Rust and what's this thing for?

Abc Xyz
@dura_lex

Agenda

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

Foreword

Motivation

Motivation

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

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?

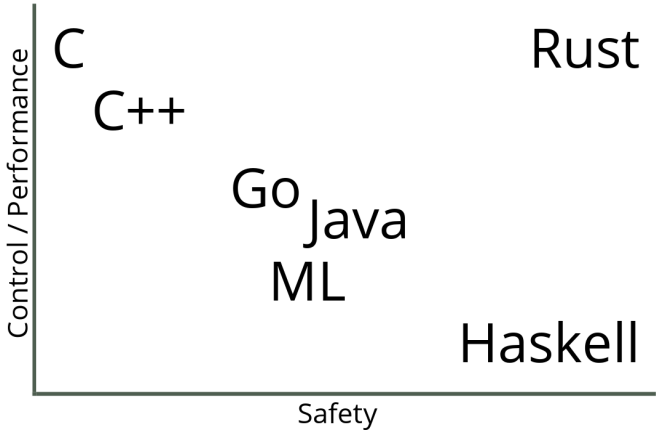
- Performance
  - Fast, memory-efficient
  - No runtime or garbage collector
  - Zero-cost abstractions
- Reliability
  - Rich type system
  - Ownership model
- Productivity
  - Documentation
  - Friendly compiler
  - Top-notch tooling

(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

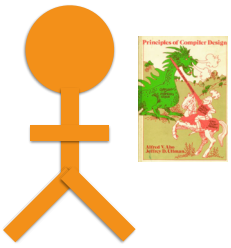
Nicholas Matsakis

Ownership

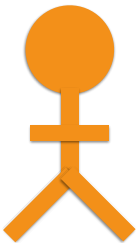
n. The act, state, or right of possessing something.

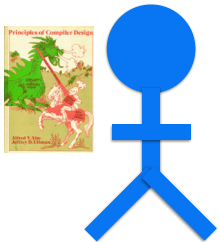
Borrow

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

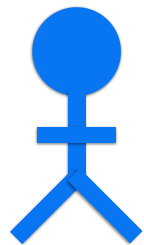
```
fn main() {
  let name = format!("…");
  helper(name);
  helper(name);
}
```

```
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);
}
```

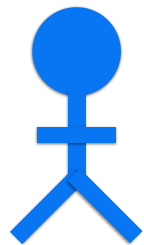
```
fn helper(name: String) {
  println!(..);
}
```

Take ownership of a String

Ownership

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

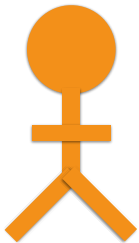
```
fn helper(name: String) {
  println!(..);
}
```



Ownership

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

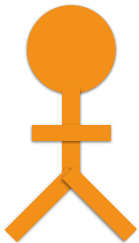
```
fn helper(name: String) {
  println!(..);
}
```



Ownership

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

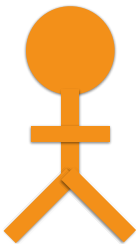
```
fn helper(name: String) {
  println!(..);
}
```



Ownership

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

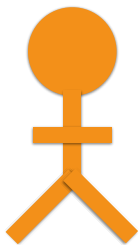
```
fn helper(name: String) {
  println!(..);
}
```



Ownership

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

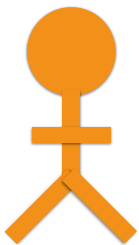
```
fn helper(name: String) {
  println!(..);
}
```



Ownership

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

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

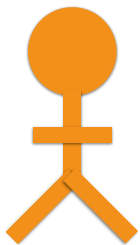


Ownership

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

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

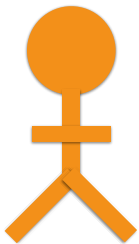
Error: use of moved value: `name`



Ownership

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

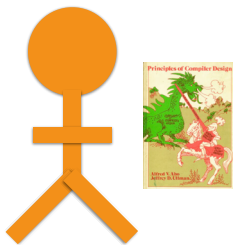
```
void helper(Vector name) {
  …
}
```



“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);
}
```

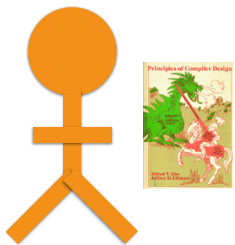
```
void helper(Vector name) {
  …
}
```

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

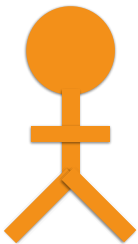
```
void main() {
  Vector name = …;
  helper(name);
  helper(name);
}
```

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

“Ownership” in Java

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

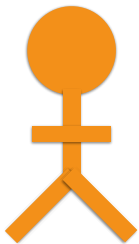
```
void helper(Vector name) {
  …
}
```



“Ownership” in Java

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

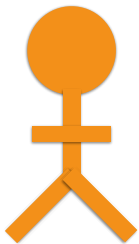
```
void helper(Vector name) {
  …
}
```



“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);
}
```

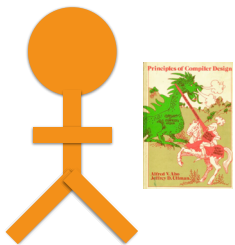
```
void helper(Vector name) {
  new Thread(…);
}
```

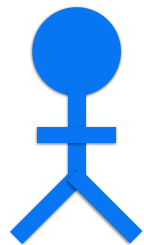
“Ownership” in Java

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);
}
```

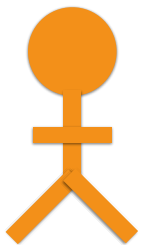
```
fn helper(name: String) {
  println!(..);
}
```

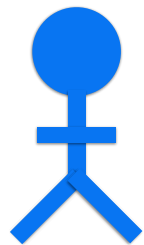
Clone

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

Copy the String

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

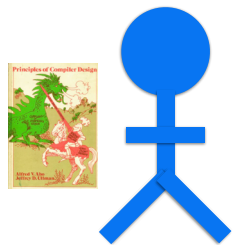




Clone

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

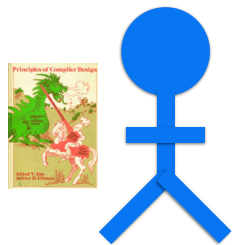
```
fn helper(name: String) {
  println!(..);
}
```



Clone

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

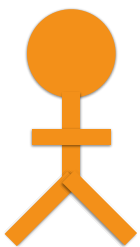
```
fn helper(name: String) {
  println!(..);
}
```

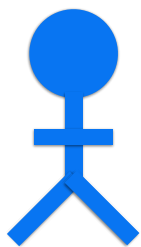


Clone

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

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

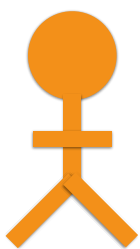


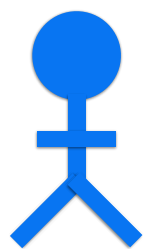


Clone

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

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

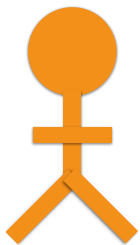




Clone

```
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);
}
```

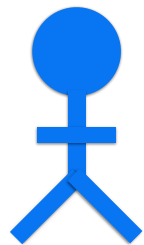
```
fn helper(count: i32) {
    println!(..);
}
```

Copy (auto-Clone)

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

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

i32 is a Copy type

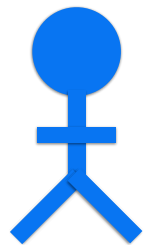


Copy (auto-Clone)

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

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

i32 is a Copy type

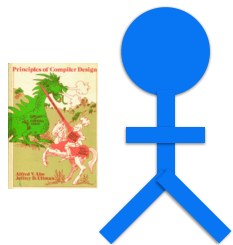


Copy (auto-Clone)

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

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

i32 is a Copy type

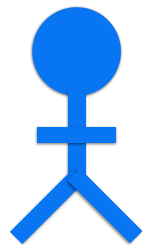


Copy (auto-Clone)

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

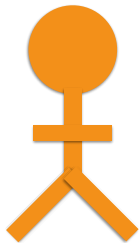
i32 is a Copy type

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

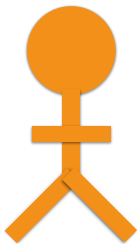


Borrowing: Shared Borrows

Borrowing: Shared Borrows

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

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

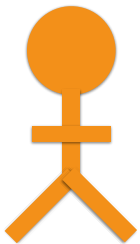


Shared borrow

```
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



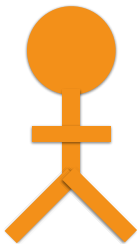
Shared borrow

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

Lend the string, creating a reference

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

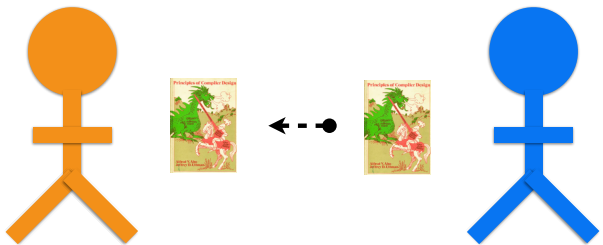
Change type to a **reference** to a String



Shared borrow

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

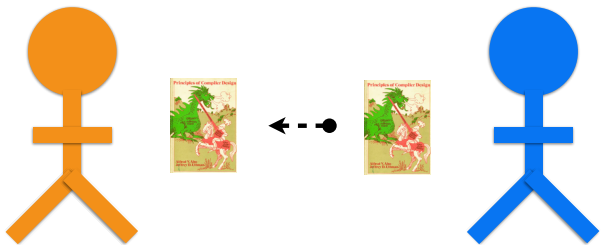
```
fn helper(name: &String) {
  println!(..);
}
```



Shared borrow

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

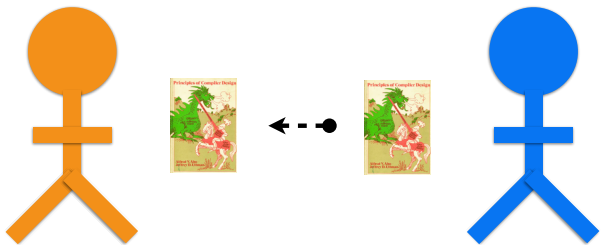
```
fn helper(name: &String) {
  println!(..);
}
```



Shared borrow

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

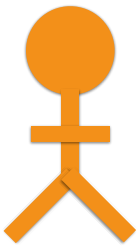
```
fn helper(name: &String) {
  println!(..);
}
```



Shared borrow

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

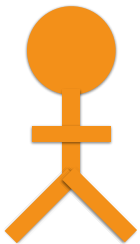
```
fn helper(name: &String) {
  println!(..);
}
```



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

http://is.gd/no0tTH

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

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

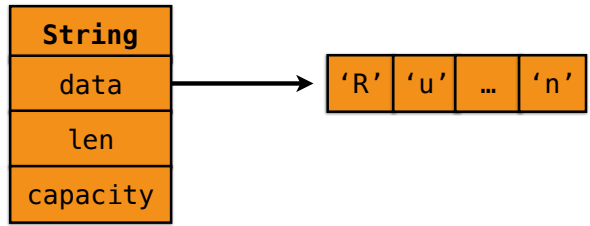
Looks like other languages:

- 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!(..);
}
```



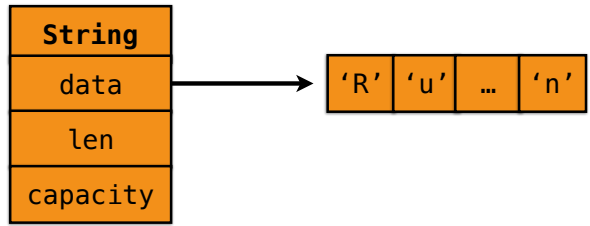
Looks like other languages:

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

But no copying at runtime.

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

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



Looks like other languages:

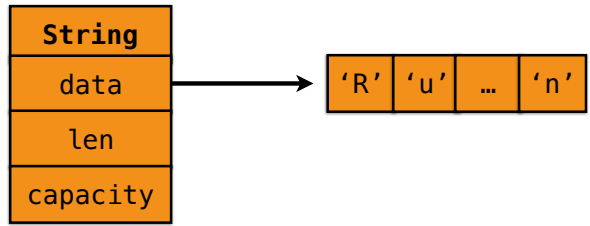
- Python: name[1:]
- Ruby: name[1..-1]

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]

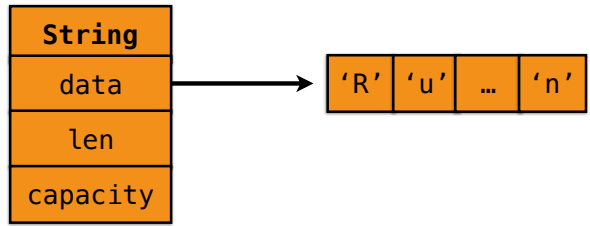
But no copying at runtime.

```
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]

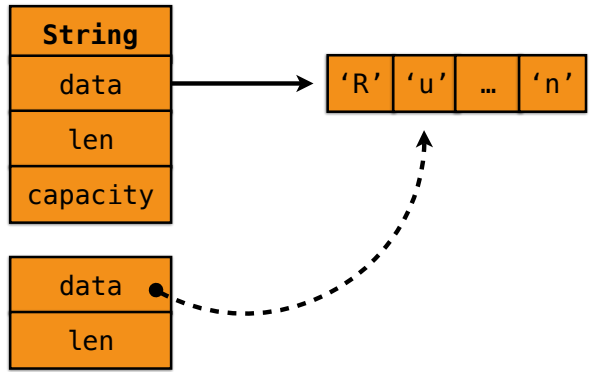
But no copying at runtime.

```
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`

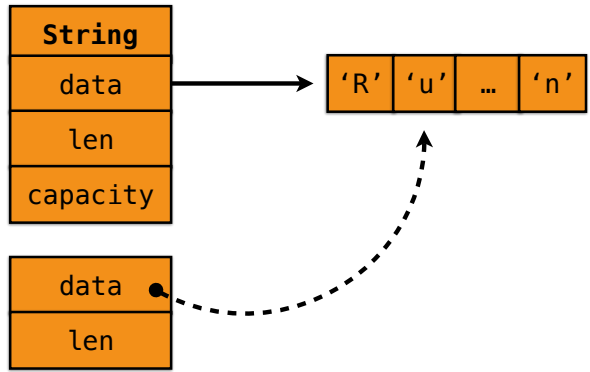


Looks like other languages:
- Python: name[1:]
- Ruby: name[1..-1]

But no copying at runtime.

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

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

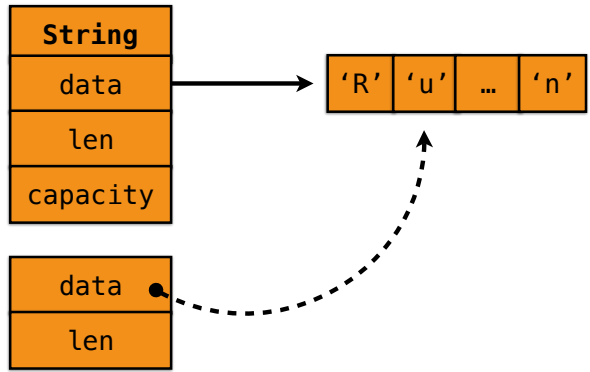


Looks like other languages:
- Python: name[1:]
- Ruby: name[1..-1]

But no copying at runtime.

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

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

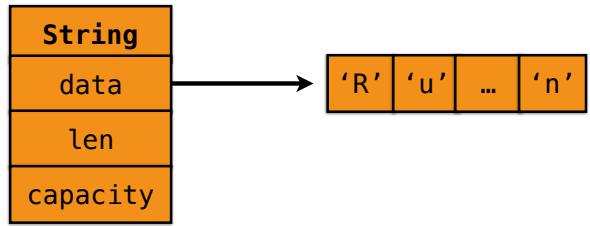


Looks like other languages:
- Python: name[1:]
- Ruby: name[1..-1]

But no copying at runtime.

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

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

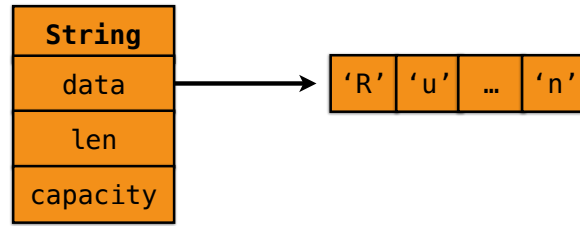


Looks like other languages:
- Python: name[1:]
- Ruby: name[1..-1]

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();
}
```

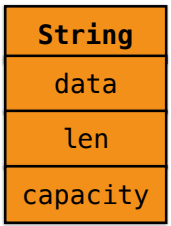
Iterator over slices borrowed from **line**.

No copying, no allocations.

High-level code, low-level efficiency

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

Iterator over slices borrowed from **line**.



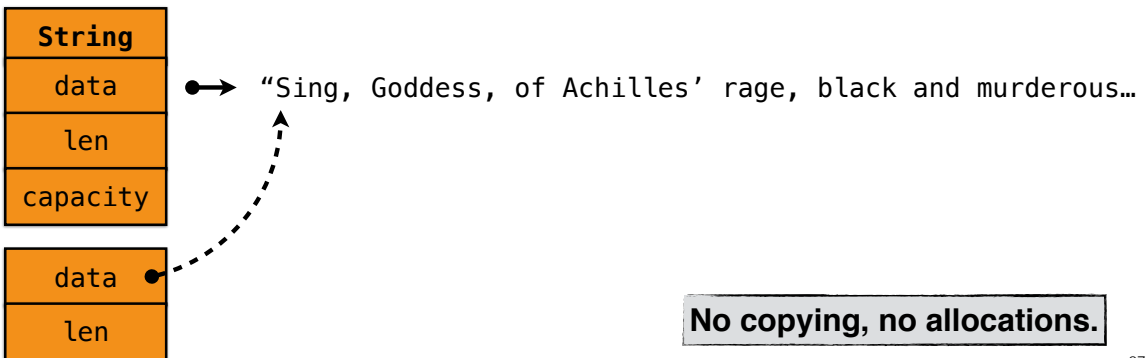
→ “Sing, Goddess, of Achilles’ rage, black and murderous…

No copying, no allocations.

High-level code, low-level efficiency

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

Iterator over slices borrowed from **line**.

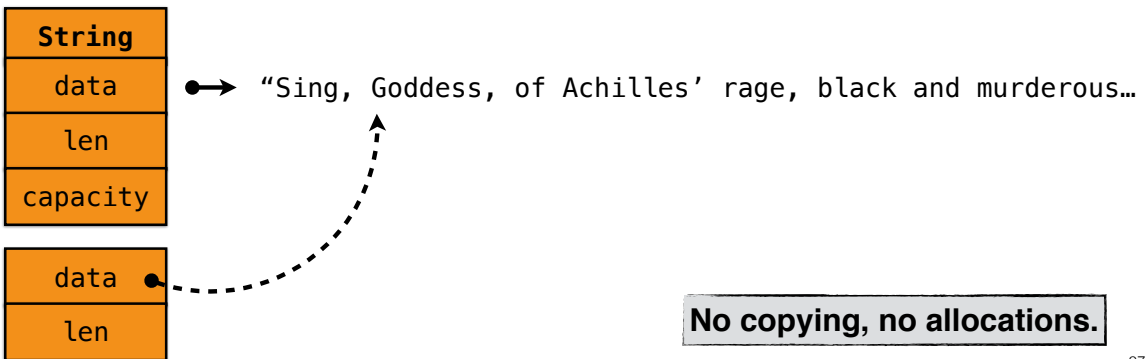


No copying, no allocations.

High-level code, low-level efficiency

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

Iterator over slices borrowed from **line**.

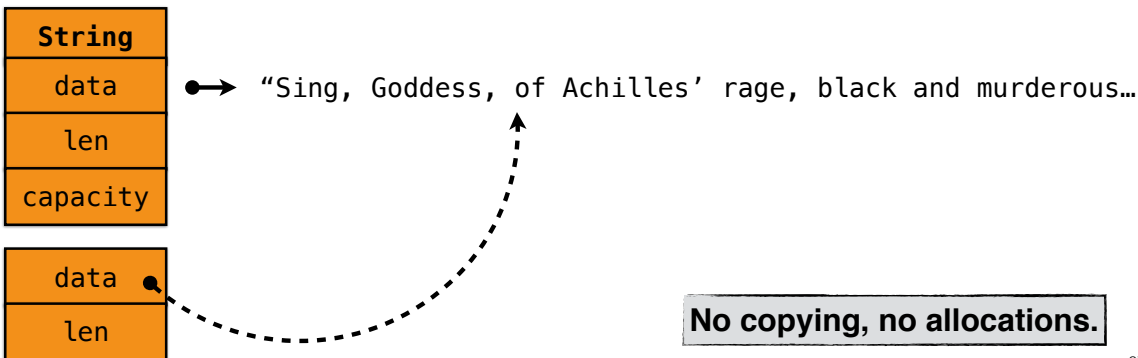


No copying, no allocations.

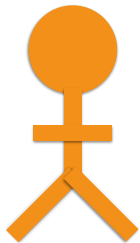
High-level code, low-level efficiency

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

Iterator over slices borrowed from **line**.



Borrowing: Mutable Borrows

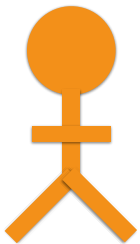


Borrowing: Mutable Borrows

Borrowing: Mutable Borrows

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

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

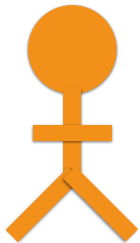


Mutable borrow

```
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



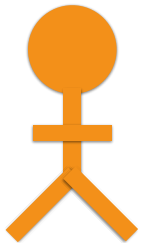
Mutable borrow

```
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**

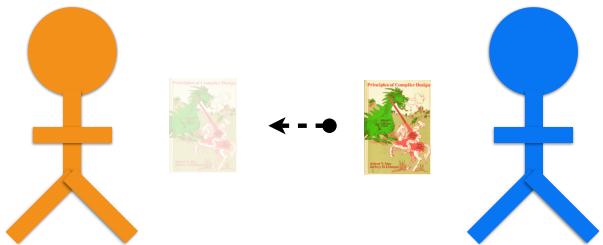
```
fn update(name: &mut String) {
  name.push_str(“…”);
}
```

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);
}
```

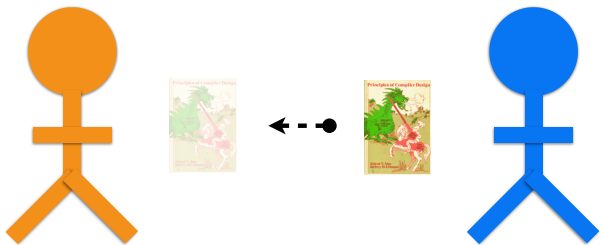
```
fn update(name: &mut String) {
  name.push_str(“…”);
}
```

Mutate string in place

Mutable borrow

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

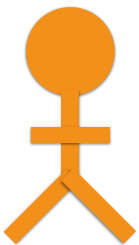
```
fn update(name: &mut String) {
  name.push_str(“…”);
}
```



Mutable borrow

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

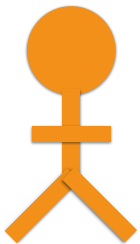
```
fn update(name: &mut String) {
  name.push_str(“…”);
}
```



Mutable borrow

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

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

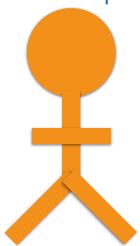


Mutable borrow

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

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

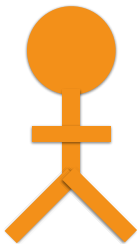
Prints the updated string.



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?

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

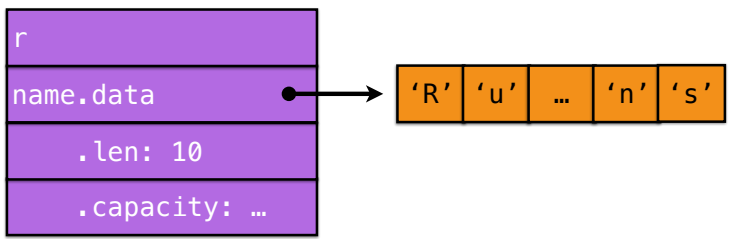
```
fn main() {
  let r;
  {
    let name = format!(“…”);
    r = &name;
  }
  println!(“{}”, r);
}
```

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

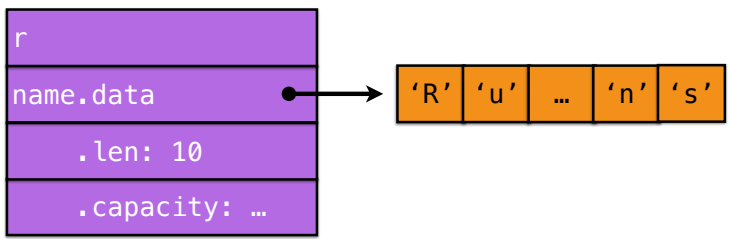
```
fn main() {
  let r;
  {
    let name = format!("…");
    r = &name;
  }
  println!("{}", r);
}
```

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

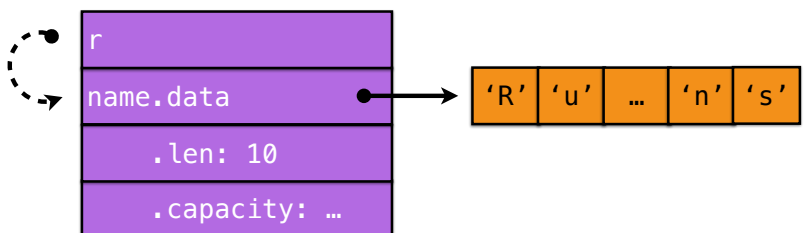
```
fn main() {
  let r;
  {
    let name = format!(“…”);
    r = &name;
  }
  println!(“{}”, r);
}
```



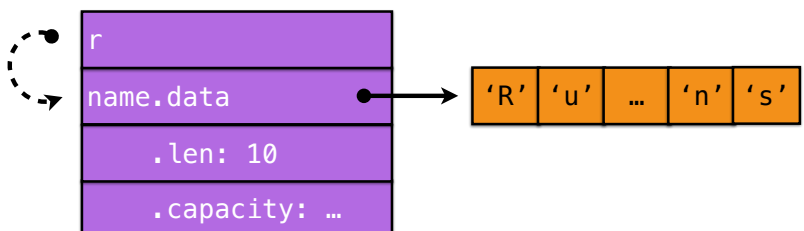
```
fn main() {
  let r;
  {
    let name = format!("…");
    r = &name;
  }
  println!("{}", r);
}
```



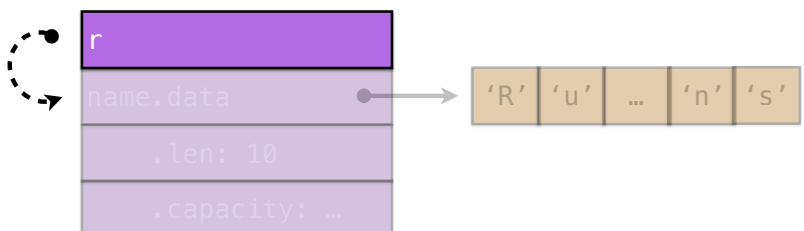
```
fn main() {
  let r;
  {
    let name = format!(“…”);
    r = &name;
  }
  println!(“{}”, r);
}
```



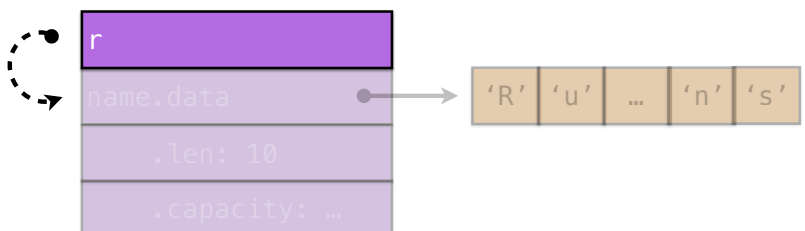
```
fn main() {
  let r;
  {
    let name = format!("…");
    r = &name;
  }
  println!("{}", r);
}
```



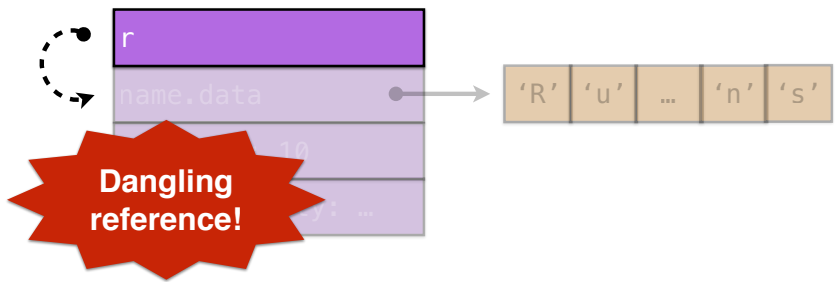
```
fn main() {
  let r;
  {
    let name = format!("…");
    r = &name;
  }
  println!("{}", r);
}
```



```
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

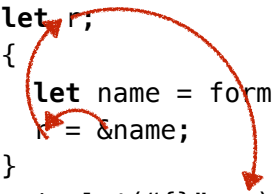
```
fn main() {
  let r;
  {
    let name = format!("…");
    r = &name;
  }
  println!("{}", r);
}
```

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

'l

Lifetime: span of code where reference is used.

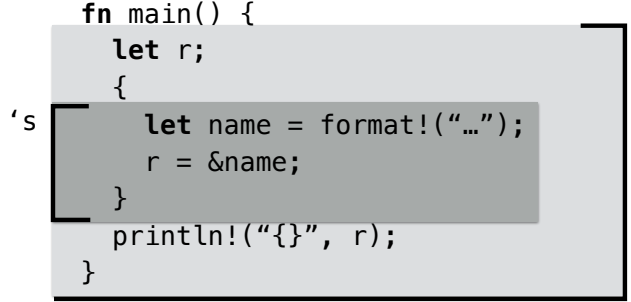
```
fn main() {
  let r;
  {
    let name = format!("…");
    r = &name;
  }
  println!("{}", r);
}
```

'l

Lifetime: span of code where reference is used.

compared against

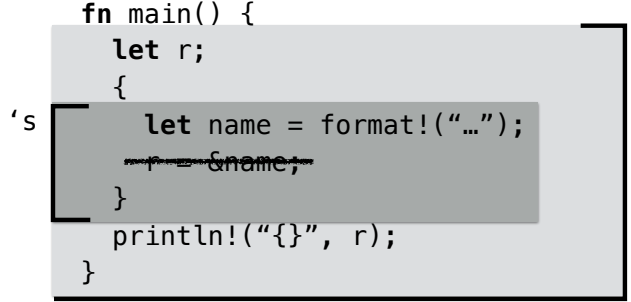
Scope of data being borrowed (here, `name`)



Lifetime: span of code where reference is used.

compared against

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



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;
            ^~~~
```

http://is.gd/HJyO7A

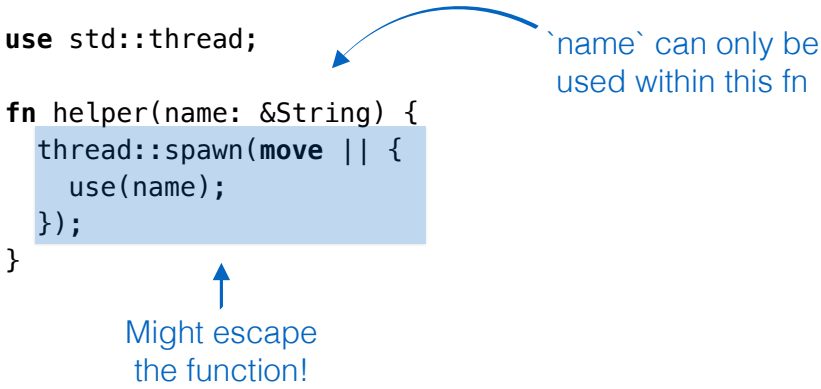
```rust
use std::thread;

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!

```
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

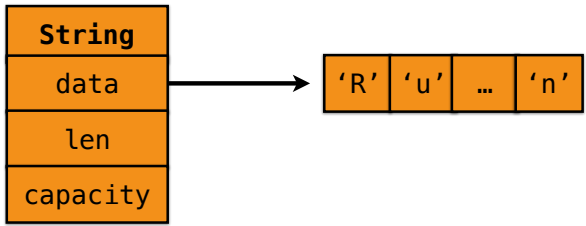
```
error: the type `[…]` does not fulfill the required lifetime
    thread::spawn(move || {
    ^~~~~~~~~~~~~
note: type must outlive the static lifetime
```

Dangers of mutation

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

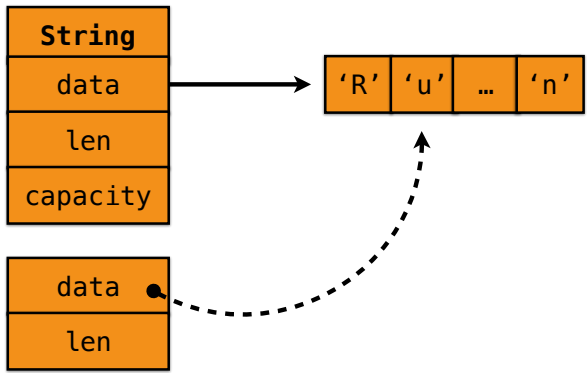
Dangers of mutation

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



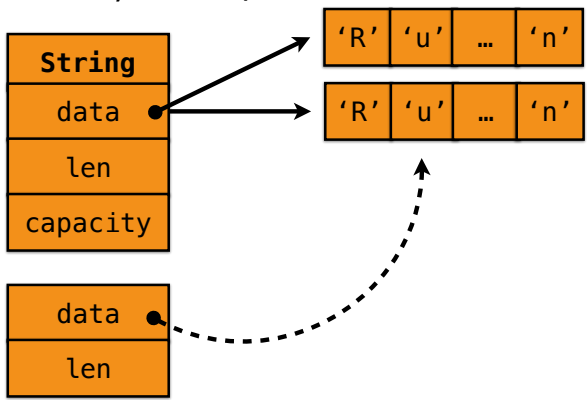
Dangers of mutation

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



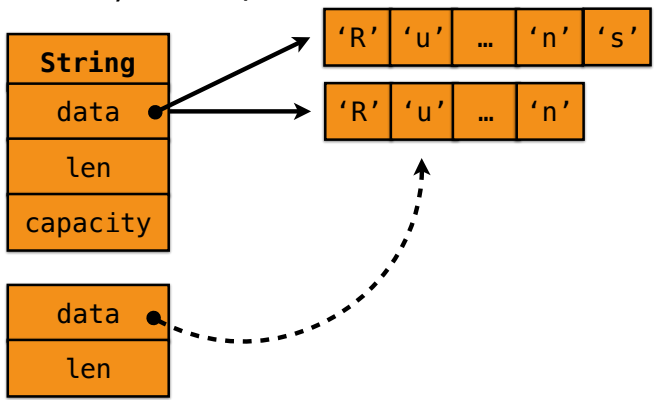
Dangers of mutation

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



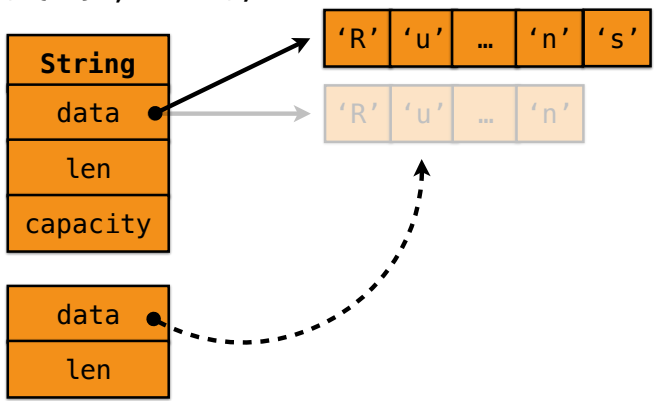
Dangers of mutation

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



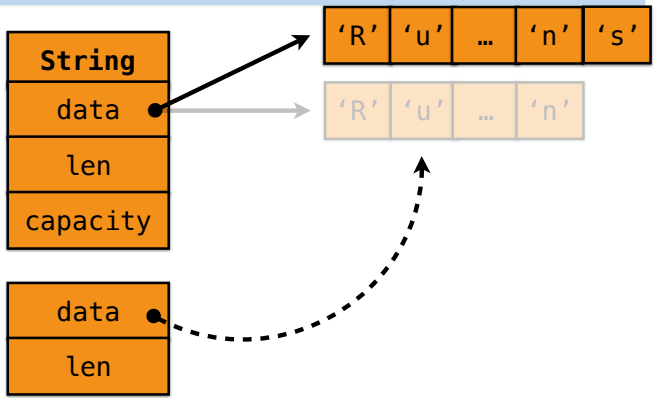
Dangers of mutation

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



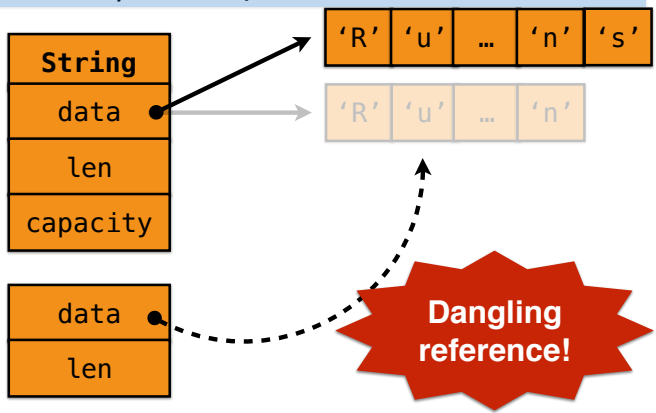
Dangers of mutation

```
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);
}
```

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

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”);
  ^~~~~
```

http://is.gd/MCPVWg

```
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

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

Rust References

```
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)
```

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

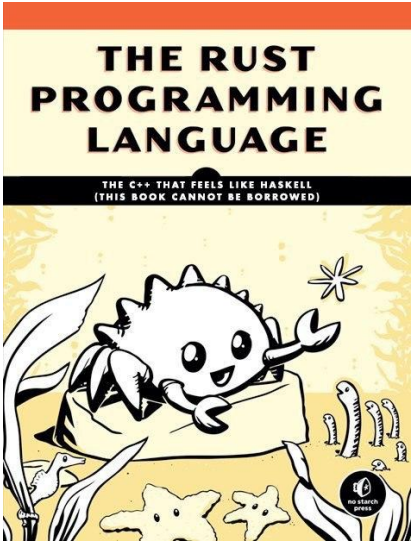
Rust

```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


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

Syntax

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
}
```

```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)

Ecosystem

Ecosystem

Community

Community

Meetups

telegram

Rust in week

gitter

reddit

IRC

matrix

ru, en, all

Ecosystem

Rustup

Ecosystem

Cargo

Cargo

Cross platform

Tests

Benchmarks

Examples

Docs

Ecosystem

Additional tools

Additional tools

cargo:

- fuzz - format - llvm - asm - graph - deps - etc

Ecosystem

IDE

Popularity

Popularity

Friends

Popularity

Popular software

- CLI tools
- Web
- Servo

Summary

Summary

Questions?

References i