



Rust and what's this thing for



Abc Xyz
@dura_lex

1. Foreword

2. What is Rust?

3. (Un)safe

4. Syntax

5. Ecosystem

6. Pitfalls

7. Experience

8. Summary

Foreword







- 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





What is Rust?

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

— Wikipedia

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

— www.rust-lang.org (2015)

“Empowering everyone to build reliable and efficient software”.

— www.rust-lang.org

What is 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

What is Rust?

Why Rust?

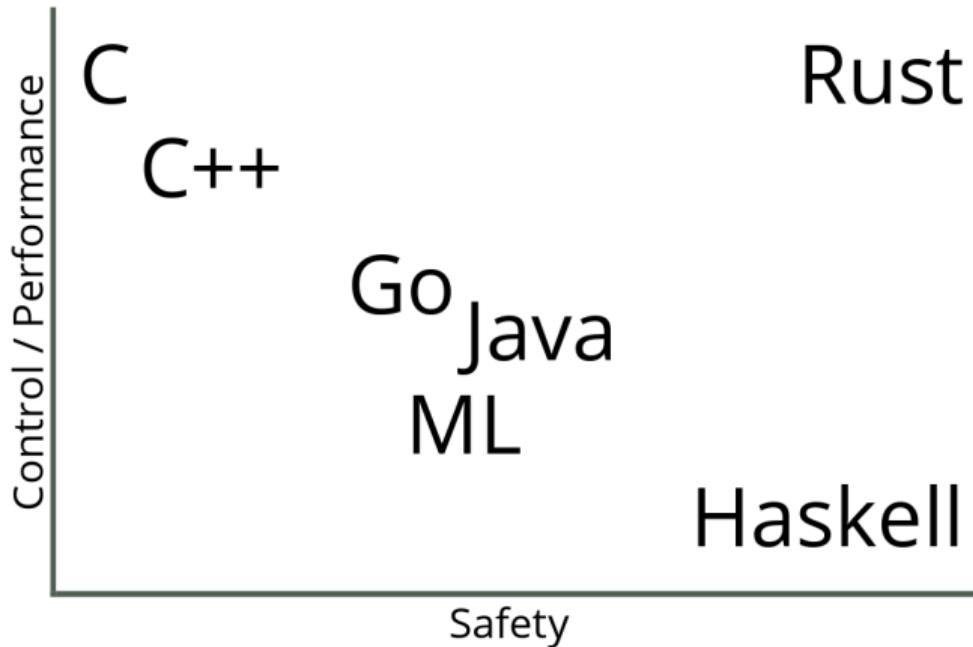


- Performance
 - Fast, memory-efficient
 - No runtime or garbage collector
 - Zero-cost abstractions
- Reliability
 - Rich type system
 - Ownership model
- Productivity
 - Documentation
 - Friendly compiler
 - Top-notch tooling

(Un)safe

(Un)safe

Control vs Safety



(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

(Un)safe

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

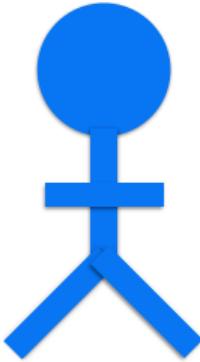
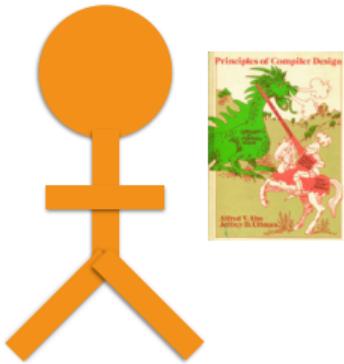
Nicholas Matsakis

Ownership

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

Borrow

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



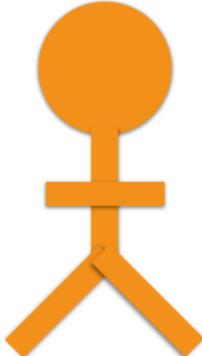
Ownership



Ownership

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

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

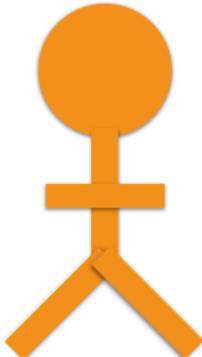


Ownership

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



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



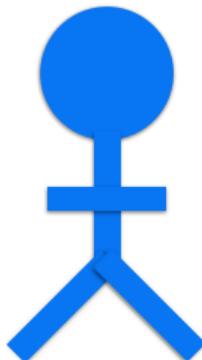
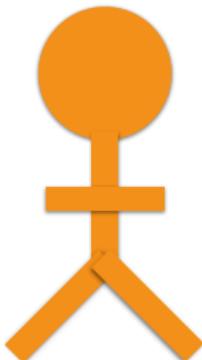
Ownership

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



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

Take ownership
of a String



Ownership

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

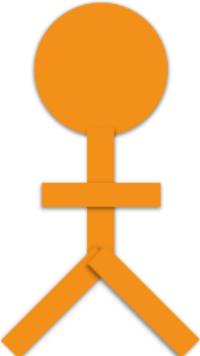


```
fn helper(name: String) {  
    println!(...);  
}
```



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!(...);  
}  
 
```



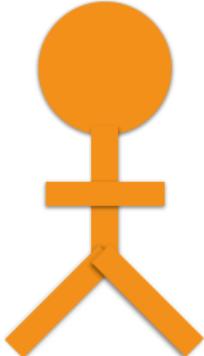
Ownership

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



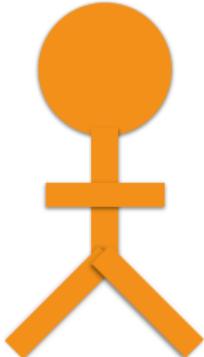
Ownership

```
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!(...);  
}
```



Ownership

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

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

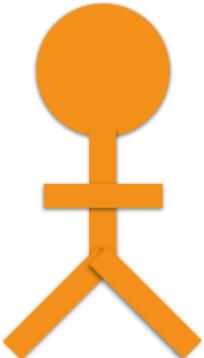
Error: use of moved value: `name`



Ownership

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

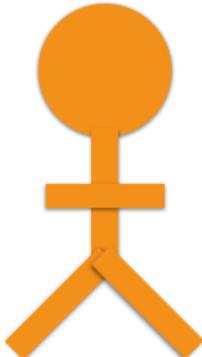
```
void helper(Vector name) {  
    ...  
}
```



“Ownership” in Java

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

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



“Ownership” in Java

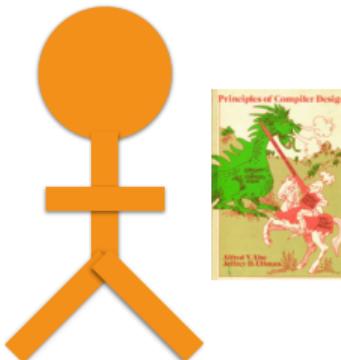
```
void main() {  
    Vector name = ...;  
    → helper(name);  
    helper(name);  
}
```

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

```
    ...  
}
```



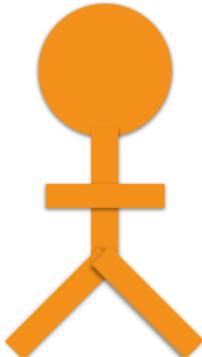
Take reference
to Vector



“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);  
}
```

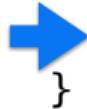


```
void helper(Vector name) {  
    ...  
}
```

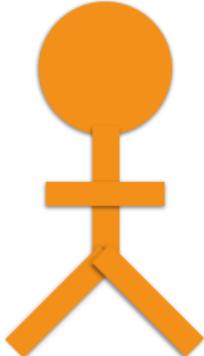


“Ownership” in Java

```
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

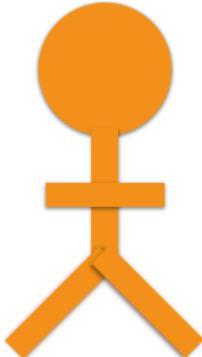
```
void main() {  
    Vector name = ...;  
    helper(name);  
    helper(name);  
}
```

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

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!(...);  
}
```

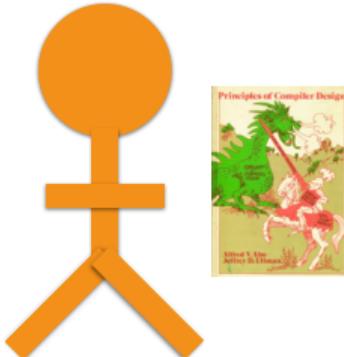


Clone

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

Copy the String

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



Clone

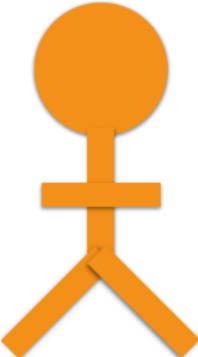
```
fn main() {  
    let name = format!("...");  
    → helper(name.clone());  
    helper(name);  
}
```

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



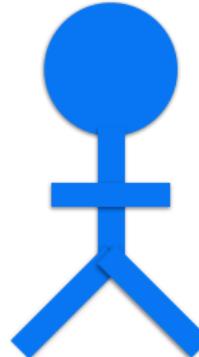
Clone

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



Clone

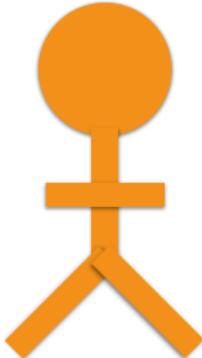
```
fn main() {  
    let name = format!("...");  
    helper(name.clone());  
    helper(name);  
}  
  
fn helper(name: String) {  
    println!(...);  
}
```



Clone

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

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

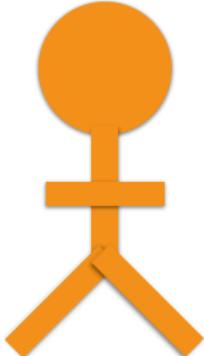


Clone

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



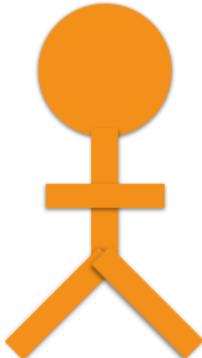
```
fn helper(name: String) {  
    println!(...);  
}
```



Copy (auto-Clone)

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

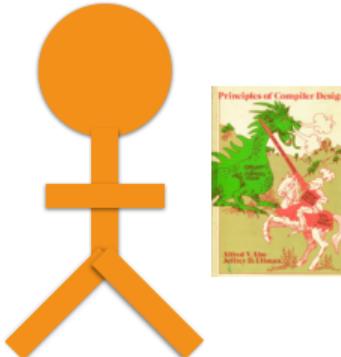
```
fn helper(count: i32) {  
    println!(..);  
}
```



Copy (auto-Clone)

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

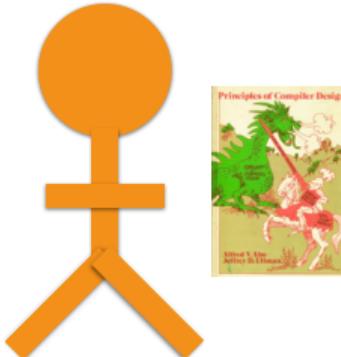
```
fn helper(count: i32) {  
    println!(..);  
}  
i32 is a Copy type
```



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)

```
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
```



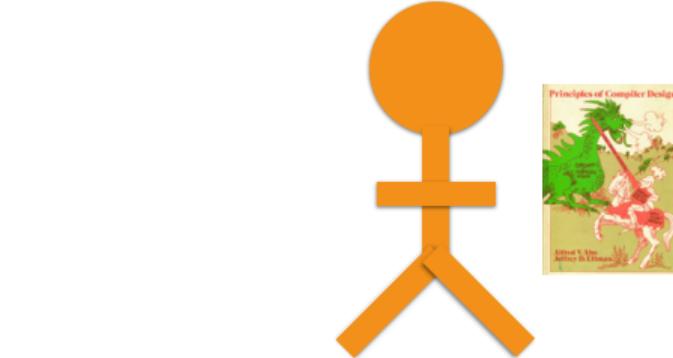
Copy (auto-Clone)

```
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



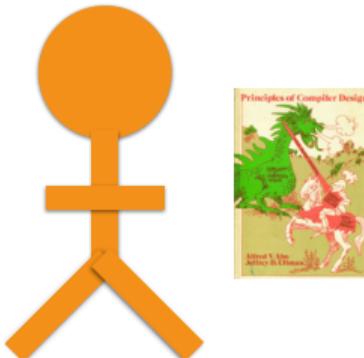
Shared borrow

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

Lend the string,
creating a reference

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

Change type to a
reference to a String



Shared borrow

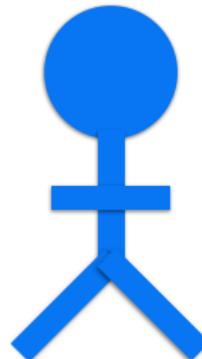
```
fn main() {  
    let name = format!("...");  
    let reference = &name;  
    ➔ helper(reference);  
    helper(reference);  
}
```

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



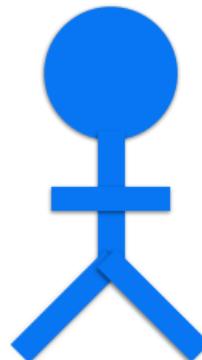
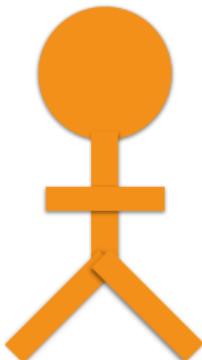
Shared borrow

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



Shared borrow

```
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

```
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

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



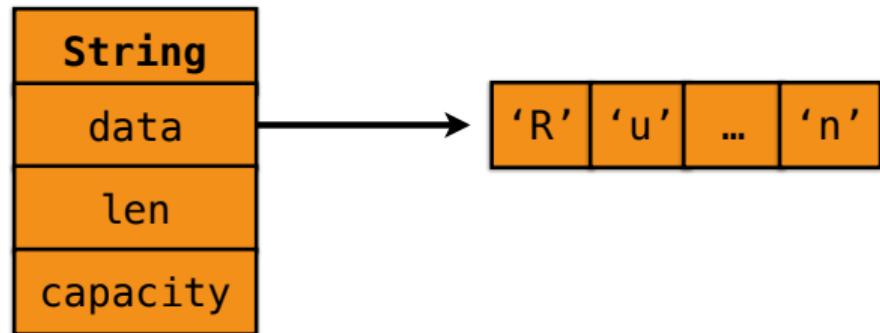
Waterloo, Cassius Coolidge, c. 1906

```
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!(...);  
}
```

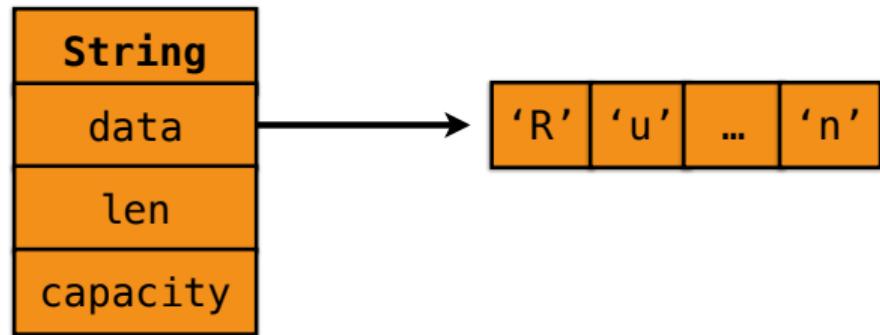


Looks like other languages:
• Python: `name[1:]`
• Ruby: `name[1..-1]`
But no copying at runtime.

```
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.

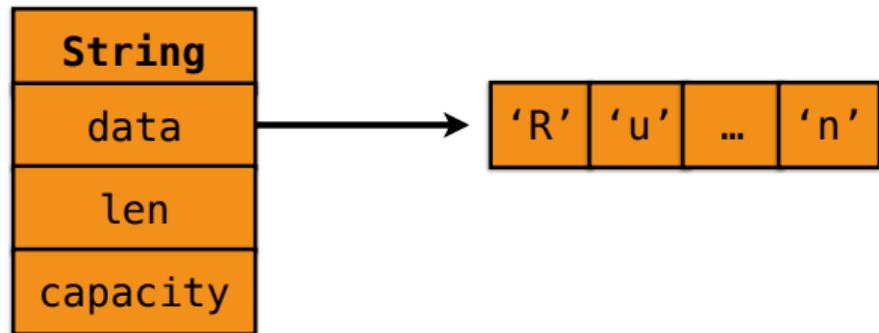


```
fn main() {  
    let name = format!("...");  
    helper(&name[1..]);  
    helper(&name);  
}
```

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

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.

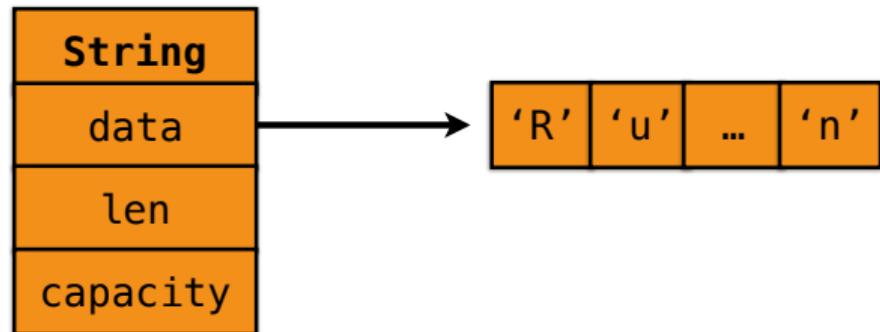


```
fn main() {  
    let name = format!("...");  
    helper(&name[1..]);  
    helper(&name);  
}
```

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.

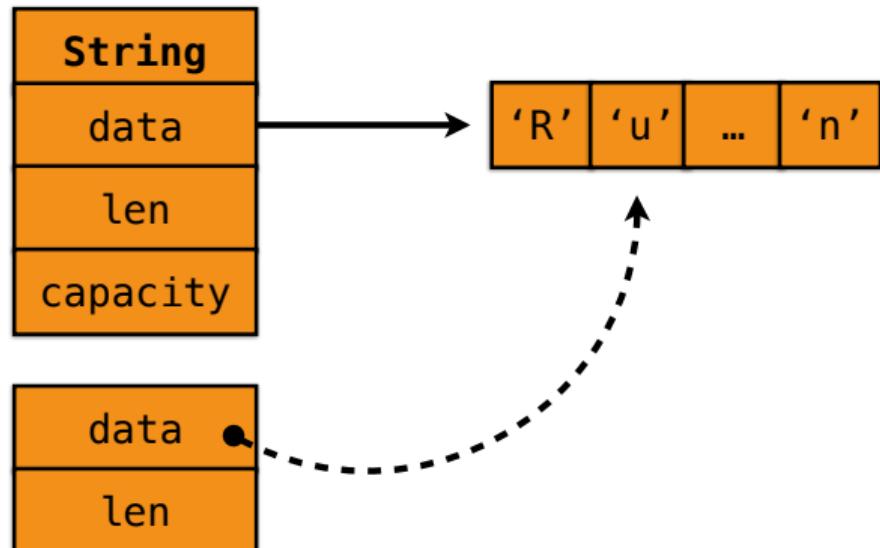
```
fn main() {  
    let name = format!("...");  
    helper(&name[1..]);  
    helper(&name);  
}
```

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.

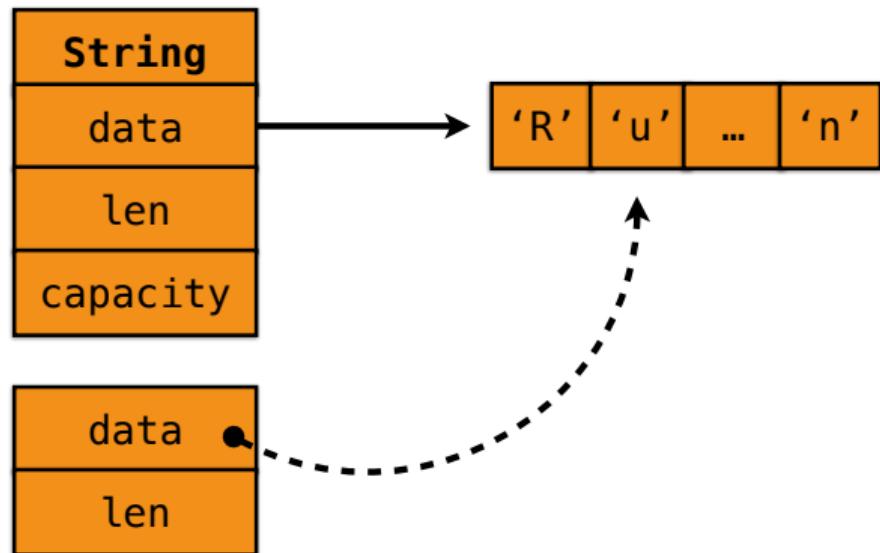


```
fn main() {  
    let name = format!("...");  
    helper(&name[1..]);  
    helper(&name);  
}
```



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

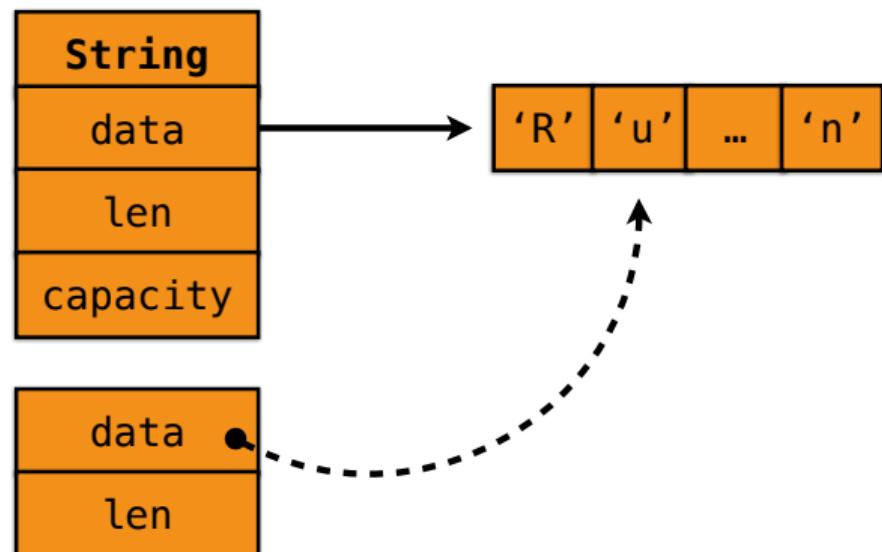
Looks like other languages:
• Python: name[1:]
• Ruby: name[1..-1]
But no copying at runtime.



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

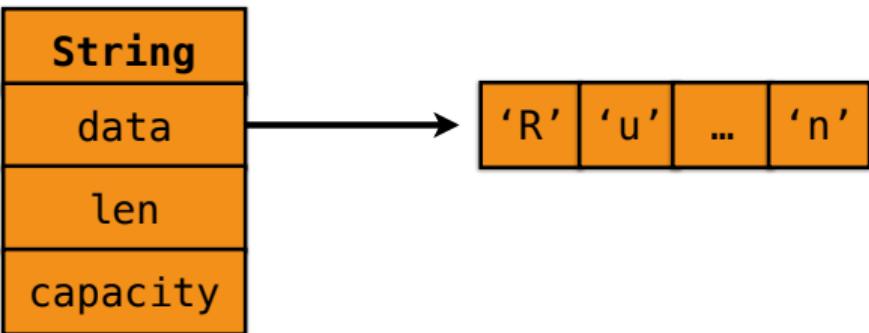
```
fn helper(name: &str) {  
    println!(...);  
}
```

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



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

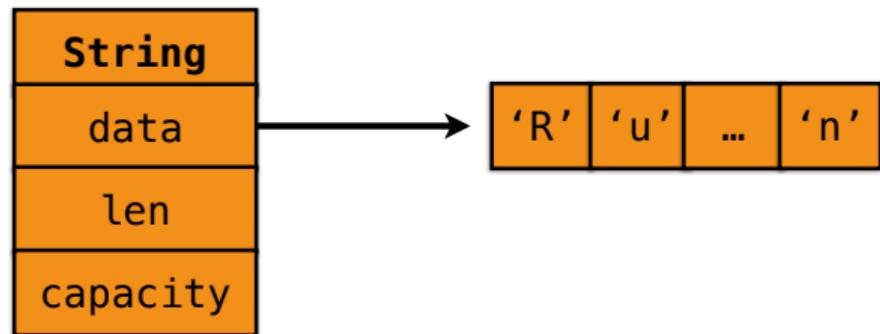
```
fn helper(name: &str) {  
    println!(...);  
}
```



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.

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`.

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`.

String
data
len
capacity

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

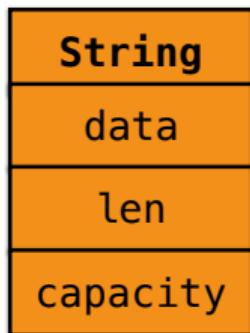
data
len

No copying, no allocations.

High-level code, low-level efficiency

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

Iterator over slices
borrowed from `line`.



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

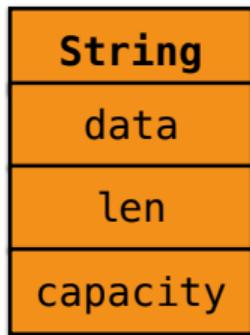


No copying, no allocations.

High-level code, low-level efficiency

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

Iterator over slices
borrowed from `line`.



→ "Sing, Goddess, of Achilles' rage, black and murderous..."



No copying, no allocations.



Borrowing: Mutable Borrows



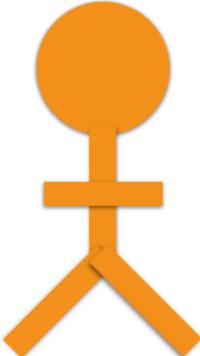
Borrowing: Mutable Borrows



Borrowing: Mutable Borrows

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

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



Mutable borrow

```
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



Mutable borrow

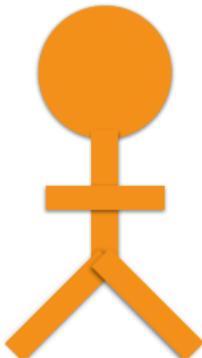
```
fn main() {  
    let mut name = ...;  
    update(&mut name);  
    println!("{}", name);  
}
```



Lend the string
mutably

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

Take a **mutable**
reference to a String



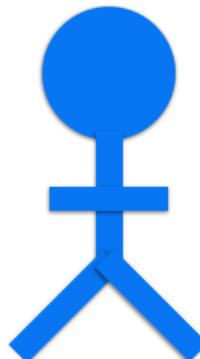
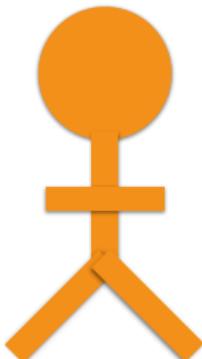
Mutable borrow

```
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("...");  
}
```

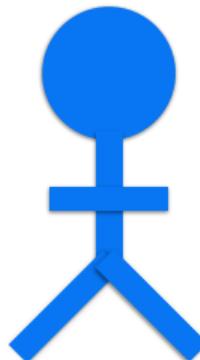
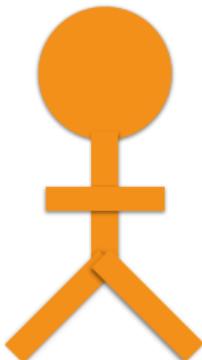


Mutable borrow

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

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

Mutate string
in place



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);  
}
```

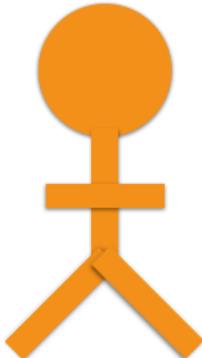
```
fn update(name: &mut String) {  
    name.push_str("...");  
}
```



Mutable borrow

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

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



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

```
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

`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?



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

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

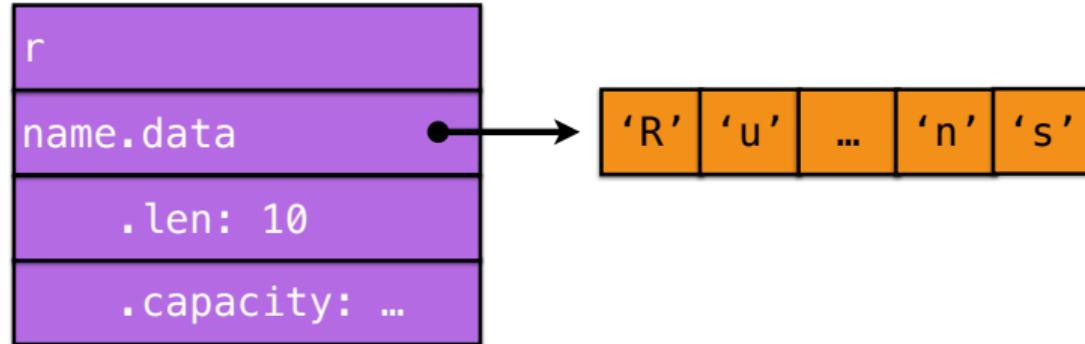
```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```

r

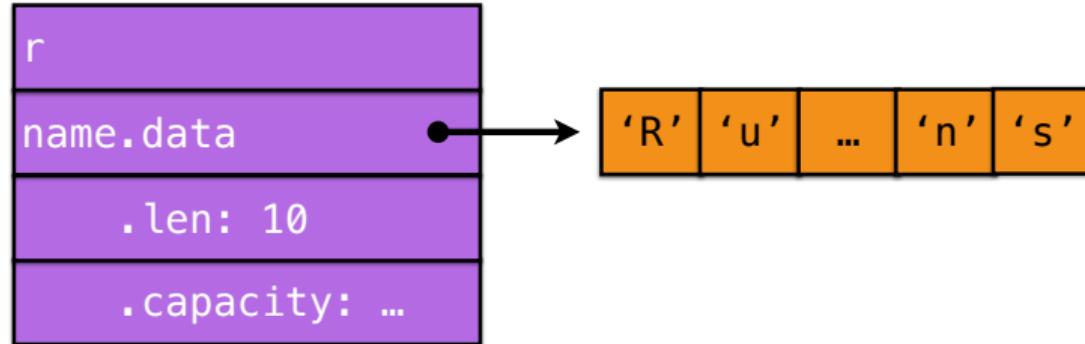
```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```

r

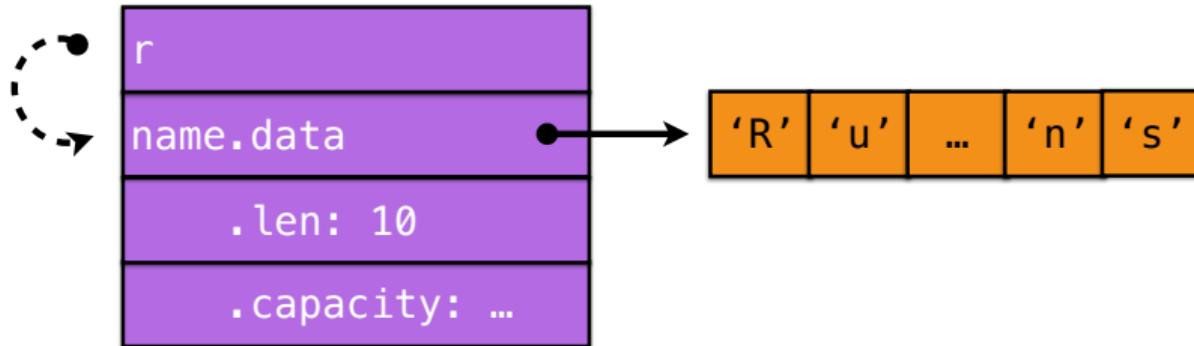
```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```



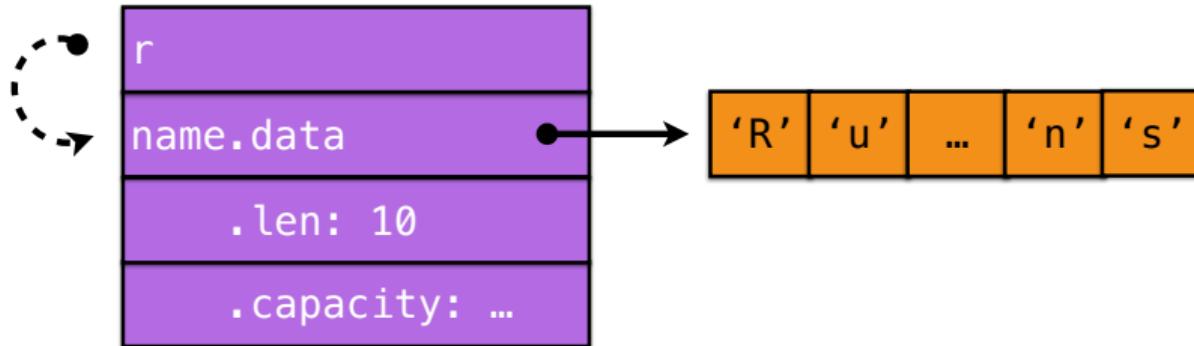
```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```



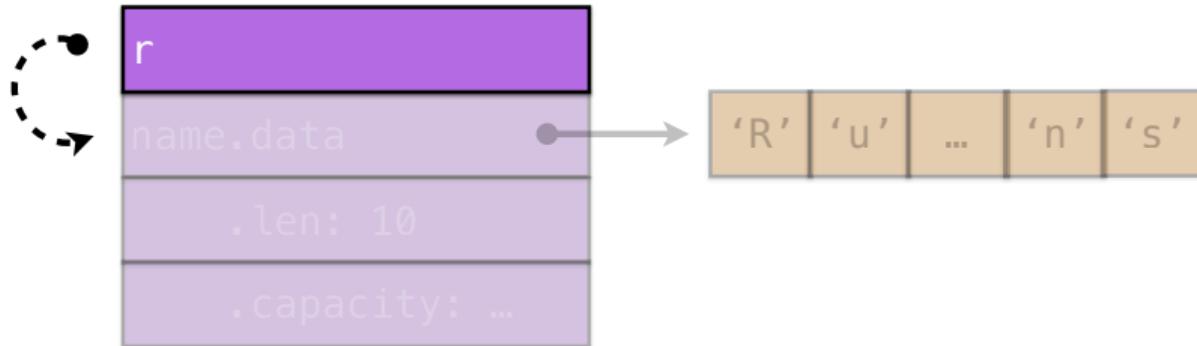
```
fn main() {  
    let r;  
    {  
        let name = format!("...");  
        r = &name;  
    }  
    println!("{}", r);  
}
```



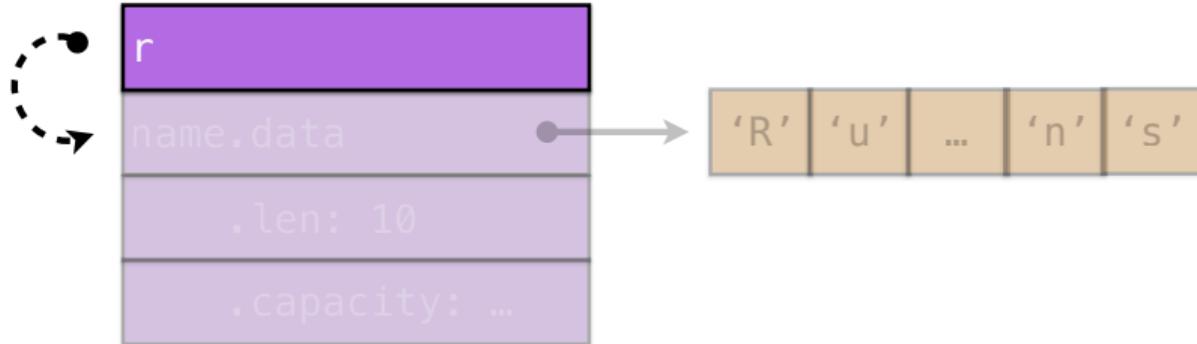
```
fn main() {
    let r;
    {
        let name = format!("...");
        r = &name;
    }
    println!("{}{}", r);
}
```



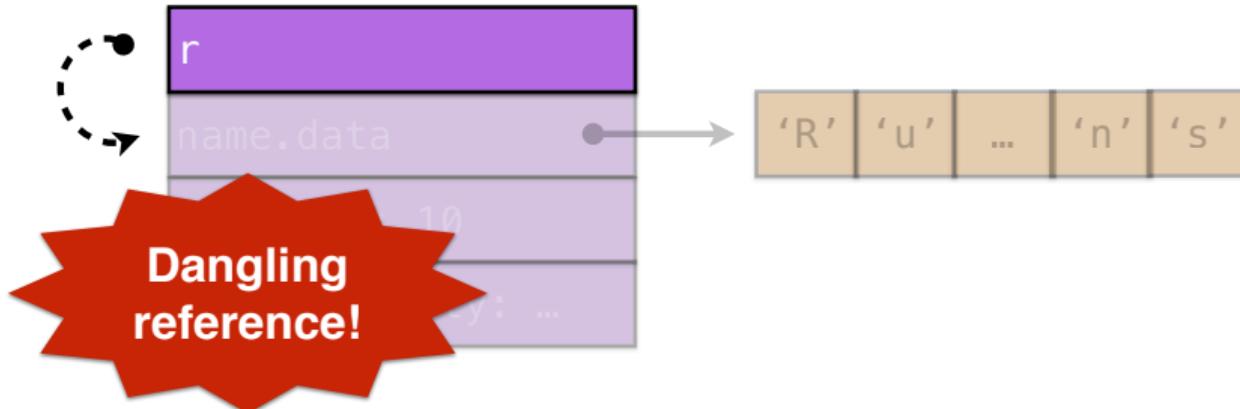
```
fn main() {
    let r;
    {
        let name = format!("...");
        r = &name;
    }
    println!("{}{}", r);
}
```



```
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);
}
```

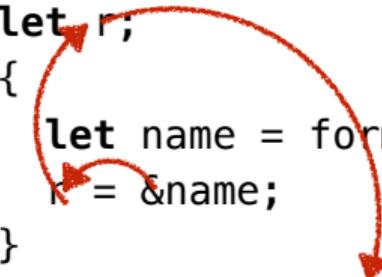
```
fn main() {
    let r;
    {
        let name = format!("...");
        r = &name;
    }
    println!("{}", r);
}
```

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() {  
    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.

compared against

Scope of data being borrowed (here, `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;  
    {  
        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;  
     ^~~~
```

```
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!

```
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
```

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

name` can only be used within this fn

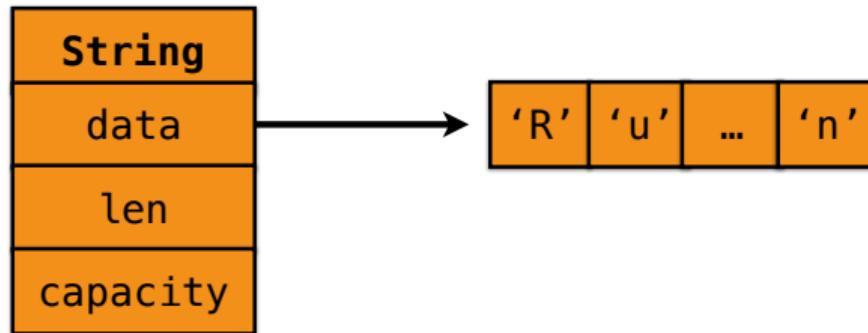
```
error: the type ` [...]` does not fulfill the required lifetime  
    thread::spawn(move || {  
        ^~~~~~  
note: type must outlive the static lifetime
```

Dangers of mutation

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

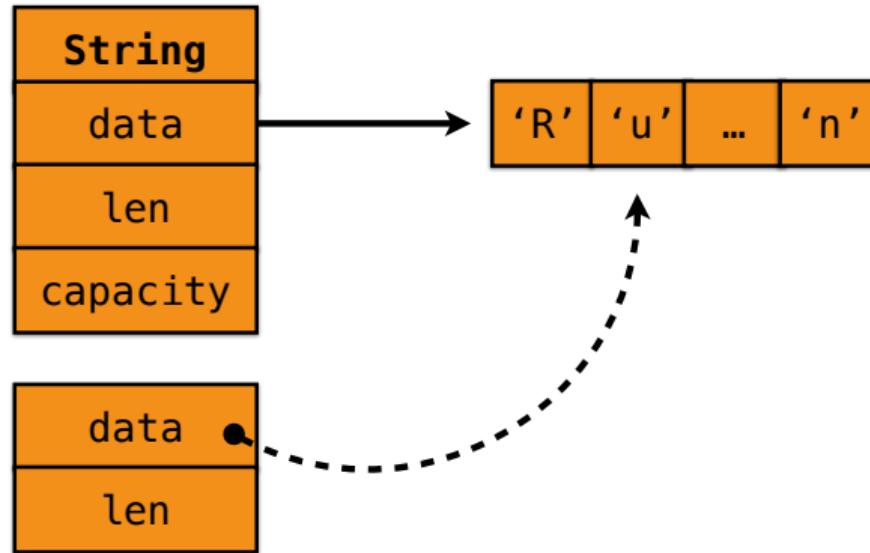
Dangers of mutation

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



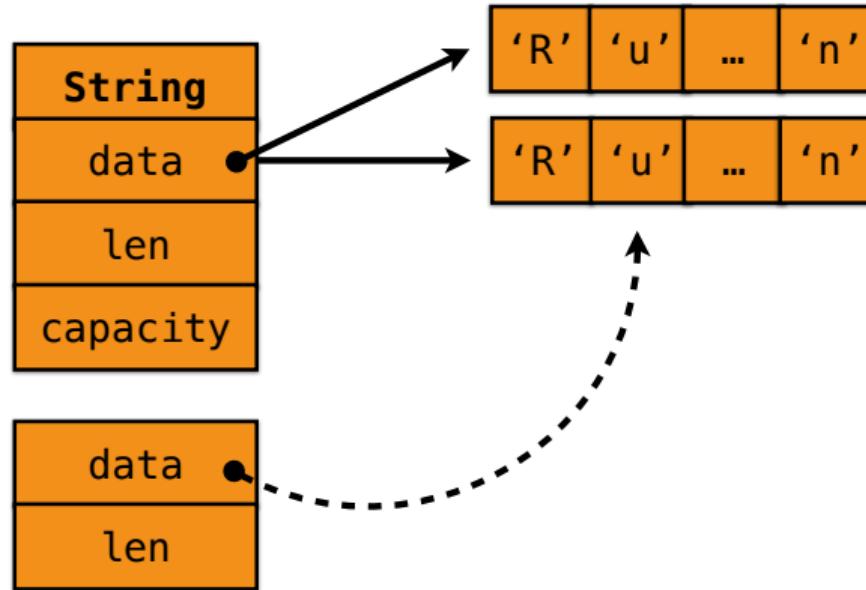
Dangers of mutation

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



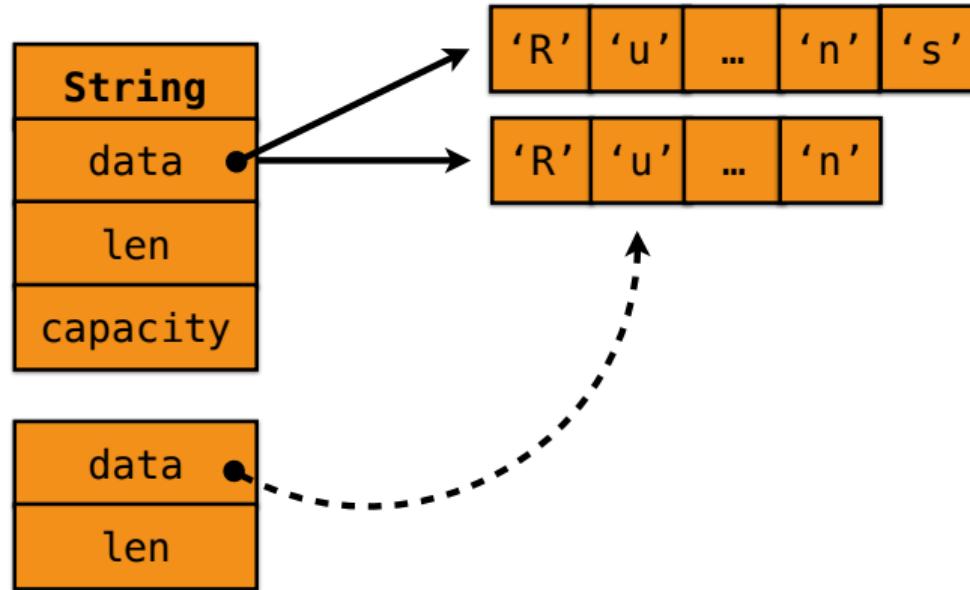
Dangers of mutation

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



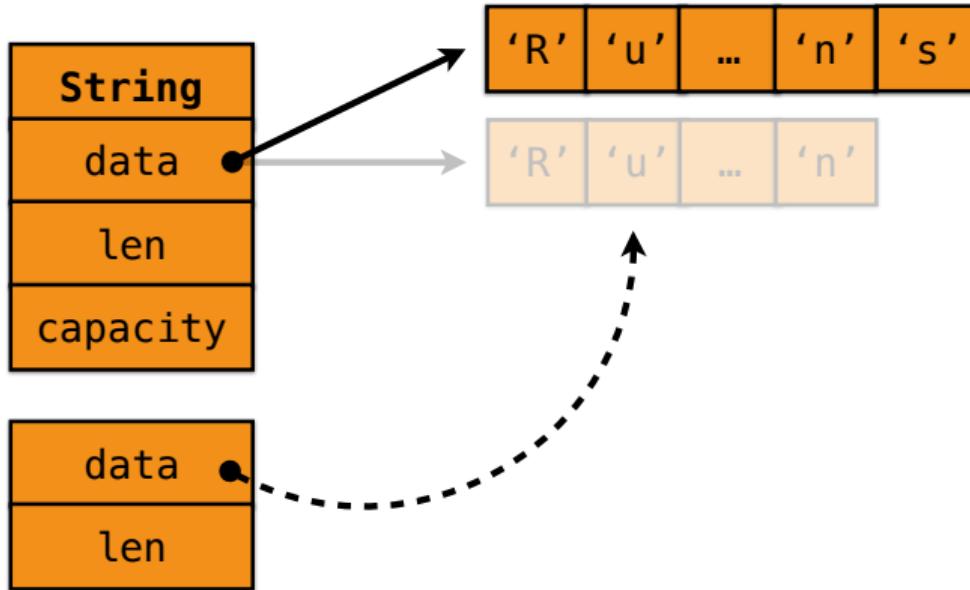
Dangers of mutation

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



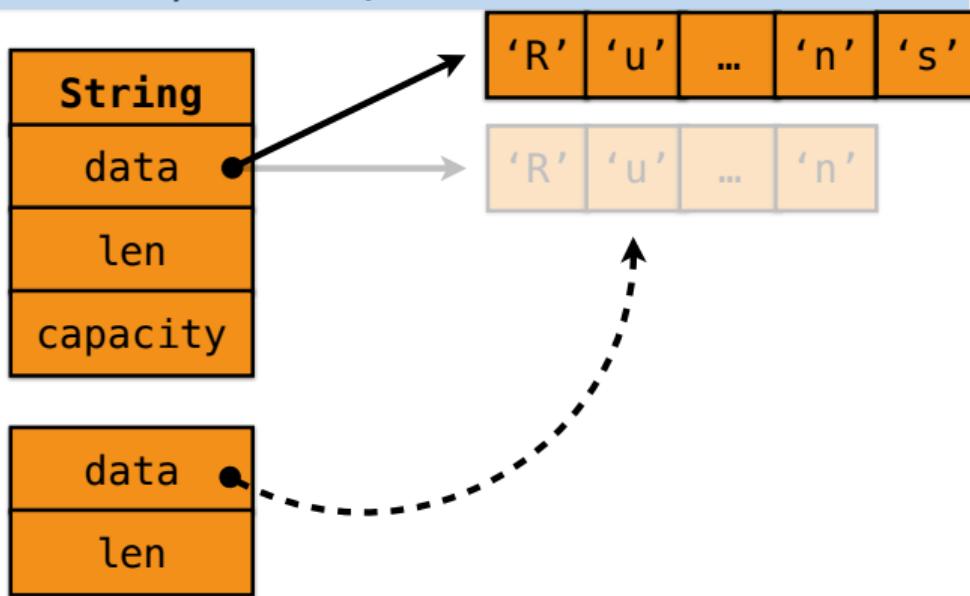
Dangers of mutation

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



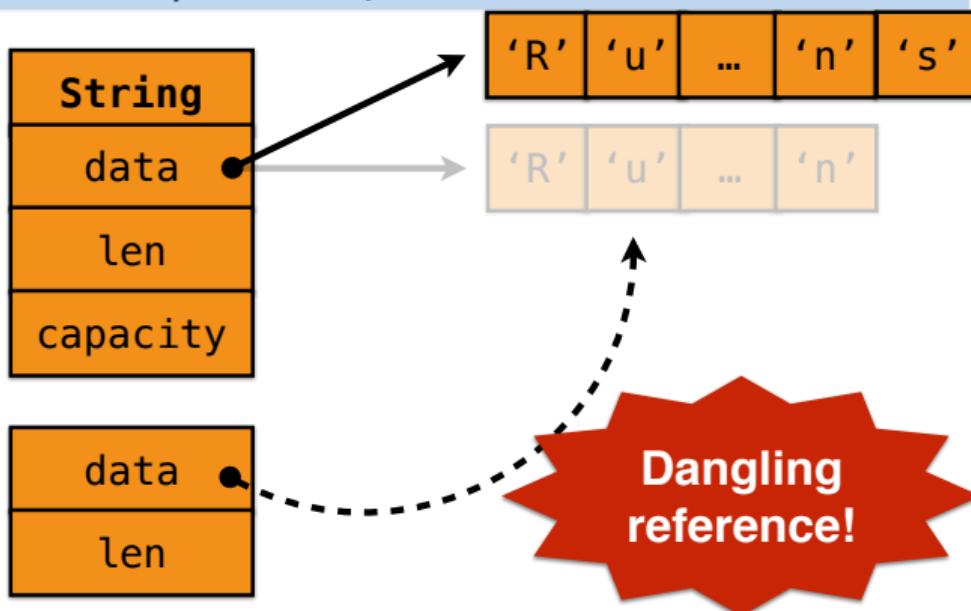
Dangers of mutation

```
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

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

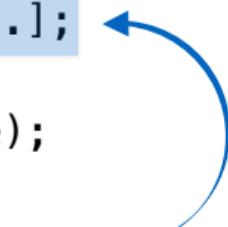
```
fn main() {  
    let mut buffer: String = format!("Rustacean");  
    let slice = &buffer[1..];  
    buffer.push_str("s");  
    println!("{}:?", slice);  
}
```

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

```
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

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 }
```

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
```

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}
```

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 }
```

C

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

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
```

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)
```

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
```

(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)

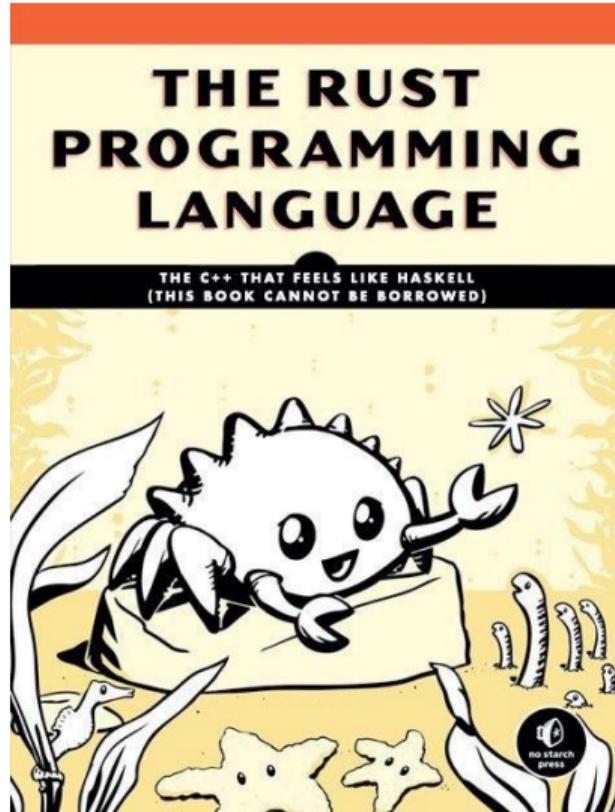
Libraries leverage **ownership and traits** to avoid data races



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 }
```

Syntax



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 }
```

```
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 }
```

Syntax

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         pa3Becmu_JIOXA();
10    }
11    _ => { },
12 }
```

```
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) => {
7       exploit(vuln);
8     }
9     None => println!("Better luck next time."),
10    }
```

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

```
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 &&'
```



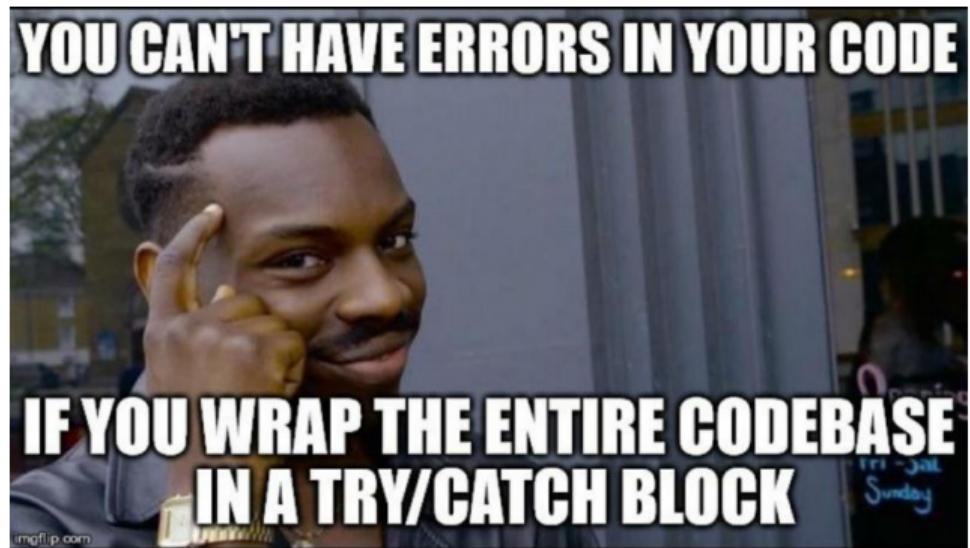
Syntax

Error handling

- Return code (C, Go)



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



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

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

- 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;j"—"æ



Panic

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

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 ...
```

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 }
```

Result

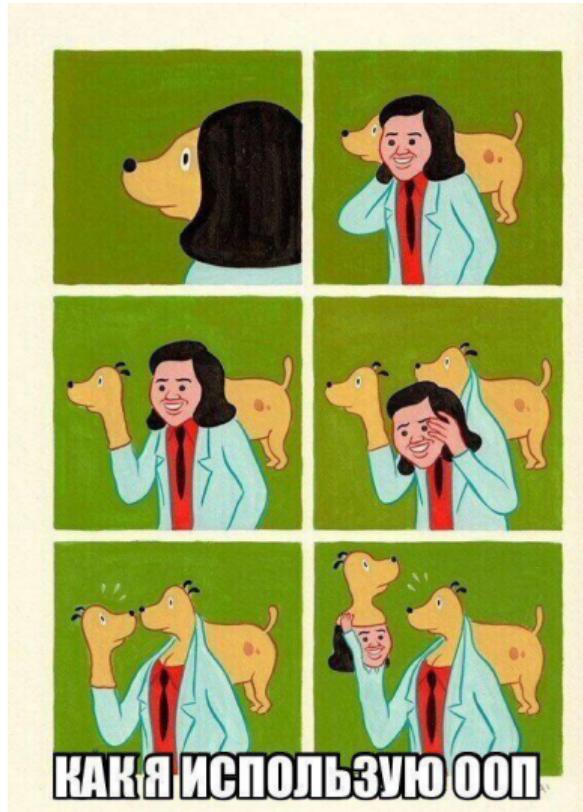
```
1 fn hack_world(world: World) -> Result<Power, u32> {
2     hack_program(&program)?;
3
4     for program in &world.programs() {
5         hack_program(program).map(install_spy).map(get_money)?;
6     }
7 }
```

Syntax

Structs



Проблемы при использовании ООП



```
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 }
```

```
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  }
```

```
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  }
```

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);
```

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);
```

Syntax

Other

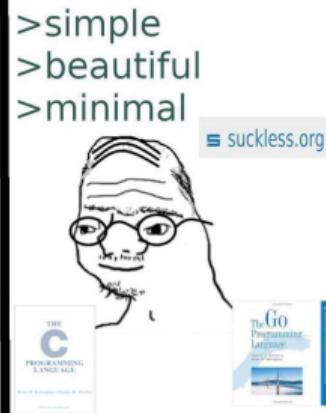
- Generics



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

Traits Polymorphism Generics Concepts

	<code>innocuous</code>	set: added constructor for set with a size hint
<code>b16set</code>		set: added constructor for set with a size hint
<code>b32set</code>		set: added constructor for set with a size hint
<code>b64set</code>		set: added constructor for set with a size hint
<code>f16set</code>		set: added constructor for set with a size hint
<code>f32set</code>		set: added constructor for set with a size hint
<code>f64set</code>		set: added constructor for set with a size hint
<code>i16set</code>		set: added constructor for set with a size hint
<code>i32set</code>		set: added constructor for set with a size hint
<code>i64set</code>		set: added constructor for set with a size hint
<code>internalset</code>		set: added constructor for set with a size hint
<code>iset</code>		set: added constructor for set with a size hint
<code>s16set</code>		set: added constructor for set with a size hint
<code>s32set</code>		set: added constructor for set with a size hint
<code>s64set</code>		set: added constructor for set with a size hint
<code>u16set</code>		set: added constructor for set with a size hint
<code>u32set</code>		set: added constructor for set with a size hint
<code>u64set</code>		set: added constructor for set with a size hint



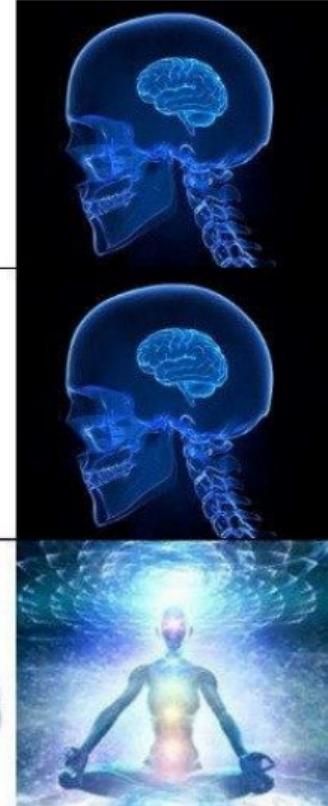
- Generics
- Traits
 - as interfaces
 - for code reuse
 - for operator overloading
- Trait objects
- Closures (`|x| 2 * x`)

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

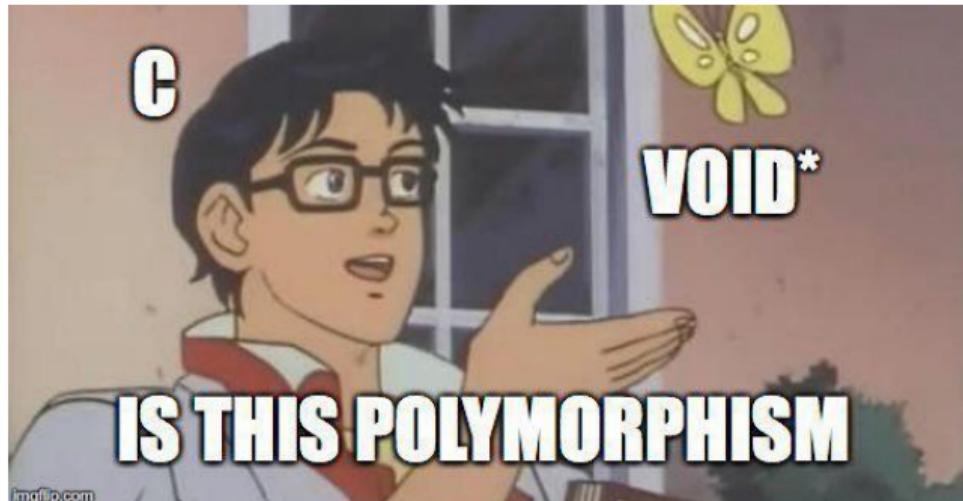
SMART POINTERS

RAW POINTERS

RAW POINTERS TO SMART POINTERS



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



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

Ecosystem

A background network graph composed of numerous small, semi-transparent grey nodes connected by thin white lines, forming a complex web-like structure.

Ecosystem

Community

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



- 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)



- 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

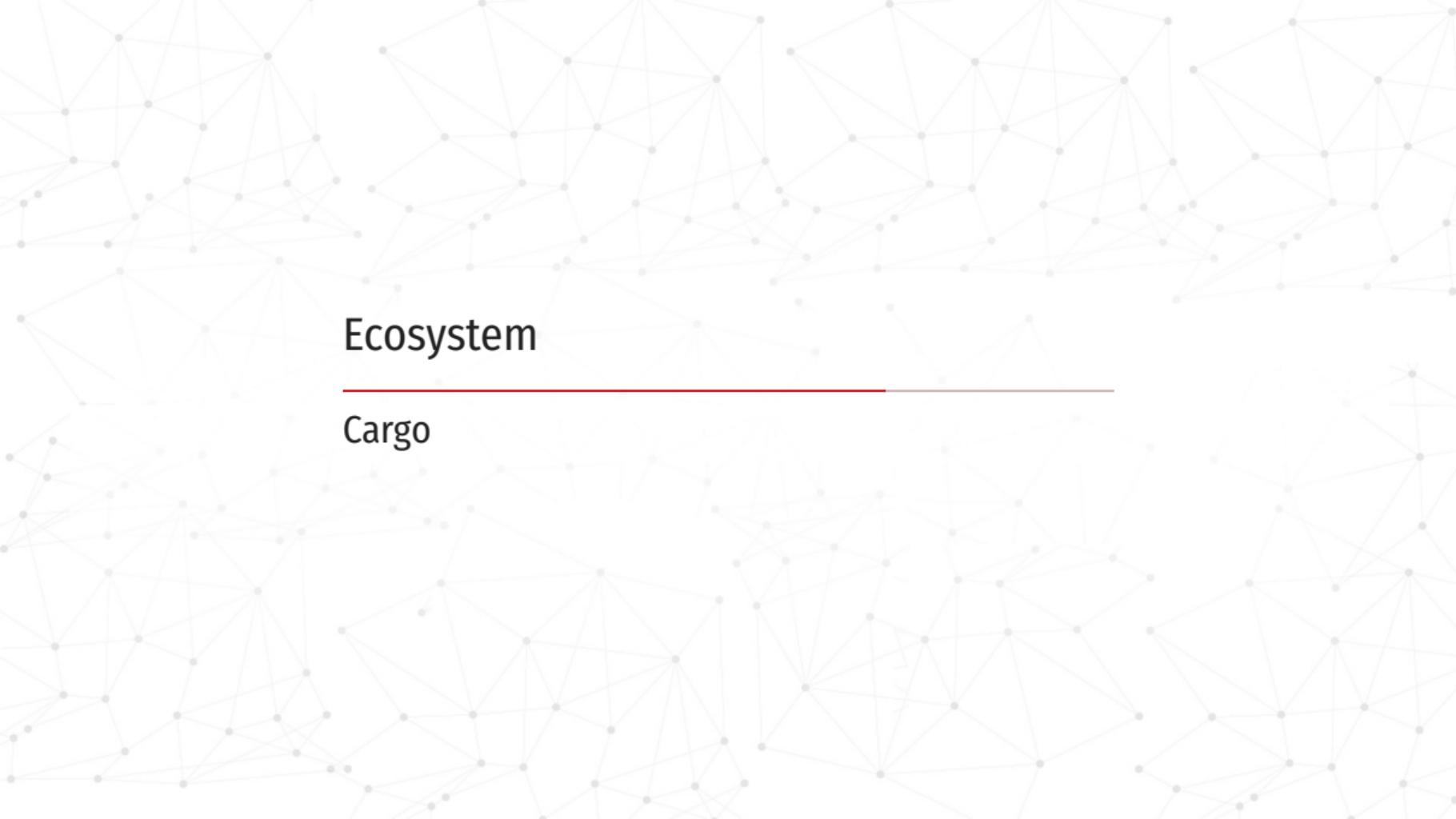


- Rust Working Groups
 - Networking services
 - WebAssembly
 - CLI Apps
 - Embedded Devices
 - Lang and compiler working groups (WG-NLL, WG-UCG, WG-Traits and etc)
- This week in Rust (This week in ProjectName)
- Meetups
- Chats (IRC, Telegram, Gitter, Matrix, Jabber)



- 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)



A background network graph composed of numerous small, semi-transparent grey dots connected by thin white lines, forming a complex web-like structure.

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



Ecosystem

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!

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



Ecosystem

Rustup

Rustup install

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

Toolchain format

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

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

<date> = YYYY-MM-DD

<host> = <target-triple>

Install nightly toolchain

\$ rustup toolchain install nightly

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
- ...

Ecosystem

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

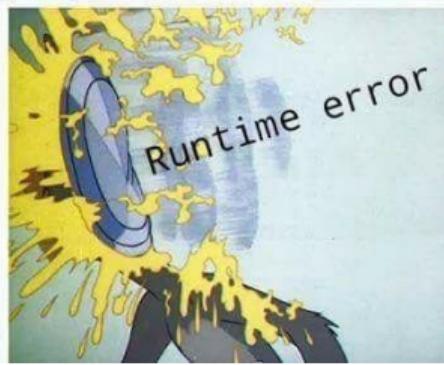
Pitfalls

Pitfalls

Compile time errors

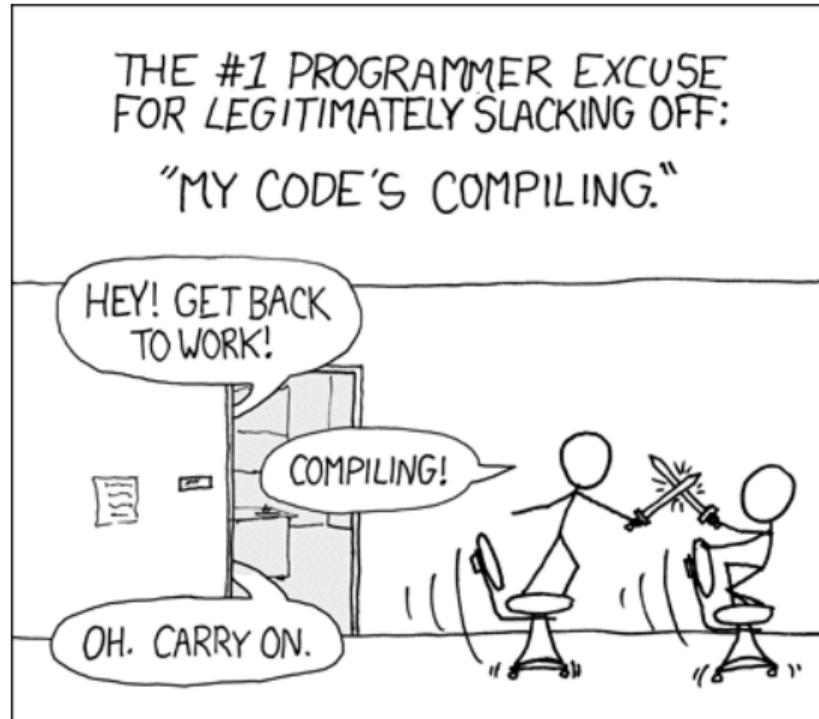


Compile time errors



Pitfalls

Compilation times

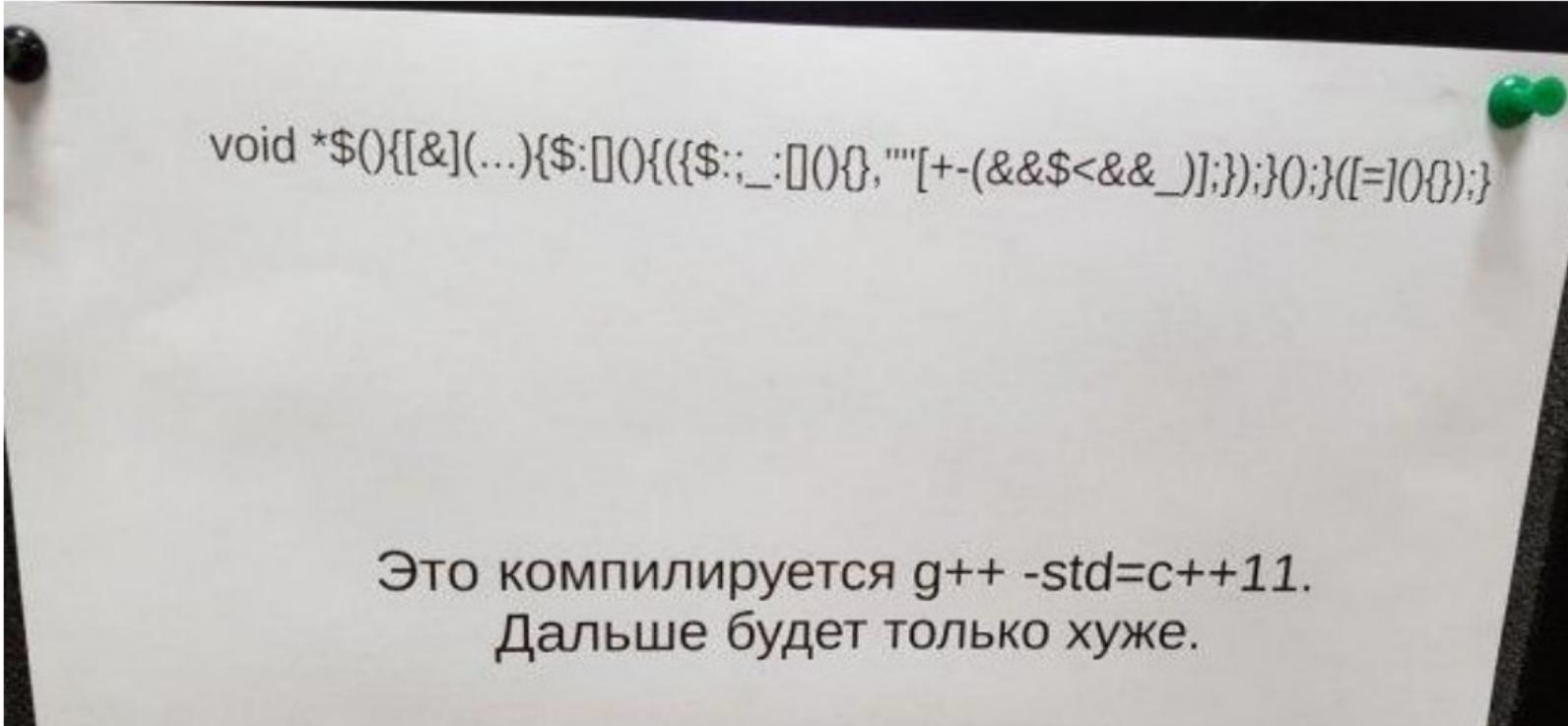


Pitfalls

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 }
```



```
void *$0{[&](...){$:_(){({$:_(){},""+[+-(&&$<&&_)];)();}(){[=]();};}
```

Это компилируется g++ -std=c++11.
Дальше будет только хуже.

Pitfalls

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

Pitfalls

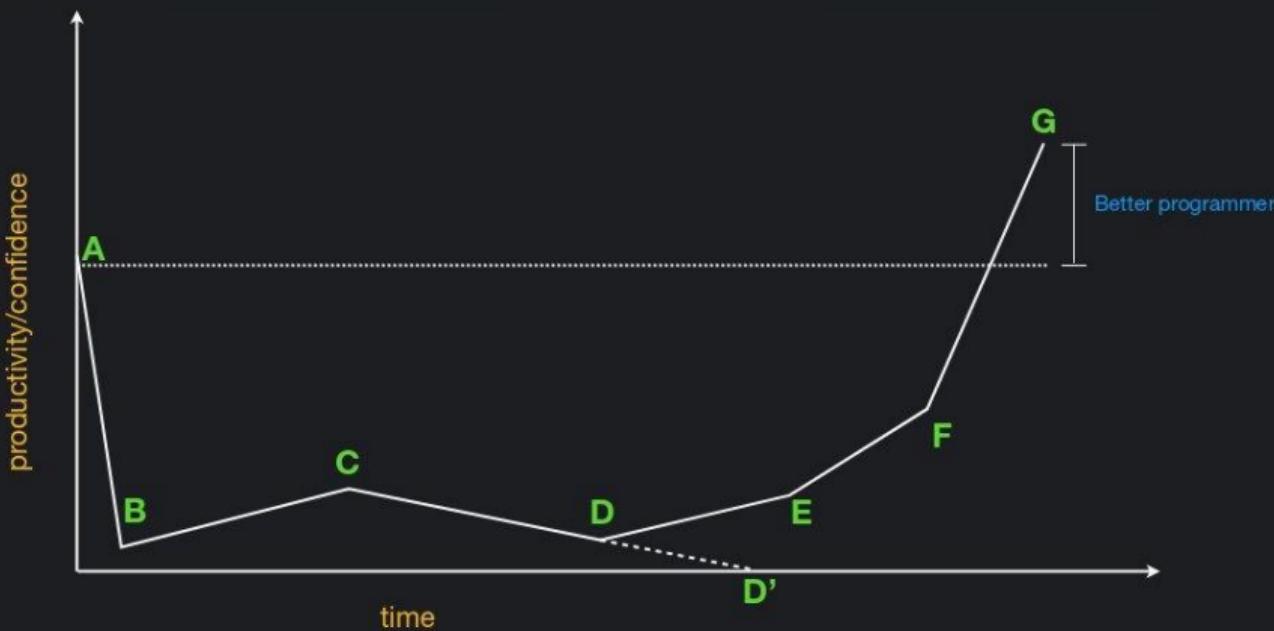
Ecosystem immaturity



Experience

Experience

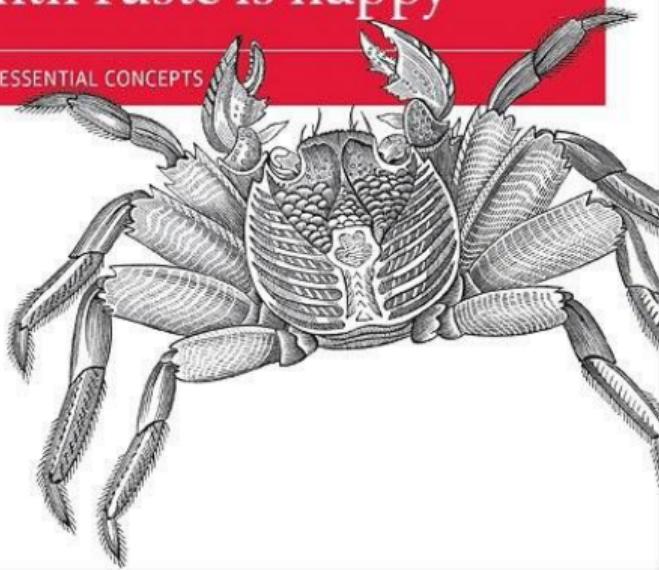
Learning curve

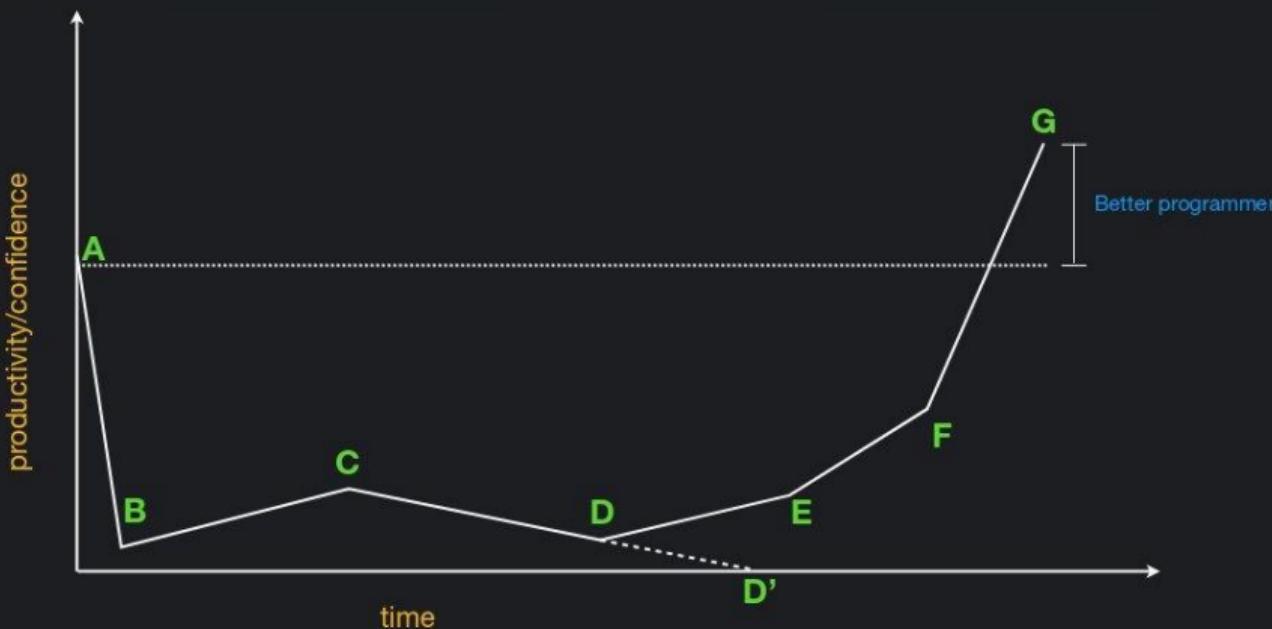


- A I know X, How hard Rust can be?
- B Borrow checker: Will my code ever compile? This is hard
- C Borrow checker thing is not that bad. I still cannot write non-trivial code
- D Even more errors while writing non-trivial code. Who decided to use Rust?
- D' I give up. Rust is too hard. Will use Go instead
- 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

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

50 ESSENTIAL CONCEPTS





- 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

Format of errors or compiler driven development

Format of errors or compiler driven development



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,
2                 from test_err.cpp:3:
3 test_err.cpp: In function ‘int main()’:
4 test_err.cpp:17:5: error: invalid use of void expression
5     assert(std::holds_alternative<std::string>(y)); // succeeds
6 ^~~~~~
```

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 `.`~, `;`~,  
|   `?`~, or an operator here  
6 6 | }  
7 | ^ unexpected token  
8  
9 error: aborting due to previous error
```

Experience

Zero-cost abstractions

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

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



Experience

Undefined behaviour

C

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

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.

- 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
- ...

- 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
- ...

- 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.

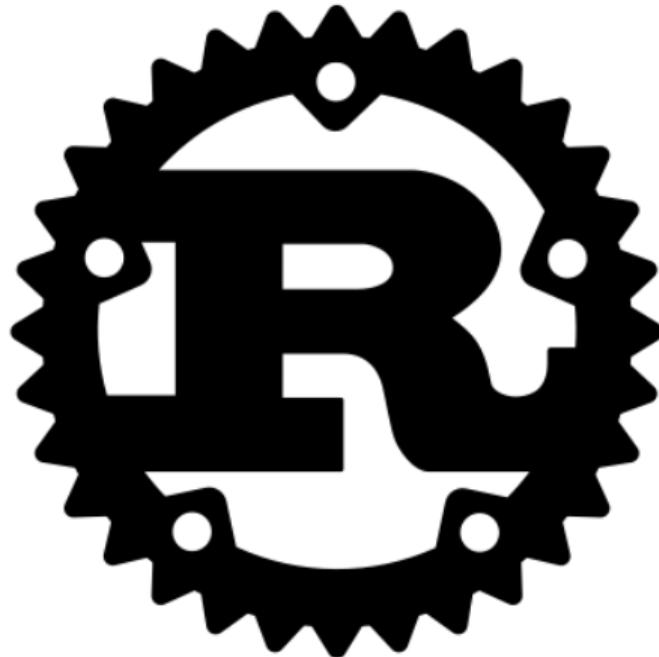
Rust will force you to be a good programmer, you like it or not.

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

Rust community is just the best.

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

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



Questions?

- `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
- ...

 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 }
```

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();
```

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