Rust and what's this thing for?

Abc Xyz
@dura_lex

Agenda

1. Foreword
2. What is Rust?
3. (Un)safe
4. Summary

Foreword

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
- 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?

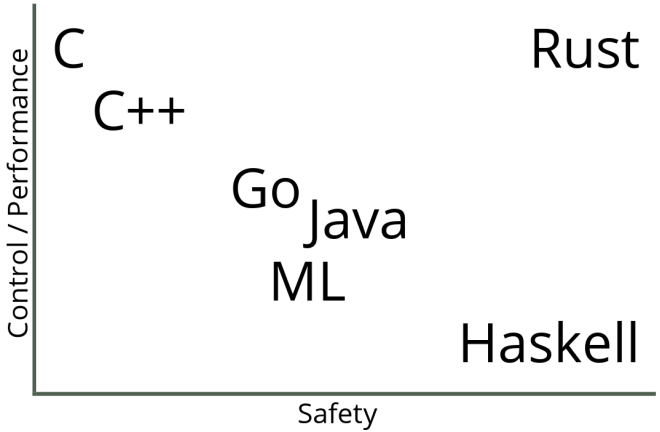
- Performance
  - Fast, memory-efficient
  - No runtime or garbage collector
  - Zero-cost abstractions
- Reliability
  - Rich type system
  - Ownership model
- Productivity
  - Documentation
  - Friendly compiler
  - Top-notch tooling

(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

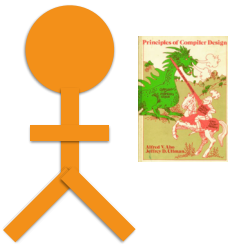
Nicholas Matsakis

Ownership

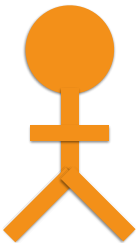
n. The act, state, or right of possessing something.

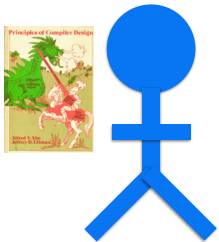
Borrow

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

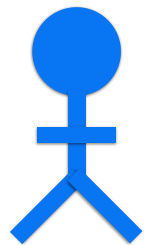
```
fn main() {
  let name = format!("…");
  helper(name);
  helper(name);
}
```

```
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);
}
```

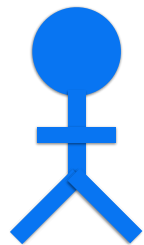
```
fn helper(name: String) {
  println!(..);
}
```

Take ownership of a String

Ownership

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

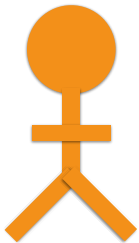
```
fn helper(name: String) {
  println!(..);
}
```



Ownership

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

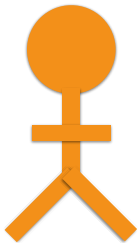
```
fn helper(name: String) {
  println!(..);
}
```



Ownership

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

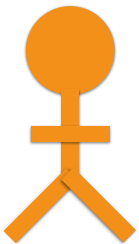
```
fn helper(name: String) {
  println!(..);
}
```



Ownership

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

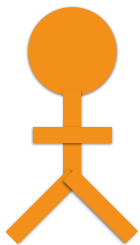
```
fn helper(name: String) {
  println!(..);
}
```



Ownership

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

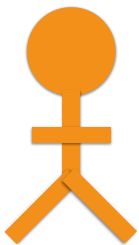
```
fn helper(name: String) {
  println!(..);
}
```



Ownership

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

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

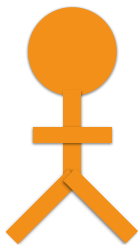


Ownership

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

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

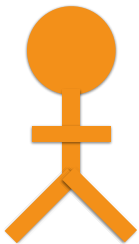
Error: use of moved value: `name`



Ownership

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

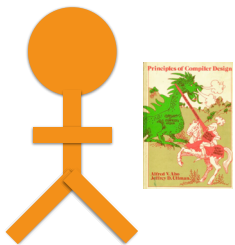
```
void helper(Vector name) {
  …
}
```



“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);
}
```

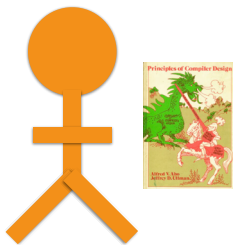
```
void helper(Vector name) {
  …
}
```

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

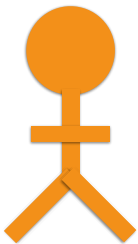
```
void main() {
  Vector name = …;
  helper(name);
  helper(name);
}
```

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

“Ownership” in Java

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

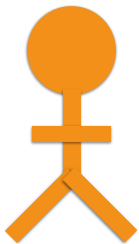
```
void helper(Vector name) {
  …
}
```



“Ownership” in Java

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

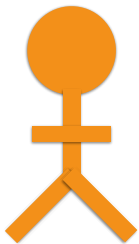
```
void helper(Vector name) {
  …
}
```



“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);
}
```

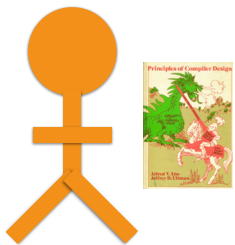
```
void helper(Vector name) {
  new Thread(…);
}
```

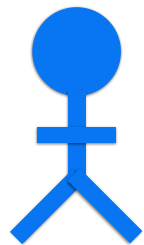
“Ownership” in Java

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);
}
```

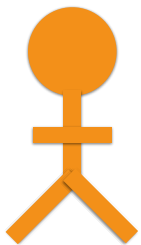
```
fn helper(name: String) {
  println!(..);
}
```

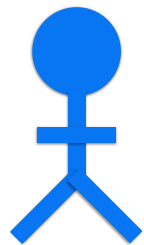
Clone

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

Copy the String

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

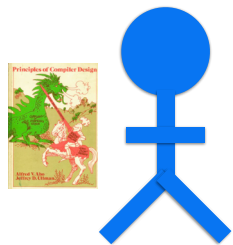




Clone

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

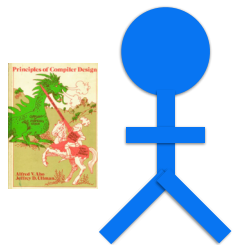
```
fn helper(name: String) {
  println!(..);
}
```



Clone

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

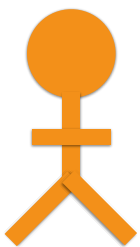
```
fn helper(name: String) {
  println!(..);
}
```

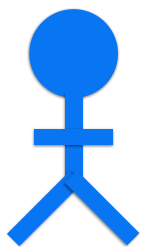


Clone

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

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

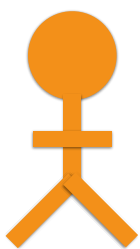


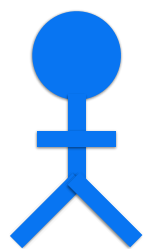


Clone

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

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

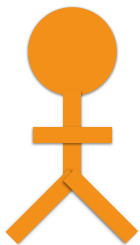




Clone

```
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);
}
```

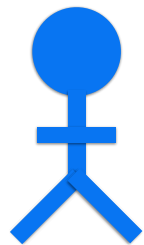
```
fn helper(count: i32) {
    println!(..);
}
```

Copy (auto-Clone)

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

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

i32 is a Copy type

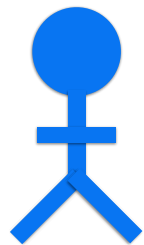


Copy (auto-Clone)

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

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

i32 is a Copy type

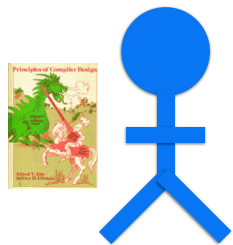


Copy (auto-Clone)

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

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

i32 is a Copy type

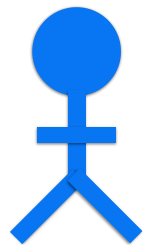


Copy (auto-Clone)

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

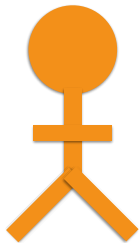
i32 is a Copy type

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

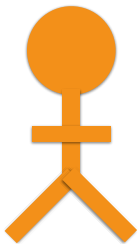


Borrowing: Shared Borrows

Borrowing: Shared Borrows

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

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

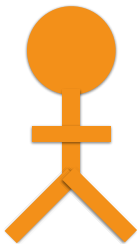


Shared borrow

```
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



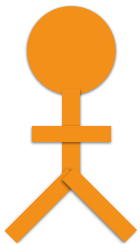
Shared borrow

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

Lend the string, creating a reference

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

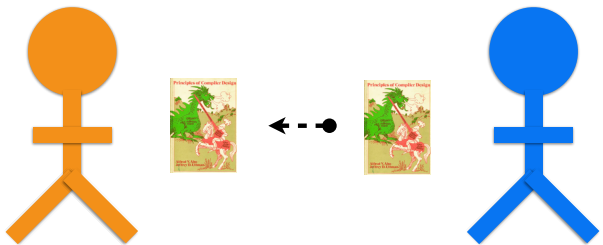
Change type to a **reference** to a String



Shared borrow

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

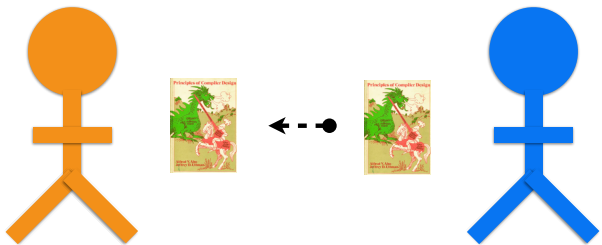
```
fn helper(name: &String) {
  println!(..);
}
```



Shared borrow

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

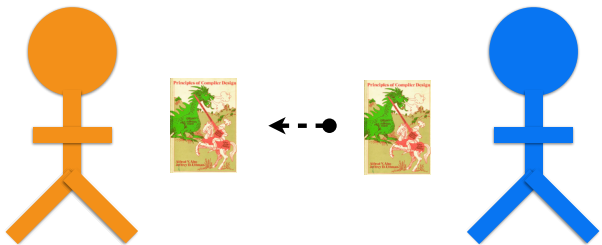
```
fn helper(name: &String) {
  println!(..);
}
```



Shared borrow

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

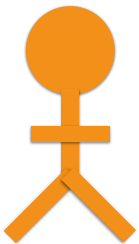
```
fn helper(name: &String) {
  println!(..);
}
```



Shared borrow

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

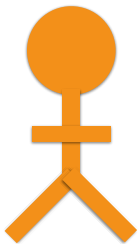
```
fn helper(name: &String) {
  println!(..);
}
```



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!(..);
}
```

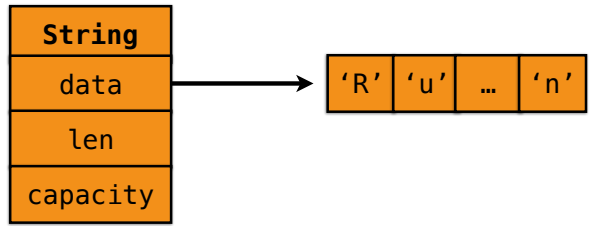
Looks like other languages:

- 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!(..);
}
```



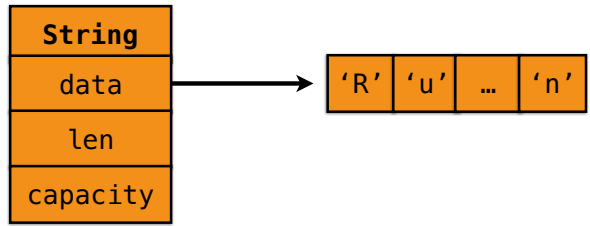
Looks like other languages:

- 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:

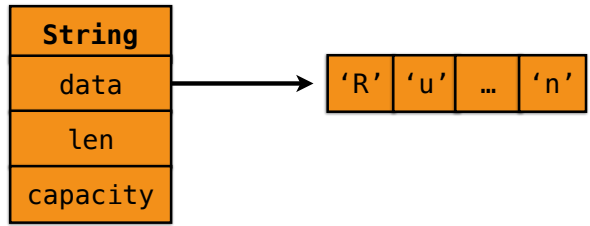
- Python: name[1:]
- Ruby: name[1..-1]

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]

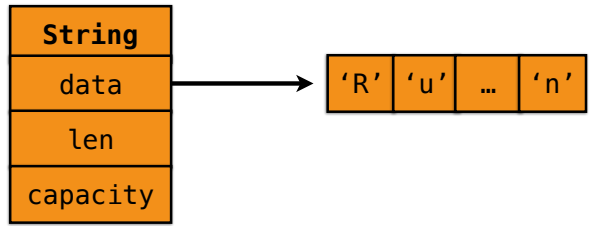
But no copying at runtime.

```
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]

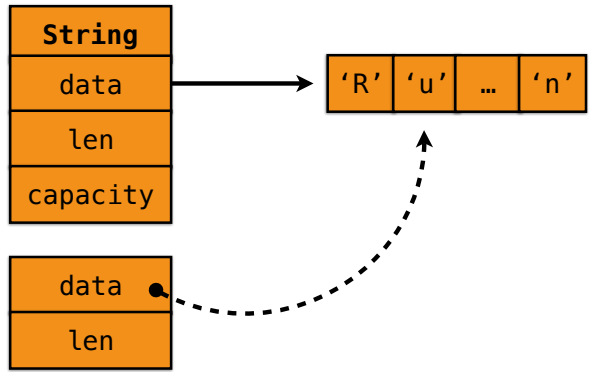
But no copying at runtime.

```
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`

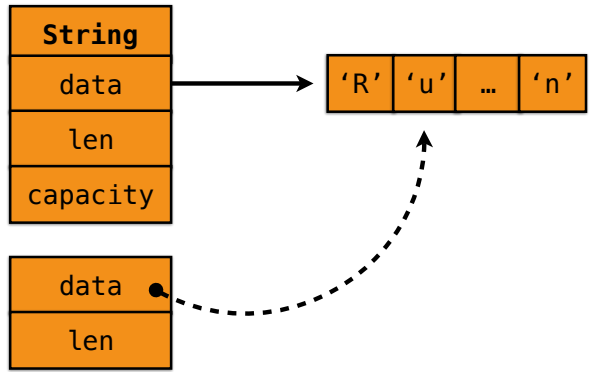


Looks like other languages:
- Python: name[1:]
- Ruby: name[1..-1]

But no copying at runtime.

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

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

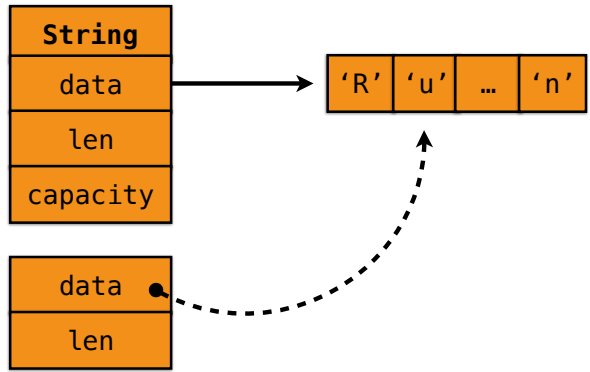


Looks like other languages:
- Python: name[1:]
- Ruby: name[1..-1]

But no copying at runtime.

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

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

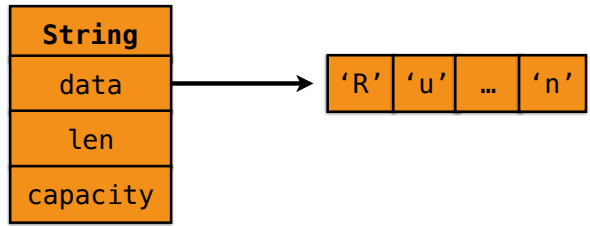


Looks like other languages:
- Python: name[1:]
- Ruby: name[1..-1]

But no copying at runtime.

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

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

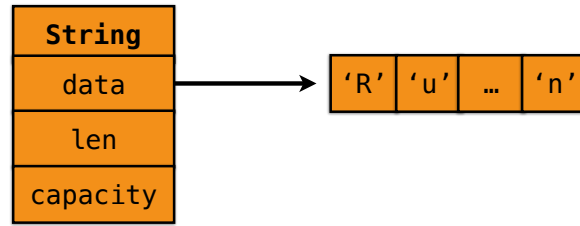


Looks like other languages:
- Python: name[1:]
- Ruby: name[1..-1]

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();
}
```

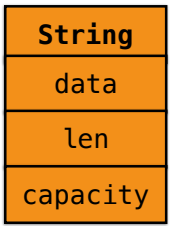
Iterator over slices borrowed from **line**.

No copying, no allocations.

High-level code, low-level efficiency

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

Iterator over slices borrowed from **line**.



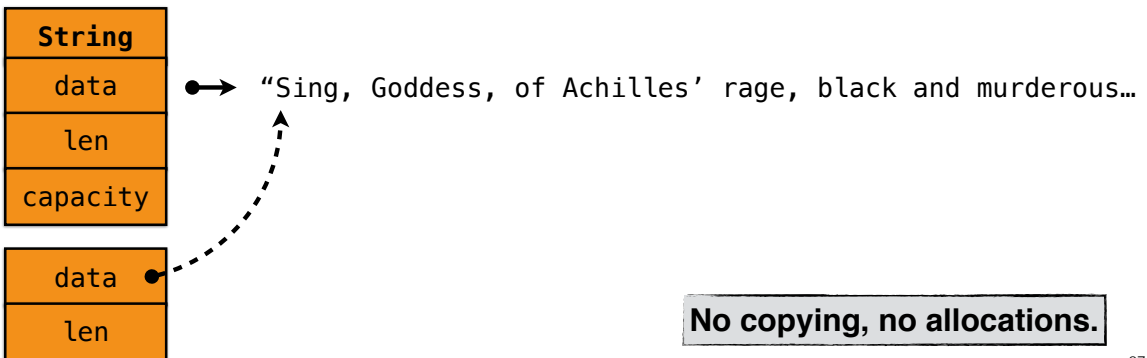
→ “Sing, Goddess, of Achilles’ rage, black and murderous…

No copying, no allocations.

High-level code, low-level efficiency

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

Iterator over slices borrowed from **line**.

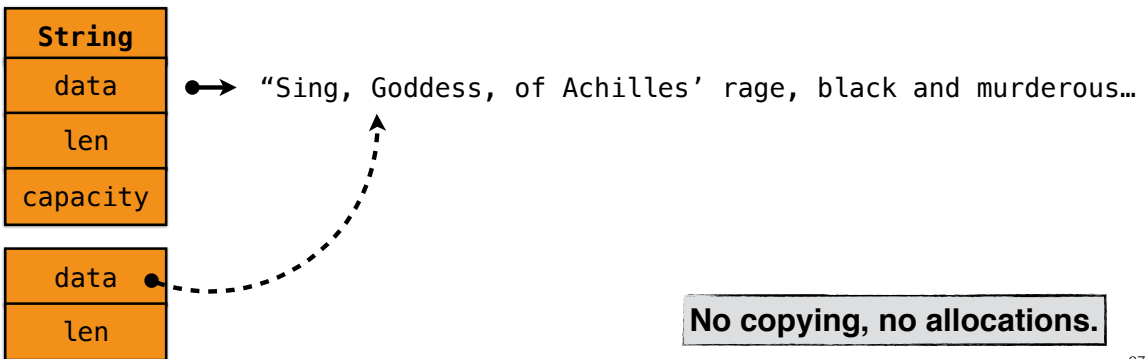


No copying, no allocations.

High-level code, low-level efficiency

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

Iterator over slices borrowed from **line**.

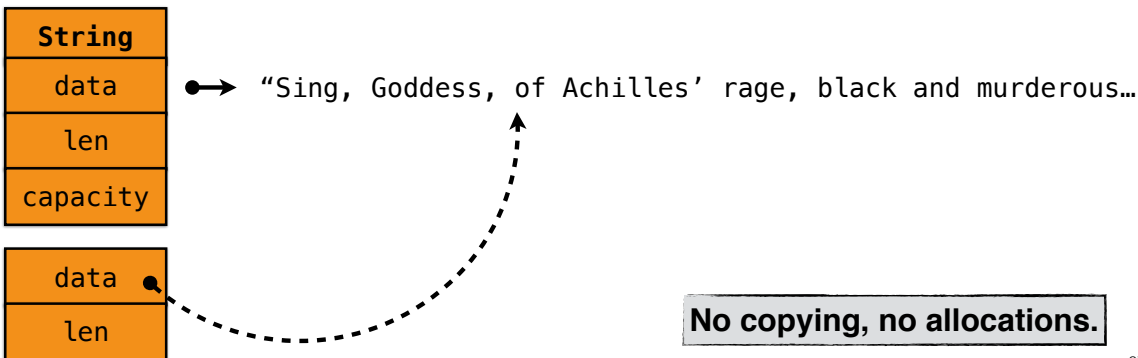


No copying, no allocations.

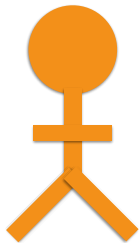
High-level code, low-level efficiency

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

Iterator over slices borrowed from **line**.



Borrowing: Mutable Borrows

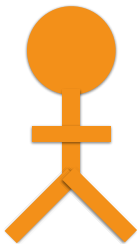


Borrowing: Mutable Borrows

Borrowing: Mutable Borrows

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

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

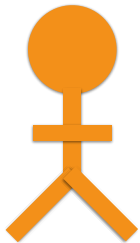


Mutable borrow

```
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



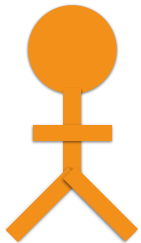
Mutable borrow

```
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**

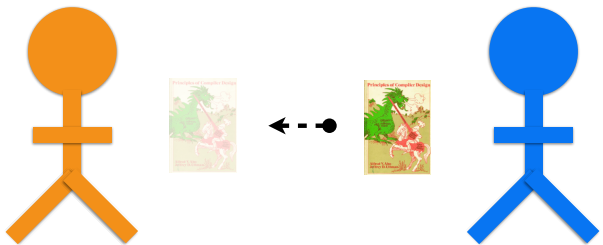
```
fn update(name: &mut String) {
  name.push_str(“…”);
}
```

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);
}
```

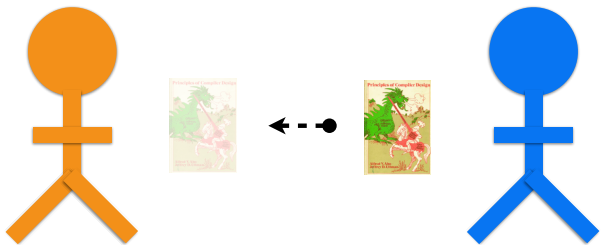
```
fn update(name: &mut String) {
  name.push_str(“…”);
}
```

Mutate string in place

Mutable borrow

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

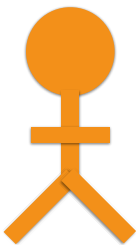
```
fn update(name: &mut String) {
  name.push_str(“…”);
}
```



Mutable borrow

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

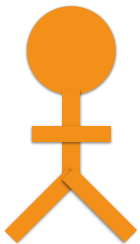
```
fn update(name: &mut String) {
  name.push_str(“…”);
}
```



Mutable borrow

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

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

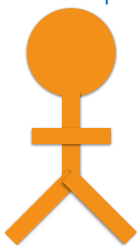


Mutable borrow

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

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

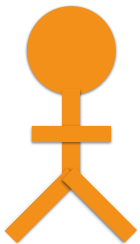
Prints the updated string.



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

How do we get safety?

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

Summary

Summary

Questions?

References i