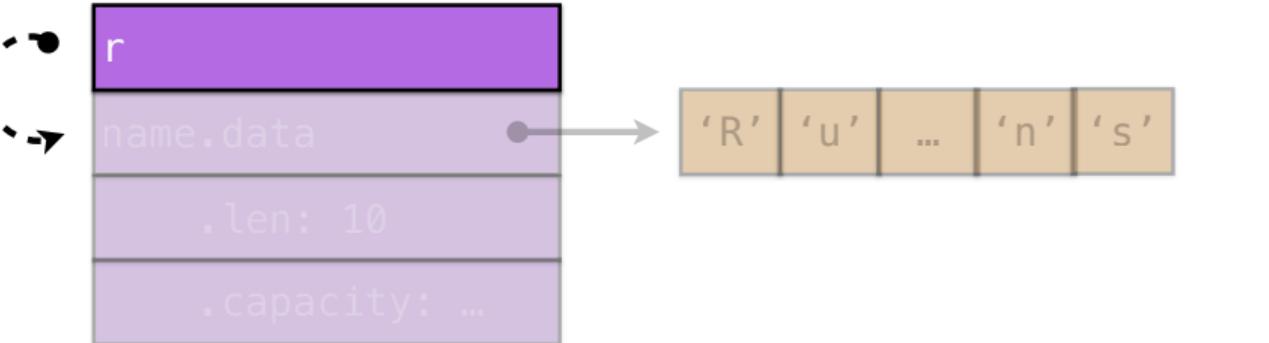
```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```

Продолжаем выполнение...



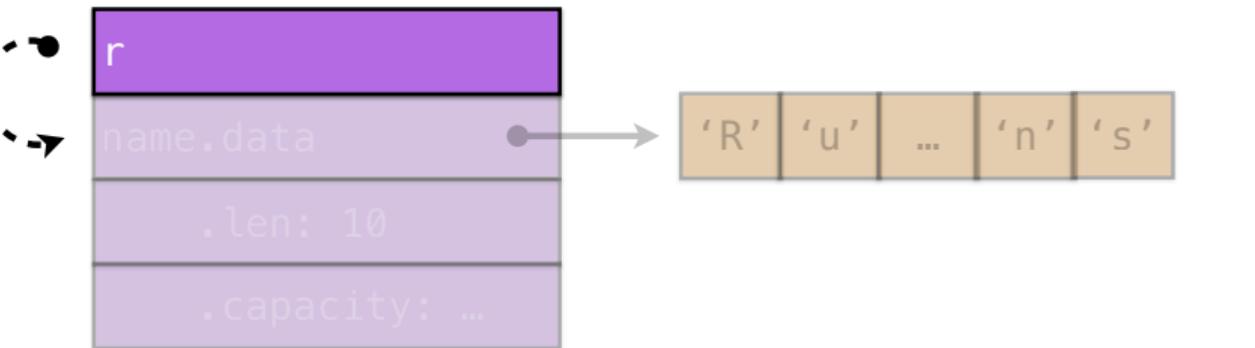
```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```

...и выходим из области видимости переменной name.



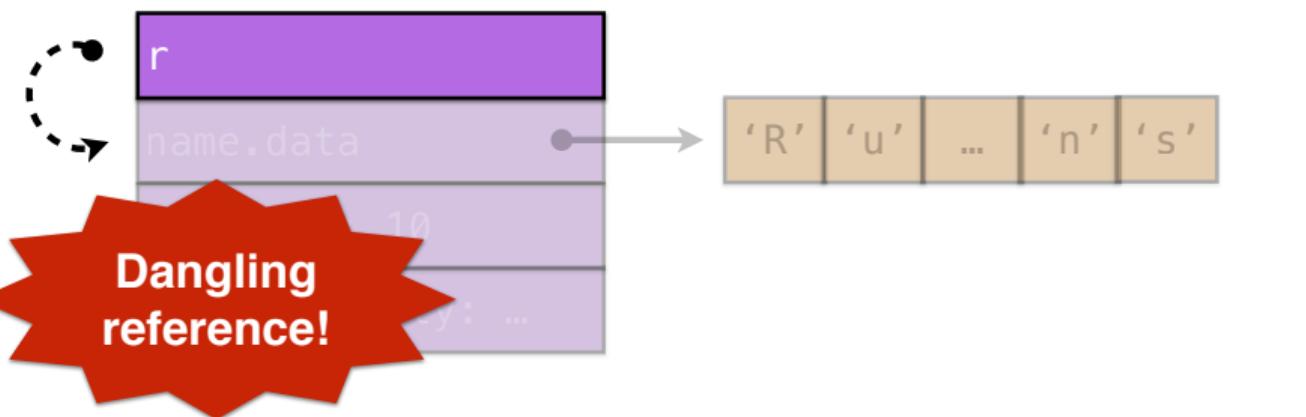
```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```

Пытаемся распечатать содержимое переменной `r`.



```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```

Упс! Висячая ссылка!



Давайте посмотрим, как Rust с этим разбирается.

```
fn main() {
    let r;
    {
        let name = format!("..");
        r = &name;
    }
    println!("{}", r);
}
```

В Rust есть понятие **время жизни ссылки** — это промежутки в коде, где ссылка используется.

```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```

Lifetime: span of code where reference is used.

Сначала ссылка присваивается переменной `r`.

```
fn main() {
    let r;
    {
        let name = format!("...");
        r = &name;
    }
    println!("{}", r);
}
```

Lifetime: span of code where reference is used.

Переменная `r`, объявленная выше, в свою очередь используется при выводе в терминал.

```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```

Lifetime: span of code where reference is used.

Вся эта область и будет являться временем жизни ссылки.

```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```

Lifetime: span of code where reference is used.

```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```

Есть ещё одно понятие – **область видимости**, область, где созданные данные могут быть использованы.

Lifetime: span of code where reference is used.

compared against

Scope of data being borrowed (here, `name`)

Вот область видимости переменной name.

```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```

Lifetime: span of code where reference is used.

compared against

Scope of data being borrowed (here, `name`)

```
fn main() {  
    let r;  
    {  
        's  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```

Производя сравнения времени жизни переменной с областью видимости ссылки, Rust приходит к выводу, что здесь ошибка (на этапе компиляции).

Lifetime: span of code where reference is used.

compared against

Scope of data being borrowed (here, `name`)

```
error: `name` does not live long enough  
r = &name;  
     ^~~~
```

Посмотрим, как Rust справляется с ошибками, возникающими при работе с потоками.

```
use std::thread;

fn helper(name: &String) {
    thread::spawn(move || {
        use(name);
    });
}
```

```
use std::thread;  
  
fn helper(name: &String) {  
    thread::spawn(move || {  
        use(name);  
    });  
}
```

name` can only be
used within this fn

У нас есть параметр функции `name`, оперировать которым мы можем только в области видимости функции, т. е. время жизни ссылки будет – область видимости данной функции.

```
use std::thread;  
  
fn helper(name: &String) {  
    thread::spawn(move || {  
        use(name);  
    });  
}
```

Might escape
the function!

name` can only be
used within this fn

Если мы создадим поток и будем использовать переменную name во вновь созданном потоке, то мы должны понимать, что поток будет исполняться вне данной функции.

```
use std::thread;

fn helper(name: &String) {
    thread::spawn(move || {
        use(name);
    });
}
```

Might escape
the function!

name` can only be
used within this fn

```
error: the type ` [...]` does not fulfill the required lifetime
      thread::spawn(move || {
      ^~~~~~
note: type must outlive the static lifetime
```

Что приведёт к ошибке — использование ссылки, которая возможно уже не существует. Rust нам об этом скажет уже на этапе компиляции.

```
use std::thread;

fn helper(name: &String) {
    thread::spawn(move || {
        use(name);
    });
}
```

name` can only be
used within this fn

```
error: the type ` [...]` does not fulfill the required lifetime
      thread::spawn(move || {
      ^~~~~~
note: type must outlive the static lifetime
```

Мы можем задать **статическое время жизни ссылки** — это значит, что ссылка будет жить, пока программа не закончит своё выполнение.

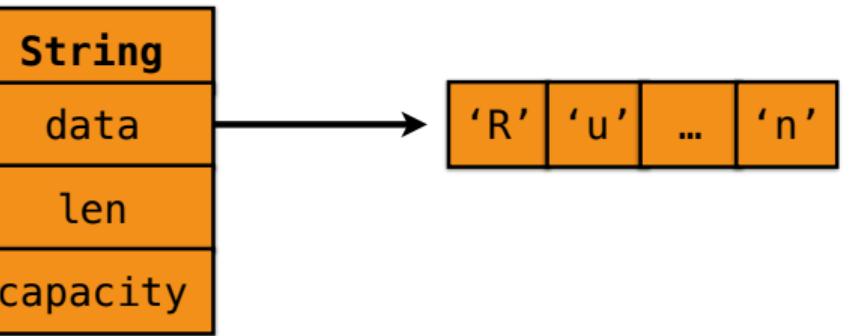
Dangers of mutation

```
let mut buffer: String = format!("Rustacean");
let slice = &buffer[1..];
buffer.push_str("s");
println!("{}:?", slice);
```

У нас создаётся **изменяемая** переменная типа `String`.

Dangers of mutation

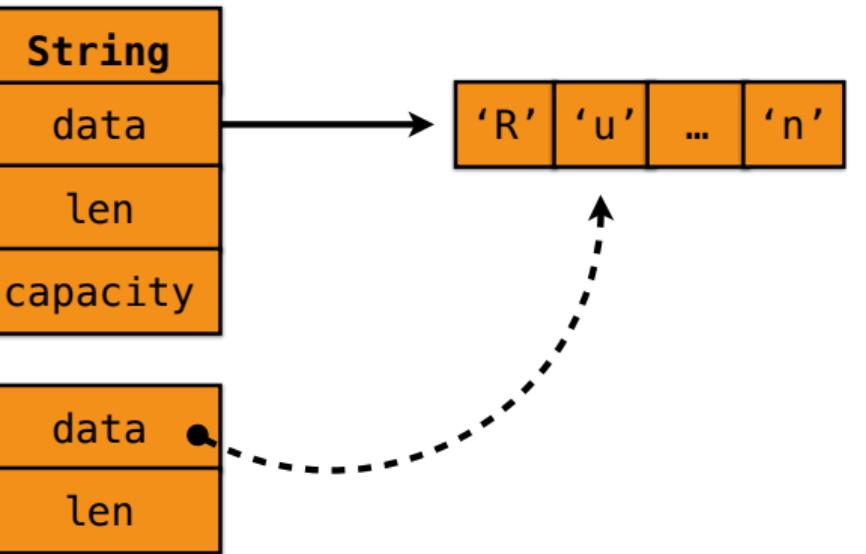
```
let mut buffer: String = format!("Rustacean");
let slice = &buffer[1..];
buffer.push_str("s");
println!("{}:?", slice);
```



Затем мы создаём ссылку на первый элемент строки.

Dangers of mutation

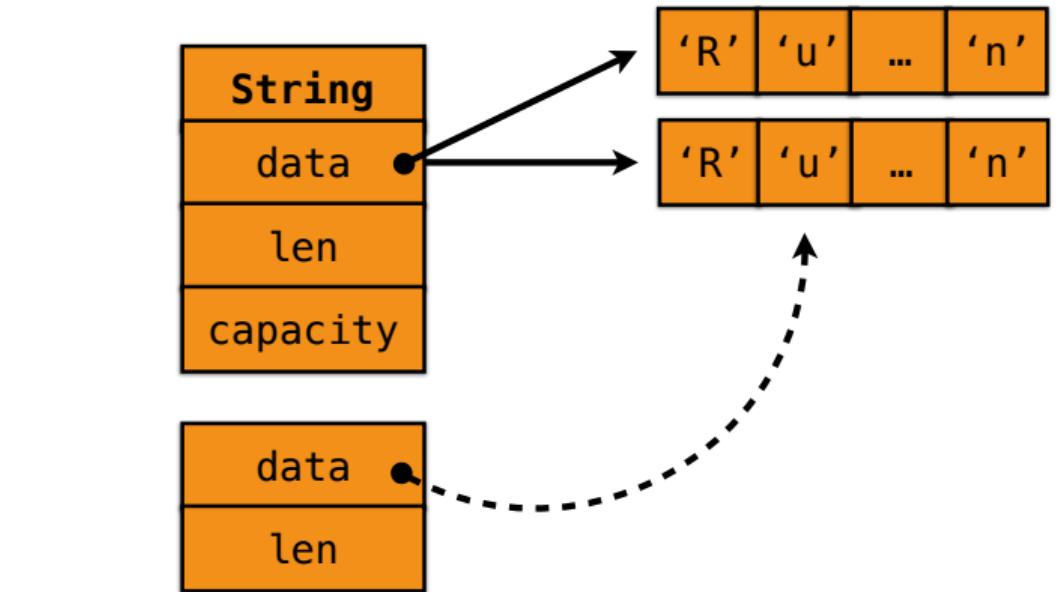
```
let mut buffer: String = format!("Rustacean");
let slice = &buffer[1..];
buffer.push_str("s");
println!("{}:?", slice);
```



Потом нам вздумалось изменить строку.

Dangers of mutation

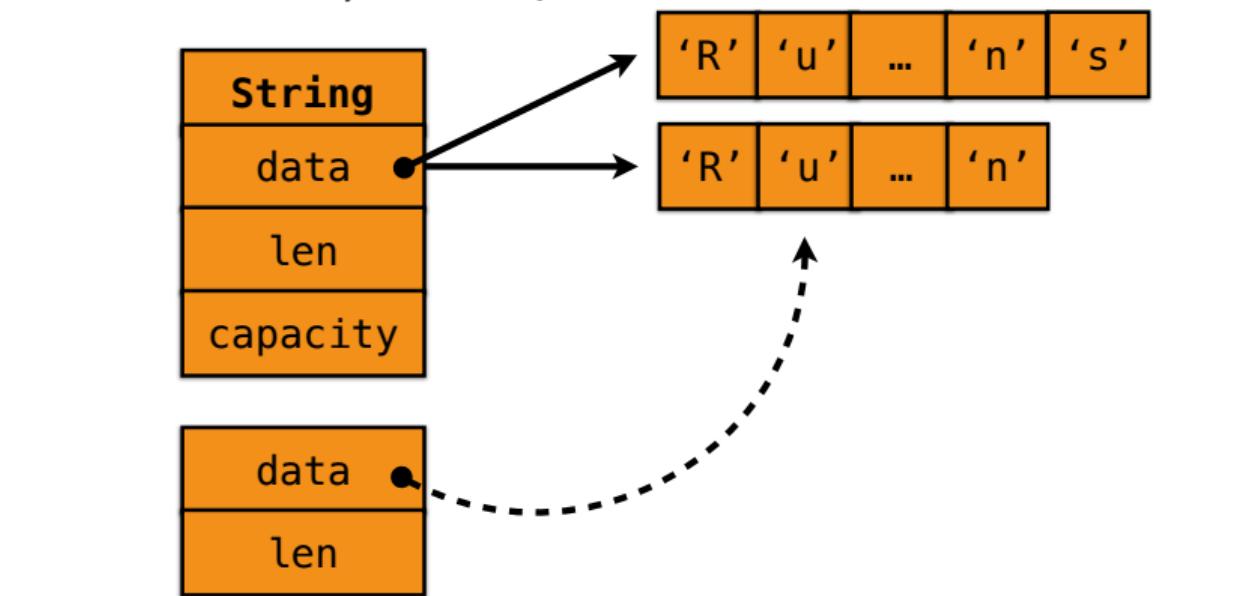
```
let mut buffer: String = format!("Rustacean");
let slice = &buffer[1..];
buffer.push_str("s");
println!("{}:?", slice);
```



Добавили в конец s.

Dangers of mutation

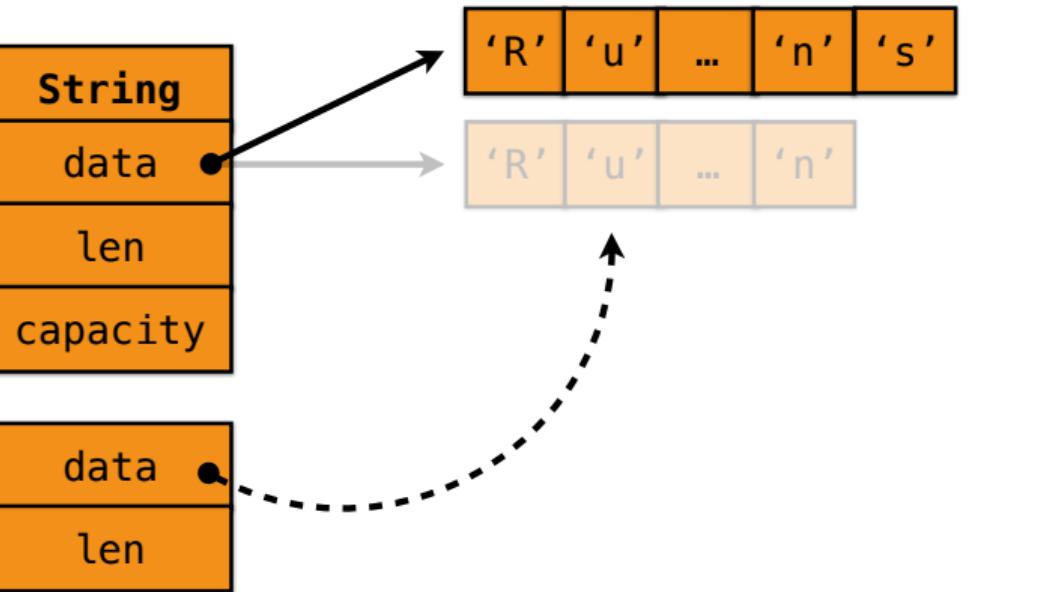
```
let mut buffer: String = format!("Rustacean");
let slice = &buffer[1..];
buffer.push_str("s");
println!("{}:?", slice);
```



Первоначальные данные с кучи у нас пропадут, т. к. мы, возможно, сделали реаллокацию.

Dangers of mutation

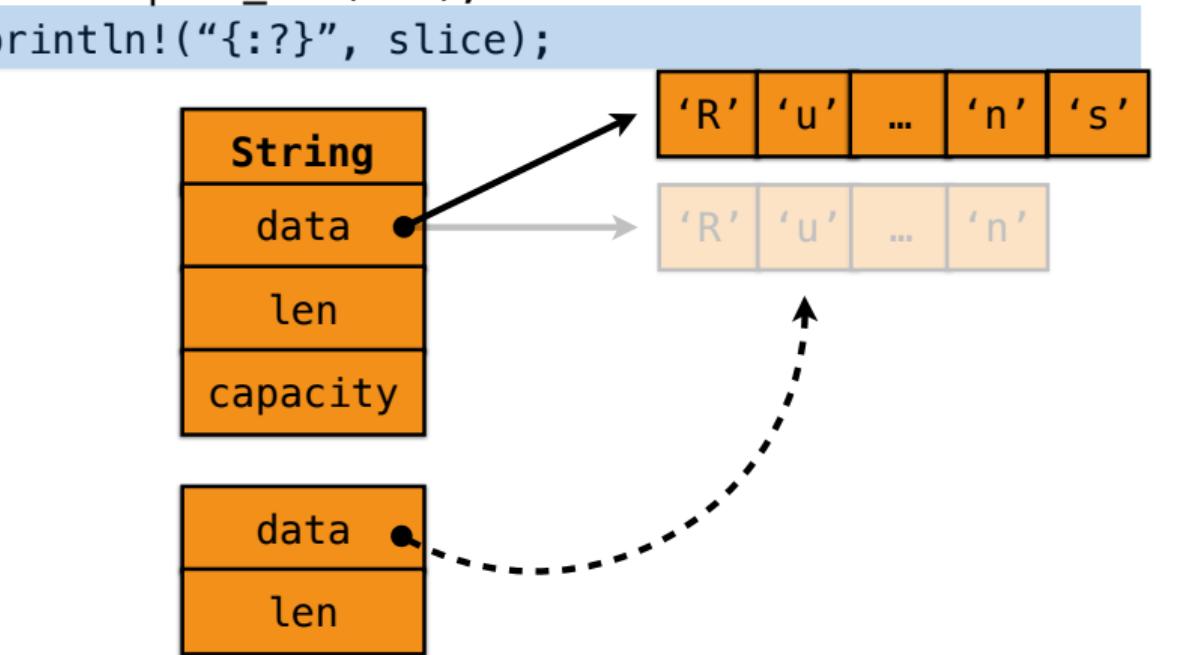
```
let mut buffer: String = format!("Rustacean");
let slice = &buffer[1..];
buffer.push_str("s");
println!("{}:?", slice);
```



Что будет, если мы попытаемся вывести содержимое по ссылке slice?

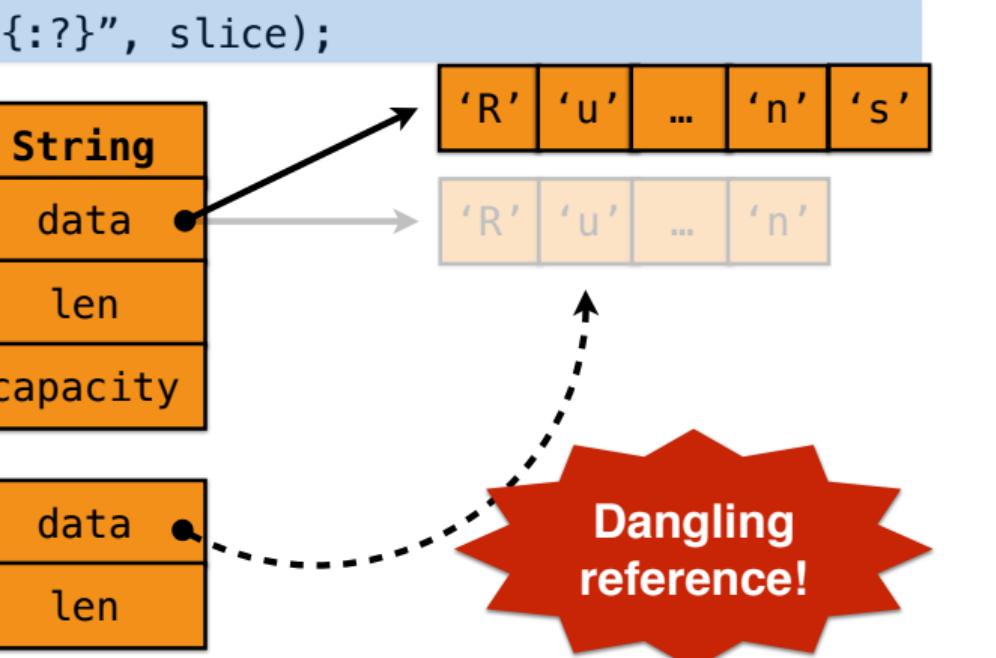
Dangers of mutation

```
let mut buffer: String = format!("Rustacean");
let slice = &buffer[1..];
buffer.push_str("s");
println!("{}:?", slice);
```



Dangers of mutation

```
let mut buffer: String = format!("Rustacean");
let slice = &buffer[1..];
buffer.push_str("s");
println!("{}:?", slice);
```



Rust solution

Compile-time read-write-lock:

Creating a shared reference to X “**read locks**” X.

- Other readers OK.
- No writers.
- Lock lasts until reference goes out of scope.

Creating a mutable reference to X “**writes locks**” X.

- No other readers or writers.
- Lock lasts until reference goes out of scope.

Never have a reader/writer at same time.

Как же Rust поступает в данном случае?

Во время компиляции проверяется:

- Если есть одна **ссылка на чтение**, то:
 - создание ещё ссылок на чтение ошибок не вызовет;
 - создание **ссылок на изменение** запрещено;
 - данные правила будут работать до тех пор, пока **жива ссылка**.
- Если есть одна **ссылка на изменение**, то:
 - у неё **эксклюзивные права**, никаких новых ссылок быть не может;
 - данные правила будут работать до тех пор, пока **жива ссылка**.

Таким образом, у нас ни при каких обстоятельствах **не может быть ссылок на чтение и изменение одновременно**.

Возвращаемся к нашему примеру с висячей ссылкой, полученной в результате изменения объекта.

Dangers of mutation

```
fn main() {  
    let mut buffer: String = format!("Rustacean");  
    let slice = &buffer[1..];  
    buffer.push_str("s");  
    println!("{}:?", slice);  
}
```

Когда мы создаём ссылку на чтение, мы автоматически блокируем создание ссылок на изменение.

Dangers of mutation

```
fn main() {  
    let mut buffer: String = format!("Rustacean");  
    let slice = &buffer[1..];  
    buffer.push_str("s");  
    println!("{}:?", slice);  
}
```



Borrow “locks”
`buffer` until `slice`
goes out of scope

Dangers of mutation

```
fn main() {  
    let mut buffer: String = format!("Rustacean");  
    let slice = &buffer[1..]; ←  
    buffer.push_str("s");  
    println!("{}:?", slice);  
}
```

Borrow “locks”
`buffer` until `slice`
goes out of scope

Dangers of mutation

```
fn main() {  
    let mut buffer: String = format!("Rustacean");  
    let slice = &buffer[1..];  
    buffer.push_str("s");  
    println!("{}:?", slice);  
}
```

Borrow “locks”
`buffer` until `slice`
goes out of scope

```
error: cannot borrow `buffer` as mutable  
      because it is also borrowed as immutable  
      buffer.push_str("s");  
      ^~~~~~
```

При вызове метода `push_str` создаётся изменяемая ссылка, а это запрещено, результат — ошибка на этапе компиляции.

Рассмотрим немного другой пример.

```
fn main() {
    let mut buffer: String = format!("Rustacean");
    for i in 0 .. buffer.len() {
        let slice = &buffer[i..];
        buffer.push_str("s");
        println!("{}:{}?", slice);
    }
    buffer.push_str("s");
}
```

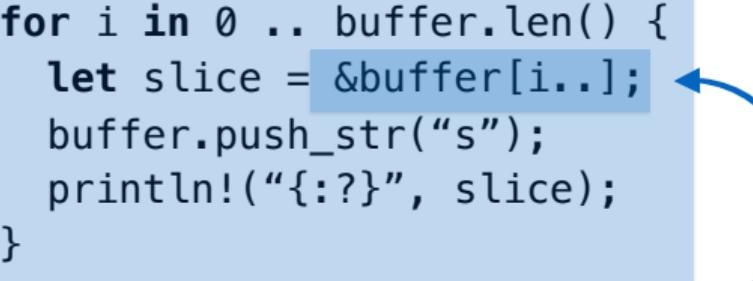
Здесь также создаётся ссылка на чтение.

```
fn main() {  
    let mut buffer: String = format!("Rustacean");  
    for i in 0 .. buffer.len() {  
        let slice = &buffer[i..]; ←  
        buffer.push_str("s");  
        println!("{}:?", slice);  
    }  
    buffer.push_str("s");  
}
```

Borrow “locks”
`buffer` until `slice`
goes out of scope

Но время жизни ссылки уже другое.

```
fn main() {  
    let mut buffer: String = format!("Rustacean");  
    for i in 0 .. buffer.len() {  
        let slice = &buffer[i..];  
        buffer.push_str("s");  
        println!("{}:{}?", slice);  
    }  
    buffer.push_str("s");  
}
```



Borrow “locks”
`buffer` until `slice`
goes out of scope

```
fn main() {  
    let mut buffer: String = format!("Rustacean");  
    for i in 0 .. buffer.len() {  
        let slice = &buffer[i..];  
        buffer.push_str("s")  
        println!("{}:?", slice);  
    }  
    buffer.push_str("s");  
}
```

Borrow “locks”
`buffer` until `slice`
goes out of scope

Но за границами жизни ссылки на чтение мы вправе создавать ссылки какие захотим.

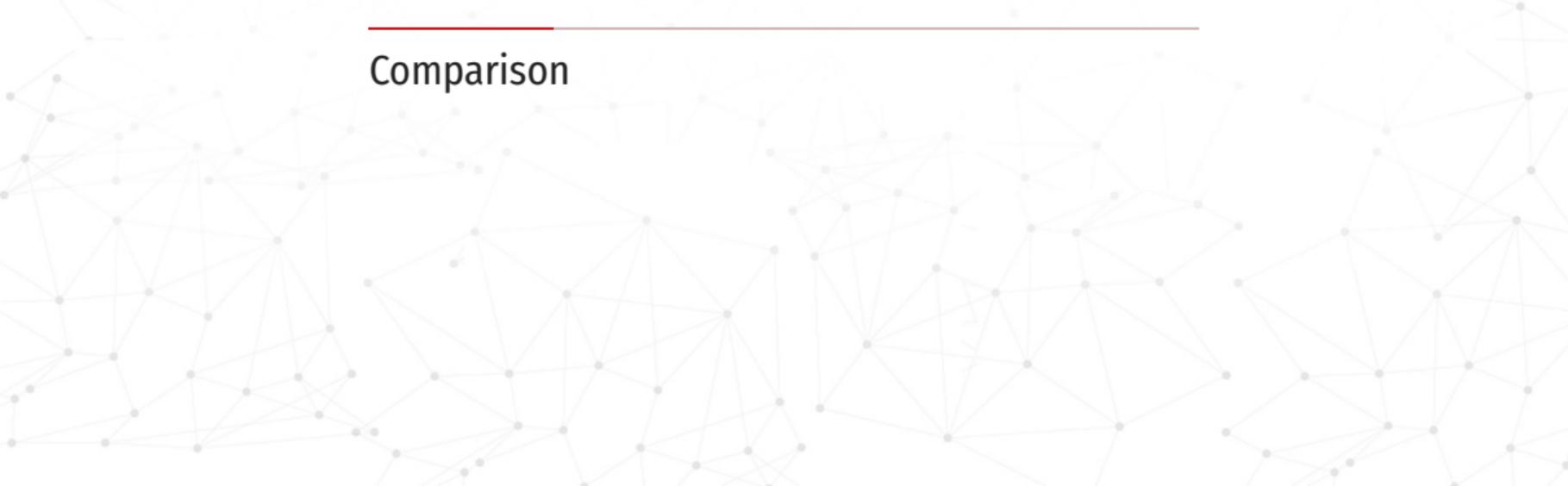
```
fn main() {  
    let mut buffer: String = format!("Rustacean");  
    for i in 0 .. buffer.len() {  
        let slice = &buffer[i..];  
        buffer.push_str("s")  
        println!("{}:?", slice);  
    }  
    buffer.push_str("s");  
}
```

Borrow “locks”
`buffer` until `slice`
goes out of scope

OK: `buffer` is not borrowed here



(Un)safe



Comparison

NULL dereferences

C

```
1 uint8_t* pointer = (uint8_t*) malloc(SIZE); // Might return NULL
2 for(int i = 0; i < SIZE; ++i) {
3     pointer[i] = i; // Might cause a Segmentation Fault
4 }
```

Rust

```
1 let mut vec = vec![0 as u8; SIZE];
2 for i in 0..SIZE { // As C code
3     vec[i] = i;
4 }
```

Как мы можем наблюдать, в C коде мы можем получить **SEGFAULT на каждом шагу**, т. к. `malloc` и другие функции могут вернуть `null`.

Если же писать на Rust в стиле, похожем на C, то в данной версии мы не можем получить `null` априори, т. к. **null в Rust нет**.

В случае, если аллокация памяти будет неудачной, то мы либо получим **панику**, либо **сможем отловить ошибку**.

NULL dereferences (continue)

В функциональном стиле всё ещё проще и без ошибок.
Если же работать с ссылками, то мы также **никогда не получим NULL**, потому что в Rust нет NULL вообще, в отличие от других языков.

Functional Rust

```
1 let vec: Vec<u8> = (0..10).collect();
```

Rust References

```
1 let my_var: u32 = 42;
2 let my_ref: &u32 = &my_var; // References ALWAYS point
                           // to valid data
4 let my_var2 = *my_ref; // An example for a Dereference
```

Use after free (C)

C

```
1 uint8_t* pointer = (uint8_t*) malloc(SIZE);
2 // ...
3 if (err) {
4     abort = 1;
5     free(pointer);
6 }
7 // ...
8 if (abort) {
9     logError("operation aborted", pointer);
10}
```

Сначала мы **выделяем память** на куче, затем **освобождаем память**, потом пытается **читать память**, которая уже освобождена.
Поведение будет **неоднозначным**, возможно, мы даже не заметим ошибки до поры до времени.

Use after free (Rust)

Rust

```
1 let vec: Vec<u32> = Vec::new();
2 {
3     {
4         let vec_1 = vec; // vec's ownership has been moved
5     } // the Vec will be freed (dropped) here
6 }
```

В Rust, как и в C++ работает RAI^I (получение ресурса есть инициализация), поэтому ошибок UAF в Rust быть не может. Кроме этого, как говорилось выше, работают и другие правила, например, концепция владения.

Dangling pointers (C)

Создаём в функции переменную на стеке, затем возвращаем ссылку на эту переменную — у нас висячая ссылка. Ссылка указывает на неизвестные данные на стеке.

C

```
1 uint8_t* get_dangling_pointer(void) {  
2     uint8_t array[4] = {0};  
3     return &array[0];  
4 }
```

Dangling pointers (Rust)

Rust

```
1 fn get_dangling_pointer() -> &u8 {  
2     let array = [0; 4];  
3     &array[0]  
4 }
```

Compile time error

```
|  
1 | fn get_dangling_pointer() -> &u8 {  
|           ^ help: consider giving it a 'static lifetime:  
|           → `&'static`  
|  
= help: this function's return type contains a borrowed value, but there is no  
→ value for it to be borrowed from
```

В Rust при попытке вернуть ссылку на переменные, находящиеся на стеке, возникнет ошибка при компиляции.

Компилятор советует указать статическое время жизни ссылки для того, чтобы она была доступна при выходе из функции.

Out of bounds (C)

При попытке получить значение вне массива, мы получим случайное значение на стеке.

C

```
1 void print_out_of_bounds(void) {
2     uint8_t array[4] = {0};
3     printf("%u\r\n", array[4]);
4 }
5 // prints memory that's outside `array` (on the stack)
```

Out of bounds (Rust)

Rust

```
1 fn print_panics() {  
2     let array = [0; 4];  
3     println!("{}", array[4]);  
4 }
```

Compile time error

```
error: index out of bounds: the len is 4 but the index is 4  
--> test.rs:8:20  
|  
3 |     println!("{}", array[4]);  
|          ^^^^^^  
|  
= note: #[deny(const_err)] on by default
```

В случае с Rust, если существует возможность на стадии компиляции **определить границы массива**, то ошибка **будет отловлена уже на стадии компиляции**.

Если же компилятор **не может определить**, по какому индексу происходит чтение, то в случае чтение за границами массива возникнет ошибки во время исполнения — **паника**.



(Un)safe

Concurrency

Originally: Rust had message passing built into the language

Now: library-based, multi-paradigm

- rayon (parallel processing, thread pool)
- tokio, futures (I/O, async)
- coroutine, coio (coroutine)
- crossbeam, mio (low-level concurrency)

Libraries leverage **ownership and traits** to avoid data races



На уровне стандартной библиотеки Rust имеет реализацию параллелизма в виде передачи сообщений, также имеются вспомогательные типы: мьютексы, атомарные типы и прочее, сейчас на стадии обкатки `async-await`.

Но существует множество решений в виде библиотек, которые реализовывают параллелизм в виде множества различных парадигм. Все библиотеки лишены багов типа гонка данных, благодаря использованию внутри концепций владения и типажей.

Concurrent quicksort

Rust

```
1 fn qsort(vec: &mut [i32]) {  
2     if vec.len() <= 1 { return; }  
3     let pivot = vec[random(vec.len())];  
4     let mid = vec.partition(vec, pivot);  
5     let (less, greater) = vec.split_at_mut(mid);  
6  
7     rayon::join(|| qsort(less),  
8                 || qsort(greater));  
9  
10 }
```

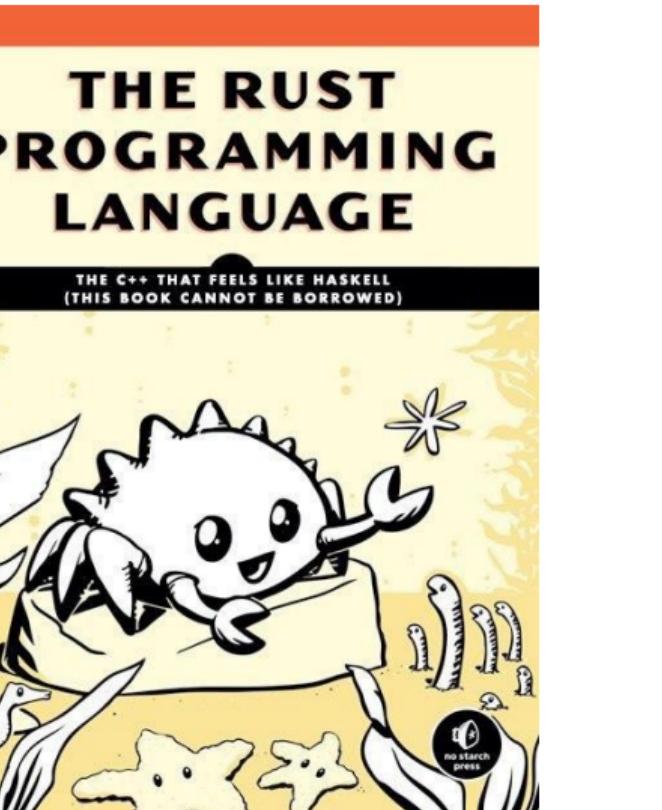
Кто и как часто писал распараллеленные программы? Я писал очень мало.

Не требуется понимать весь код, который представлен здесь. Достаточно понять, как легко с помощью библиотеки `rayon` (среднеуровневая библиотека) здесь реализована распараллеленная быстрая сортировка.

На **других ЯП** распараллеленная быстрая сортировка, как правило, выглядит гораздо **сложнее**, в Rust же она лишена таких багов, как **data races**, благодаря вышеназванным концепциям.

Больше о параллелизме я говорить не буду, так как данная тема займет всё время, да и я не шарю на столько, чтобы про него вешать.

Syntax



Я не буду рассказывать о всех нюансах синтаксиса, а расскажу только о тех, **которые могут быть интересны и несложны в понимании**.

Как я уже говорил – Rust похож на смесь C++ и Haskell, а также других ML языков, поэтому Rust, как в принципе и многие другие языки, немного императивный и немного функциональный. Для меня идеально, т. к. считаю, что и та, и та парадигмы должны присутствовать в хорошем языке.

Syntax

Concepts

```
1 //! # Main
2 //! Module docs
3
4 /// Docs
5 // Comments
6 fn main() {
7     let x = 31337;
8     println!("The value of x is: {}", x); // 31337
9     let mut y: u8 = 5;
10    y = x as u8;
11    println!("The value of y is: {}", y); // 105
12 }
```

Есть встроенная система документации, о которой расскажу позднее.

От функциональных языков в Rust пришла **иммутабельность по-умолчанию**: все переменные, объекты, функции и т. п. по-умолчанию неизменяемые.

Rust обладает **статической сильной типизацией** с возможностью **вывода типов** (type inference). Используется **система типов Хиндли-Милнера** (впервые использовалась в ML языках), благодаря которой и работает вывод типов, а также другие особенности, о которых позже.

Приведение типов в Rust автоматическое для безопасных случаев, в остальных – ручной кастинг двух видов **as** и **transmute**.

```
1 fn nsa(is_hack: bool, backdoor: &str, blue_pill: String) -> f64 {  
2     for c in blue_pill.chars() {  
3         print!("{} ", c);  
4     }  
5     if is_hack {  
6         loop { break 3.1337; }  
7     } else if backdoor.len() > 3 {  
8         42.0 - 42.0  
9     } else {  
10        3.14  
11    }  
12 }
```

В Rust **всё есть выражение** (expression).

Пример задания параметров функции и возвращаемого значения.

Для управления потоком используются: `while`, `loop`, `if-else`, `for` – проходит по элементам, например, по итераторам.

Syntax

Enums (Algebraic data type)

Enums (Algebraic data type)

```
1 enum Pohek {
2     XSS(XssType),
3     SocialEngineering,
4     Phishing,
5     // ...
6 }
7
8 enum XssType {
9     Reflected,
10    Stored,
11    // ...
12 }
```

```
1 match pohek {
2     Pohek::XSS(xss_type) =>
3     {
4         hack_by_xss(xss_type);
5     },
6     Pohek::SocialEngineering |
7     Pohek::Phishing =>
8     {
9         разВесми_ЖИОХА();
10    }
11    - => {},
12 }
```

Enum в Rust используется для создания **алгебраических типов** — можно создать один тип, который будет **включать себя сумму типов**. Да, во множестве языков есть алгебраические типы, но в Rust с ними также удобно работать, как и в других функциональных языках.

Одним из ярких примеров удобства является **конструкция match**, заменяет switch, даёт гораздо больше возможностей, работает с деструктурированием данных, отслеживает полное покрытие всех типов, позволяет задавать условия и т. д.

```
1 fn find_vulnerability(program: &Program) -> Option<Vulnerability>
2   ↪ { . . . }
3
4 fn hack_program(program: &mut Program) {
5   match find_vulnerability(&program) {
6     Some(vuln) => exploit(vuln),
7     None => println!("Better luck next time."),
8   }
}
```

Когда мы пытаемся найти уязвимость, мы можем её либо найти, либо нет, так вот когда мы не находим уязвимость, нам ничего не возвращается, и мы об этом должны знать.

Благодаря алгебраическим типам, Rust **обходится без Null**. Вместо него используется алгебраический тип **Option**, который может возвращать либо объект типа **Some с данными внутри**, либо объект типа **None**, который и является заменой Null.

Всё это опять же пришло из функциональных языков.

Optional (C++)

- std::optional
- std::variant
- std::any
- std::pair

Кто разрабатывает на C++? Вы знали, что там есть подобные алгебраические типы?
Если вы не знали, в C++ тоже есть подобные алгебраические типы, а также `std::optional` и `std::variant` с C++17, но там не всё так просто:

- иногда приходится проверять возвращаемый тип;
- приходится использовать либо boost, либо C++17, что вносит некоторую сумятицу;
- в случае ошибки же **выходит какой-то ад (next slide)**.

Чего, кто, кого? Где ошибка?

На самом деле так со всем в C++, где используются шаблоны или параметризированные типы.

```
error C2664: 'void
std::vector<block,std::allocator<_Ty>>::p
ush_back(const block &)' cannot convert
argument 1 from 'std::
_Vector_iterator<std::_Vector_val<std::
_Simple_types<block>>>' to 'block &&'
```

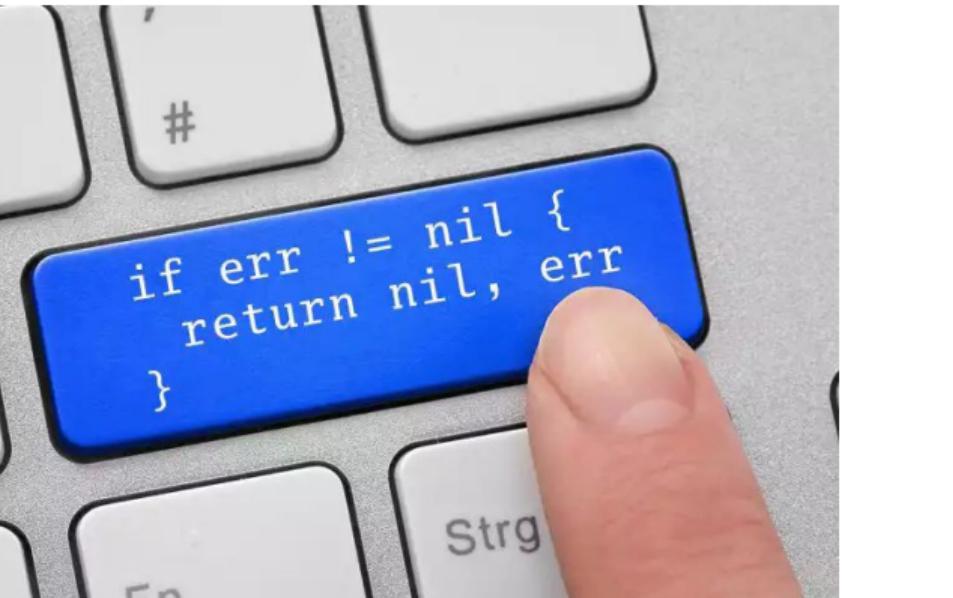


Syntax

Error handling

Error handling

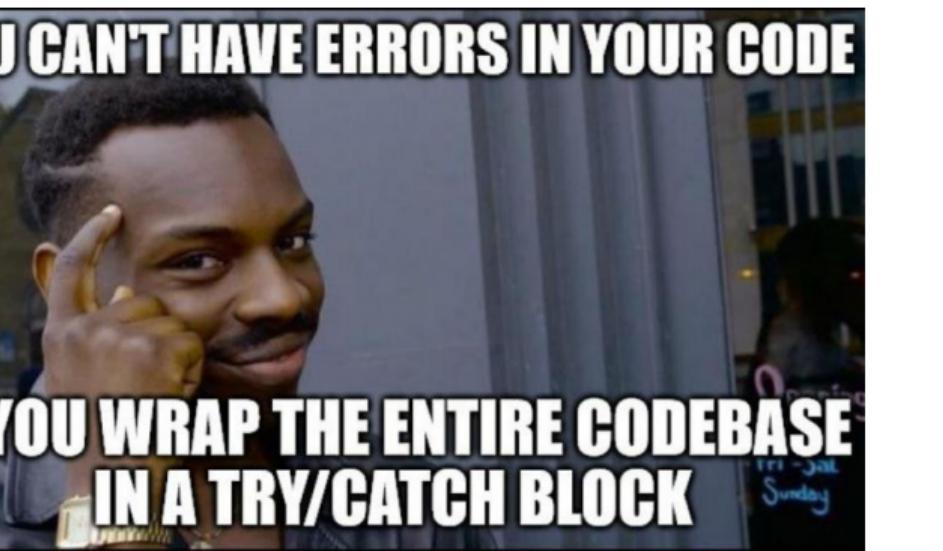
- Return code (C, Go)



Древний способ отлова ошибок, но кто-то до сих пор внедряет его в новые языки. Никаких гарантий, что ты не забудешь написать проверку, к тому же нужно знать все коды ошибок.

Error handling

- Return code (C, Go)
- Exceptions (C++, Python)



Самый популярный на данное время способ отлова ошибок.
Меня смущает один факт — как вы узнаете, какой может быть exception и вызывает ли эта функция exception?

Error handling

- Return code (C, Go)
- Exceptions (C++, Python)
- Global variable (custom)

Задаётся глобальная переменная, в которую записываются все ошибки.
Встречал только в реализациях различных фреймворков, особенно на JS.

Error handling

- Return code (C, Go)
- Exceptions (C++, Python)
- Global variable (custom)
- Design by Contract (SPARK)

Проверка входных и выходных параметров заданному условию.
Крутой способ, конечно, но это уже по части языка SPARK или контрактного программирования.

Error handling

Это уже наш вариант.

- Return code (C, Go)
- Exceptions (C++, Python)
- Global variable (custom)
- Design by Contract (SPARK)
- Error (success) indicator (Haskell)

slaps roof of language standard
this bad boy can fit so much
undefined behavior in itx6²I2÷Pts}I—iO≥D@f—k—0_¶2;í"—"æ



Все знают, что в С и С++ есть **неопределённое поведение**, мало того, у каждого компилятора **своё поведение** не определено. В Rust нет понятия неопределенного поведения в принципе, бывают только баги компилятора (nightly), которые исправляются, все остальные возможные ошибки так или иначе отлавливаются.

Есть определённый род ошибок, который отлавливается Rust во время выполнения. Иногда в программе возникают подобного рода ошибки, которые программист **не может обработать или не хочет**, так как считает, что программа должна завершиться, если произойдёт такого рода ошибки. Пример: прикладное приложение не смогло выделить память на куче, в таком случае стоит завершить приложение, т. к., видимо, что-то не в порядке с ОС.

Panic

```
1 fn main() {  
2     let v = vec![1, 2, 3];  
3  
4     v[99];  
5 }
```

Вот такое информативное сообщение мы получим, можем изменить его по своему желанию. Можно даже сделать так, чтобы отправлялись сообщения на почту с логами. Кроме всего прочего Rust позволяет нам раскрутить стек, до места, где произошла ошибка, указывая при этом точное местоположение.

Output

```
1 thread 'main' panicked at 'index out of bounds: the len is 3 but
   ↳ the index is 99', /checkout/src/liballoc/vec.rs:1555:10
2 note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

Output

```
1 ...
2: std::panicking::default_hook::{closure}
3     at /checkout/src/libstd/sys_common/backtrace.rs:60
4     at /checkout/src/libstd/panicking.rs:381
5 ...
6 11: panic::main
7     at src/main.rs:4
8 12: __rust_maybe_catch_panic
9     at /checkout/src/libpanic_unwind/lib.rs:99
10 13: std::rt::lang_start
11     at /checkout/src/libstd/panicking.rs:459
12     at /checkout/src/libstd/panic.rs:361
13     at /checkout/src/libstd/rt.rs:61
14 14: main
15 ...
```

По раскрученному стеку очень просто понять, где возникла ошибка и по какой причине. Это вам не segmentation fault непонятный.

Некоторые ошибки могут быть не настолько критичными, чтобы паниковать, поэтому вводится **новый алгебраический тип**, который может содержать в себе либо **возвращаемый объект** (в случае успеха), либо **объект-ошибку**.

Это удобно, т. к. позволяет нам узнать **полную информацию об ошибке** в случае чего, а также задавать абсолютно **любые типы в качестве возвращаемых**.

Result

```
1 enum Result<T, E> {  
2     Ok(T),  
3     Err(E),  
4 }
```

Result

```
1 pub fn hack_program(program: &Program) -> Result<Shell> { ... }

2

3 match hack_program(&program) {
4     Ok(shell) => connect(shell),
5     Err(error) => {
6         // Do something with error
7     }
8 }
```

Вернёмся к нашей **крутой программе**, которая ломает всё, что плохо лежит.
Когда мы пытаемся взломать программу, мы **можем получить ошибку**, а **можем получить шелл**.
В случае, если получаем шелл, то подключаемся, если же взломать не получилось, то либо
пытаемся понять почему (**обрабатывая ошибку**), либо **пробрасываем ошибку** дальше, либо
завершаем программу и грустим, либо всё, что вам угодно.

Для прорасывания ошибки мы можем использовать `?`.
Также мы можем создавать цепочки вызовов функций, которые будут выполняться в случае успеха.
Можно ещё много всего показывать и рассказывать, но времени нет.

Result

```
1 fn hack_world(world: World) -> Result<Power, u32> {
2     hack_program(&program)?;
3
4     for program in &world.programs() {
5         hack_program(program).map(install_spy).map(get_money)?;
6     }
7 }
```

Syntax

Structs

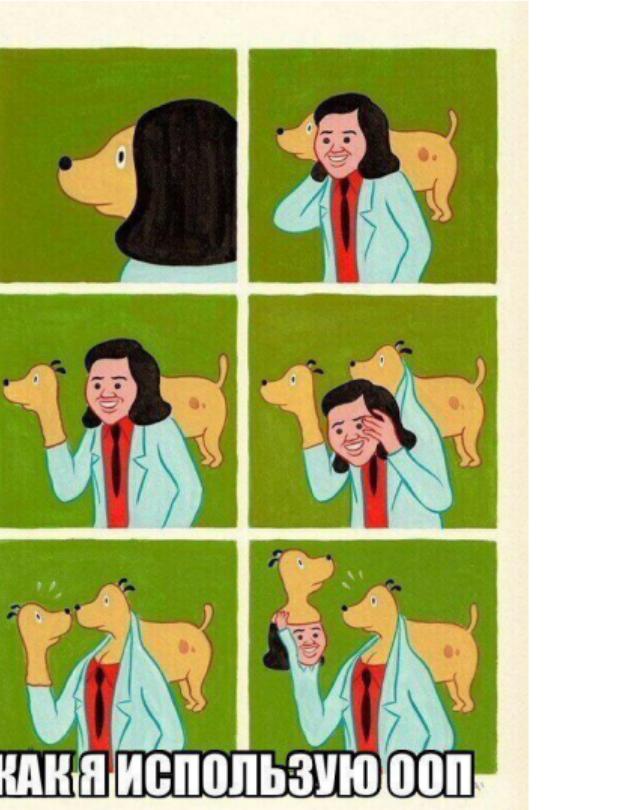
Object-oriented programming



Проблемы при использовании ООП

В C++ есть классы, в Python есть классы, во множестве языков есть парадигма ООП, а это объекты, методы, поля и т. п., а если быть точным: инкапсуляция, наследование, полиморфизм и другие концепции. Лично мне парадигма ООП приносила много проблем, я её не понимал, считал её неудобной и ущербной.

Object-oriented programming



Я читал много книг по ООП и **пытался понять, как нужно писать правильно**. Одно время даже на Smalltalk хотел программировать, чтобы всё понять как следует. Из книг я понял одно, что **не ООП плох, а я туп**.

Кто до конца понял ООП и считает, что это именно та парадигма программирования, которой **стоит придерживаться?**

В Rust же **чистого ООП**, которые привыкли многие видеть, — **нет**. Да, там есть некоторые концепции, например полиморфизм, инкапсуляция, но, например, чистого наследования там как такового нет. **Там другой подход**, который мне как раз таки пришёлся по душе. Конечно, это всё дело вкуса, но мало ли, кому-то не нравится ООП парадигма, посмотрите на Rust или на функциональные языки, потому что даже на C++ можно писать в другой парадигме.

Из-за того, что в Rust нет чистого ООП многие **люди приходят в язык**, пытаются писать **не идиоматичный код**, компилятор **бьёт** их больно по рукам, они плачут и ругают Rust.

```
1 struct Hacker {  
2     nickname: String,  
3     scope: Scope,  
4     cves: Vec<u32>,  
5 }  
6  
7 enum Scope {  
8     Fuzzing,  
9     Developing,  
10    Exploiting,  
11    Reversing,  
12 }
```

В Rust есть структуры, если хотите, можете думать, что это что-то типа классов. Но это не классы, это всего лишь абстракция над данными.

```
1 impl Hacker {  
2     fn new(nickname: String, scope: Scope) -> Hacker {  
3         Hacker {  
4             nickname: nickname,  
5             scope: scope,  
6             cves: Vec::new(),  
7         }  
8     }  
9 }
```

Задать создание структуры можно таким образом, если хотите, можете называть это конструктором, так будет проще.

Более компактная версия.

```
1 impl Hacker {  
2     fn new(nickname: String, scope: Scope) -> Self {  
3         Hacker {  
4             nickname, scope,  
5             cves: Vec::new(),  
6         }  
7     }  
8 }
```

```
1 impl Hacker {
2     fn add_cve(&mut self, cve: u32) {
3         self.cves.push(cve);
4     }
5     fn cves(&self) -> &Vec<u32> {
6         &self.cves
7     }
8 }
```

Таким образом задаются методы. На вход кроме обычных параметров подаётся `self`, т. е. мы подаём экземпляр самого объекта. Вот здесь типичный C++. У нас появляются ассоциированные со структурой методы.

Syntax

Macros

Declarative macros

```
1 #[macro_export]
2 macro_rules! vec {
3     ( $( $x:expr ),* ) => {
4         {
5             let mut temp_vec = Vec::new();
6             $(temp_vec.push($x));*
7             temp_vec
8         }
9     };
10 }
11 let vec_int = vec!(1, 2, 3, 4);
12 let vec_str = vec!("H", "a", "c", "k", "e", "r");
13 println!("{} {}", vec_int, vec_str);
```

В Rust **макросы — это очень мощный инструмент**. Конечно, они не такие, как в Lisp, но как по мне — не менее мощные.

В Rust два типа макросов — **декларативные и процедурные**.

Пример декларативного — такой же, как в C, только основаны они на AST представлении, отчего ошибки в них может **отловить компилятор**, они **гиgienически чистые**, как в Scheme.

Procedural macros (Custom #[derive])

```
1 #[derive(Hackable)]
2 struct System;
```

Procedural macros (Attribute-like)

```
1 #[route(GET, "/")]
2 fn index() {...}
```

Procedural macros (Function-like)

```
1 let sql = sql!(SELECT * FROM posts WHERE id=1);
```

Также есть процедурные макросы, которые в свою очередь делятся на:

- #[derive] макросы,
- макросы-атрибуты,
- макросы-функции.

Syntax

Other

- Generics



Я про многое не смогу рассказать. Например, про параметризованные типы, в Rust называются дженериками.

- Generics
- Traits
 - as interfaces
 - for code reuse
 - for operator overloading
- Trait objects

Traits Polymorphism Generics Concepts

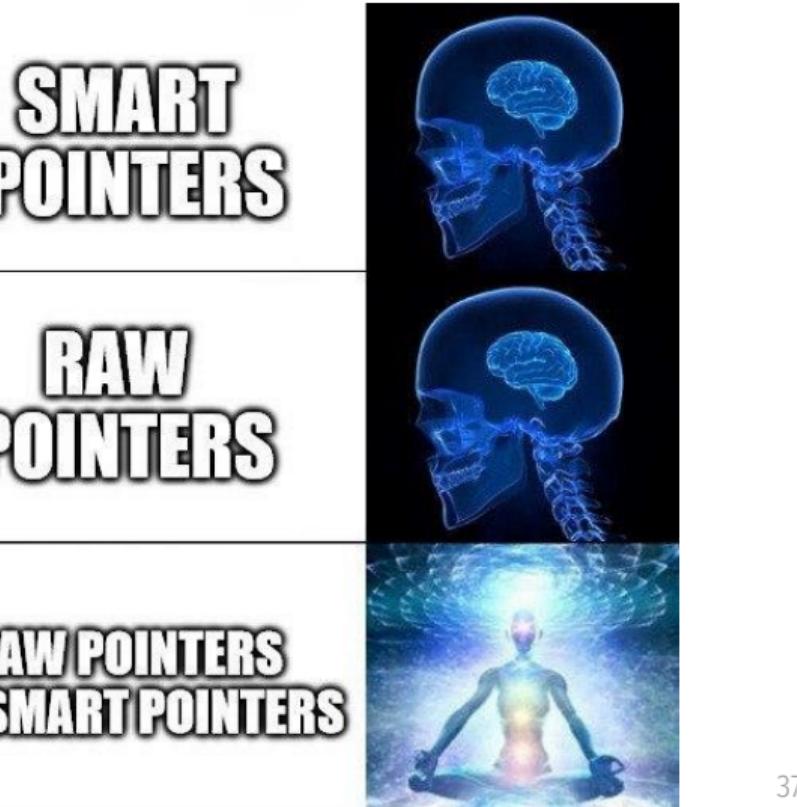


Также не смогу рассказать про трейты (типажи), кстати, кто не знал, в C++ тоже есть типажи, но они не такие, как в Rust. В Rust типажи больше похожи на typeclass из Haskell.
В Rust **нет** всех тех проблем, о которых нам пришлось беспокоиться на C++. Мы можем больше не думать о том, как что-то теряется, когда функция вызывается каким-то образом, и какое влияние оказывает **виртуальная диспетчеризация** на наш код. В Rust все работает в едином стиле, независимо от типа. Таким образом **целый класс детских ошибок просто исчезает**.
В связи с этим, не смогу рассказать большой пласт информации, т. к. многое в Rust связано на типажах.

- Generics
- Traits
 - as interfaces
 - for code reuse
 - for operator overloading
- Trait objects
- Closures (`|x| 2 * x`)

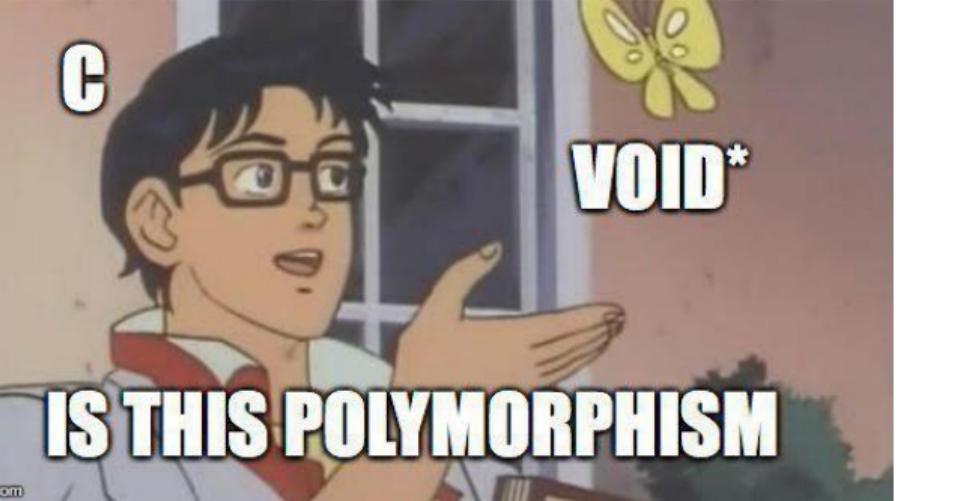
Про замыкания, которые основаны на типажах. Весьма интересно реализованы, ведь GC нет и замыкания не похожи на замыкания из C++.

- Generics
- Traits
 - as interfaces
 - for code reuse
 - for operator overloading
- Trait objects
- Closures (`|x| 2 * x`)
- Common collections
- Smart pointers



Про коллекции структур из стандартной библиотеки и их весьма эффективную реализацию.
Про саму стандартную библиотеку как таковую.
Про умные указатели не расскажу, хотя там всё очень интересно, учитывая, что Rust позволяет достаточно просто создавать мультипоточные программы не без помощи умных указателей.

- Generics
- Traits
 - as interfaces
 - for code reuse
 - for operator overloading
- Trait objects
- Closures (`|x| 2 * x`)
- Common collections
- Smart pointers
- Polymorphism, encapsulation



Также не расскажу, как в Rust реализовывается полиморфизм и инкапсуляция.

- Generics
- Traits
 - as interfaces
 - for code reuse
 - for operator overloading
- Trait objects
- Closures (`|x| 2 * x`)
- Common collections
- Smart pointers
- Polymorphism, encapsulation
- Unsafe
- ...

Unsafe код — место, где вы сможете писать, как на C/C++ и компилятор будет меньше ругаться. И многое другое, например, какие используются **оптимизации на уровне компилятора и промежуточного представления**, которые никак не могут использоваться в C++ по причине обратной совместимости с С. Всё рассказать невозможно.



Ecosystem



Ecosystem

Community

- Rust Working Groups
 - Networking services
 - WebAssembly
 - CLI Apss
 - Embedded Devices
 - Lang and compiler working groups (WG-NLL, WG-UCG, WG-Traits and etc)



Кроме того, что Rust – open source разработка, как я уже говорил, и все фичи обсуждаются и принимаются всеми, **существуют рабочие группы**. Каждая группа отвечает за свою область, вступить в них можно, если активно проявляешь себя в той или иной области. Каждая группа отчитывается за проделанную работу, а также составляет планы на будущее.

- Rust Working Groups
 - Networking services
 - WebAssembly
 - CLI Apps
 - Embedded Devices
 - Lang and compiler working groups (WG-NLL, WG-UCG, WG-Traits and etc)
- This week in Rust (This week in ProjectName)



Каждую неделю постятся новости, полезные ссылки, статьи, цитаты, информация о сходках, о том, какие изменения в компиляторе разработали, какие RFC принимаются, что стабилизируется, работа, проекты, которым нужна помощь и о прочем. Не требуется собирать свежую информацию по крупицам со всего интернета.

Кроме этого по некоторым проектам также каждую неделю постится информация.

- Rust Working Groups
 - Networking services
 - WebAssembly
 - CLI Apss
 - Embedded Devices
 - Lang and compiler working groups (WG-NLL, WG-UCG, WG-Traits and etc)
- This week in Rust (This week in ProjectName)
- Meetups



По Rust очень много сходок по всему миру. Кроме того, что за год проходит по **две официальные конференции**, во всех странах мира создаются небольшие сходки. В Питере тоже есть такая, собираемся в JetBrains примерно 2 раза месяц. В Москве – организатор Kaspersky.

- Rust Working Groups
 - Networking services
 - WebAssembly
 - CLI Apps
 - Embedded Devices
 - Lang and compiler working groups (WG-NLL, WG-UCG, WG-Traits and etc)
- This week in Rust (This week in ProjectName)
- Meetups
- Chats (IRC, Telegram, Gitter, Matrix, Jabber)



В чатах создано множество тематических групп. В том числе русскоязычных: и по Web разработке, и по embedded, и просто по Rust.

- Rust Working Groups
 - Networking services
 - WebAssembly
 - CLI Apps
 - Embedded Devices
 - Lang and compiler working groups (WG-NLL, WG-UCG, WG-Traits and etc)
- This week in Rust (This week in ProjectName)
- Meetups
- Chats (IRC, Telegram, Gitter, Matrix, Jabber)
- Blogs (Official, Read Rust, Core Developers, many others)



По Rust очень много информации в блогах. Есть много структурированной информации на официальных сайтах, есть блоги core разработчиков и множество других. Core разработчики устраивают офисные часы — раз в 2 недели задаются животрепещущий вопрос, разработчик чётко с примерами отвечает.

Rust сообщество очень дружелюбное! По вопросам в остальных языках могут и заклевать.



Ecosystem

Cargo

Best package manager!

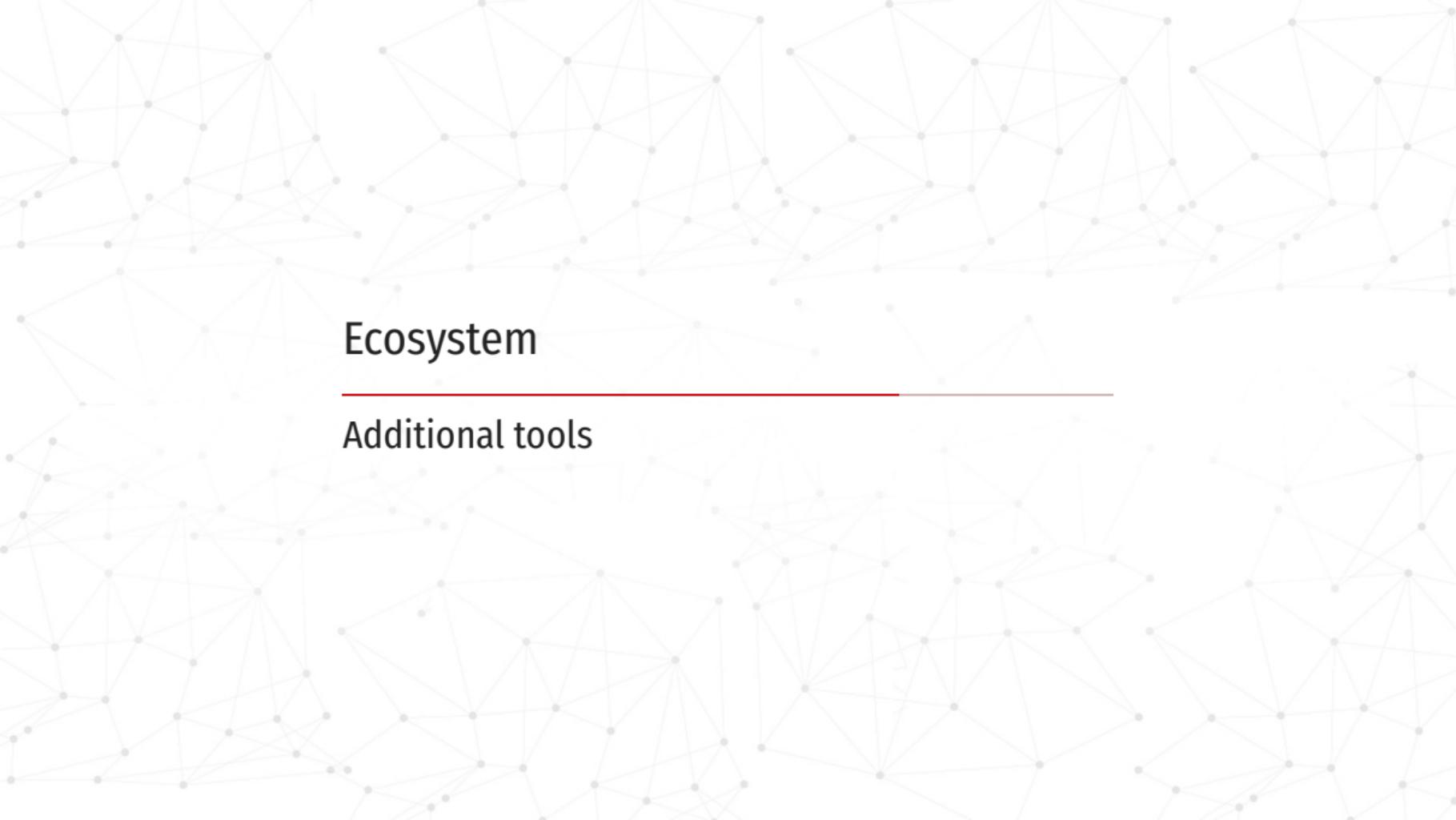
- Create structure of project
- Check and update dependencies
- Check, build and compile project
- Search and install packages
- Compile and run examples
- Generate docs
- Compile and run tests (in docs too)
- Compile and run benchmarks
- etc



- Cargo создаёт каркас проекта
- Следит за зависимостями, обновляет устаревшие по запросу
- Проводит проверку, собирает и компилирует
- Ищет и устанавливает в систему проекты
- Компилирует и запускает примеры
- Генерирует документацию
- Компилирует и запускает тесты
- Компилирует и запускает бенчмарки

В отличие от многих других языков, в Rust пакетный менеджер **разрабатывался вместе с компилятором языка**. Поэтому Cargo — **это стандарт**. Отсутствие зоопарка систем сборки позволяет снизить порог входления в чужие проекты за счёт одинаковой структуры проектов и единого унифицированного способа сборки зависимостей.

npm (js), pip (python), maven (java), hex (elixir), bundler (ruby), nuget (.NET), CPAN (perl), cabal (haskell), OPAM (OCaml), Elm, cmake, make, gult — отстой.



Ecosystem

Additional tools

Additional tools

Cargo plugins

- cargo-asm
- cargo-call-stack
- cargo-clippy
- cargo-fmt
- cargo-fuzz
- cargo-geiger
- cargo-graph
- cargo-install-update
- cargo-llvm-ir
- cargo-profdata
- cargo-size
- many others!

Для cargo можно устанавливать плагины.

- для показа CFG, DFG
- для автоматического форматирования всего проекта
- для автоматического фаззинга
- для нахождения неидиоматичного кода с последующим автоматическим исправлением
- для генерации промежуточного представления
- для профилирования
- ещё куча всего!

Например, вот так просто вызываются санитайзеры для кода.

Sanitize code

```
1 $ RUSTFLAGS="-Z sanitizer=address" cargo test
2 $ RUSTFLAGS="-Z sanitizer=leak" cargo test
3 $ RUSTFLAGS="-Z sanitizer=memory" cargo test
4 $ RUSTFLAGS="-Z sanitizer=thread" cargo test
```

- Rust documentation
- Crates.io – all packages
- Docs.rs – all documentation
- Rust book
- Rust Playground



Как у любого уважающего себя языка, у Rust есть **online документация** по самому языку и по стандартной библиотеке.

Также есть удобный **сайт со всеми пакетами**.

А к нему есть официальный сайт с **автогенерируемой документацией** по всем пакетам. И всё в **одном стиле!**

Есть уже **второе издание online книги** по Rust, также есть печатная версия. Советую ознакомиться. Кроме этой книги, есть ещё **пара официальных** для более продвинутых.

Кроме того, есть **официально поддерживаемая** площадка для компилирования кода **online**.



Ecosystem

Rustup

Rustup

Rustup install

```
$ curl https://sh.rustup.rs -sSf | sh
```

Rustup – это **инсталлятор Rust**, если можно так выразиться. Через него вы можете установить компоненты языка (компилятор, стандартную библиотеку, утилиты для разработки, тулчайны, контролировать версии компилятора).

Вот такой простой командой вы можете установить rustup или через системный пакетный менеджер.

Toolchain format

<channel> [<date>] [<host>]

<channel> = stable|beta|nightly|<version>

<date> = YYYY-MM-DD

<host> = <target-triple>

Install nightly toolchain

\$ rustup toolchain install nightly

У Rust официально есть три версии тулчайна:

- stable
- beta
- nightly

Для кросскомпиляции проекта надо всего две команды!

Cross compile

```
$ rustup target add mips64el-unknown-linux-gnuabi64  
$ cargo build --target=mips64el-unknown-linux-gnuabi64
```

- aarch64-apple-ios
- aarch64-fuchsia
- arm-unknown-linux-gnueabihf
- armv5te-unknown-linux-musleabi
- asmjs-unknown-emscripten
- i686-pc-windows-msvc
- powerpc-unknown-linux-gnu
- riscv32imac-unknown-none-elf
- sparcv9-sun-solaris
- wasm32-unknown-emscripten
- x86_64-unknown-redox
- ...

Существует многочисленная поддержка архитектур, ОС и библиотек, в том числе **bare metal устройств**.

Кроме инструментов, про которые я рассказал, **есть и другие**, но на них уже нет времени.



Ecosystem

Rust in production

Rust in production

Hundreds of companies around the world are using Rust in production today for fast, low-resource, cross-platform solutions

- Mozilla
- Cloudflare
- Microsoft
- Facebook
- Ready at Dawn Studios
- CoreOS, Inc.
- The GNOME Project
- Coursera
- Unity
- Google
- npm, Inc.
- Amazon
- Red hat
- Frostbite Engine
- Parity
- Canonical
- System 76
- Wire

- Samsung
- Dropbox
- Twitter
- Electronic Arts
- Discord
- Atlassian
- Baidu
- Reddit
- many others

Может это и не заметно, но уже множество компаний использует Rust в production, но официально известно лишь нескольких сотнях.

- GameDev (движки, SDK, 3D моделирование, полностью игры), **Ready At Dawn (God of War)**
- везде внутренние утилиты на C/C++
- ОС, файловые системы
- виртуальные машины, песочницы, системы управления ВМ
- ПО для анализа исходного кода, бинарного кода
- кодеки, декодировщики (современные кодеки)
- P2P, mesh сети
- Web сервера
- криптовалюта
- прикладное ПО
- на голом железе
- UEFI, Intel SGX, TrustZone
- системы защиты реального времени (WAF, Anti-DDOS, антивирусы)
- протоколы, криптография
- поисковые движки
- некоторые используют везде: начиная от частей ОС, кончая утилитами (замена скриптам)

Pitfalls

Pitfalls

Compile time errors

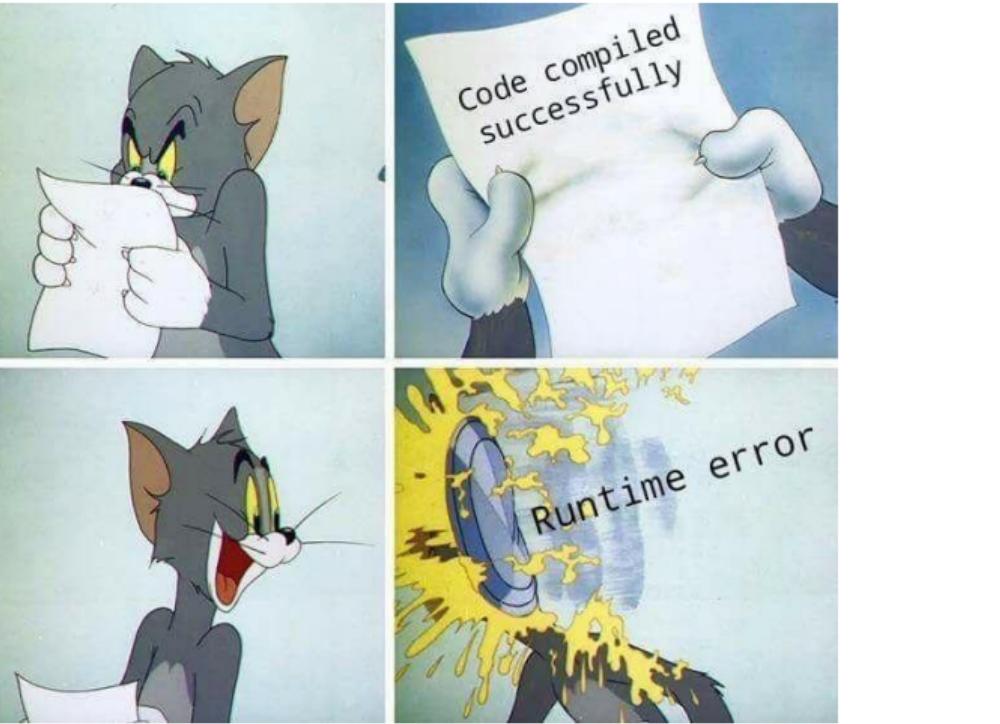
Compile time errors



Rust похож на Haskell системой комплексных типов, поэтому многие ошибки выявляются на этапе компиляции.

Отсюда как раз таки и вытекает **минус Rust по версии многих новичков**. Компилятор очень **больно и много бьёт** по рукам. Многие жалуются, что они делают всё правильно, а компилятор всё равно ругается, особенно это касается момента разработки, когда требуется расставить всё время жизни объектов. Это называется **борьба с borrow checker**.

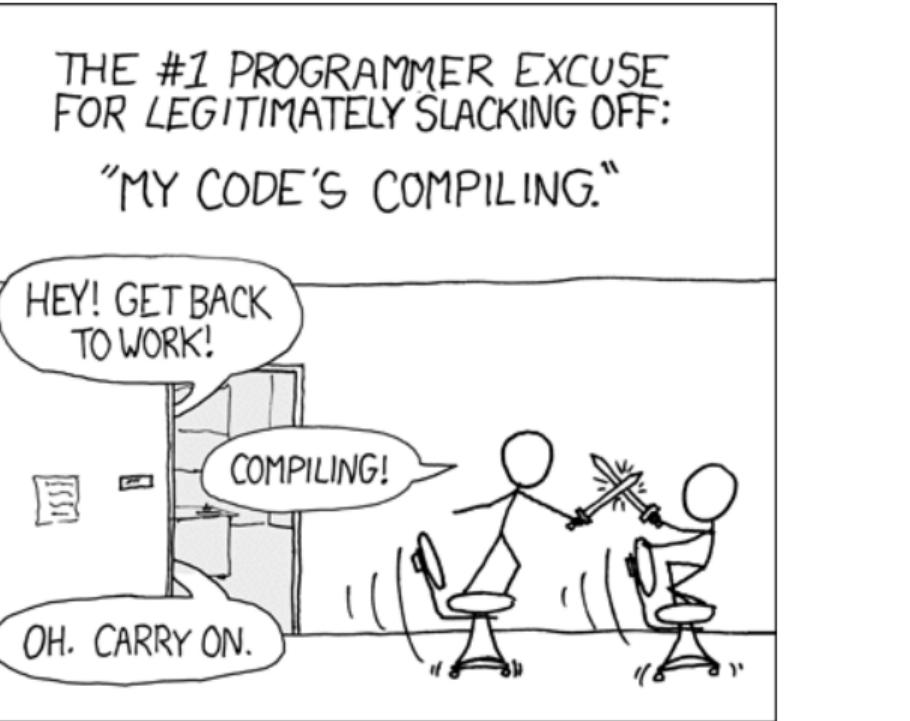
Compile time errors



Вообще, Rust такой язык, что **если программа скомпилировалась без ошибок**, это значит, что в 98% случаев программа будет работать **без ошибок в runtime**. И у вас не возникнет такой ситуации, как на слайде.

Pitfalls

Compilation times



Медленное время компиляции в Rust обусловлено тем, что **атомарной единицей** компиляции является не один файл (как в C/C++), а **целый crate** (или библиотека). Вопрос сейчас решается, уже разработано несколько уровней **промежуточного представления**, позволяющих проводить **инкрементную компиляцию**, существуют инструменты **кэширования промежуточных состояний** компиляции. К тому же в cargo встроена возможность **проверки кода на ошибки**, которая отрабатывает очень быстро, в IDE ошибки отображаются сразу же.

Pitfalls

Complex syntax

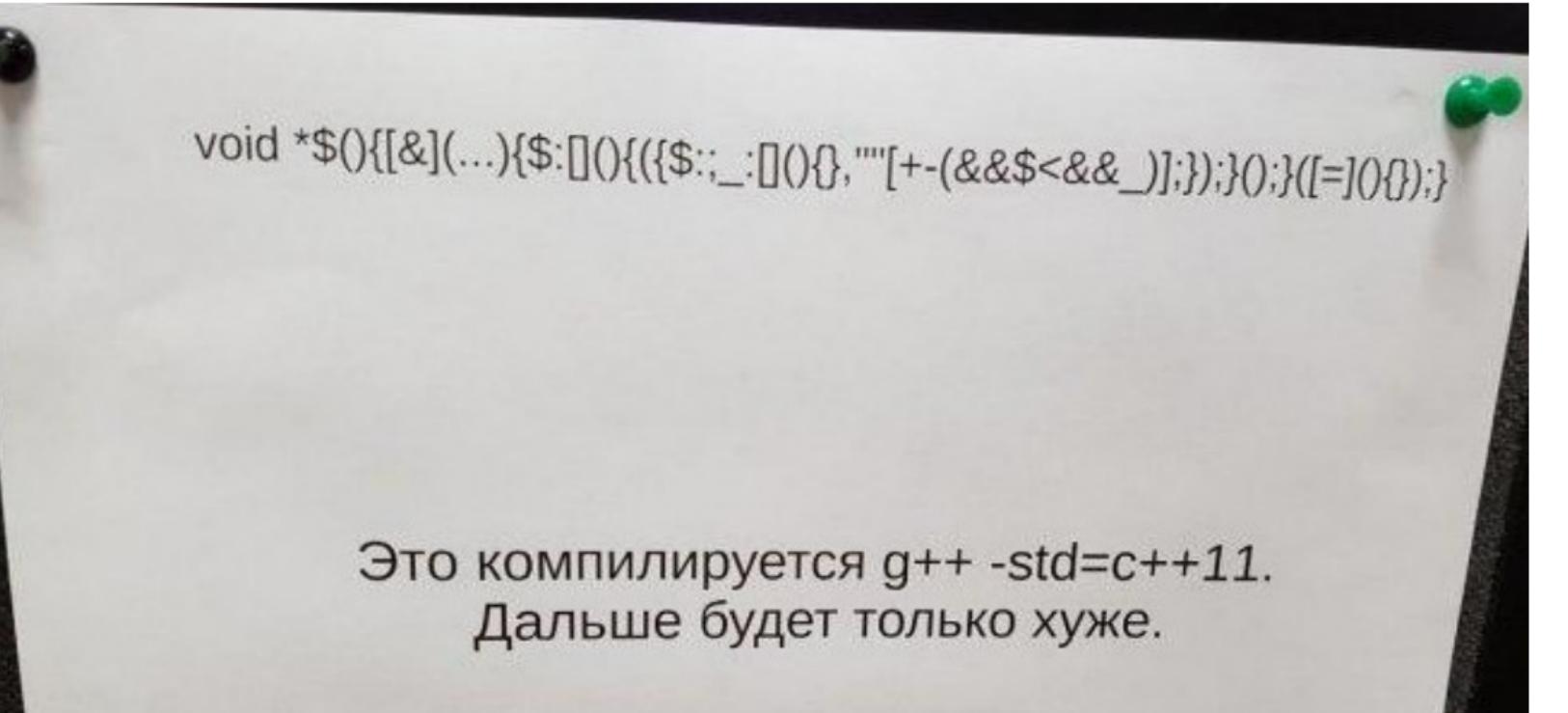
Complex syntax

Rust

```
1 impl<'a, 'gcx, 'tcx> InferCtxt<'a, 'gcx, 'tcx> {
2     pub fn replace_bound_vars_with_placeholders<T>(&self,
3         binder: &ty::Binder<T>) -> (T, PlaceholderMap<'tcx>)
4     where T: TypeFoldable<'tcx>
5     {
6         let next_universe = self.create_next_universe();
7         let fld_r = |br| {
8             self.tcx.mk_region(ty::RePlaceholder(ty::PlaceholderRegion {
9                 universe: next_universe,
10                name: br,
11            }))
12        };
13    ...
14 }
15 }
```

Незнакомых с Rust очень сильно **отпугивает сложный синтаксис** — зачем все эти закорючки, зачем сокращать слова, зачем столько скобок и т. д. Суть в том, что это **язык без GC**, со своей, особой техникой отслеживания времени жизни объекта, отсюда такой комплексный синтаксис, к которому привыкаешь. Если хотите писать на безопасном и быстром языке, от этого никуда не деться.

Лично по моему опыту — когда я начинал изучать Rust, он не был таким популярным, в итоге на форумах никто не брюзжал о том, что синтаксис плох. И когда я изучал Rust у меня не было практически никакого мнения о нём, я знал, для чего он и что может, но как он изучается — в курсе не был. В итоге синтаксис мне не показался чем-то сложным, большие сложности у меня возникли **при осознании построения правильных и идиоматичных абстракций** в Rust. Стоит заметить, что **раньше синтаксис был ещё комплекснее**, это сейчас делают всё, чтобы компилятор понимал, чего хочет программист. Прелесть Rust в том, что в нём сложный процесс написания кода. То есть вы часть ошибок ловите ещё до этапа компиляции. То есть вы вынуждены больше думать наперёд, и в итоге получаете меньше проблем.



И вообще, многое зависит от самого разработчика, как он сам напишет код, потому что уродливо можно написать на любом языке, каким бы лаконичным он ни был.
Лично мне синтаксис Rust никогда не казался сложным или непонятным, даже наоборот, я нахожу его более понятным.

- Чётко видно где и какие методы используются, где они реализованы (а не блуждать в include файлах).
- Чётко понимаю, какой тип у переменной.
- Понимаю какой знак к чему относится (C pointers).
- Код легче читать, потому что всё обернуто в абстракции типов.
- Ясно что и куда передаётся в разных потоках.

Pitfalls

Barriers to entry

Barriers to entry

Typically scope:

- Object-oriented programming
- Garbage collected programming language
- Dynamic programming language

Rust scope:

- No object-oriented programming
- No garbage collector
- No dynamic typing

Изучать **новое** всегда **сложно** и **времязатратно**, а в Rust для типичного программиста будет **много** **всего нового**. По сути придётся изучать **не просто новый язык** программирования (что уже само по себе не легко), а также **статическую систему типов**, **понятия стека и кучи**, **понятие времени жизни объекта**, **дженерики**, **алгебраические типы**, **отвыкнуть от ООП**, **немного погрузиться в функциональное программирование** и т. д.

Pitfalls

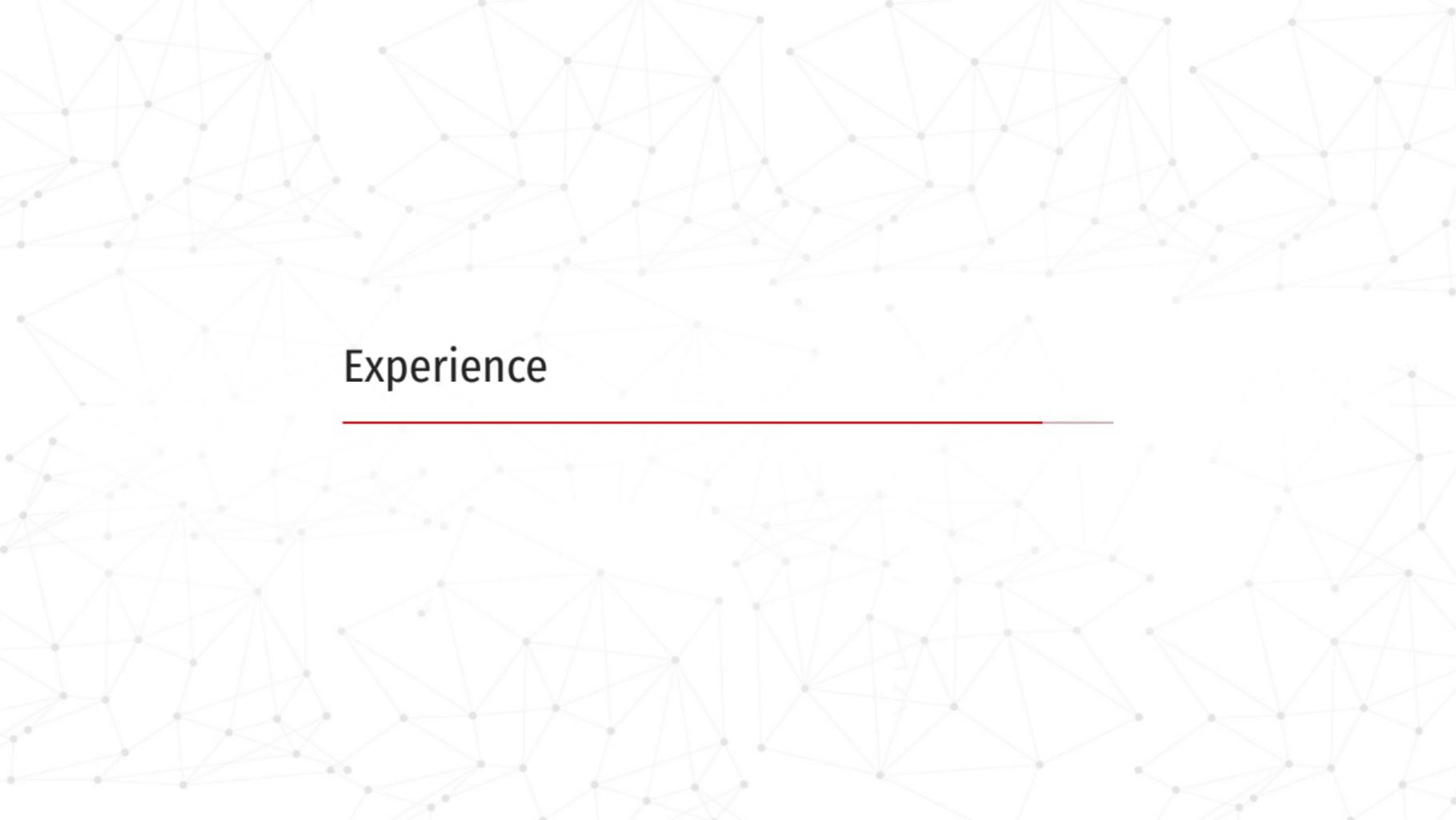
Ecosystem immaturity

Ecosystem immaturity



Тут всё понятно — язык как таковой **ещё молодой**. Далеко не для всех проблем есть решения, реализованные на Rust. **Разработчики железа** пока не спешат **выкатывать SDK** для Rust. Не все **популярные фреймворки внедряют байндинги** для Rust. Иногда приходится реализовывать что-то самому вместо того, чтобы взять готовую библиотеку.

Но Rust **идёт очень уверенным темпом**. Может быть **после моего доклада** кто-то напишет классную утилиту, библиотеку, фреймворк на Rust.

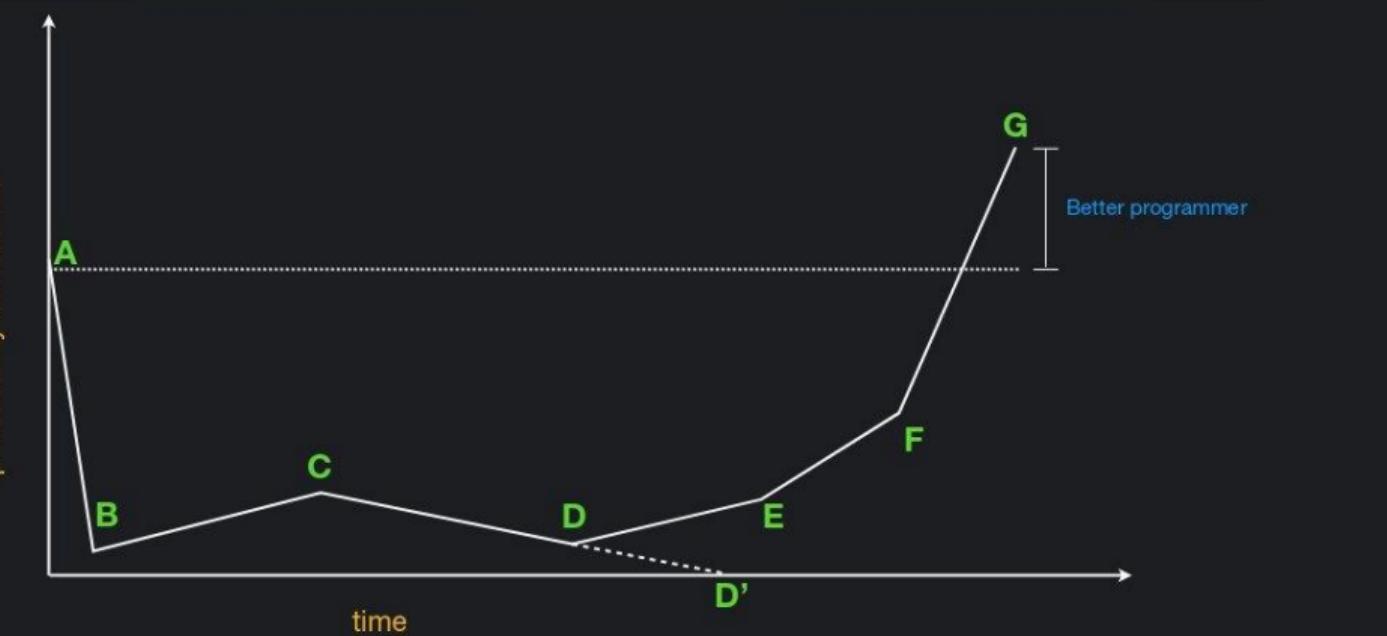


Experience



Experience

Learning curve



- A I know X, How hard Rust can be?
- B Borrow checker: Will my code ever compile? This is hard
- C Borrow checker thing is not that bad. I still cannot write non-trivial code
- D Even more errors while writing non-trivial code. Who decided to use Rust?
- D' I give up. Rust is too hard. Will use Go instead
- E Now I see how things fit. Compiler is indeed my friend.
- F Discover the wonders e.g Rayon. Refactoring is a pleasure.
- G Write much better code in first shot. Increase in productivity

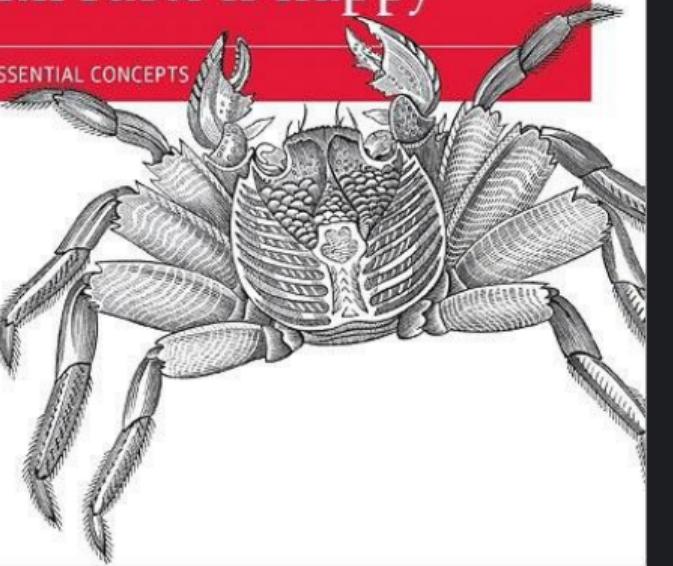
Изучать Rust не так просто, как другие языки. Сравнить кривую обучения можно, пожалуй с тем же C++.

Данный график полностью отражает **кривую моего обучения**.

- Прежде, чем изучать Rust, я почитал немного Rust book, я знал для чего этот язык предназначен и зачем он мне.
- Затем я столкнулся с тем фактом, что сложнее hello world я не могу ничего написать, компилятор в бешенстве. Я не понимаю, что ему надо.
- Методом проб и ошибок я начинаю понимать, что и куда писать, чтобы код работал. Это выглядело примерно так – **следующий слайд!**

Adding and Removing & and * at random until rustc is happy

50 ESSENTIAL CONCEPTS



Также пытаюсь расставить время жизни объектов наобум, как советует компилятор.



- A I know X. How hard Rust can be?
- B Borrow checker: Will my code ever compile? This is hard
- C Borrow checker thing is not that bad. I still cannot write non-trivial code
- D Even more errors while writing non-trivial code. Who decided to use Rust?
- D' I give up. Rust is too hard. Will use Go instead
- E Now I see how things fit. Compiler is indeed my friend.
- F Discover the wonders e.g Rayon. Refactoring is a pleasure.
- G Write much better code in first shot. Increase in productivity

- Как только я пытаюсь написать какой-то нетривиальный код — я снова в заложниках у компилятора. Тут обычно люди и уходят с Rust.
- Я начинаю понимать, что компилятор реально от меня хочет, что я делаю не так. Я начинаю строить абстракции заранее так, как этого хочет компилятор.
- Начинают **познавать что-то новое**: библиотеки, инструменты, другие подходы, парадигмы.
- Если понимаю архитектуру, то пишу код **с первого раза верно**, увеличилась **производительность** как программиста.

После того, как изучил Rust, когда пишешь на другом языке, всё равно **пытаешься думать правильно**, составлять **абстракции верно** изначально, невольно **избегаешь ошибок** при работе с памятью!

Кривая похожа на Emacs — сначала, вроде, всё легко, потом сложно, а потом много нового.

Experience

Format of errors or compiler driven development

Format of errors or compiler driven development



Есть много различных методологий разработки (через тестирование, через поведение, agile), но у меня впервые разработка на основе поведения компилятора (или вывода на ошибках).

Format of errors or compiler driven development

Пример работы с параметрическими типами. Я просто убрал символ ссылки.

C++

1 Too big, too unclear

Rust

1 error: expected type, found `'static`
2 --> test_err.rs:3:9
3 |
4 3 | Ref('static str),
5 | ^^^^^^
6
7 error: aborting due to previous error

C++

```
1 In file included from /usr/include/c++/8.2.1/cassert:44,
      from test_err.cpp:3:
2 test_err.cpp: In function ‘int main()’:
3 test_err.cpp:17:5: error: invalid use of void expression
4     assert(std::holds_alternative<std::string>(y)); // succeeds
5
6 ^~~~~~
```

Format of errors or compiler driven development

Rust

```
1 error: expected one of `.`~, `;`~, `?`~, or an operator, found `}`~  
2 --> test_err.rs:6:1  
3 |  
4 5 |     let y = S("xyz".to_string())  
5 |                         - expected one of `.`~, `;`~,  
6 |                         → `?`~, or an operator here  
6 | }  
7 | ^ unexpected token  
8  
9 error: aborting due to previous error
```

Кроме того, что компилятор говорит точно, где ошибка, он ещё может посоветовать [как её исправить](#).

Кроме всего, почти для каждой ошибки есть раздел в [одной большой книге ошибок](#), где описывается типичный пример ошибки и [гайд по её исправлению](#).



Experience

Zero-cost abstractions



Zero-cost abstractions

- Traits (static and dynamic dispatching)
- Zero sized types
- Closures
- Markers
- Higher-kinded types
- Compile-time function execution
- ...

В Rust реализовано множество техник, которые позволяют избежать накладных расходов при написании высокоуровневого кода.

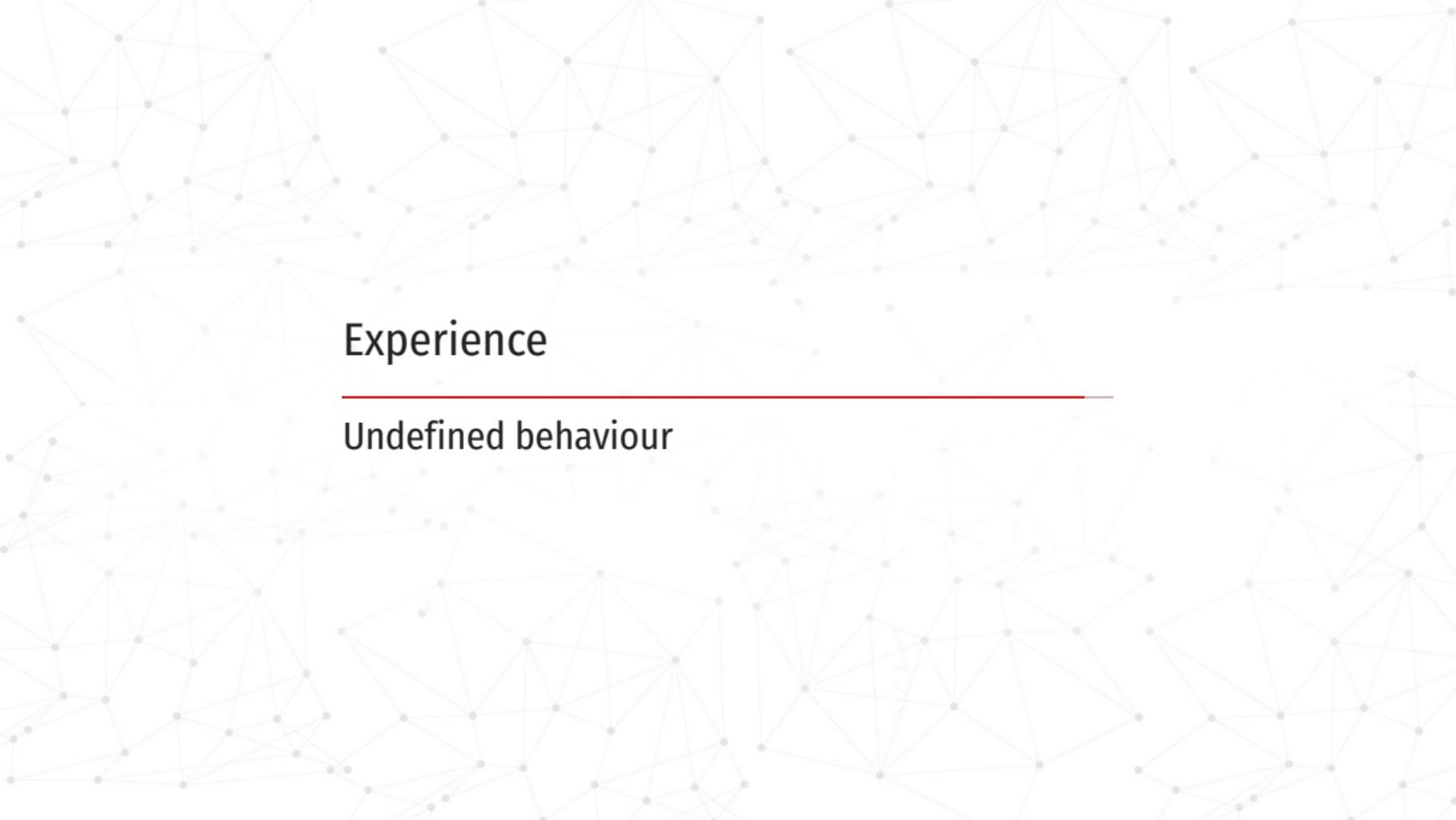
C++ также обладает zero-cost abstractions, т. е. за высокоуровневыми абстракциями прячется высокоэффективный код (абстракции без накладных расходов). Но даже самые важные C++ люди утверждают, что им далеко до Rust, т. к. чтобы достигнуть такого уровня абстракций, придётся поломать весь язык и обратную совместимость.

Zero-cost abstractions

- Python: `sum(range(1000))` – 1000 iterations and 1000 additions
- Rust: `(0..1000).sum()` – 499500



Извините, не мог сдержаться, у меня просто **полыхает от Python**.
Самым примитивным примером может послужить вычисление суммы.



Experience

Undefined behaviour



Undefined behaviour

C

```
1 memset(c, 0, sizeof(Net_Crypto));  
2 free(c);
```

После C, Rust — это как глоток свежего воздуха. Но одним из самых ярких плюсов для меня считается **отсутствие UB**.

В примере `memset` может затеряться, потому что компилятор думает: а зачем занулять, если я сейчас всё равно удалю указатель. **Это реальный пример** из криптографической библиотеки, которая была **переписана на Rust** впоследствии.

В Rust есть только **Unsound** штуки, по-другому — баги в компиляторе или в стандартной библиотеке, которые **естественно фиксятся**. UB нет в RFC, в отличие от стандарта C/C++, в котором чёрт ногу сломит.

Summary

Summary

Still have logical problems or wrong architecture, but

- Fearlessness

- no race conditions,
- no leaking resources,
- no dangling pointers,
- no unhandled exceptions,
- no NULL dereferences,
- no out of bounds,
- ...

With Rust I can focus on the real problems.

Я практически **не затронул параллелизм!**

На Rust легко программирувать надёжные системы. Да, я до сих пор буду совершать ошибки при разработке системы в целом, но зато мне не надо думать о целом классе других ошибок. Я могу полностью сконцентрироваться на архитектуре, на конечной цели.

Rust is reasonably good at pretty much everything.

- Fearlessness
- Universality
 - Embedded systems
 - WebAssembly frontend code
 - Quick and dirty utility
 - Sophisticated tool
 - 3D engine or game
 - Business server-side app
 - Mobile app
 - OS and drivers
 - ...

Rust можно использовать для написание практически всего, чего угодно. Я вам приводил примеры того, для чего использовал Rust я, а также примеры того, где используют Rust крупные компании в production.

- Fearlessness
- Universality
- Combines strengths of “best tools for the given job”

- The performance, power and control of C/C++
- Memory safety of JVM/scripting languages
- Expressive type system like OCaml/Haskell/Scala
- Automatic memory management like a GC
- Dependency management and code sharing like Node
- Error messages like Elm
- Built-in message passing like Go
- ...

Rust объединяет в себе лучшее других языков, при этом оставляет за бортом плохое.

- Но безопасный и более юзабельный.
- Но без тяжёлого runtime.
- Но без GC и его проблем как таковых.
- Взято у других и улучшено.

- Fearlessness
- Universality
- Combines strengths of “best tools for the given job”
- Ownership system

Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

Rust will force you to be a good programmer, you like it or not.

Благодаря системе владения, приходится писать код короче, быстрее, лёгким для понимания и изменения, хочу я этого или нет (иначе компилятор будет ругаться).

Summary

- Fearlessness
- Universality
- Combines strengths of “best tools for the given job”
- Ownership system
- Community and collaboration

Rust community is just the best.

Я реально не встречал сообщества более лучшего, чем у Rust. Вы можете просто кинуть ссылку на Playground и попросить сделать code review. И вам его сделают, какой бы уровень у вас не был.

Мейнтейнеры самые доброжелательные, многие пофиксят баги, которые ты попросишь, потому что писать на Rust в кайф, хочется, чтобы твой код был максимально чистым и рабочим.

- Fearlessness
- Universality
- Combines strengths of “best tools for the given job”
- Ownership system
- Community and collaboration
- **Tooling**
- rustup
- cargo
- xargo
- rls
- racer
- rustfmt
- ...

Все тулы для Rust кажутся идиоматичными и составляют единое целое. Пока не попробуешь – не поймёшь.

- Fearlessness
- Universality
- Combines strengths of “best tools for the given job”
- Ownership system
- Community and collaboration
- Tooling
- Keeps getting better



Что ни новая версия Rust — то стабилизация какой-то крутой фичи. Многие языки программирования копируют концепции Rust.
Заинтересовавшимся **могу накидать кучу ссылок**, где Rust описывается профессиональными разработчиками на C/C++, а также где Rust приходит на замену чего-либо (C/C++ или Python).

Questions?



Smart pointers

- `Box<T>` – for allocating values on the heap
- `Rc<T>/Arc<T>, Weak<T>` – a reference counting type that enables multiple ownership
- `Ref<T>, RefMut<T>, RefCell<T>/Mutex<T>` – a type that enforces the borrowing rules at runtime instead of compile time
- ...

Smart pointers

cons of Lisp

```
1 enum List {
2     Cons(Rc<RefCell<i32>>, Rc<List>),
3     Nil,
4 }
5 fn main() {
6     let value = Rc::new(RefCell::new(5));
7
8     let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));
9
10    let b = Cons(Rc::new(RefCell::new(6)), Rc::clone(&a));
11    let c = Cons(Rc::new(RefCell::new(10)), Rc::clone(&a));
12
13    *value.borrow_mut() += 10;
14 }
```



Closures

By reference (&T)

```
1 let color = "green";
2 // A closure to print `color` which immediately borrows (`&`)
3 // `color` and stores the borrow and closure in the `print`
4 // variable. It will remain borrowed until `print` goes out of
5 // scope. `println!` only requires `by reference` so it doesn't
6 // impose anything more restrictive.
7 let print = || println!("`color`: {}", color);
8
9 // Call the closure using the borrow.
10 print();
11 print();
```



By mutable reference (&mut T)

```
1 let mut count = 0;
2 // A closure to increment `count` could take either `&mut count`
3 // or `count` but `&mut count` is less restrictive so it takes
4 // that. Immediately borrows `count`.
5 let mut inc = || {
6     count += 1;
7     println!("`count`: {}", count);
8 };
9 // Call the closure.
10 inc();
11 inc();
12 //let _reborrow = &mut count;
```



By value (T)

```
1 // A non-copy type.  
2 let movable = Box::new(3);  
3 // `mem::drop` requires `T` so this must take by value. A copy type  
4 // would copy into the closure leaving the original untouched.  
5 // A non-copy must move and so `movable` immediately moves into  
6 // the closure.  
7 let consume = || {  
8     println!("`movable`: {:?}", movable);  
9     std::mem::drop(movable);  
10};  
11 // `consume` consumes the variable so this can only be called once.  
12 consume();  
13 //consume();
```

Closures

- Fn: the closure captures by reference (`&T`)
- FnMut: the closure captures by mutable reference (`&mut T`)
- FnOnce: the closure captures by value (`T`)