



Beginner Tutorial: Ownership and Borrowing

Steve Klabnik



Hack without fear!


```
class ::String
  def blank?
    /\A[[:space:]]*\z/ == self
  end
end
```

Performance

Ruby:
964K iter/sec


```

static VALUE
rb_str_blank_as(VALUE str)
{
    rb_encoding *enc;
    char *s, *e;

    enc = STR_ENC_GET(str);
    s = RSTRING_PTR(str);
    if (!s || RSTRING_LEN(str) == 0) return Qtrue;

    e = RSTRING_END(str);
    while (s < e) {
        int n;
        unsigned int cc = rb_enc_codepoint_len(s, e, &n, enc);

        switch (cc) {
            case 9:
            case 0xa:
            case 0xb:
            case 0xc:
            case 0xd:
            case 0x20:
            case 0x85:
            case 0xa0:
            case 0x1680:
            case 0x2000:
            case 0x2001:
            case 0x2002:
            case 0x2003:
            case 0x2004:

```

```

            case 0x2005:
            case 0x2006:
            case 0x2007:
            case 0x2008:
            case 0x2009:
            case 0x200a:
            case 0x2028:
            case 0x2029:
            case 0x202f:
            case 0x205f:
            case 0x3000:
                #if ruby_version_before_2_2()
                    case 0x180e:
                #endif
                    /* found */
                    break;
            default:
                return Qfalse;
        }
        s += n;
    }
    return Qtrue;
}

```

https://github.com/SamSaffron/fast_blank

Performance

Ruby:
964K iter/sec

10x!

C:
10.5M iter/sec


```
class ::String
  def blank?
    /\A[[:space:]]*\z/ == self
  end
end
```

```
extern "C" fn fast_blank(buf: Buf) -> bool {
  buf.as_slice().chars().all(|c| c.is_whitespace())
}
```

Get Rust
string slice

Get iterator over
each character

Are all characters
whitespace?

Performance

Ruby:
964K iter/sec

C:
10.5M iter/sec

Rust:
11M iter/sec

Parallel Programming

```
fn load_images(paths: &[PathBuf]) -> Vec<Image> {  
    paths.iter() ← For each path...  
        .map(|path| {  
            Image::load(path) ← ...load an image...  
        })  
    .collect() ← ...create and return  
                a vector.  
}
```


Parallel Programming

`extern crate rayon;`  Third-party library

```
fn load_images(paths: &[PathBuf]) -> Vec<Image> {  
    paths.par_iter()  ...make it parallel.  
        .map(|path| {  
            Image::load(path)  
        })  
        .collect()  
}
```

Can also do: processes with channels,
mutexes, non-blocking data structures...

Safer Parallel Programming

```
fn load_images(paths: &[PathBuf]) -> Vec<Image> {  
    let mut jpegs = 0;  
    paths.par_iter()  
        .map(|path| {  
            if path.ends_with("jpeg") { jpegs += 1; }  
            Image::load(path)  
        })  
        .collect();  
}
```

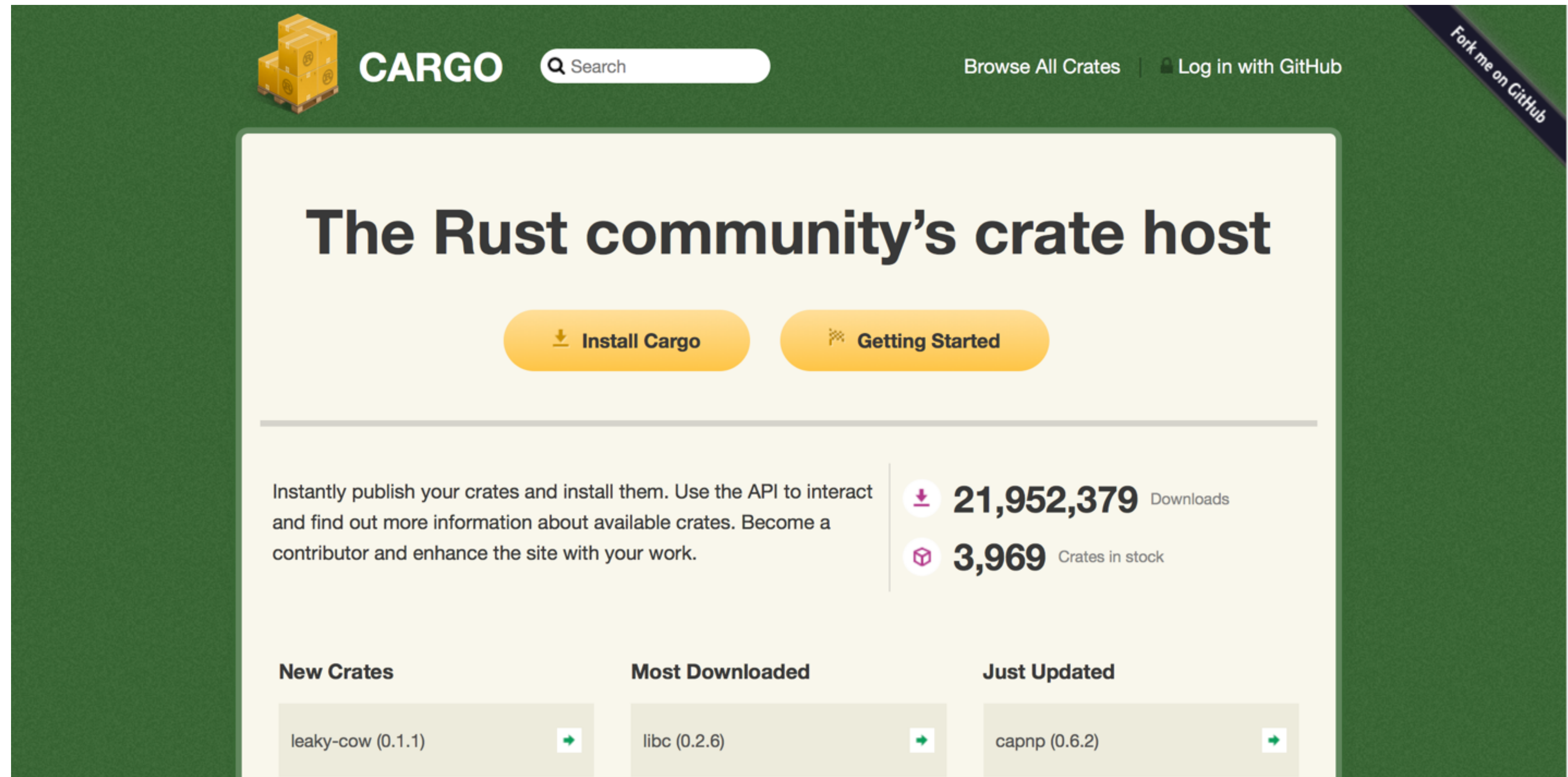


The diagram illustrates a data race. A blue arrow labeled "Data-race" points from the `let mut jpegs = 0;` line to the `jpegs += 1;` line. A large red 'X' is drawn over the entire code block, indicating that this code is unsafe and will not compile.

Will not compile

To fix: use **AtomicU32**, **Mutex**, etc.

Cargo and crates.io



The screenshot shows the Cargo website (crates.io) with a dark green header. The header contains the Cargo logo (a stack of yellow crates), the word "CARGO" in white, a search bar, and links to "Browse All Crates" and "Log in with GitHub". A purple banner in the top right corner says "Fork me on GitHub".

The main content area has a light beige background. It features the heading "The Rust community's crate host" in a large, bold, dark font. Below this heading are two yellow buttons: "Install Cargo" (with a download icon) and "Getting Started" (with a flag icon).

Below the buttons is a horizontal line. To the left of the line, there is a paragraph: "Instantly publish your crates and install them. Use the API to interact and find out more information about available crates. Become a contributor and enhance the site with your work." To the right of the line, there are two statistics: "21,952,379 Downloads" (with a download icon) and "3,969 Crates in stock" (with a crate icon).

Below the statistics are three columns: "New Crates", "Most Downloaded", and "Just Updated". Each column has a list of crates. The first crate in each column is highlighted with a green arrow icon.

New Crates	Most Downloaded	Just Updated
leaky-cow (0.1.1)	libc (0.2.6)	capnp (0.6.2)

Open and welcoming

Rust has been **open source** from the beginning.

Open governance model based on **public RFCs**.

We have an **active, amazing community**.



Hello, world!

Hello, world!

```
fn main() {  
    println!("Hello, world!");  
}
```


Exercise: **hello world**

[**http://rust-tutorials.com/exercises/**](http://rust-tutorials.com/exercises/)

Ownership

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

Borrow

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



Ownership

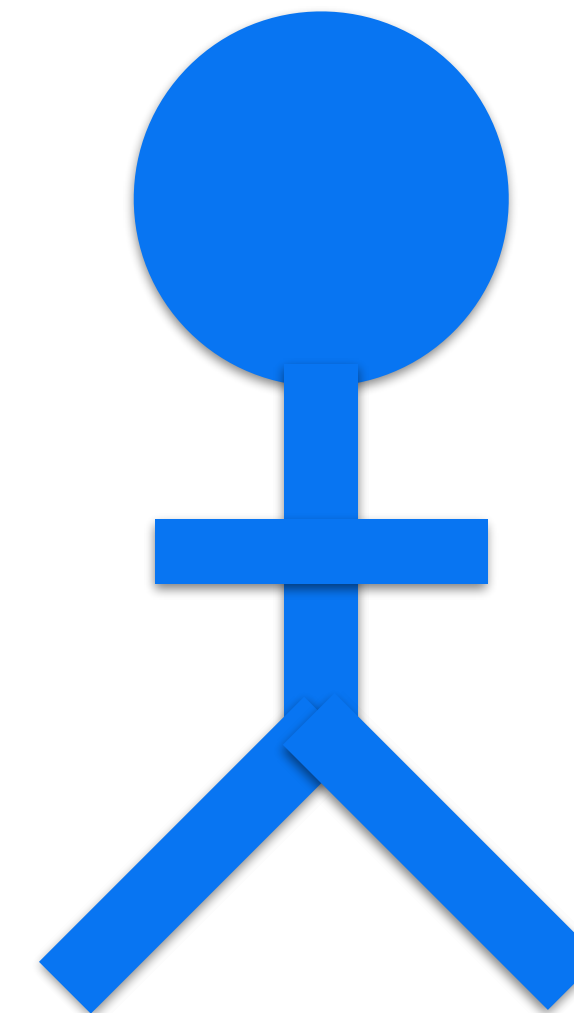
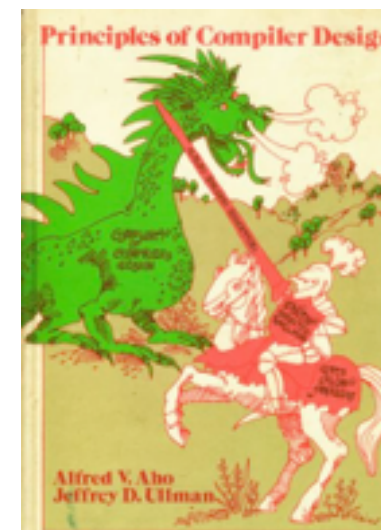
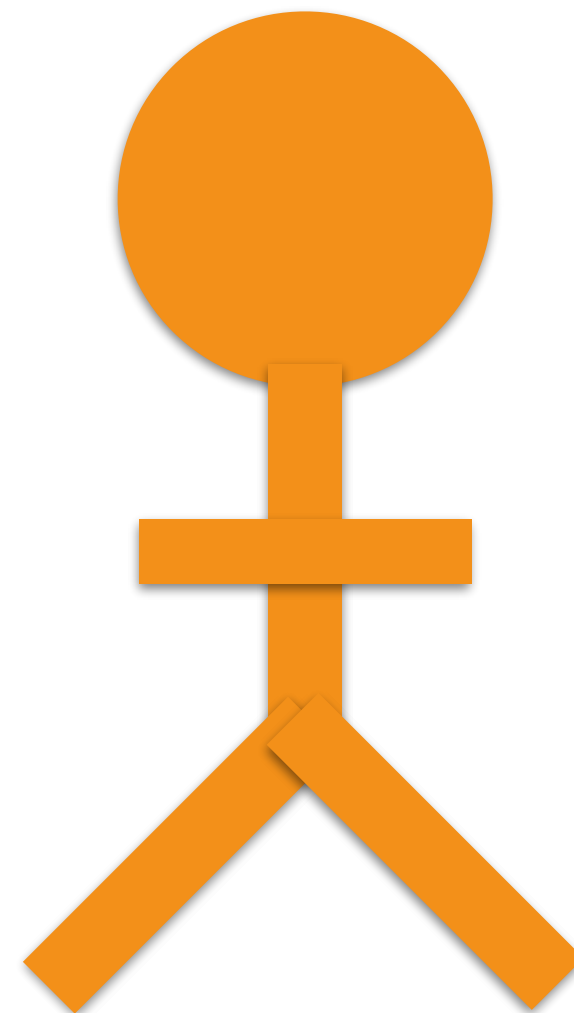
```
fn main() {  
  let name = format!("{}", ...);  
  helper(name);  
  helper(name);  
}
```

↑

Error: use of moved value: `name`

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

↑
Take ownership
of a String

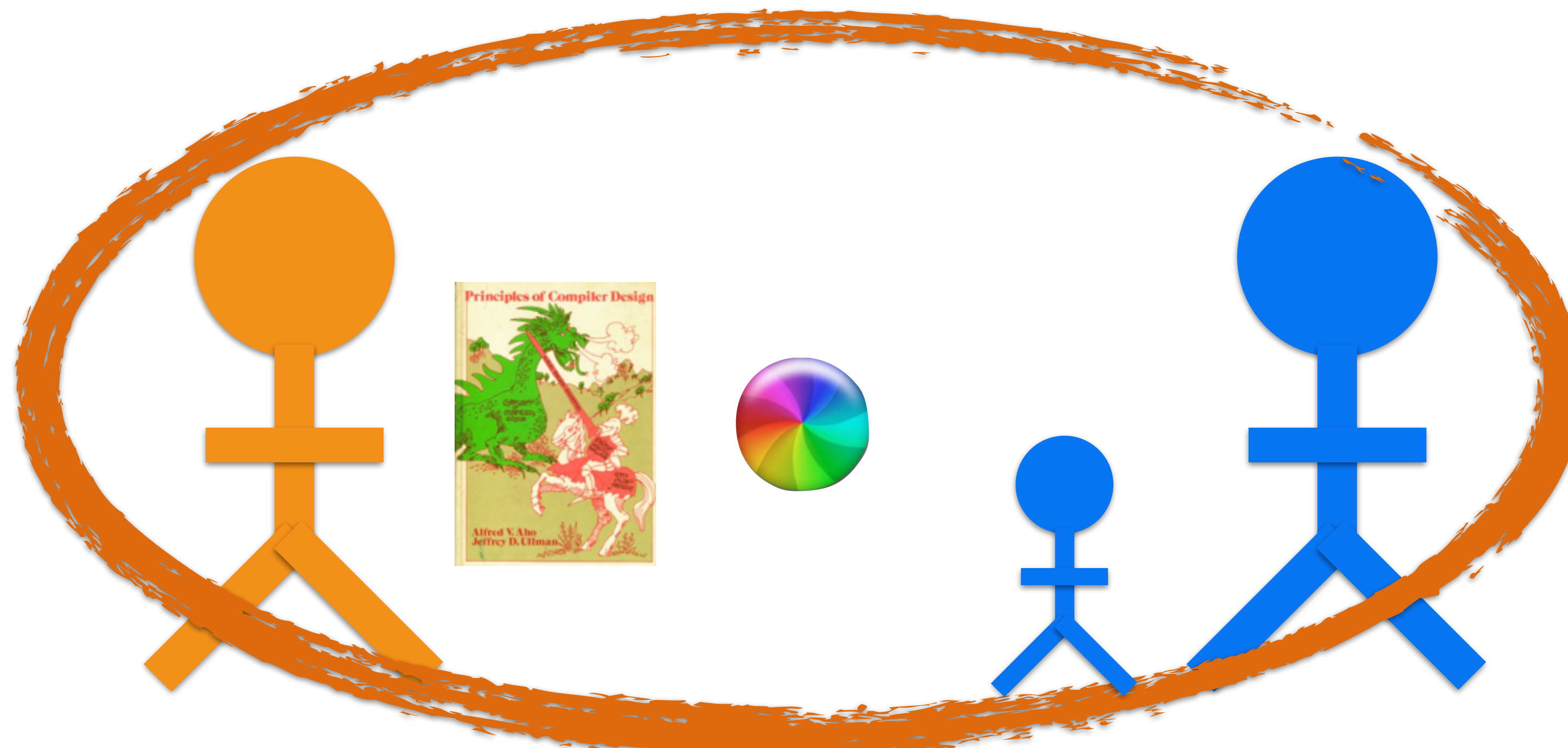


Ownership


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

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

Take **reference**
to Vector



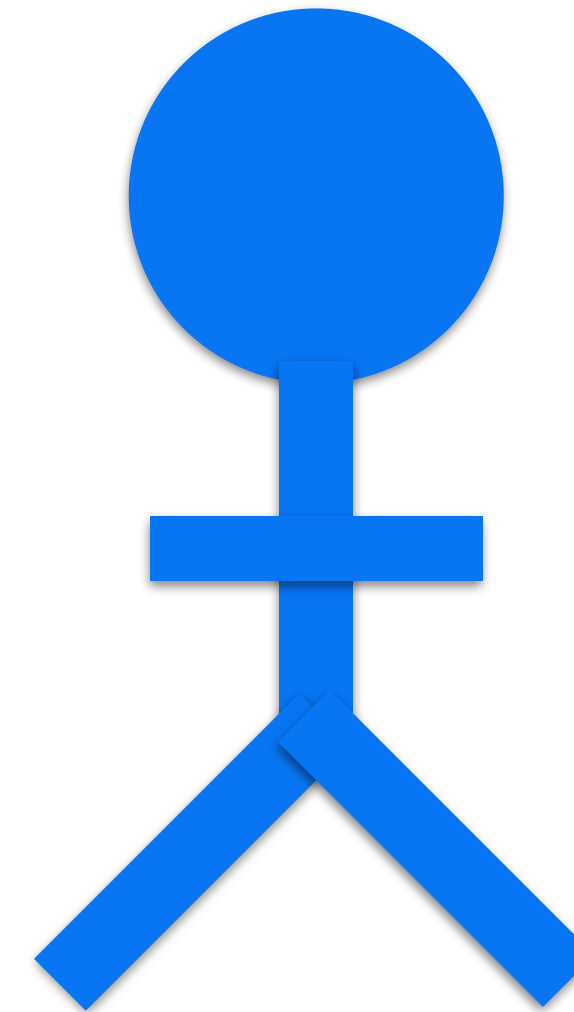
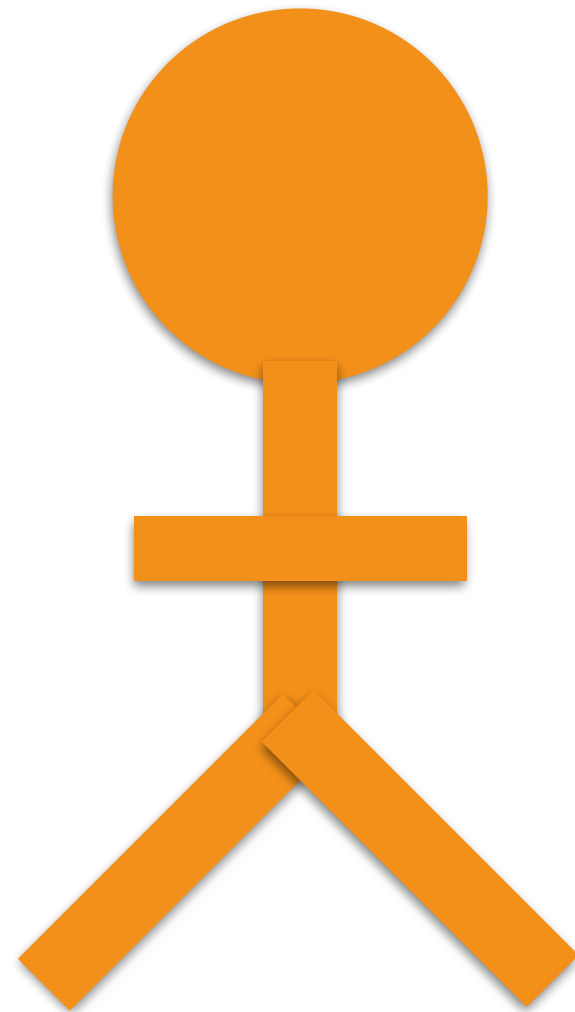
“Ownership” in Java

Clone

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

Copy the String

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

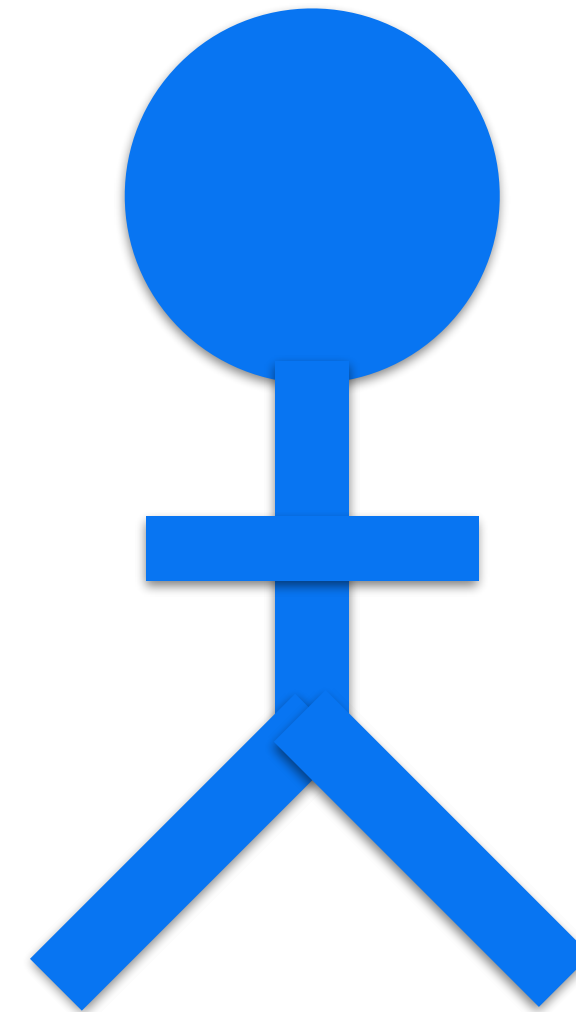
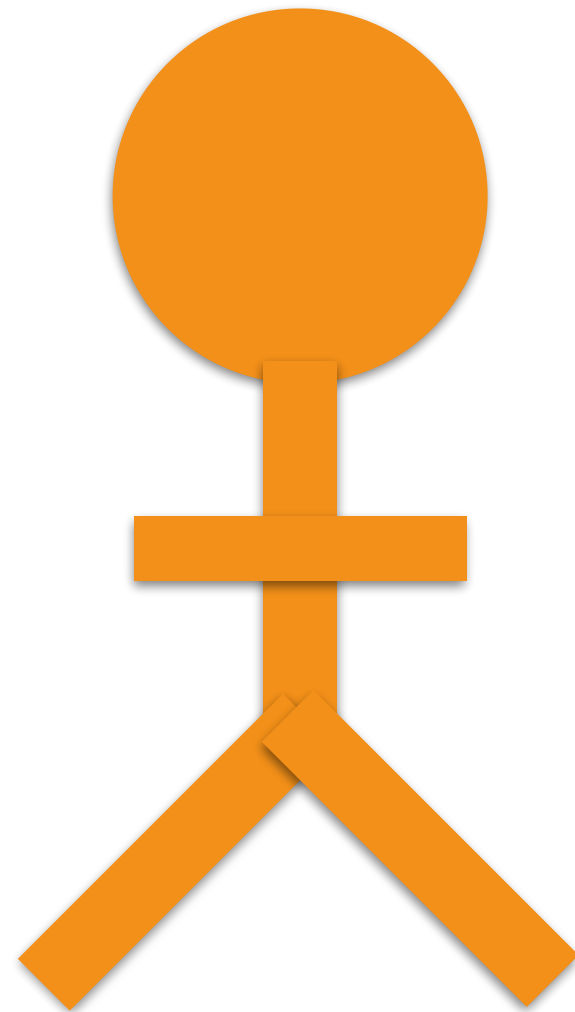


Copy (auto-Clone)

```
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*

Exercise: **ownership**

<http://rust-tutorials.com/exercises/>

Cheat sheet:

```
fn helper(name: String) // takes ownership
```

```
string.clone()           // clone the string
```

```
http://doc.rust-lang.org/std
```



Borrowing: Shared Borrows


```

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

```

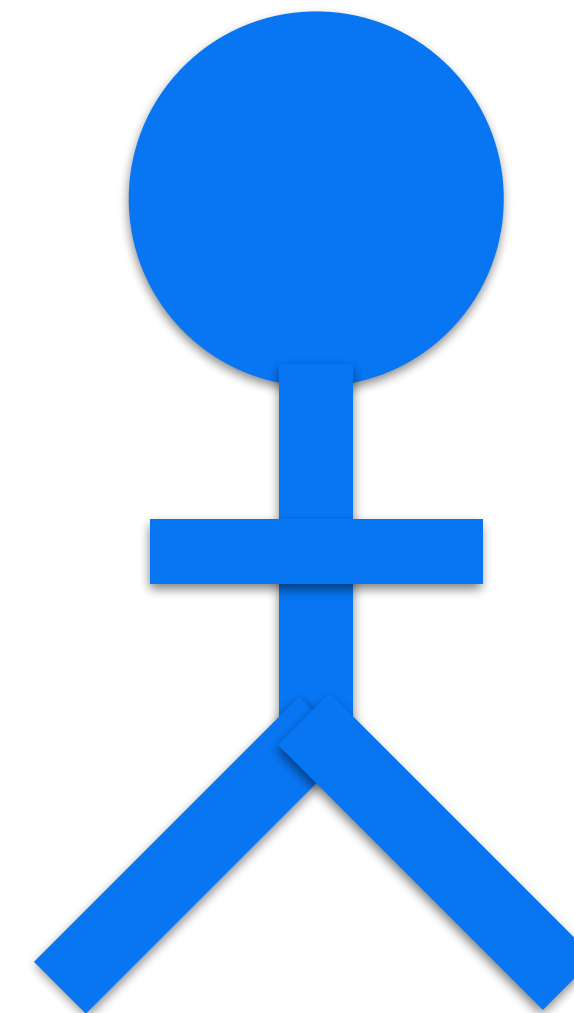
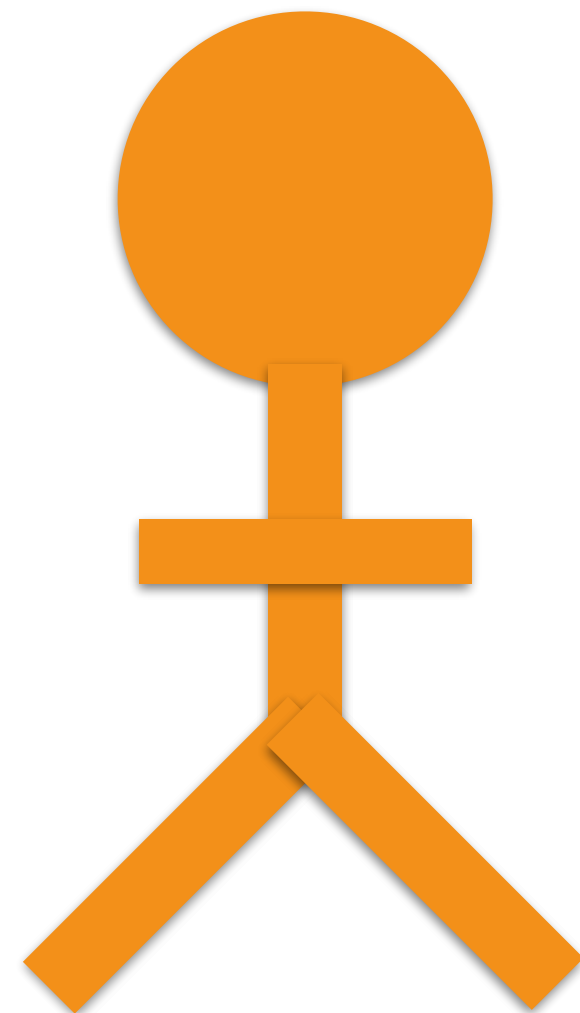
Lend the string,
creating a reference

```

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

```

Change type to a
reference to a String



Shared borrow

Shared == Immutable^{*}

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

← **OK.** Just reads.

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

← **Error.** Writes.

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

^{*} **Actually:** mutation only in **controlled circumstances**.

Play time



Waterloo, Cassius Coolidge, c. 1906

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

Lend some of
the string

Lend entire string

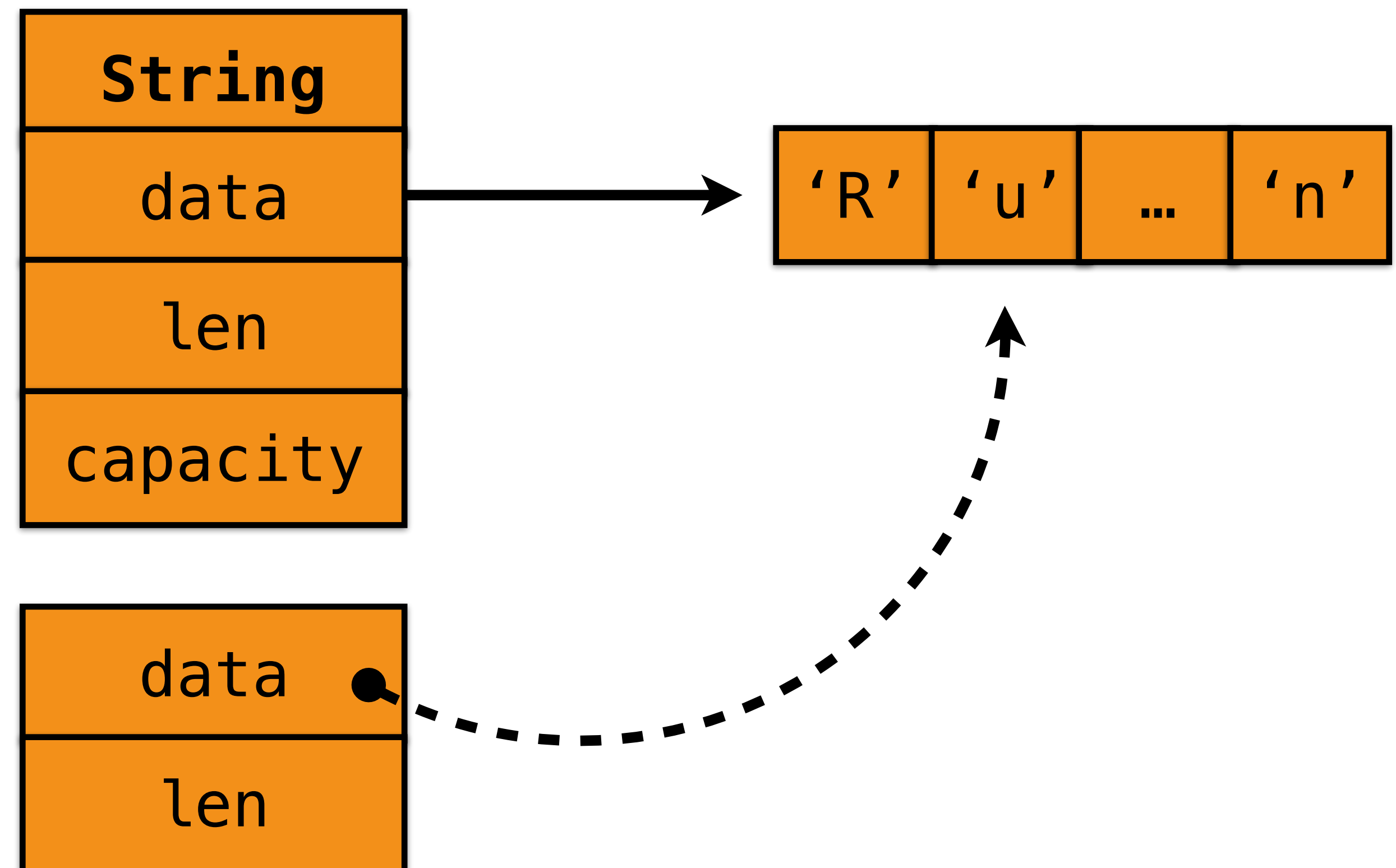
Looks like other languages:

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

But no copying at runtime.

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

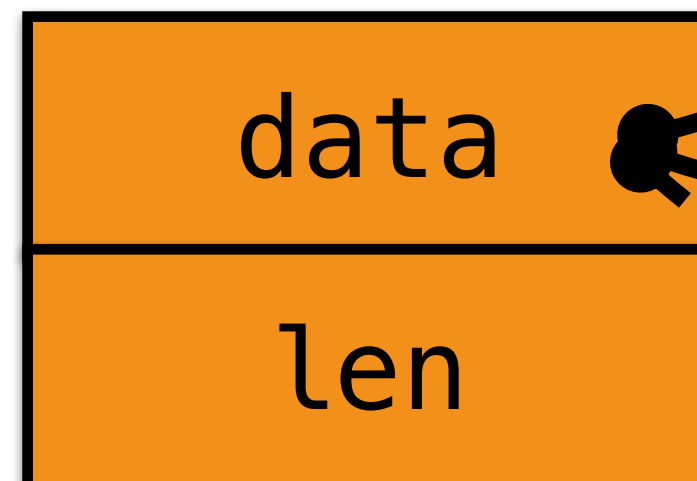
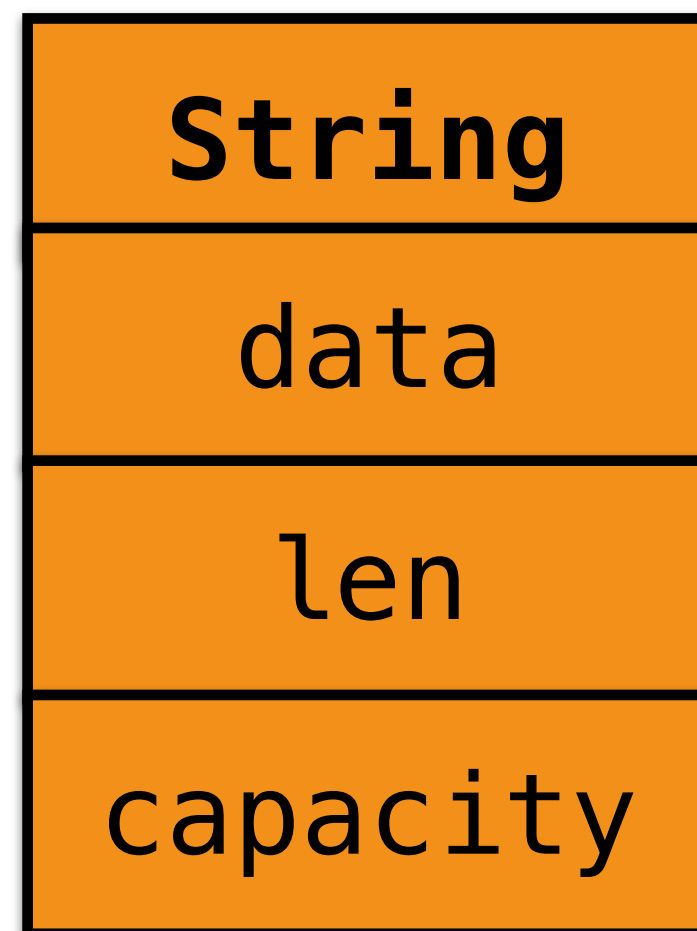
Change type from `&String`
to a **string slice**, `&str`



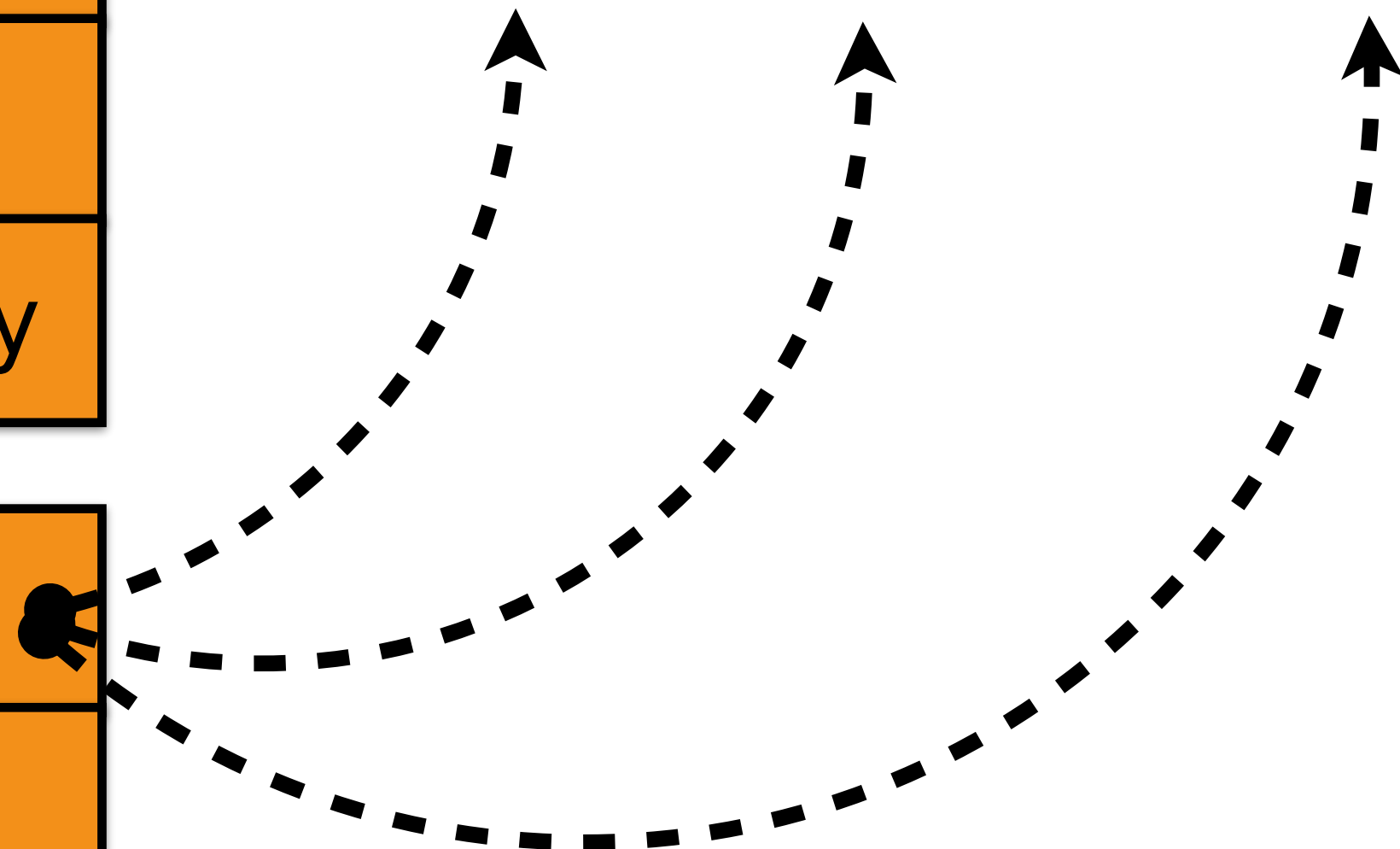
High-level code, low-level efficiency

```
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.

Exercise: **shared borrow**

[**http://rust-tutorials.com/exercises/**](http://rust-tutorials.com/exercises/)

Cheat sheet:

```
&String      // type of shared reference  
&str         // type of string slice
```

```
fn greet(name: &String) {...}
```

```
&name        // shared borrow  
&name[x..y]   // slice expression
```

<http://doc.rust-lang.org/std>



Borrowing: Mutable Borrows

```

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

```

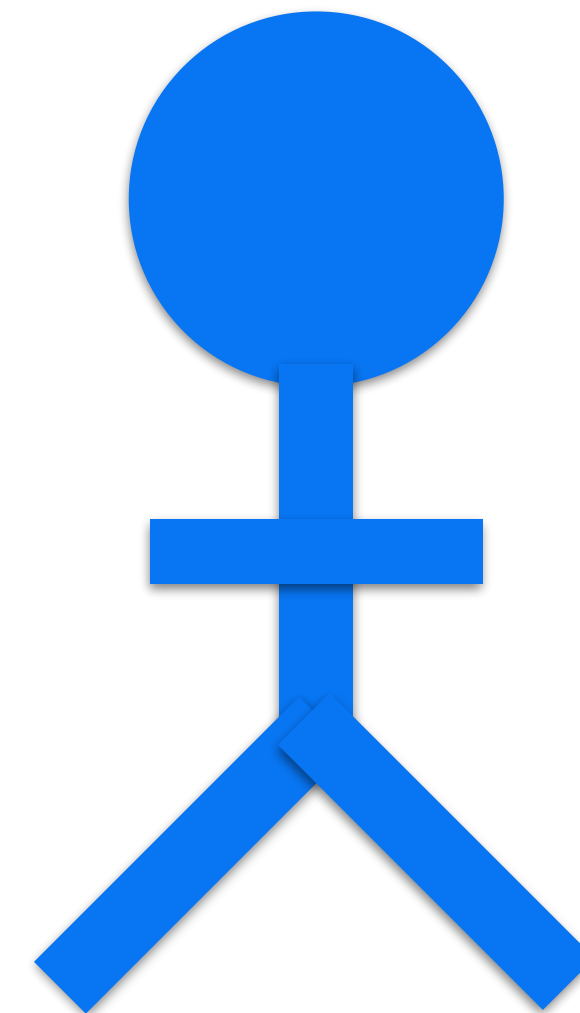
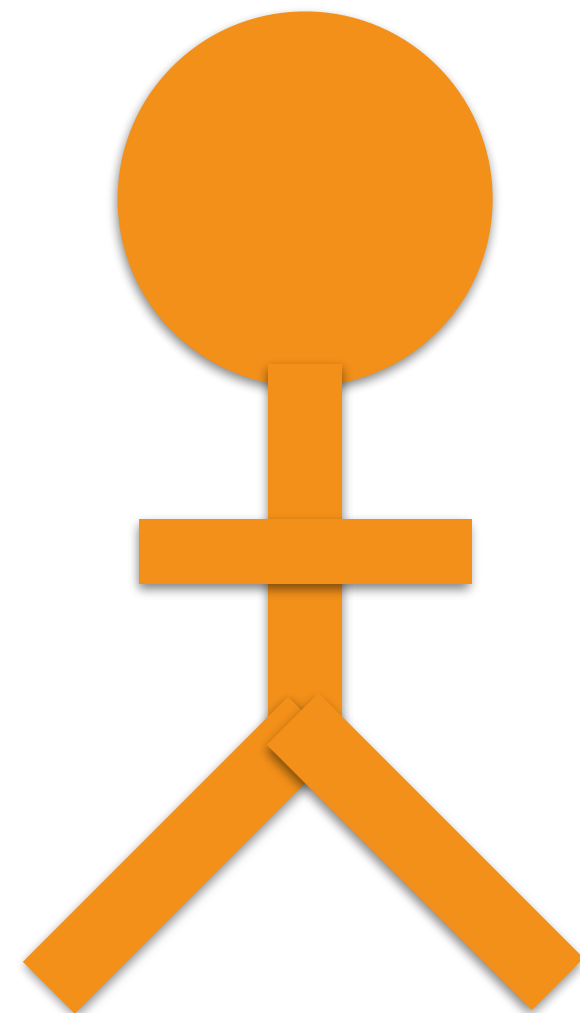
Lend the string
mutably Prints the
 updated string.

```

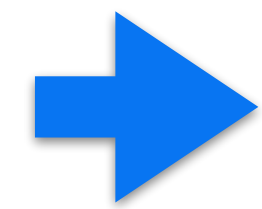
fn update(name: &mut String) {
    name.push_str("...");
}

```

Take a **mutable**
reference to a String
 Mutate string
 in place



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

Play time

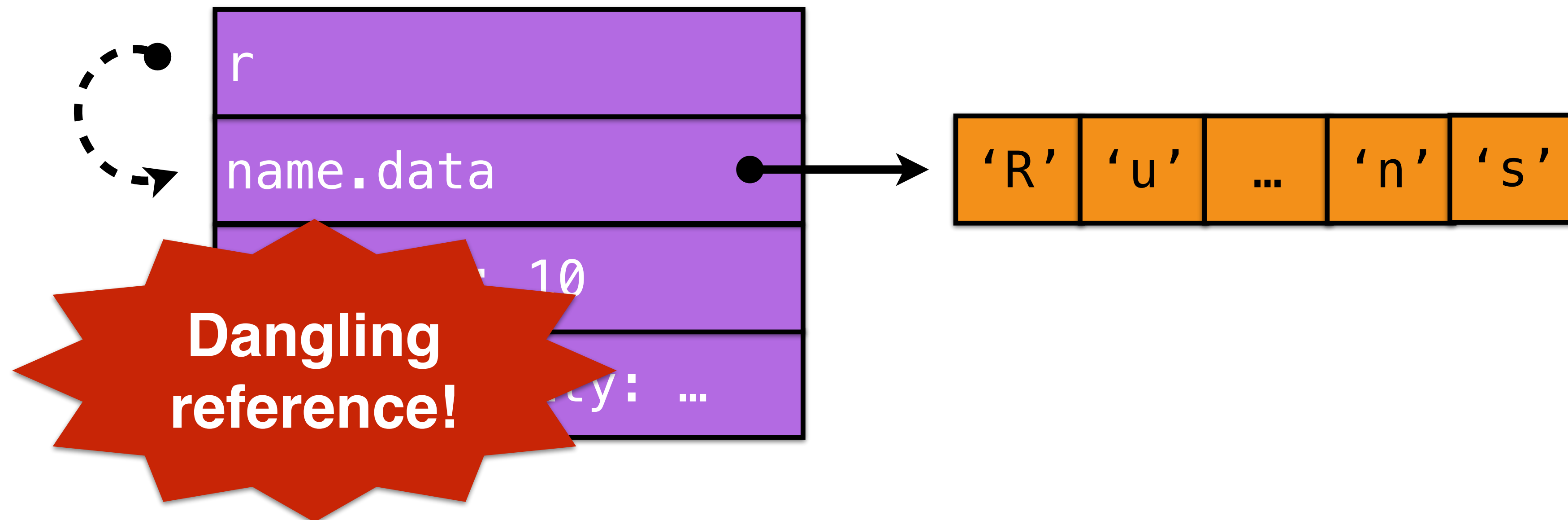


Waterloo, Cassius Coolidge, c. 1906

How do we get safety?



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




```

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

```

Diagram illustrating variable lifetimes and borrowing in Rust. The code defines a function `main` with a variable `r` and a nested block. Inside the block, `name` is created and `r` is updated to point to `name`. Red arrows show the lifetime of `r` (from its declaration to the end of the function) and the lifetime of `name` (from its creation to the end of the block). A blue highlight is under `&name` in the assignment `r = &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;
          ^~~~

```

```
use std::thread;
```

``name`` can only be
used within this fn

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

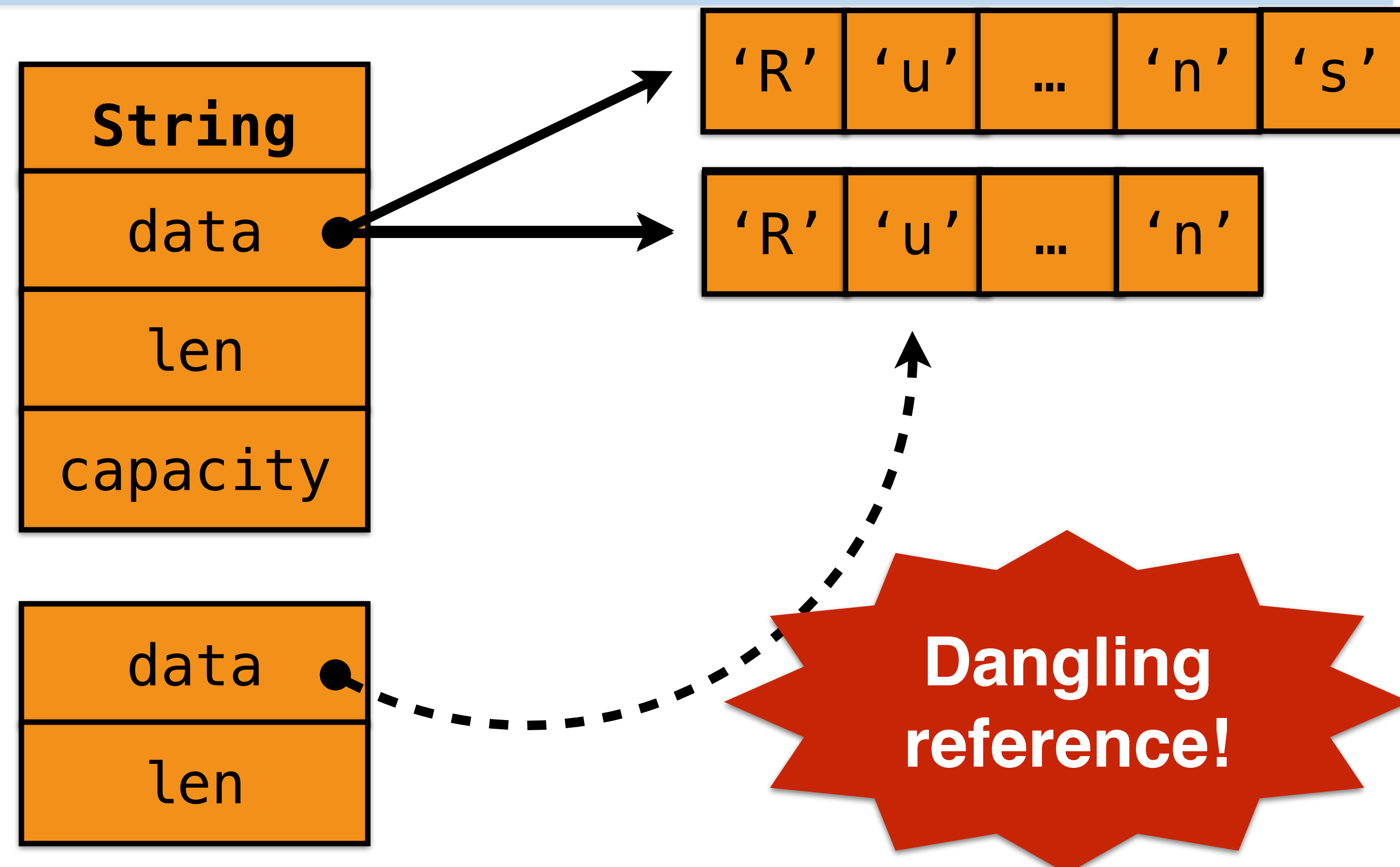
However: see **rayon**,
crossbeam, etc on crates.io

```
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);
```



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.

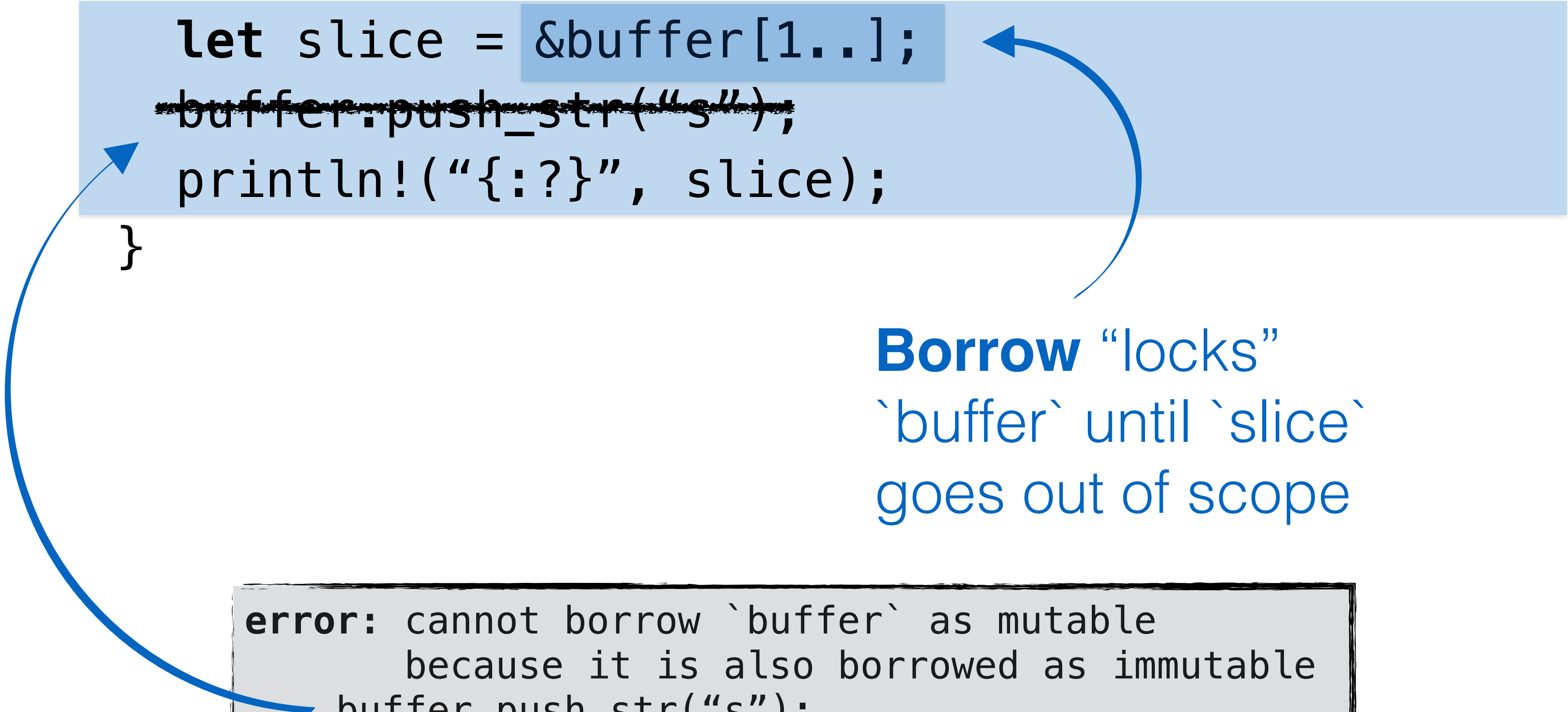
Creating a mutable reference to X “**writes locks**” X.

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

Never have a reader/writer at same time.

Dangers of mutation

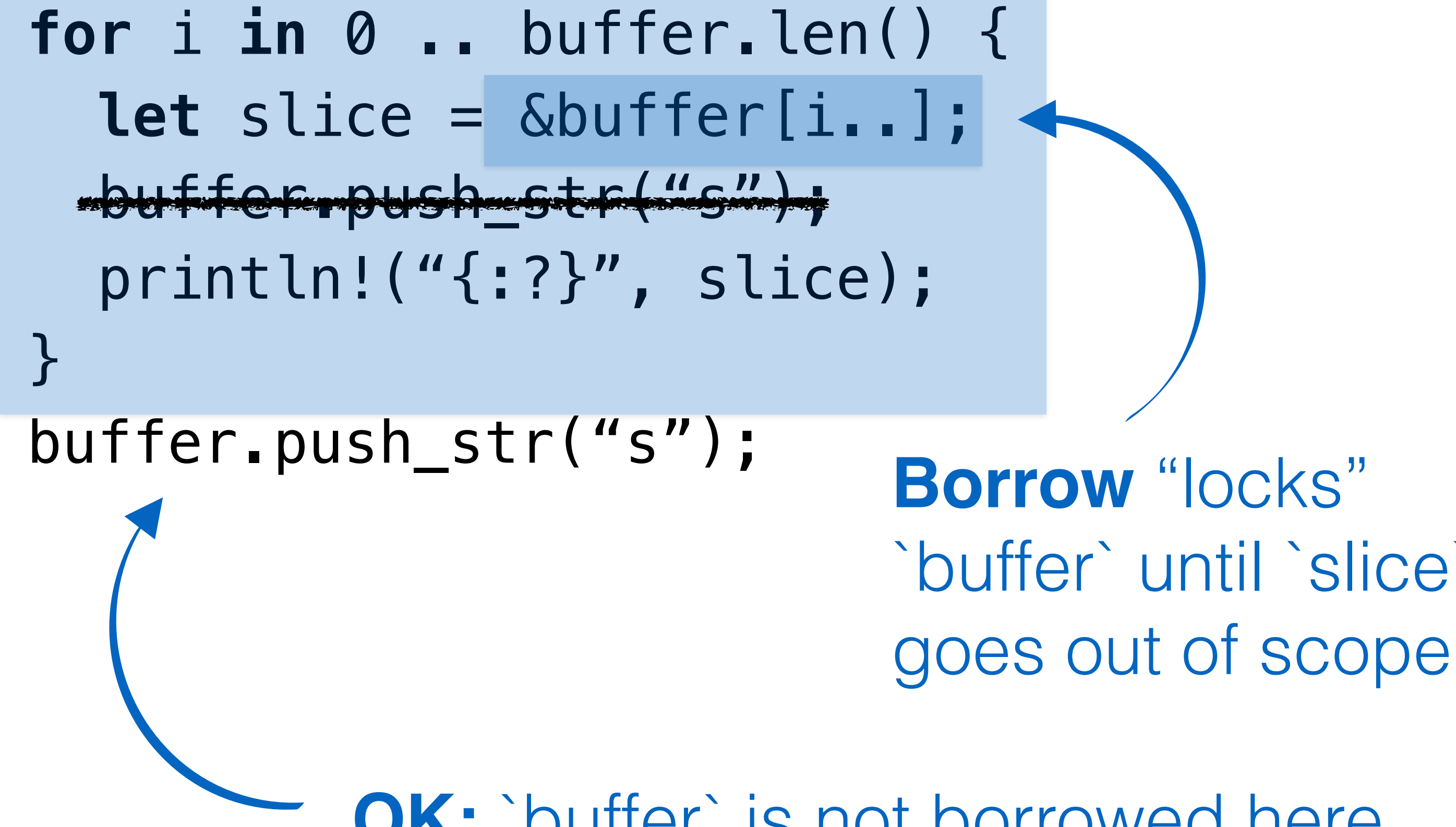
```
fn main() {  
    let mut buffer: String = format!("Rustacean");  
    let slice = &buffer[1..];  
    buffer.push_str("s");  
    println!("{:?}", slice);  
}
```



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

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

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



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

OK: `buffer` is not borrowed here

Exercise: **mutable borrow**

<http://rust-tutorials.com/exercises/>

Cheat sheet:

```
&String          // type of shared reference  
&mut String      // type of mutable reference  
&str             // type of string slice
```

```
fn greet(name: &String) {..  
fn adjust(name: &mut String) {..}
```

```
&name            // shared borrow  
&mut name        // mutable borrow  
&name[x..y]      // slice expression
```

<http://doc.rust-lang.org/std>

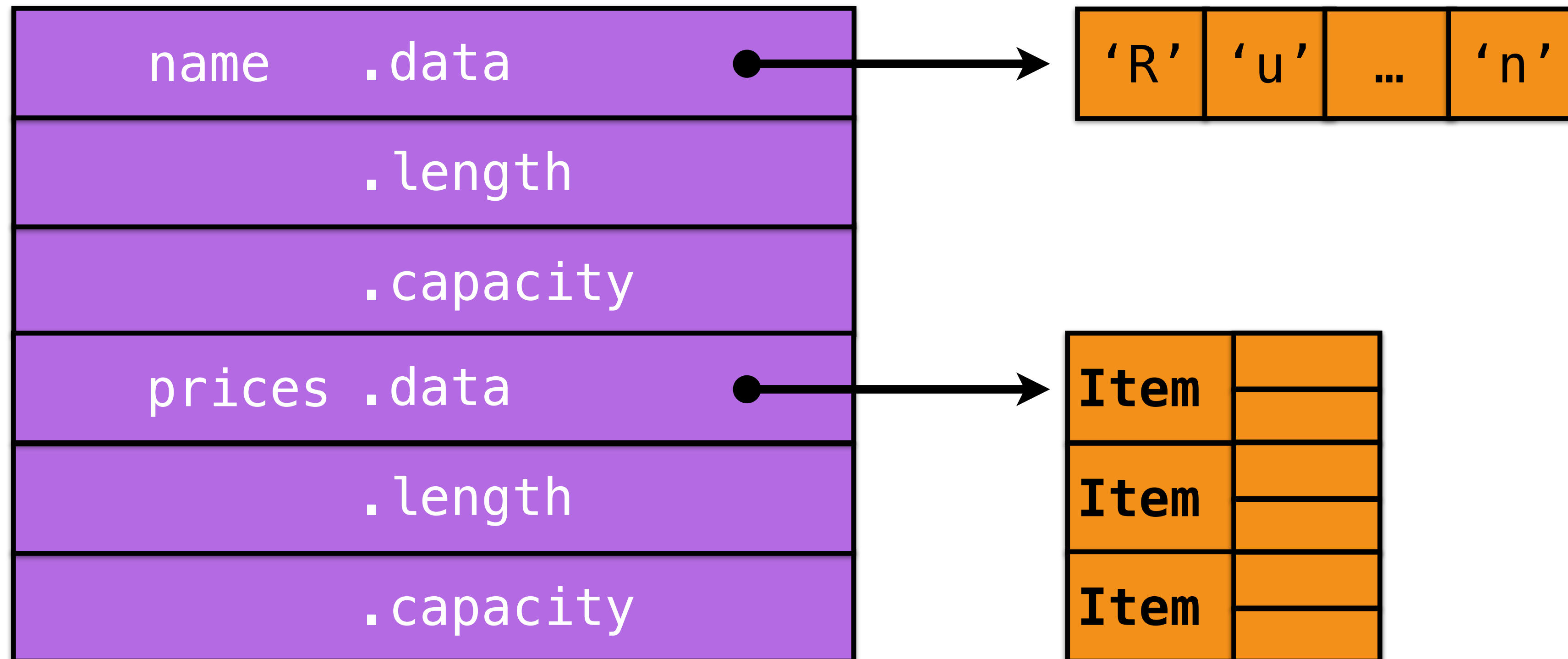
Structs and such



Shop till you drop!

Declaring a structure

```
struct Store {  
    name: String,  
    prices: Vec<Item>,  
}
```



```
struct Item {  
    name: String,  
    price: f32,  
}
```

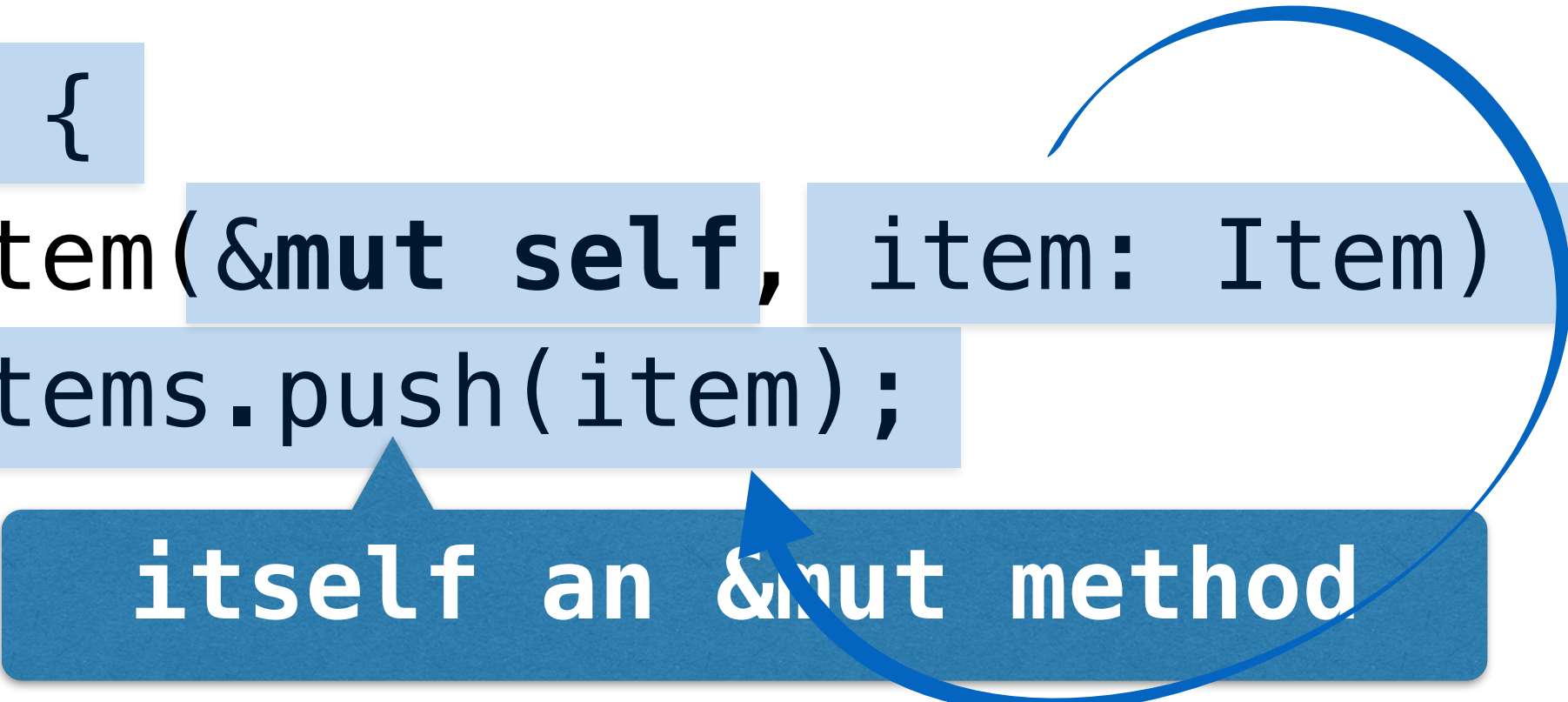
Other fundamental types

f32 f64	i8 i16 i32 i64 isize	u8 u16 u32 u64 usize	&str &[T]
floats	signed	unsigned	slices

Methods

```
struct Store { .. }  
struct Item { .. }
```

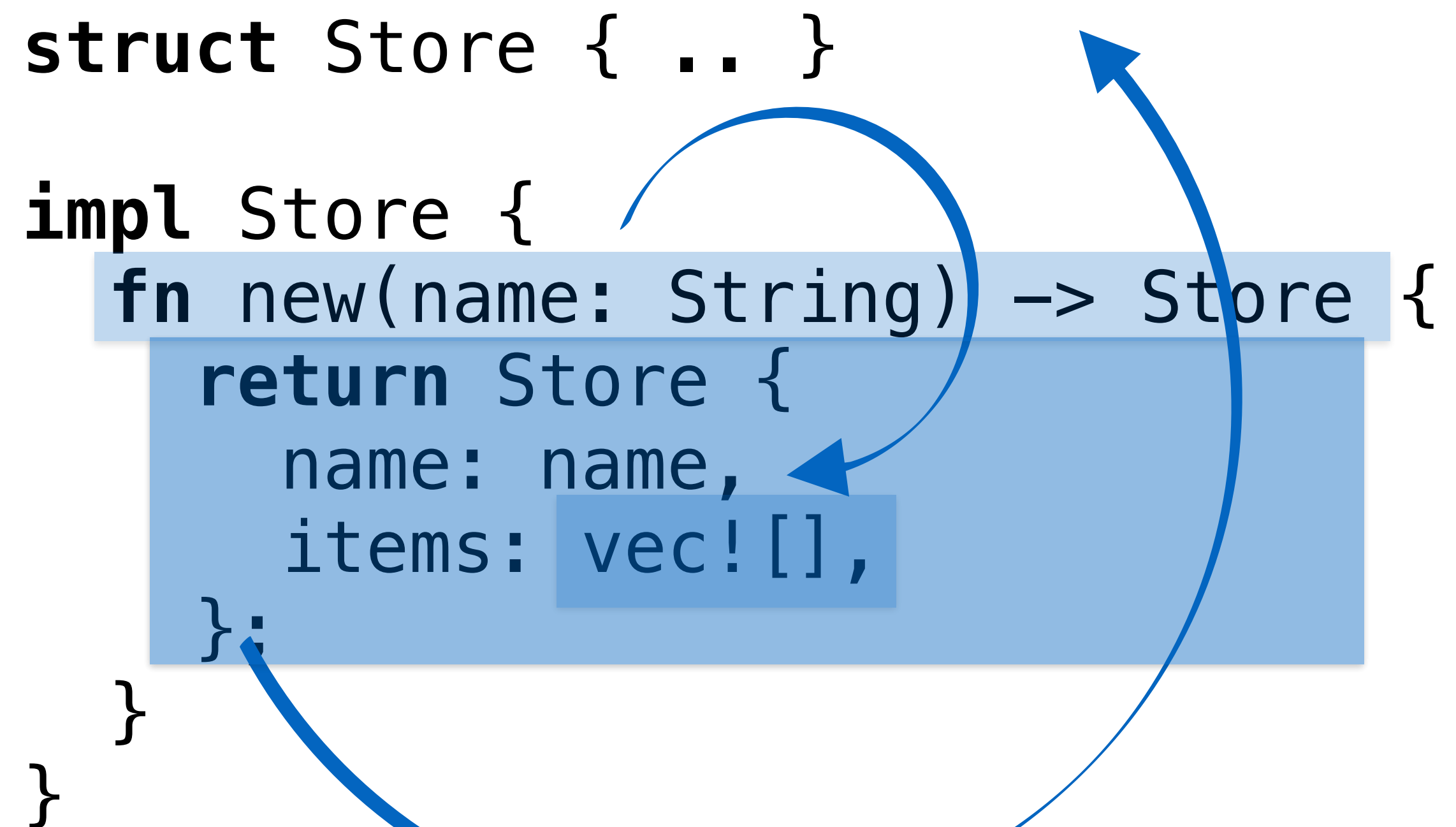
```
impl Store {  
    fn add_item(&mut self, item: Item) {  
        self.items.push(item);  
    }  
  
    fn price(&self, item_name: &str) -> f32 {  
        ... // see upcoming slide  
    }  
}
```



```
store.add_item(...); // must be let mut  
store.price(...);    // let OR let mut
```


Methods

```
struct Store { .. }  
  
impl Store {  
  fn new(name: String) -> Store {  
    return Store {  
      name: name,  
      items: vec![],  
    };  
  }  
}
```



```
Store::new(some_name)
```

Return is optional

```
struct Store { .. }  
  
impl Store {  
    fn new(name: String) -> Store {  
        Store {  
            name: name,  
            items: vec![],  
        }  
    }  
}
```

No `;` on last expression:
“return this value”

Options and Enums

```
enum Option<T> {  
    Some(T),  
    None  
}
```

```
fn main() {  
    use Option::*;  
    let v: Option<i32> = Some(22);  
    match v {  
        Some(x) => println!("v = {}", x),  
        None => println!("v = None"),  
    }  
    println!("v = {}", v.unwrap()); // risky  
}
```


For Loops

```
fn main() {  
    let v = vec![format!("Alpha"),  
                  format!("Beta"),  
                  format!("Gamma")];  
    for s in v {  
        println!("{:?}", s);  
    }  
}
```

String

Vec<String>

v:

"Alpha"

"Beta"

"Gamma"

s:

For Loops

```
fn main() {  
    let mut v = vec![format!("Alpha"),  
                     format!("Beta"),  
                     format!("Gamma")];  
  
    for s in &v {  
        println!("{:?}", s);  
    }  
  
    for s in &mut v {  
        s.push_str(".");  
    }  
}
```

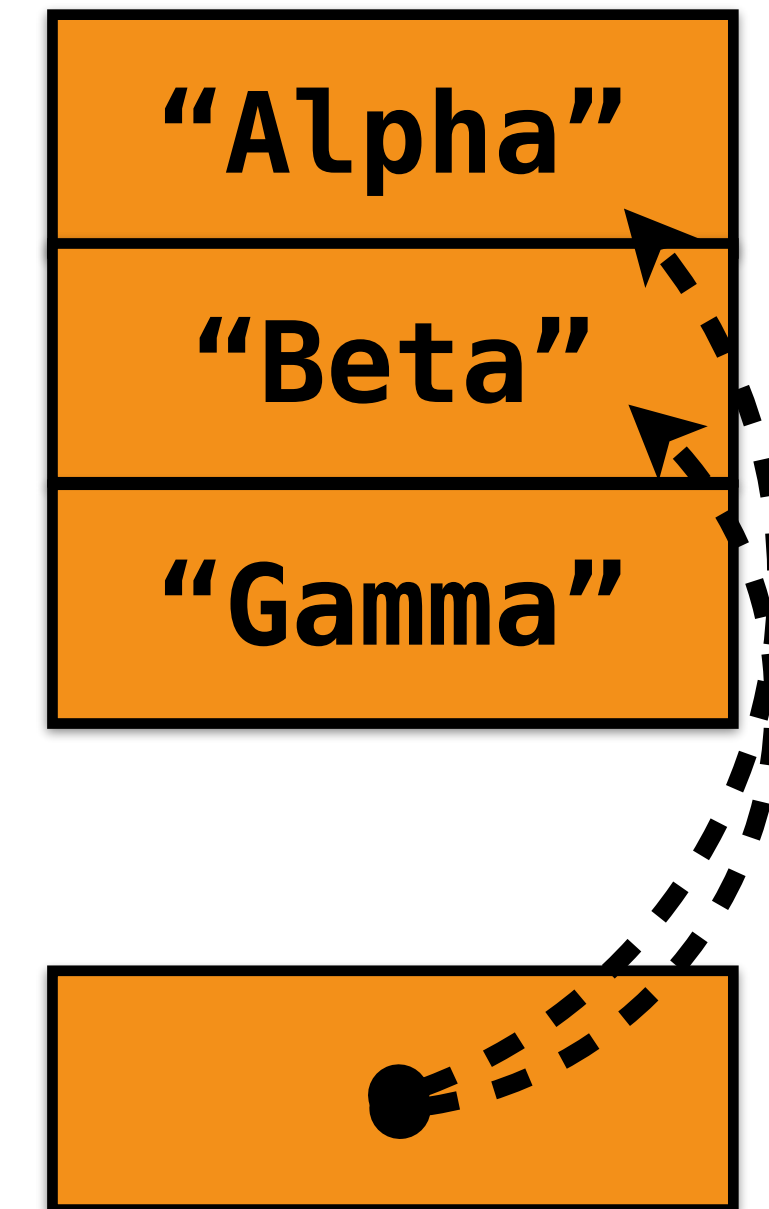
&String

&Vec<String>

&mut String

&mut Vec<String>

s:



Exercise: **structs**

<http://rust-tutorials.com/exercises>

Implement

```
fn total_price(..)
```

Cheat sheet:

```
for s in v { ... }           let mut some_var = 0.0;
```

```
for s in &v { ... }          some_var += x;
```

```
while ... { ... }            println!("{:?}", s);
```

<http://doc.rust-lang.org/std>



Thanks for listening!