Digital Security

Rust and what’s this thing for?

Abc Xyz
@dura_lex

Digital Security

Agenda

1. Foreword
2. What is Rust?
3. (Un)safe
4. Syntax
5. Ecosystem
6. Experience and Pitfalls
7. Summary

Foreword

Motivation

Motivation

Motivation


THE FIRST RULE OF RUST
TELL EVERYONE TO USE RUST

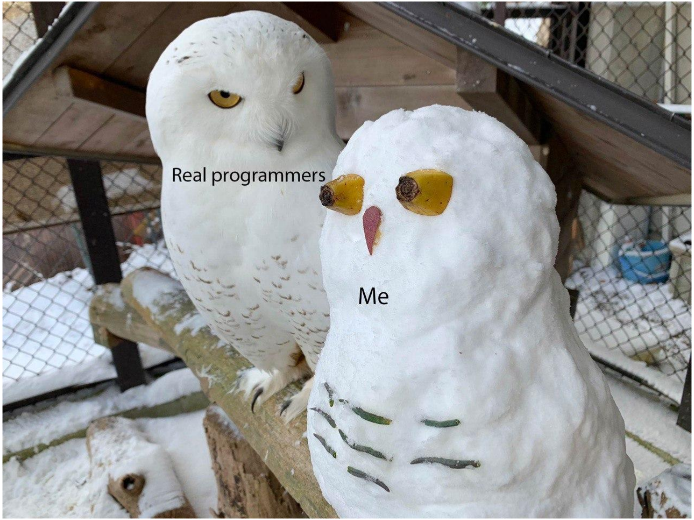

THE SECOND RULE OF RUST
TELL EVERYONE TO USE RUST


THE THIRD RULE OF RUST
STOP CODING IN RUST TO GO OUT
AND TELL EVERYONE TO USE RUST

Experience

- 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

I’m not true programmer



What is Rust?

What is Rust?

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

— Wikipedia

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)

What is Rust?

"Empowering everyone to build reliable and efficient software".

— www.rust-lang.org

What is Rust?

Quick facts about Rust

Digital Security

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

What is Rust?

Why Rust?

Why Rust?

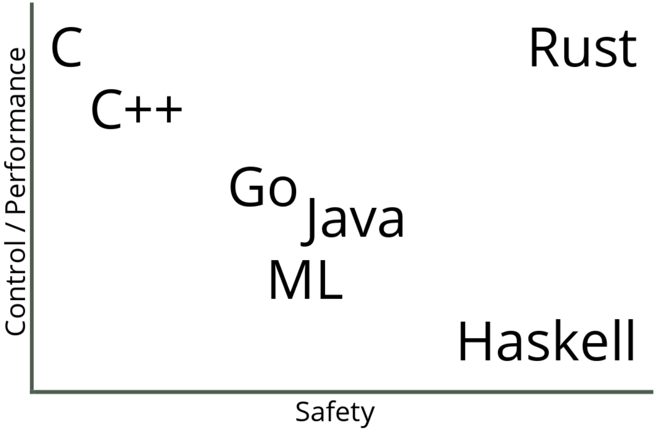
- Performance
  - Fast, memory-efficient
  - No runtime or garbage collector
  - Zero-cost abstractions
- Reliability
  - Rich type system
  - Ownership model
- Productivity
  - Documentation
  - Friendly compiler
  - Top-notch tooling

(Un)safe

(Un)safe

Control vs Safety

Control vs Safety



(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

Rust?

(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)

(Un)safe

Ownership and Borrowing

Ownership and Borrowing

Nicholas Matsakis

Ownership

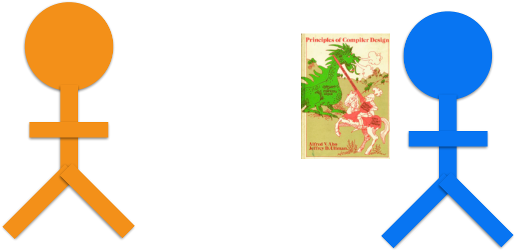
n. The act, state, or right of possessing something.

Borrow

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


Principles of Compiler Design
Alfred V. Aho
Jeffrey D. Ullman

Ownership


Principles of Compiler Design
Alfred V. Aho
Jeffrey D. Ullman

Ownership

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

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

Ownership

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

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

Ownership

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

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

Take ownership of a String

Ownership

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

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

Ownership

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

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

Ownership

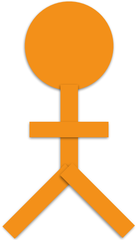
```
fn main() {
  let name = format!("…");
  helper(name);
  helper(name);
}
```

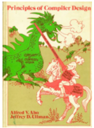
```
fn helper(name: String) {
  println!(..);
}
```

Ownership

```
fn main() {
  let name = format!(“…”);
  helper(name);
  helper(name);
}
```

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

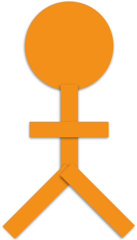




Ownership

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

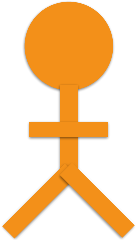
```
fn helper(name: String) {
  println!(..);
}
```



Ownership

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

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

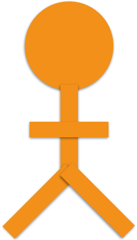


Ownership

```
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);
}
```

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

“Ownership” in Java

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

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

“Ownership” in Java

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

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

Take **reference** to Vector

"Ownership" in Java

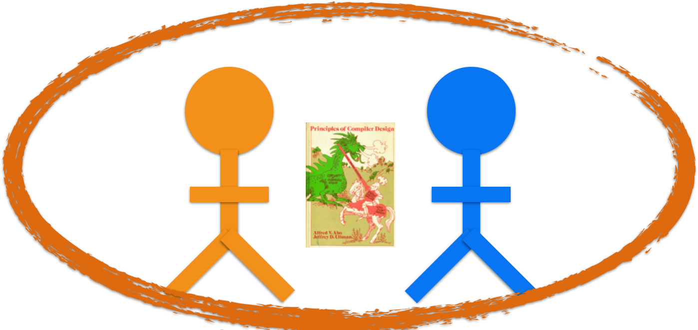
```
void main() {
  Vector name = …;
  helper(name);
  helper(name);
}
```

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

“Ownership” in Java

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

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



“Ownership” in Java

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

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

“Ownership” in Java

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

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

“Ownership” in Java

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

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

“Ownership” in Java

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

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

“Ownership” in Java

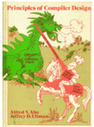
```
void main() {
  Vector name = …;
  helper(name);
  helper(name);
}
```

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

“Ownership” in Java

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

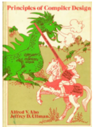
```
void helper(Vector name) {
  …
}
```



“Ownership” in Java

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

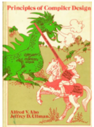
```
void helper(Vector name) {
  new Thread(…);
}
```



“Ownership” in Java

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

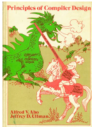
```
void helper(Vector name) {
  new Thread(…);
}
```



“Ownership” in Java

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

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



“Ownership” in Java

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

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

“Ownership” in Java

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

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

“Ownership” in Java

Clone

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

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

Clone

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

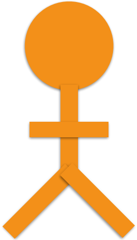
```
fn helper(name: String) {
  println!(..);
}
```

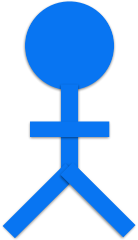
Clone

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

Copy the String

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





Clone

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

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

Clone

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

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

Clone

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

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

Clone

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

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

Clone

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

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

Copy (auto-Clone)

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

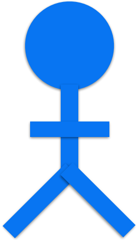
```
fn helper(count: i32) {
  println!(..);
}
```

Copy (auto-Clone)

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

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

i32 is a Copy type

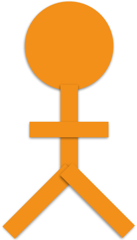


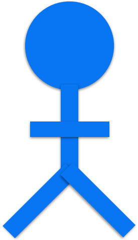
Copy (auto-Clone)

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

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

i32 is a Copy type





Copy (auto-Clone)

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

```rust
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

Copy (auto-Clone)

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

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

i32 is a Copy type

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

Example: *money*

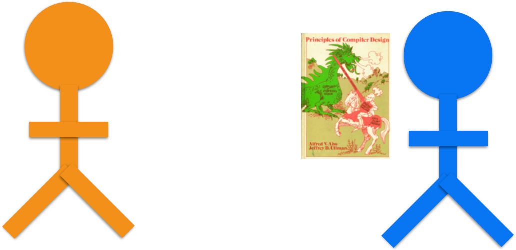
Clone: Run custom code to make a copy.

Example: *strings*

Copy: Type is implicitly copied when referenced.

Example: *integers or floating-point numbers*

Borrowing: Shared Borrows

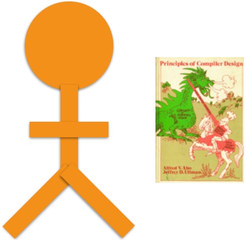


Borrowing: Shared Borrows

Borrowing: Shared Borrows

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

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

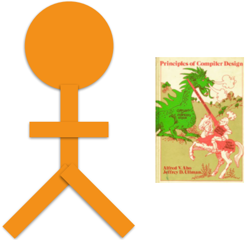


Shared borrow

```
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



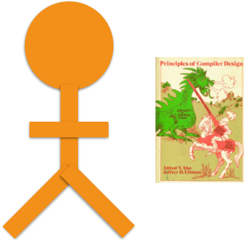
Shared borrow

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

Lend the string,
creating a reference

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

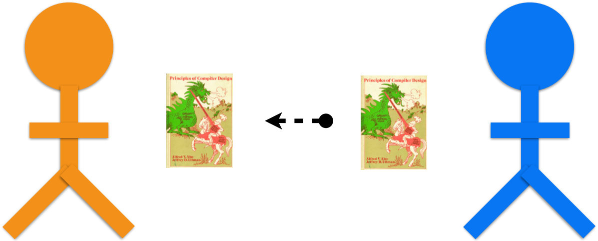
Change type to a
reference to a String



Shared borrow

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

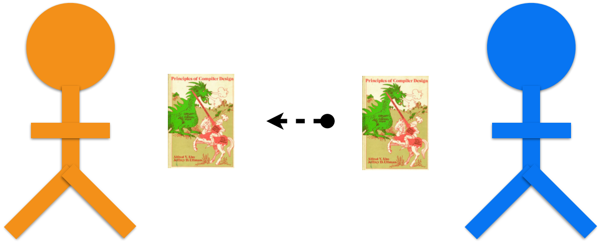
```
fn helper(name: &String) {
  println!(..);
}
```



Shared borrow

```
fn main() {
  let name = format!(“…”);
  let reference = &name;
  helper(reference);
  helper(reference);
}
```

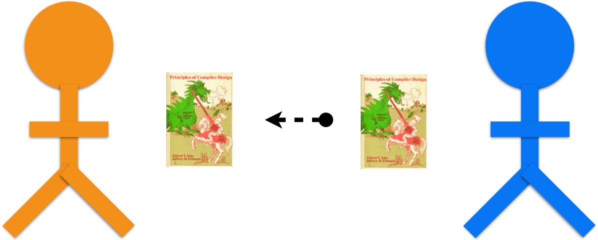
```
fn helper(name: &String) {
  println!(..);
}
```



Shared borrow

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

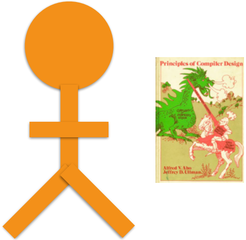
```rust
fn helper(name: &String) {
  println!(..);
}
```



Shared borrow

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

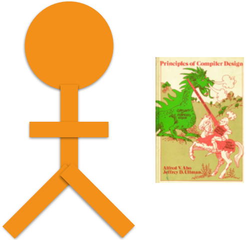
```rust
fn helper(name: &String) {
  println!(..);
}
```



Shared borrow

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

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



Shared borrow

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

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

Shared borrow

Shared == Immutable

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

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

Shared == Immutable

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

OK. Just reads.

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

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");
    ^~~~
```

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**.

Play time

Waterloo, Cassius Coolidge, c. 1906

http://is.gd/no0tTH

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

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

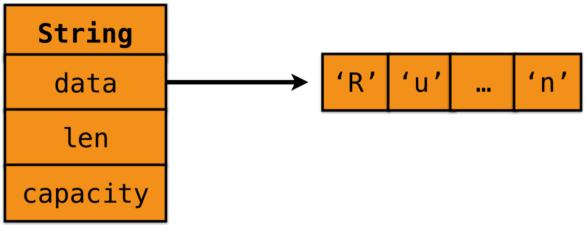
Looks like other languages:

- 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!(..);
}
```



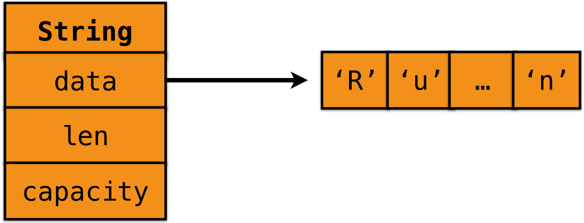
Looks like other languages:

- Python: name[1:]
- Ruby: name[1..-1]

But no copying at runtime.

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

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



Looks like other languages:

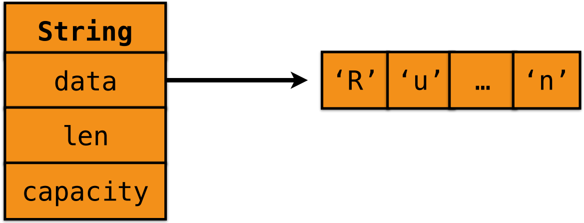
- Python: name[1:]
- Ruby: name[1..-1]

But no copying at runtime.

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

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

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



Looks like other languages:

- Python: name[1:]
- Ruby: name[1..-1]

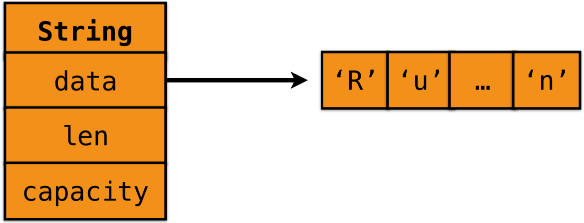
But no copying at runtime.

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

Lend some of the string

```rust
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]

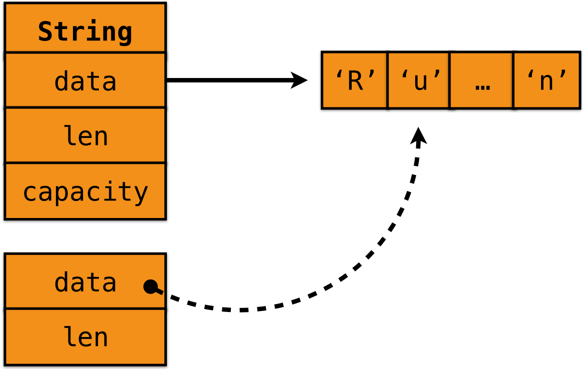
But no copying at runtime.

```
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`



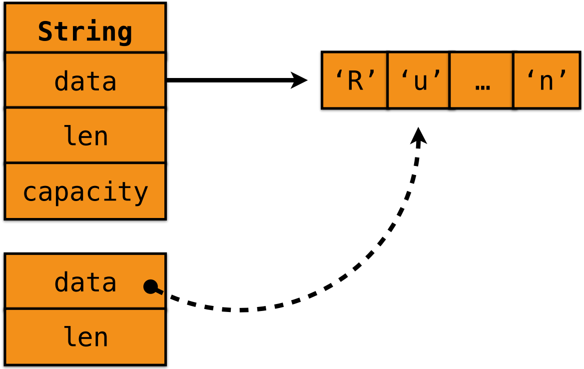
Looks like other languages:

- Python: name[1:]
- Ruby: name[1..-1]

But no copying at runtime.

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

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



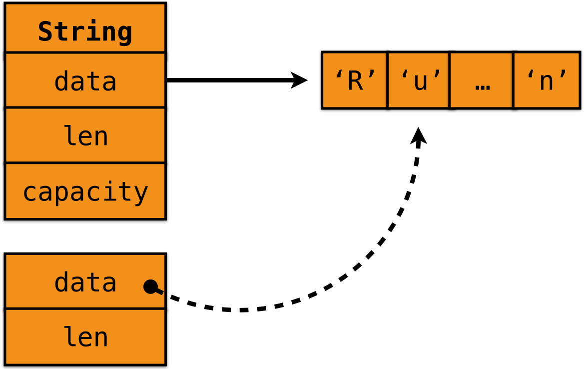
Looks like other languages:

- Python: name[1:]
- Ruby: name[1..-1]

But no copying at runtime.

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

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



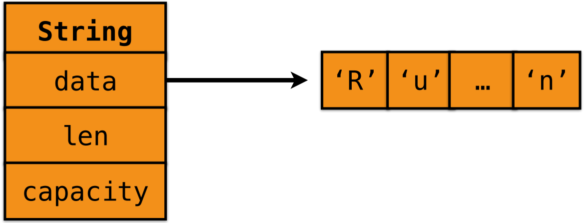
Looks like other languages:

- Python: name[1:]
- Ruby: name[1..-1]

But no copying at runtime.

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

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



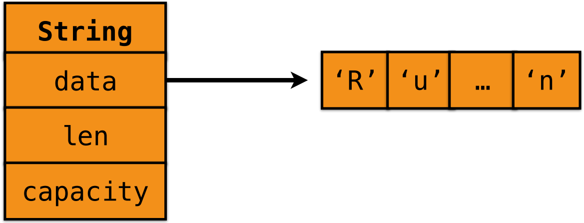
Looks like other languages:

- Python: name[1:]
- Ruby: name[1..-1]

But no copying at runtime.

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

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



Looks like other languages:

- Python: name[1:]
- Ruby: name[1..-1]

But no copying at runtime.

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();
}
```

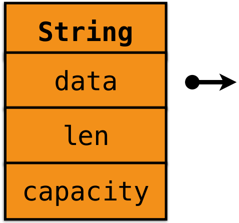
Iterator over slices borrowed from **line**.

No copying, no allocations.

High-level code, low-level efficiency

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

Iterator over slices borrowed from **line**.



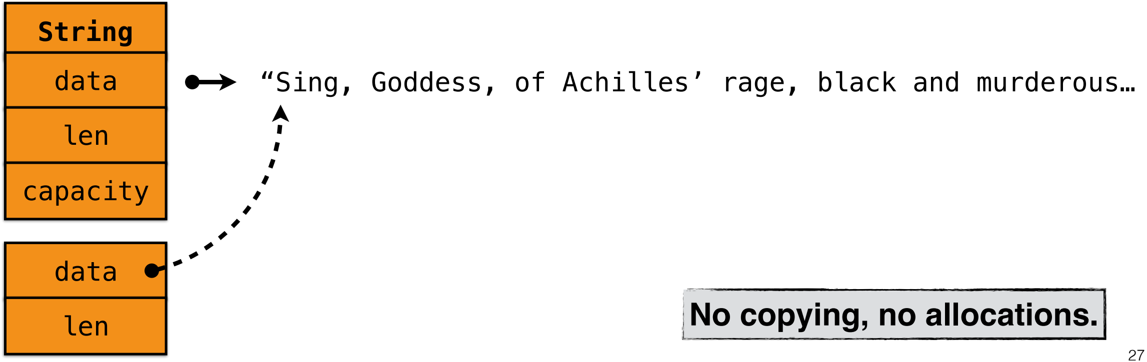
“Sing, Goddess, of Achilles’ rage, black and murderous…

No copying, no allocations.

High-level code, low-level efficiency

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

Iterator over slices borrowed from **line**.

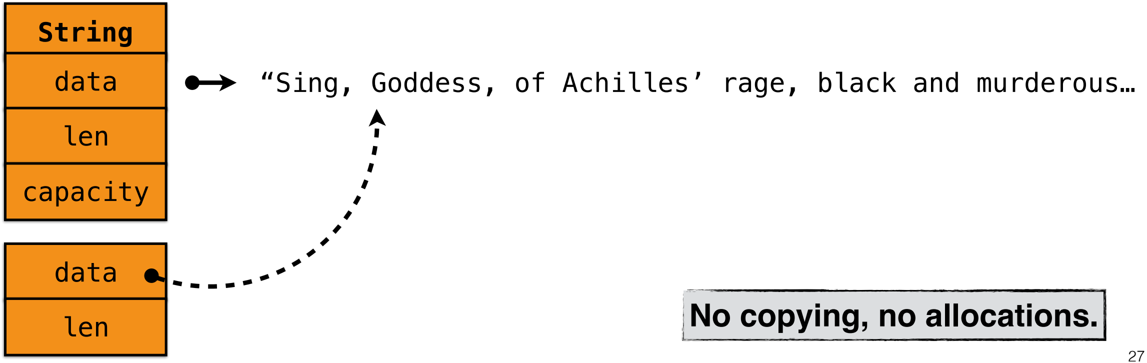


No copying, no allocations.

High-level code, low-level efficiency

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

Iterator over slices borrowed from **line**.

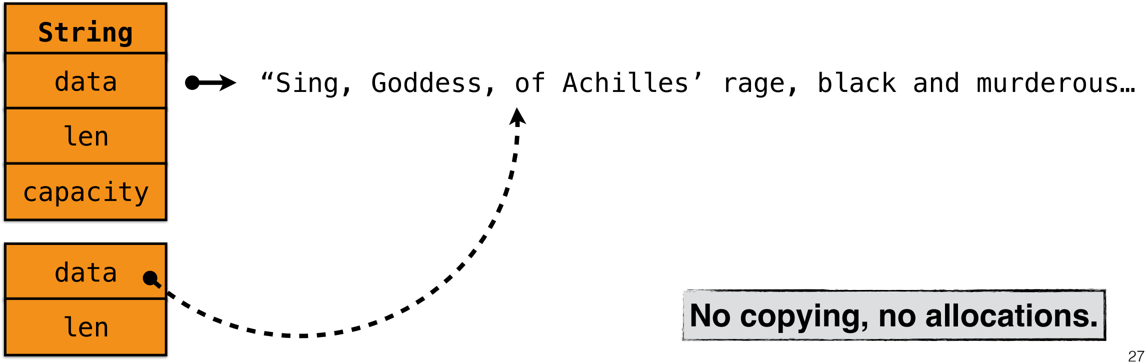


No copying, no allocations.

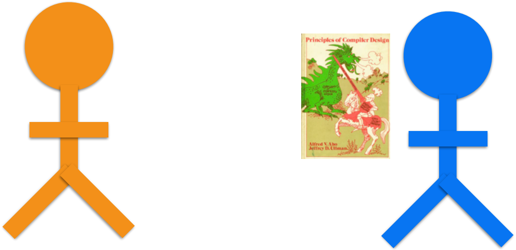
High-level code, low-level efficiency

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

Iterator over slices
borrowed from **line**.



Borrowing: Mutable Borrows

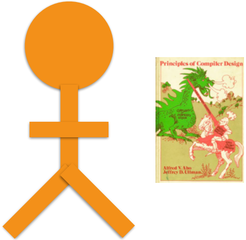


Borrowing: Mutable Borrows

Borrowing: Mutable Borrows

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

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



Mutable borrow

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

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

Take a **mutable reference** to a String

Mutable borrow

```
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

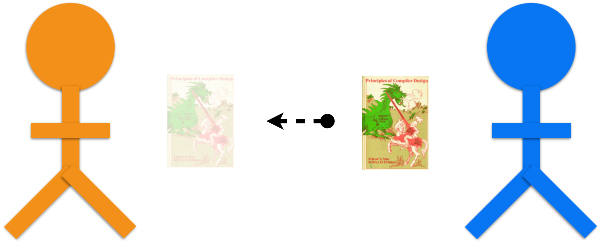
Mutable borrow

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

Lend the string **mutably**

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

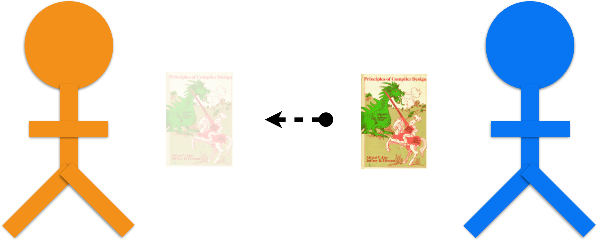
Take a **mutable reference** to a String



Mutable borrow

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

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

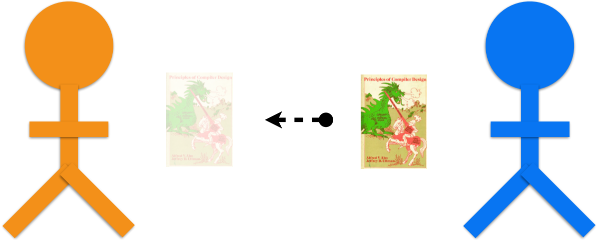


Mutable borrow

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

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

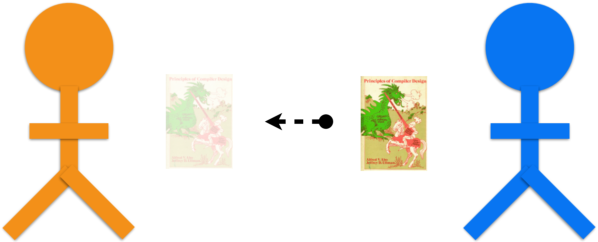
Mutate string
in place



Mutable borrow

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

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



Mutable borrow

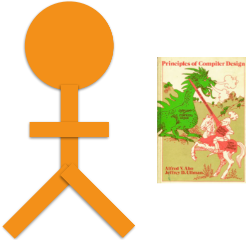
```
fn main() {
  let mut name = …;
  update(&mut name);
  println!("{}", name);
}
```

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

Mutable borrow

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

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

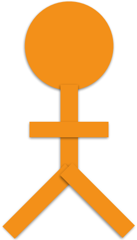


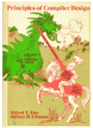
Mutable borrow

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

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

Prints the updated string.

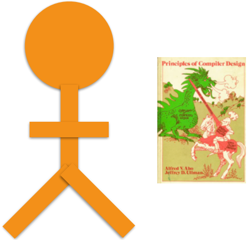




Mutable borrow

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

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



Mutable borrow

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

```rust
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

`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

(Un)safe

How do we get safety?

How do we get safety?

https://www.flickr.com/photos/langtind/2217639550/in/photostream/

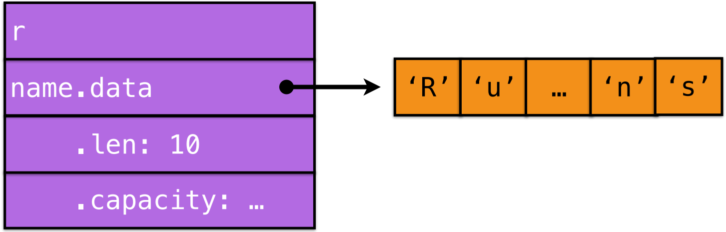
```
fn main() {
  let r;
  {
    let name = format!("…");
    r = &name;
  }
  println!("{}", r);
}
```

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

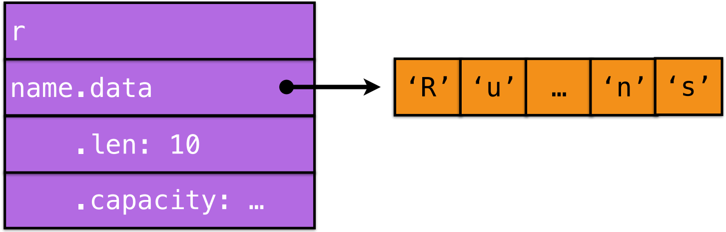
```rust
fn main() {
  let r;
  {
    let name = format!("…");
    r = &name;
  }
  println!("{}", r);
}
```

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

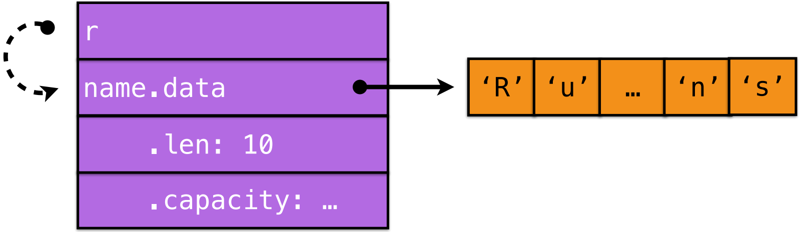
```rust
fn main() {
  let r;
  {
    let name = format!("…");
    r = &name;
  }
  println!("{}", r);
}
```



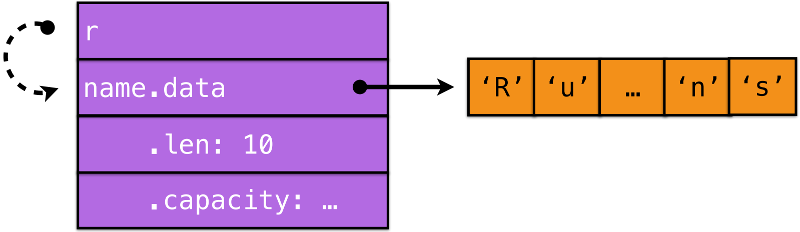
```
fn main() {
  let r;
  {
    let name = format!("…");
    r = &name;
  }
  println!("{}", r);
}
```



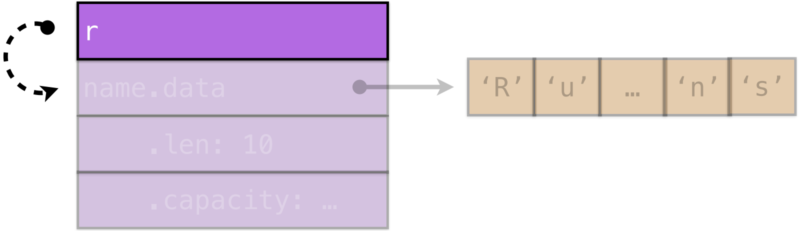
```rust
fn main() {
  let r;
  {
    let name = format!("…");
    r = &name;
  }
  println!("{}", r);
}
```



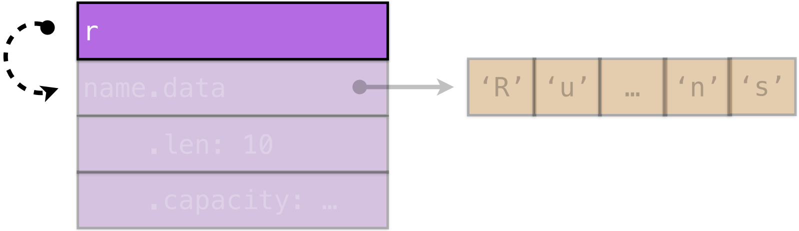
```rust
fn main() {
  let r;
  {
    let name = format!("…");
    r = &name;
  }
  println!("{}", r);
}
```



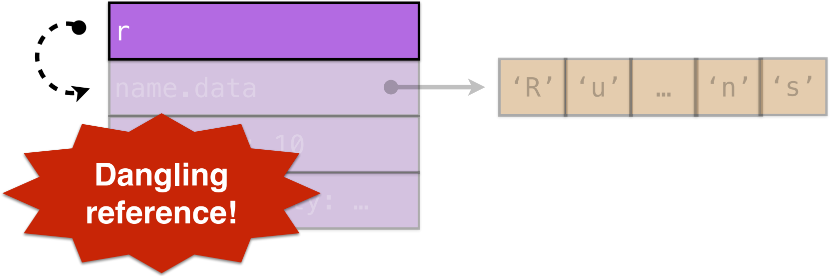
```
fn main() {
  let r;
  {
    let name = format!("…");
    r = &name;
  }
  println!("{}", r);
}
```



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



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



```
fn main() {
  let r;
  {
    let name = format!(“…”);
    r = &name;
  }
  println!(“{}”, r);
}
```

http://is.gd/HJyO7A

```
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);
}
```

Lifetime: span of code where reference is used.

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

Lifetime: span of code where reference is used.

```
fn main() {                                          'l
  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);
}
```

'l

Lifetime: span of code where reference is used.

compared against

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

‘s

```
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() {                                      'l
    let r;
    {
's      let name = format!("…");
        r = &name;
    }
    println!("{}", r);
}
```

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;
            ^~~~
```

```rust
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

```
use std::thread;

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

`name` can only be used within this fn

Might escape the function!

```rust
use std::thread;

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

`name` can only be used within this fn

Might escape the function!

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

```rust
use std::thread;

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

`name` can only be used within this fn

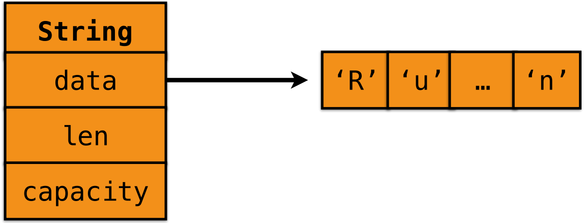
```
error: the type `[…]` does not fulfill the required lifetime
    thread::spawn(move || {
    ^~~~~~~~~~~~~
note: type must outlive the static lifetime
```

Dangers of mutation

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

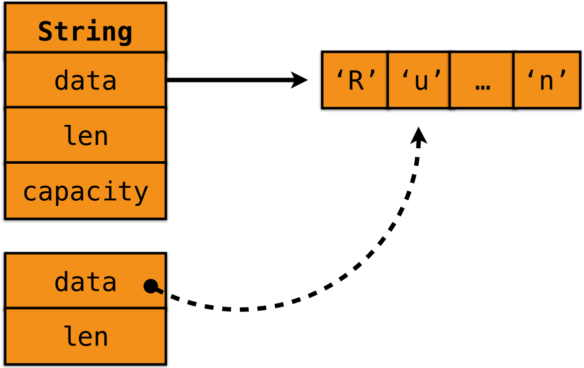
Dangers of mutation

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



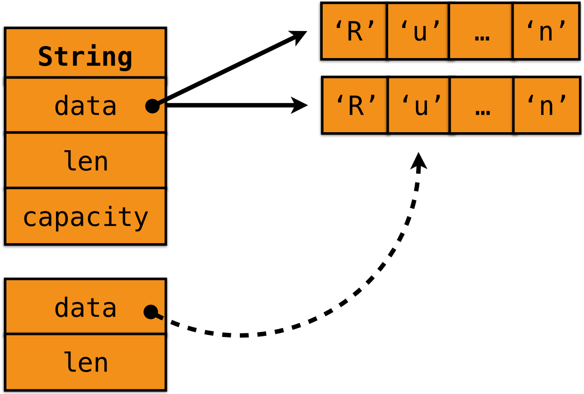
Dangers of mutation

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



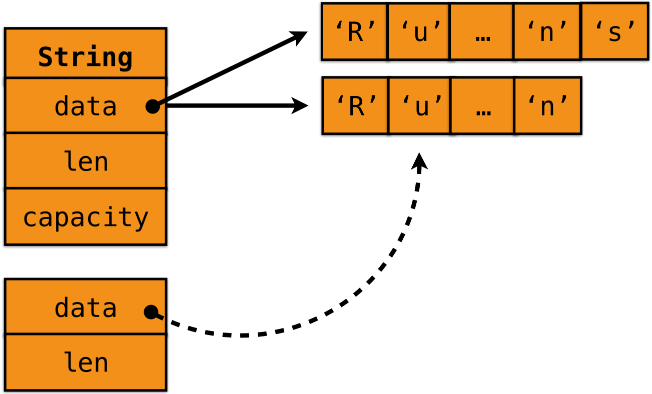
Dangers of mutation

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



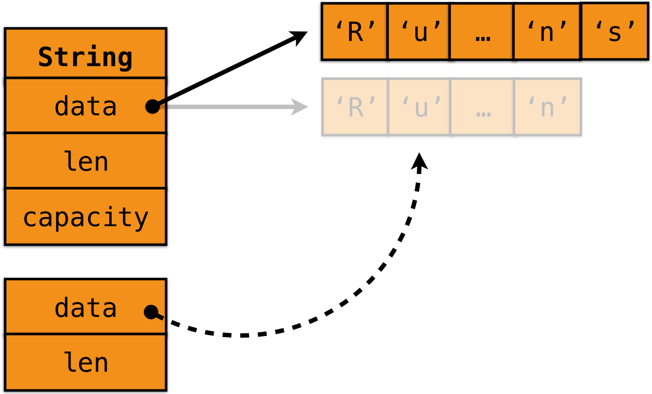
Dangers of mutation

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



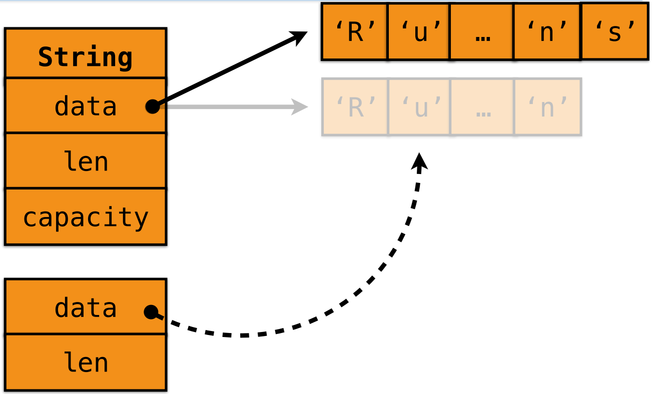
Dangers of mutation

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



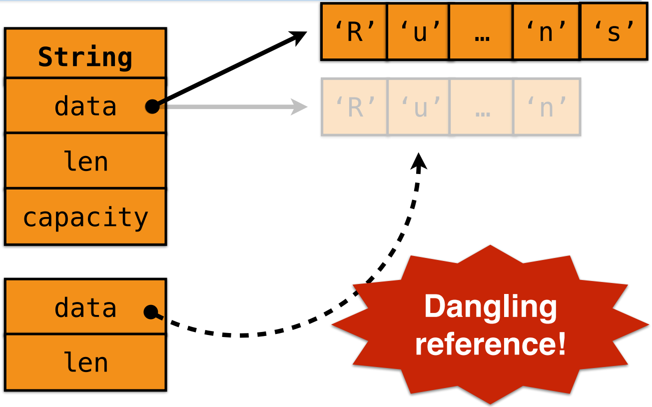
Dangers of mutation

```
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.

Dangers of mutation

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

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

http://is.gd/MCPVWg

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

http://is.gd/MCPVWg

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");
    ^~~~~~
```

```
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");
}
```

```
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

```
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

OK: \`buffer\` is not borrowed here

(Un)safe

Comparison

NULL dereferences

C

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

Rust

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

NULL dereferences (continue)

Functional Rust

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

Rust References

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

Use after free (C)

C

```c
uint8_t* pointer = (uint8_t*) malloc(SIZE);
// ...
if (err) {
  abort = 1;
  free(pointer);
}
// ...
if (abort) {
  logError("operation aborted", pointer);
}
```

Use after free (Rust)

Rust

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

Dangling pointers (C)

C

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

Dangling pointers (Rust)

Rust

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

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
```

Out of bounds (C)

C

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

Digital Security

Out of bounds (Rust)

Rust

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

(Un)safe

Concurrency

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

Concurrent quicksort

Rust

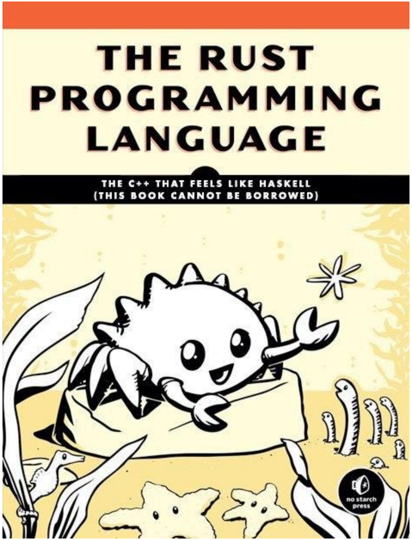
```rust
fn qsort(vec: &mut [i32]) {
  if vec.len() <= 1 { return; }
  let pivot = vec[random(vec.len())];
  let mid = vec.partition(vec, pivot);
  let (less, greater) = vec.split_at_mut(mid);

  rayon::join(|| qsort(less),
              || qsort(greater));

}
```

Syntax

Syntax


THE RUST PROGRAMMING LANGUAGE
THE C++ THAT FEELS LIKE HASKELL
(THIS BOOK CANNOT BE BORROWED)
no starch press

Syntax

Concepts

Concepts

```rust
//! # Main
//! Module docs

/// Docs
// Comments
fn main() {
    let x = 31337;
    println!("The value of x is: {}", x); // 31337
    let mut y: u8 = 5;
    y = x as u8;
    println!("The value of y is: {}", y); // 105
}
```

Concepts

```rust
fn nsa(is_hack: bool, backdoor: &str, blue_pill: String) -> f64 {
    for c in blue_pill.chars() {
        print!("{}", c);
    }
    if is_hack {
        loop { break 3.1337; }
    } else if backdoor.len() > 3 {
        42.0 - 42.0
    } else {
        3.14
    }
}
```

Syntax

Enums (Algebraic data type)

Enums (Algebraic data type)

```
enum Pohek {
  Xss(XssType),
  SocialEngineering,
  Phishing,
  // ...
}

enum XssType {
  Reflected,
  Stored,
  // ...
}
```

```
match pohek {
  Pohek::Xss(xss_type) =>
  {
    hack_by_xss(xss_type);
  },
  Pohek::SocialEngineering |
  Pohek::Phishing =>
  {
    pa3Becmu_JIOXA();
  }
  _ => {},
}
```

Option

```rust
fn find_vulnerability(program: &Program) -> Option<Vulnerability>
↪ {...}

fn hack_program(program: &mut Program) {
    match find_vulnerability(&program) {
        Some(vuln) => {
            exploit(vuln);
        }
        None => println!("Better luck next time."),
    }
}
```

Optional (C++)

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

Optional (C++)

```
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

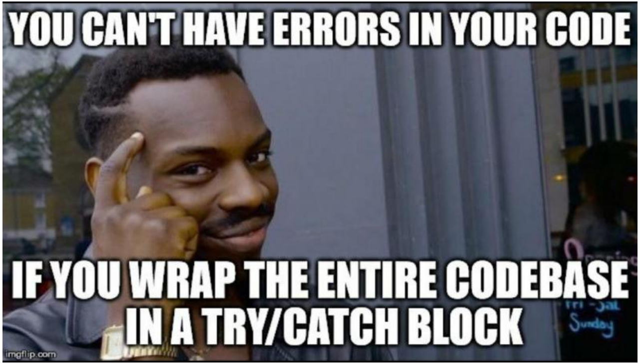
Error handling

Error handling

- Return code (C, Go)

Error handling

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



Error handling

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

Digital Security

Error handling

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

Error handling

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

Panic


slaps roof of language standard
this bad boy can fit so much
undefined behavior in itx6^{2}I2÷Pts}I▄¡O≥D@f╨k–0╖2;í"÷"æ
++

Panic

Panic

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

    v[99];
}
```

Panic

Output

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

Panic

Output

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

Result

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

Result

Result

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

match hack_program(&program) {
  Ok(shell) => connect(shell),
  Err(error) => {
    // Do something with error
  }
}
```

Result

Result

```rust
fn hack_world(world: World) -> Result<Power, u32> {
  hack_program(&program)?;

  for program in &world.programs() {
    hack_program(program).map(install_spy).map(get_money)?;
  }
}
```

Syntax

Structs

Object-oriented programming

Проблемы при использовании ООП

Object-oriented programming

Structs

```rust
struct Hacker {
  nickname: String,
  scope: Scope,
  cves: Vec<u32>,
}

enum Scope {
  Fuzzing,
  Developing,
  Exploiting,
  Reversing,
}
```

Construction

```rust
impl Hacker {
  fn new(nickname: String, scope: Scope) -> Hacker {
    Hacker {
      nickname: nickname,
      scope: scope,
      cves: Vec::new(),
    }
  }
}
```

Construction

```rust
impl Hacker {
  fn new(nickname: String, scope: Scope) -> Self {
    Hacker {
      nickname, scope,
      cves: Vec::new(),
    }
  }
}
```

Methods

```rust
impl Hacker {
  fn add_cve(&mut self, cve: u32) {
    self.cves.push(cve);
  }
  fn cves(&self) -> &Vec<u32> {
    &self.cves
  }
}
```

Syntax

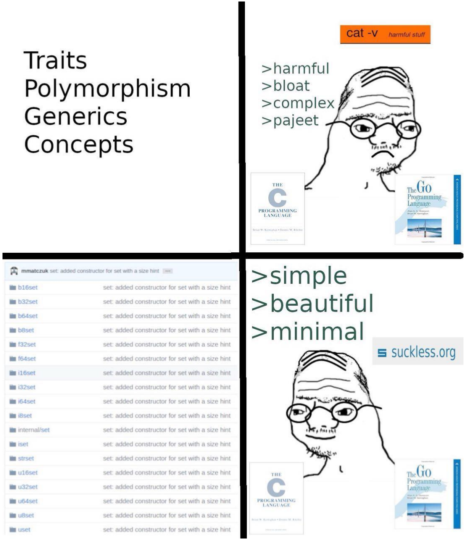
Other

Other

- Generics

Other

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

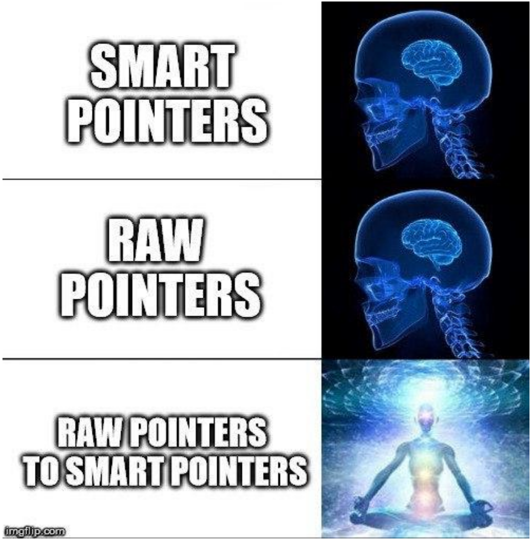


Other

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

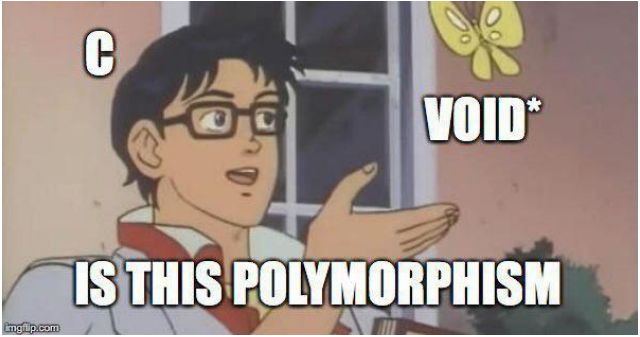
Other

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



Other

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



Ecosystem

Ecosystem

Community

Community

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

Community

- 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`)

Community

- 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

Community

- 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
- Chats (IRC, Telegram, Gitter, Matrix, Jabber)

Community

- 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
- Chats (IRC, Telegram, Gitter, Matrix, Jabber)
- Blogs (Official, Read Rust, Core Developers, many others)

Ecosystem

Cargo

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

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!

Additional tools

Sanitize code

```
$ RUSTFLAGS="-Z sanitizer=address" cargo test
$ RUSTFLAGS="-Z sanitizer=leak" cargo test
$ RUSTFLAGS="-Z sanitizer=memory" cargo test
$ RUSTFLAGS="-Z sanitizer=thread" cargo test
```

Services

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

Ecosystem

Rustup

Rustup

Rustup install

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

Rustup

Toolchain format

```
<channel>[-<date>][-<host>]

<channel>       = stable|beta|nightly|<version>
<date>          = YYYY-MM-DD
<host>          = <target-triple>
```

Install nightly toolchain

```
$ rustup toolchain install nightly
```

Rustup

Cross compile

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

Rustup

- 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
- …

Ecosystem

Rust in production

Digital Security

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

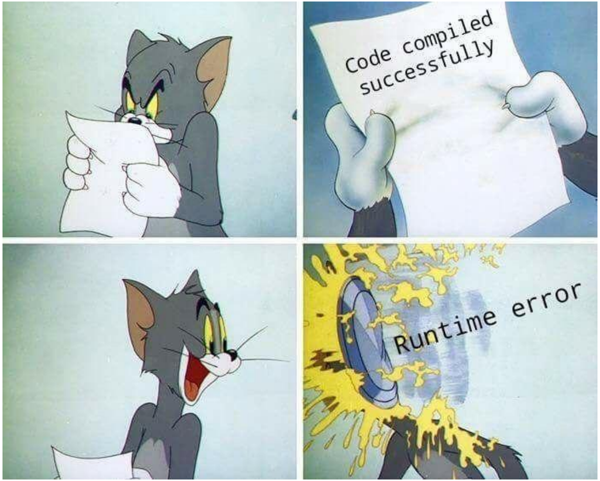
Experience and Pitfalls

Experience and Pitfalls

Compile time errors

Compile time errors

Compile time errors



Experience and Pitfalls

Compilation times

Compilation times

Experience and Pitfalls

Complex syntax

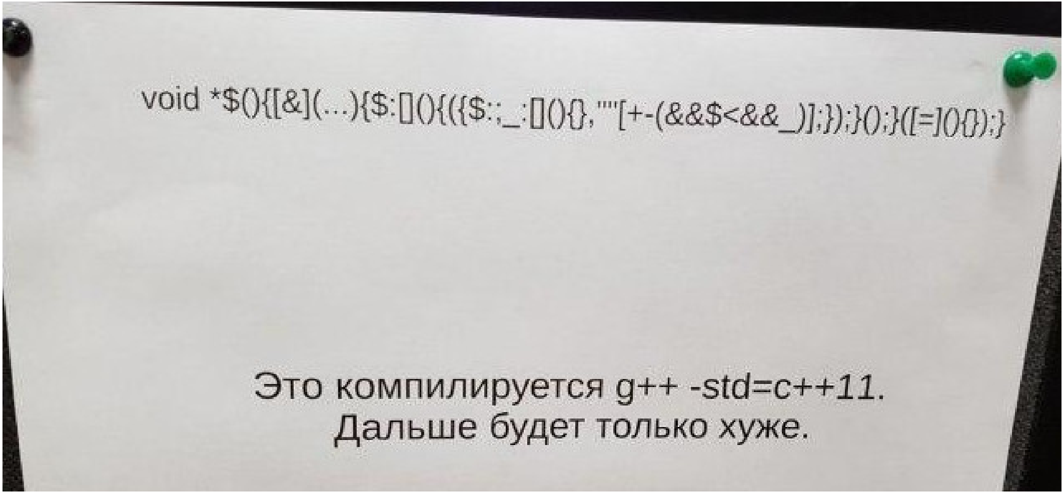
Complex syntax

Rust

```rust
impl<'a, 'gcx, 'tcx> InferCtxt<'a, 'gcx, 'tcx> {
    pub fn replace_bound_vars_with_placeholders<T>(&self,
        binder: &ty::Binder<T>
    ) -> (T, PlaceholderMap<'tcx>) where T: TypeFoldable<'tcx>
    {
        let next_universe = self.create_next_universe();

        let fld_r = |br| {
            self.tcx.mk_region(ty::RePlaceholder(ty::PlaceholderRegion {
                universe: next_universe,
                name: br,
            }))
        };
        ...
    }
}
```

Complex syntax



Experience and 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

Experience and Pitfalls

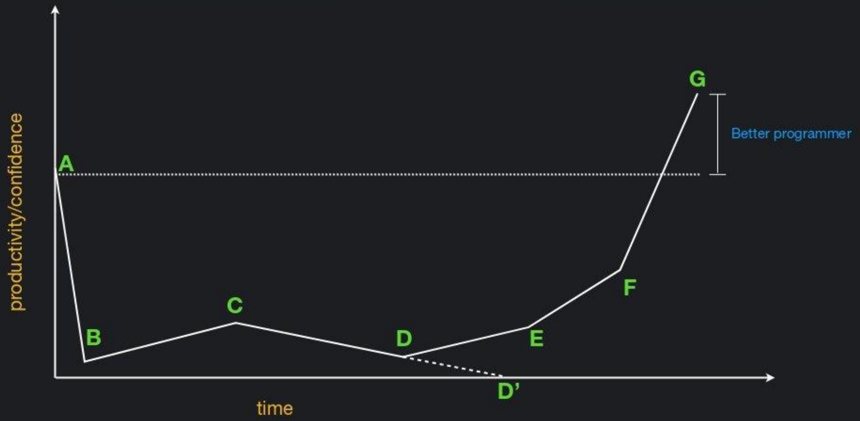
Ecosystem immaturity

Ecosystem immaturity

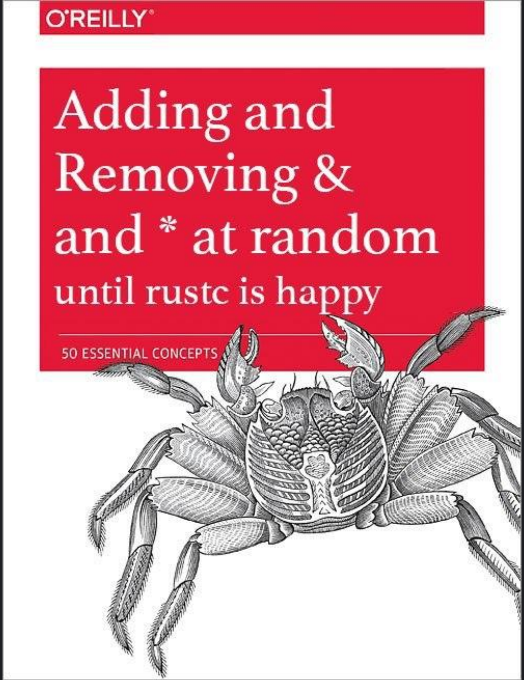
Experience and Pitfalls

Learning curve

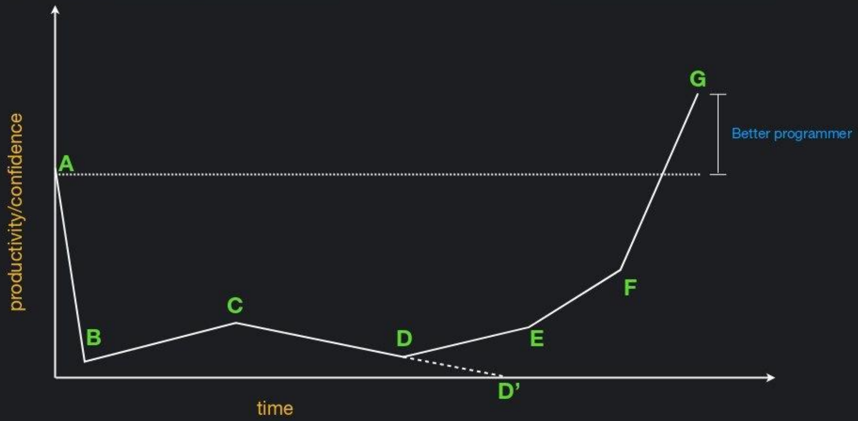
Rust: Lifecycle of Rust developer




```
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
```


O'REILLY
Adding and Removing & and * at random
until rustc is happy
50 ESSENTIAL CONCEPTS

Rust: Lifecycle of Rust developer



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

Experience and Pitfalls

Format of errors or compiler driven development

Digital Security

Format of errors or compiler driven development

Format of errors or compiler driven development

C++

```
Too big, too unclear
```

Rust

```
error: expected type, found `'static`
 --> test_err.rs:3:9
  |
3 |     Ref('static str),
  |         ^^^^^^^

error: aborting due to previous error
```

Format of errors or compiler driven development

C++

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

Format of errors or compiler driven development

Rust

```
error: expected one of `.`, `;`, `?`, or an operator, found `}`
 --> test_err.rs:6:1
  |
5 |     let y = S("xyz".to_string())
  |                                 - expected one of `.`, `;`, `?`, or an operator here
6 | }
  | ^ unexpected token

error: aborting due to previous error
```

Experience and Pitfalls

Zero-cost abstractions

Digital Security

Zero-cost abstractions

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

Zero-cost abstractions

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

Experience and Pitfalls

Undefined behaviour

Undefined behaviour

C
```c
memset(c, 0, sizeof(Net_Crypto));
free(c);
```

Summary

Digital Security

Summary

- Fearlessness

Still have logical problems or wrong architecture, but

- 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.

Summary

- Fearlessness
- Universality

Rust is reasonably good at pretty much everything.

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

Digital Security

Summary

- 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
- …

Summary

- 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.

Summary

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

Rust community is just the best.

Summary

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

- rustup
- cargo
- xargo
- rls
- racer
- rustfmt
- …

Summary

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

Questions?