Beginner Tutorial: Ownership and Borrowing

Steve Klabnik


Hack without fear!

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

Performance

Ruby:
964K iter/sec

```c
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

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

```rust
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()
        .map(|path| {
            Image::load(path)
        })
        .collect()
}
```

For each path…

…load an image…

…create and return a vector.

Parallel Programming

```
extern crate rayon;
```

Third-party library

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

…make it parallel.

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

Safer Parallel Programming

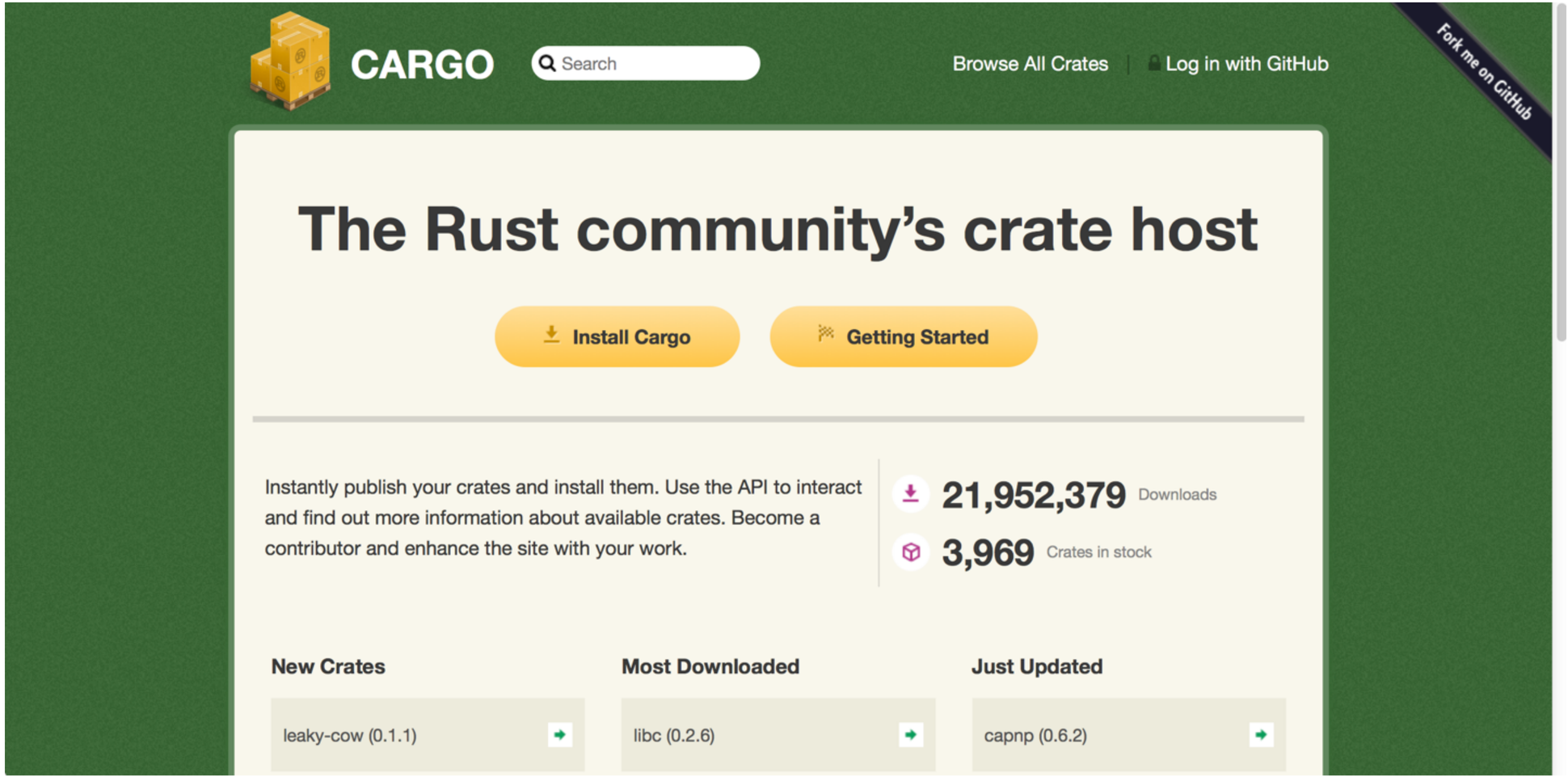
```
fn load_images(paths: &[PathBuf]) -> Vec<Image> {
    let mut jpegs = 0;
    paths.par_iter()
        .map(|path| {
            if path.ends_with(“jpeg”) { jpegs += 1; }
            Image::load(path)
        })
        .collect();
}
```

Data-race

Will not compile

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

Cargo and crates.io



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/

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

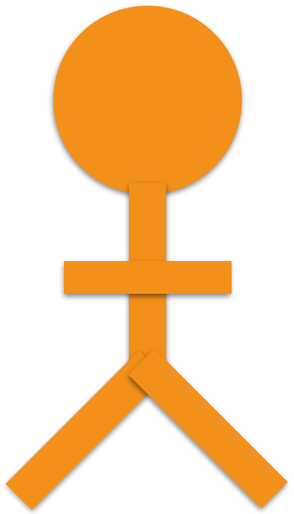
Ownership

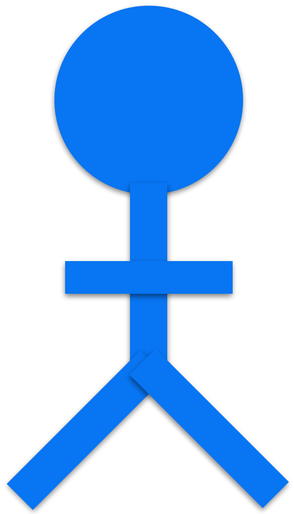
```
fn main() {
    let name = format!("…");
    helper(name);
    helper(name);
}
```

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

Take ownership of a String

Error: use of moved value: `name`





Ownership

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

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

Take **reference** to Vector

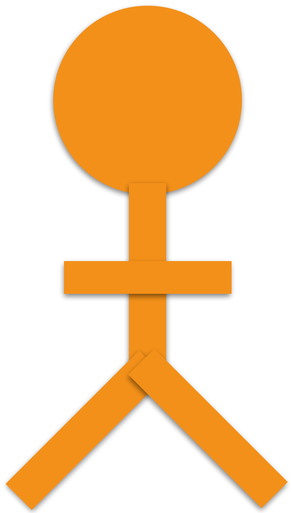
“Ownership” in Java

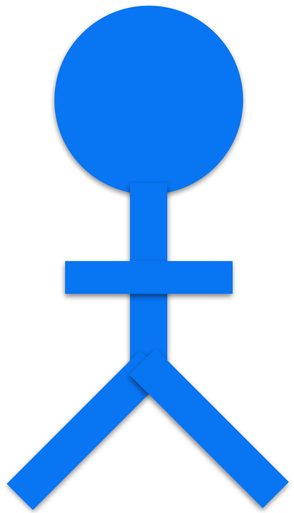
Clone

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

Copy the String

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



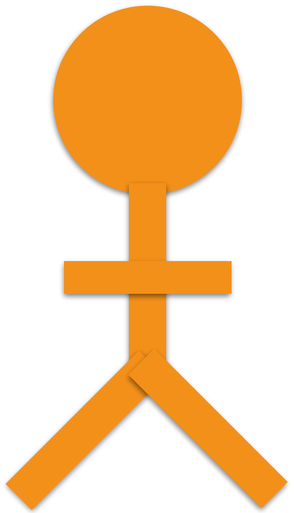


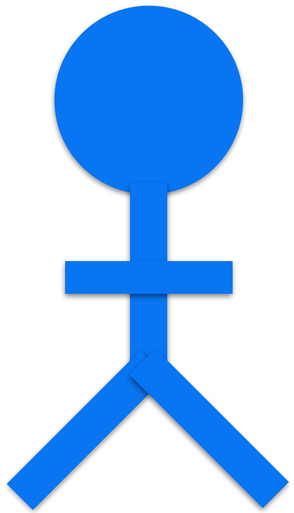
Copy (auto-Clone)

```
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

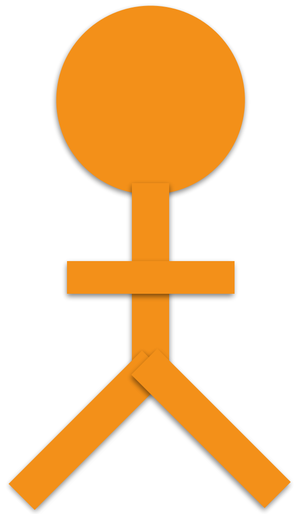
Borrowing: Shared Borrows

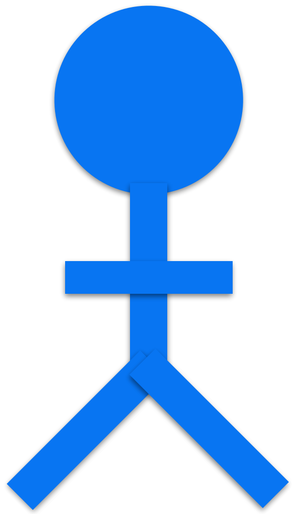
```
fn main() {
    let name = format!("…");
    let reference = &name;
    helper(reference);
    helper(reference);
}
```

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

http://is.gd/no0tTH

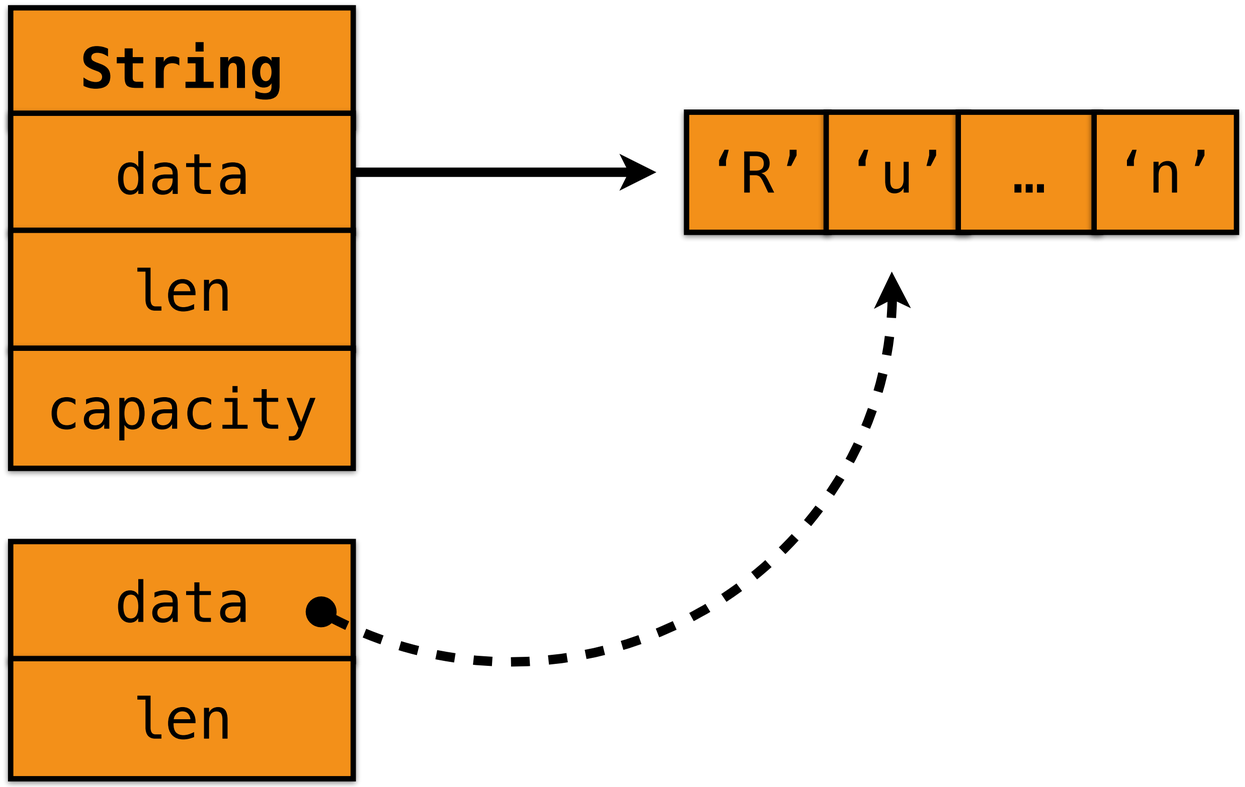
```
fn main() {
    let name = format!("…");
    helper(&name[1..]);
    helper(&name);
}
```

Lend some of the string

Lend entire string

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

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



Looks like other languages:

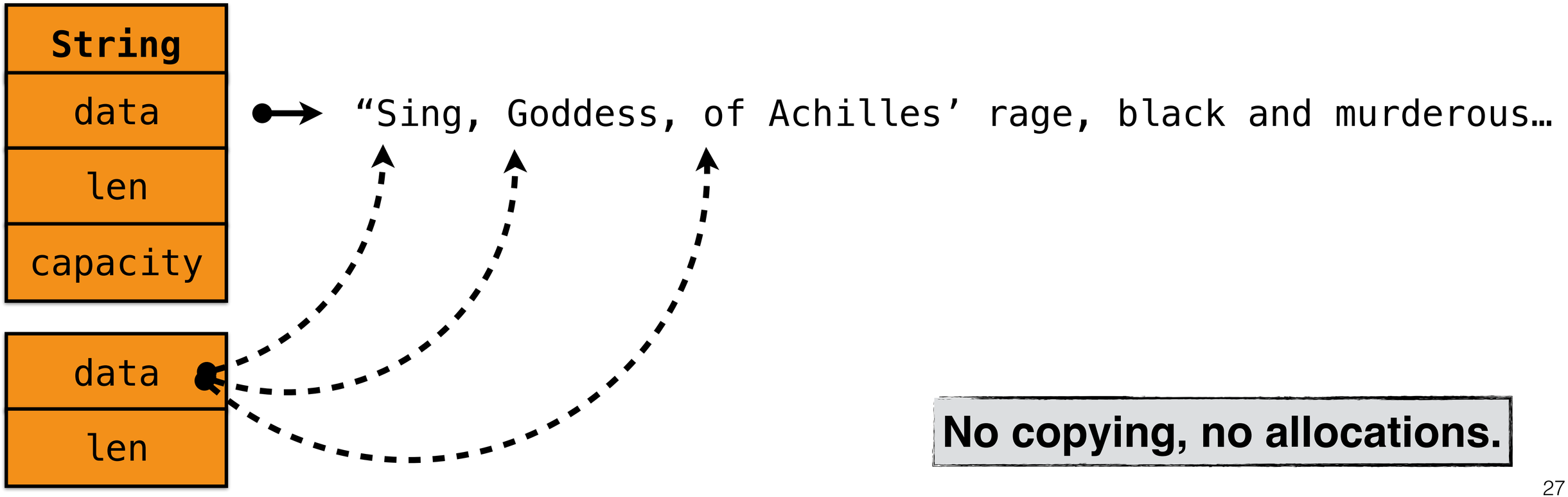
- Python: name[1:]
- Ruby: name[1..-1]

But no copying at runtime.

High-level code, low-level efficiency

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

Iterator over slices borrowed from **line**.



No copying, no allocations.

Exercise: **shared borrow**

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);
}
```

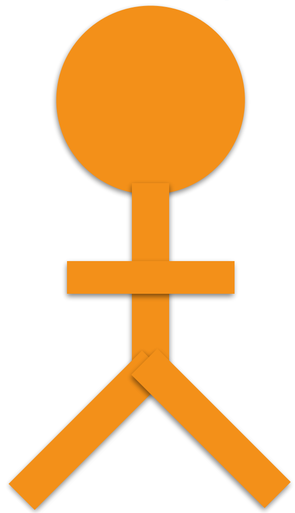
```
fn update(name: &mut String) {
  name.push_str(“…”);
}
```

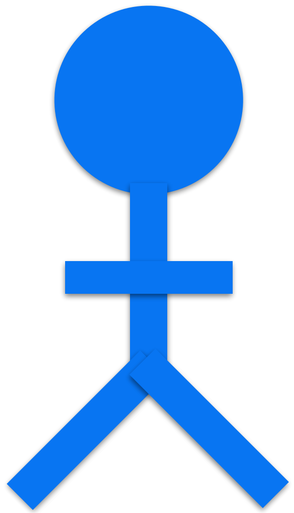
Lend the string **mutably**

Prints the updated string.

Mutate string in place

Take a **mutable reference** to a String





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

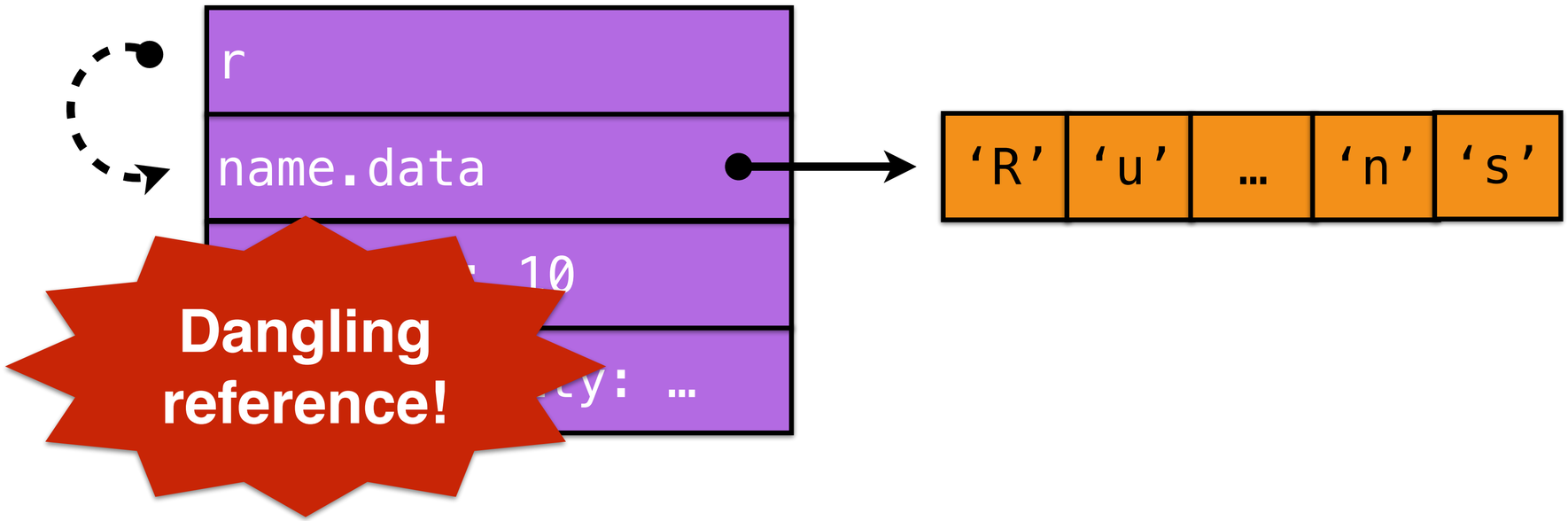
Waterloo, Cassius Coolidge, c. 1906

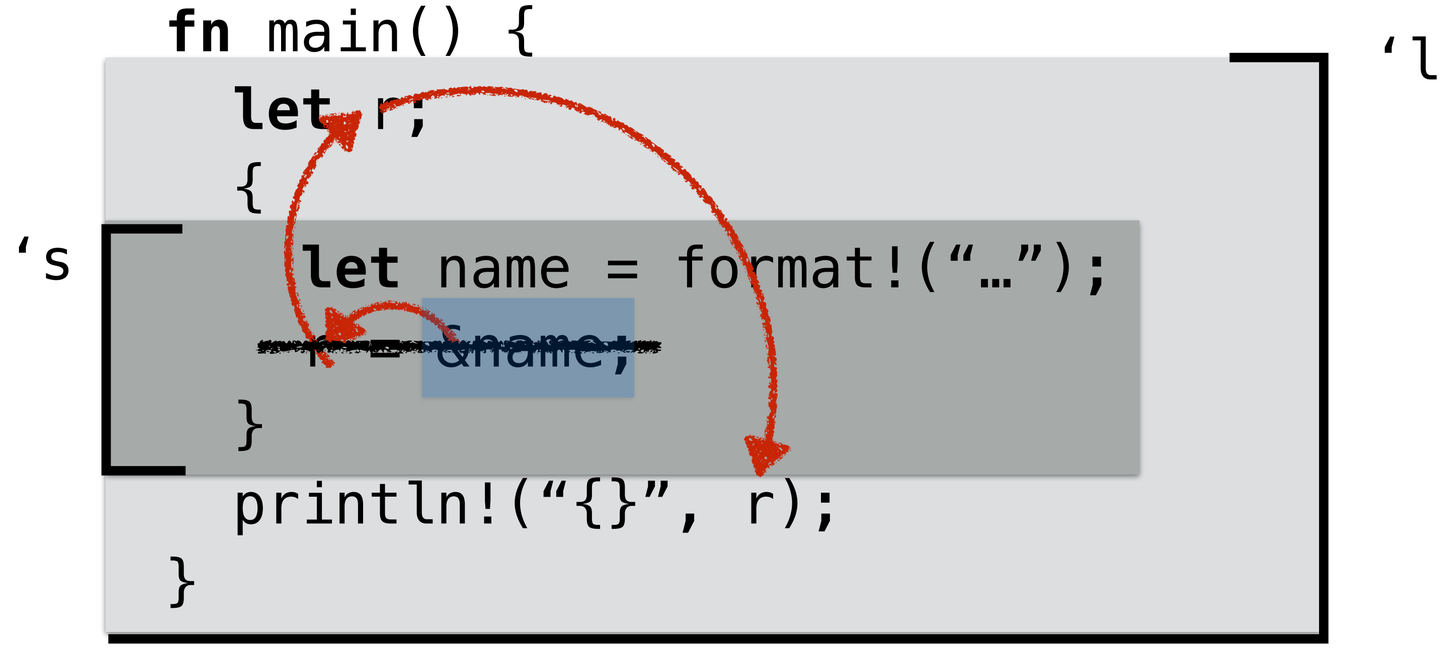
http://is.gd/no0tTH

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);
}
```





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

```
use std::thread;

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

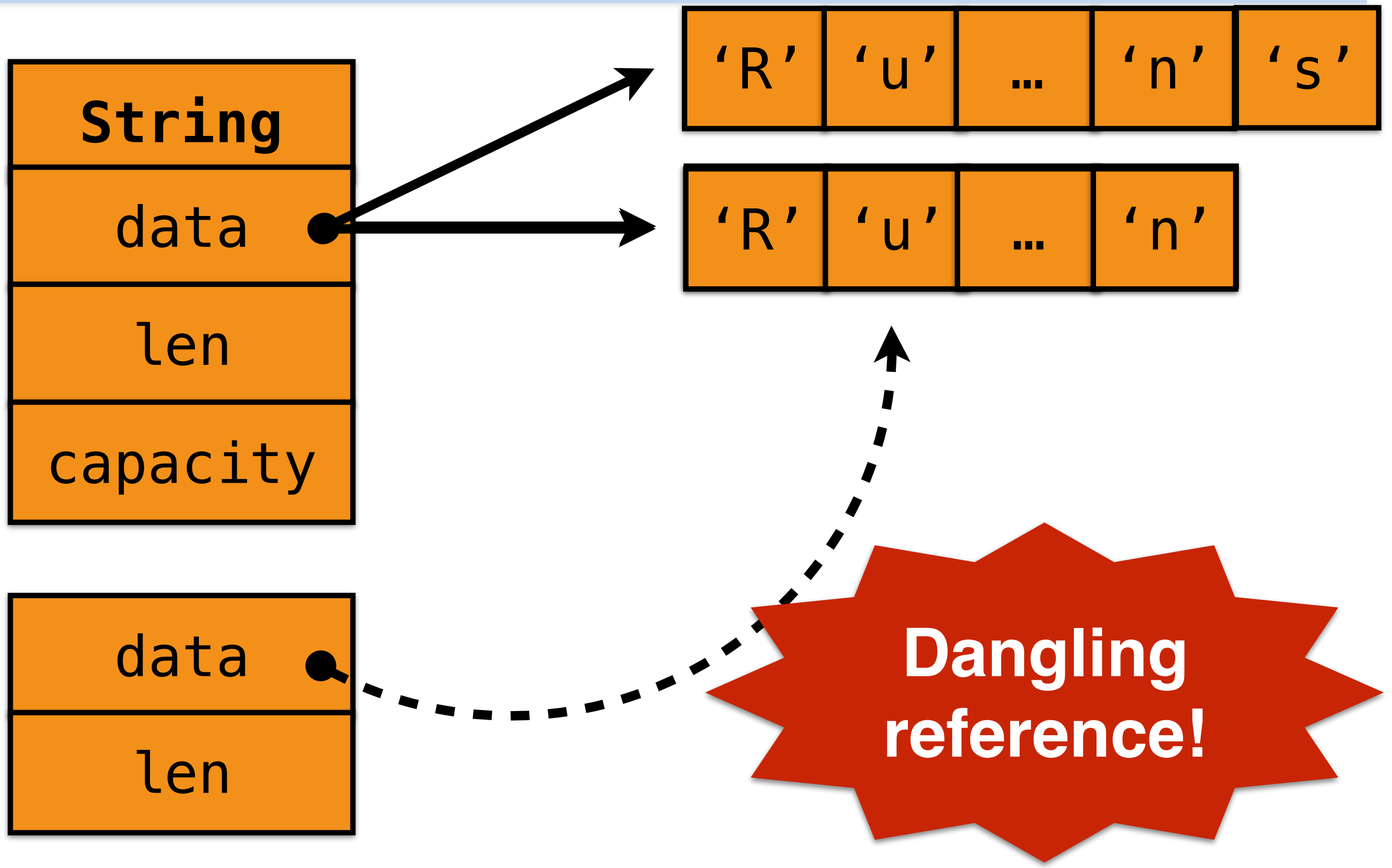
`name` can only be used within this fn

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”);
  ^~~~~~
```

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”);
}
```

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

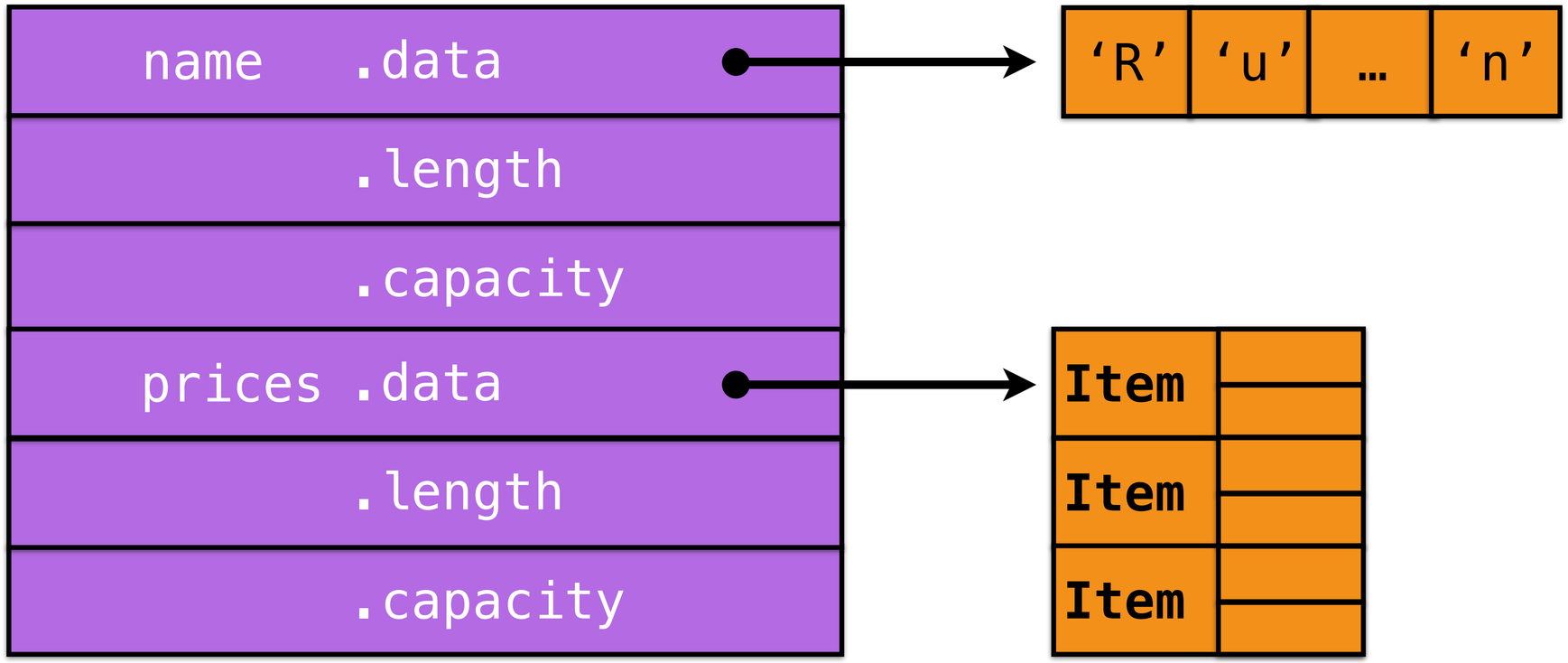
Structs and such

By QuentinUK (Own work), via Wikimedia Commons

Shop till you drop!

Declaring a structure

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



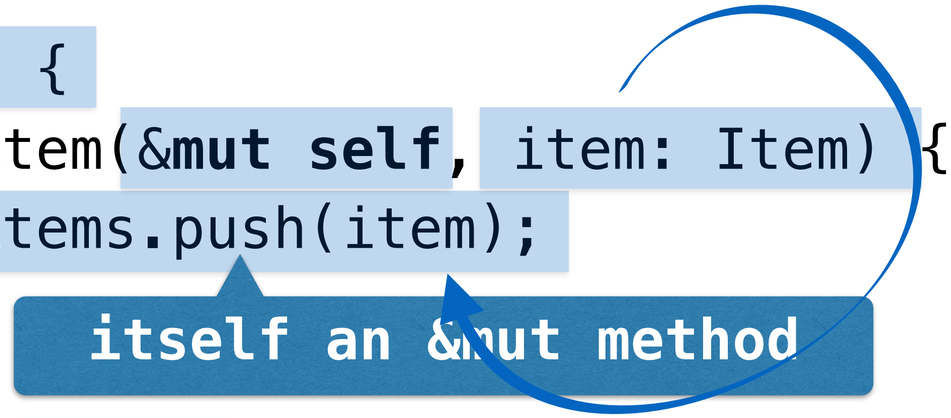
```
struct Item {
    name: String,
    price: f32,
}
```

Other fundamental types

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

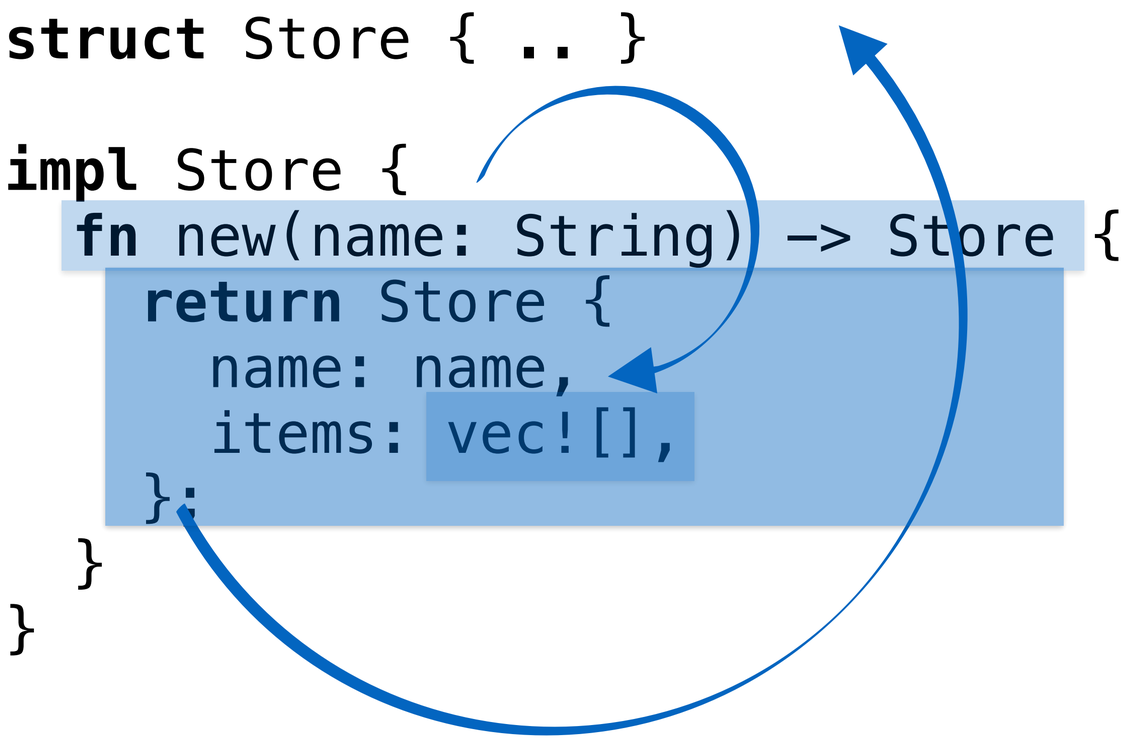
Methods

```
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




```
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>

For Loops

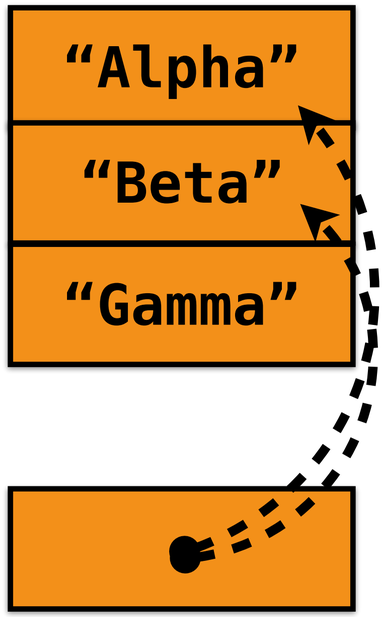
```
fn main() {
  let mut v = vec![format!(“Alpha”),
                   format!(“Beta”),
                   format!(“Gamma”)];

  for s in &v {
    println!(“{:?}”, s);
  }

  for s in &mut v {
    s.push_str(“.”);
  }
}
```

&String

&Vec<String>

&mut String

&mut Vec<String>



Exercise: **structs**

http://rust-tutorials.com/exercises

Implement

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

Cheat sheet:

```
for s in v { … }
```

```
for s in &v { … }
```

```
while … { … }
```

```
let mut some_var = 0.0;
```

```
some_var += x;
```

```
println!(“{:?}”, s);
```

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

Thanks for listening!