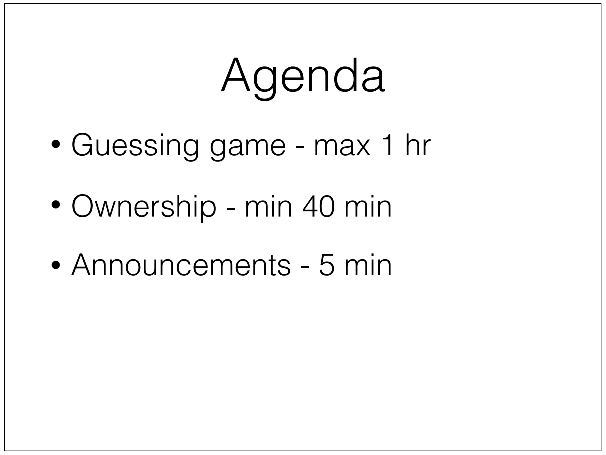
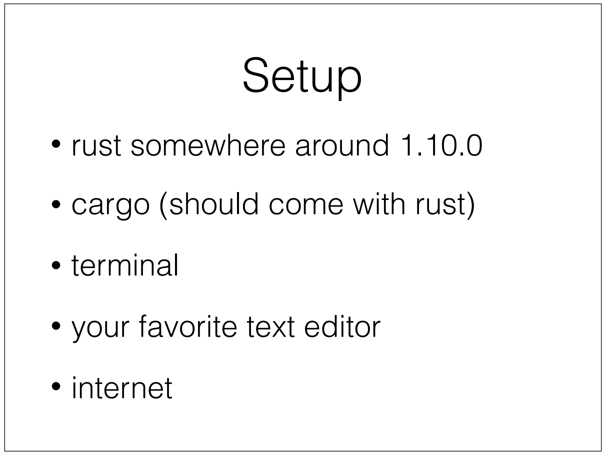
Intro to Rust

Carol (Nichols || Goulding)
@carols10cents



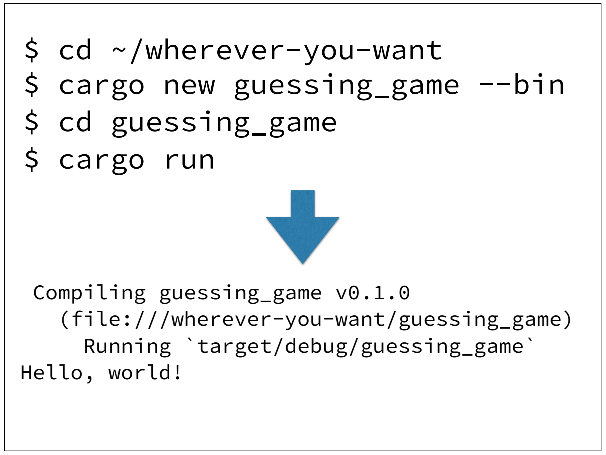
Agenda

- Guessing game - max 1 hr
- Ownership - min 40 min
- Announcements - 5 min



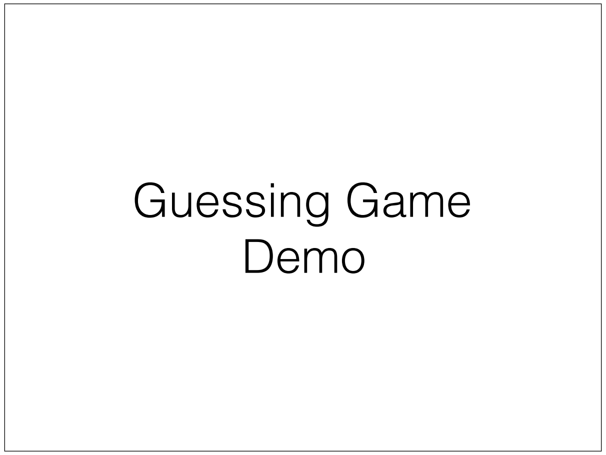
Setup

- rust somewhere around 1.10.0
- cargo (should come with rust)
- terminal
- your favorite text editor
- internet



```
$ cd ~/wherever-you-want
$ cargo new guessing_game --bin
$ cd guessing_game
$ cargo run
```

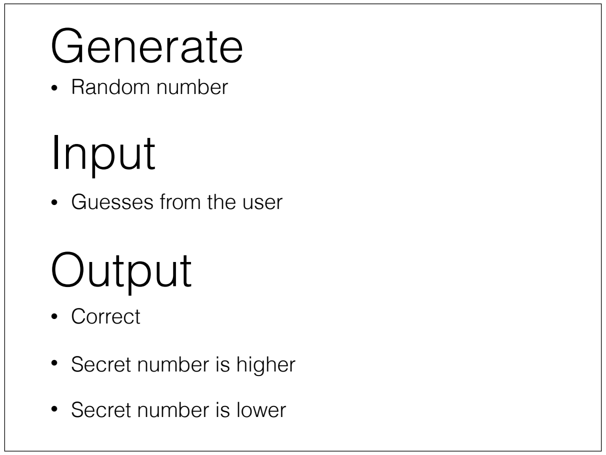
```
 Compiling guessing_game v0.1.0
   (file:///wherever-you-want/guessing_game)
     Running `target/debug/guessing_game`
Hello, world!
```



Guessing Game Demo

(do a demo)

What kinds of things will our program need to do to play this game?



Generate

- Random number

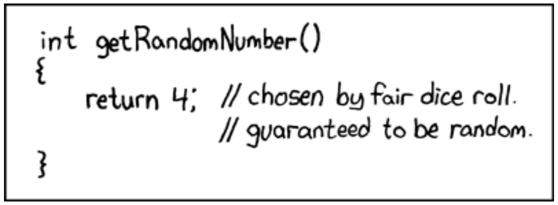
Input

- Guesses from the user

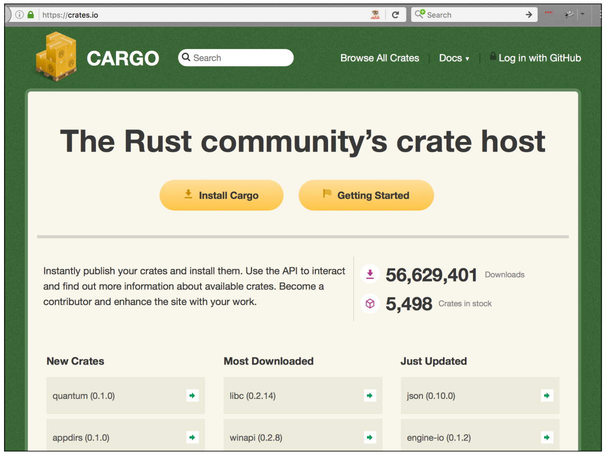
Output

- Correct
- Secret number is higher
- Secret number is lower

Let’s code!



```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```



The Rust community's crate host

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.

56,629,401 Downloads

5,498 Crates in stock

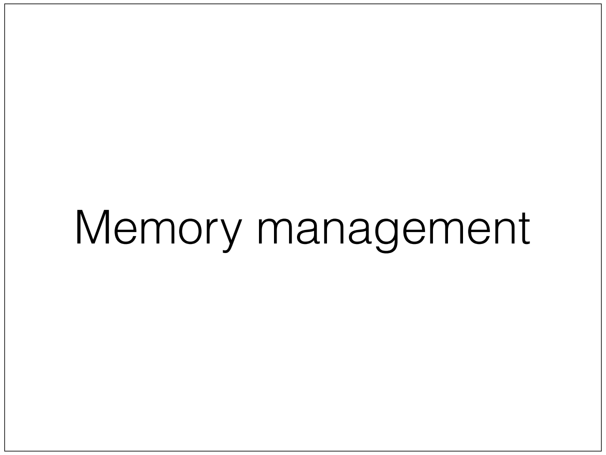
New Crates	Most Downloaded	Just Updated
quantum (0.1.0)	libc (0.2.14)	json (0.10.0)
appdirs (0.1.0)	winapi (0.2.8)	engine-io (0.1.2)

What you learned

- Output values
- Input from stdin
- Var bindings
- Mutability
- Using a crate
- match
- Result
- parse
- loop
- break

Ownership

Lots more to learn about Rust, you're all capable of looking up "how to do an if/else in rust"
Ownership is different though.

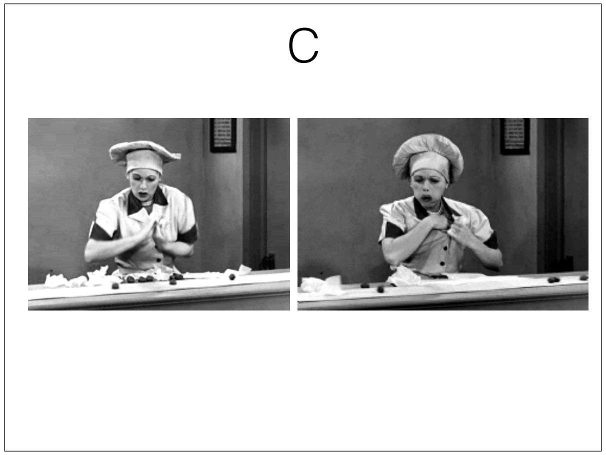


Memory management

Java

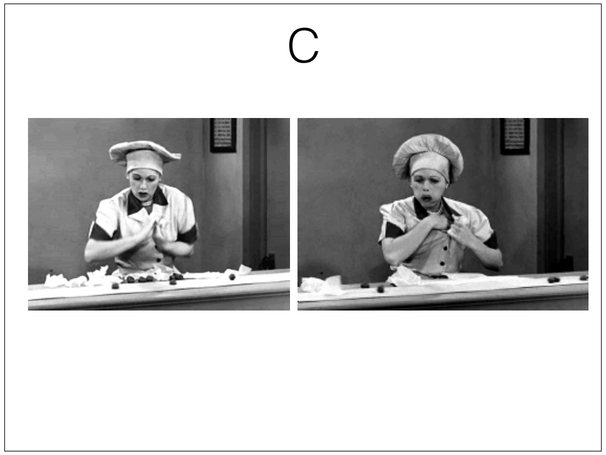
- Java has a garbage collector, like mayor peduto
- Every so often, the garbage collector comes along and frees up memory you aren’t using anymore
- you don’t have to think about it, until you do.. like when? anyone?
- memory leaks, performance tuning
- with less responsibility comes less power

Java

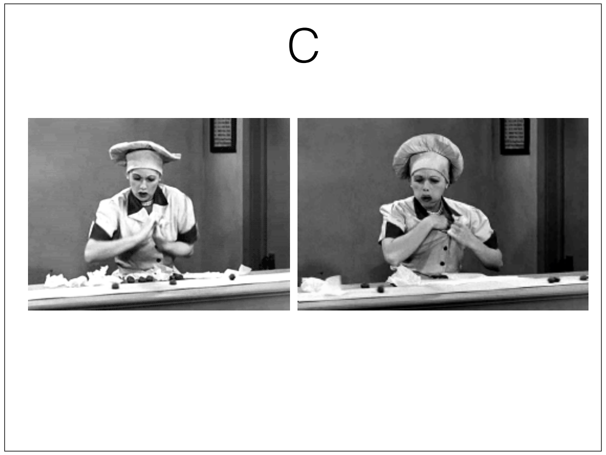


C

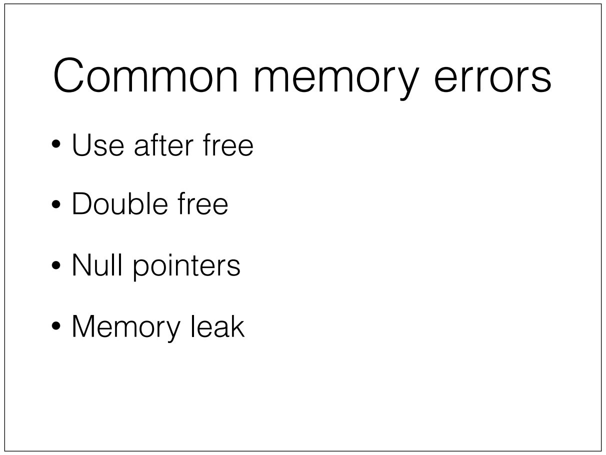
In C, you, the programmer, has to ask the operating system for specific amounts of memory, make sure that memory remains valid for all parts of your code that might use it, and remember to return that memory to the operating system exactly once when you’re done with it.



C

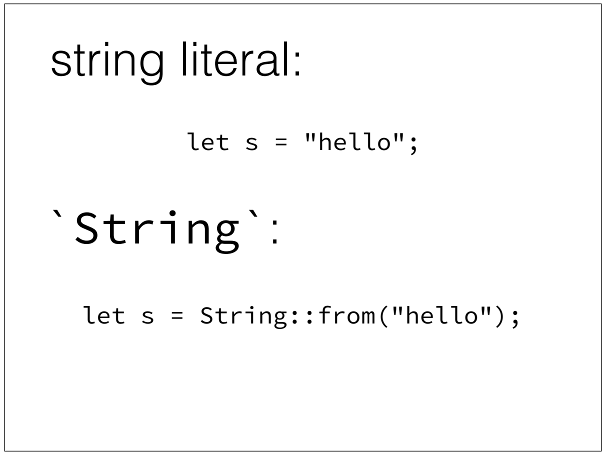


C



Common memory errors

- Use after free
- Double free
- Null pointers
- Memory leak



string literal:

```
let s = "hello";
```

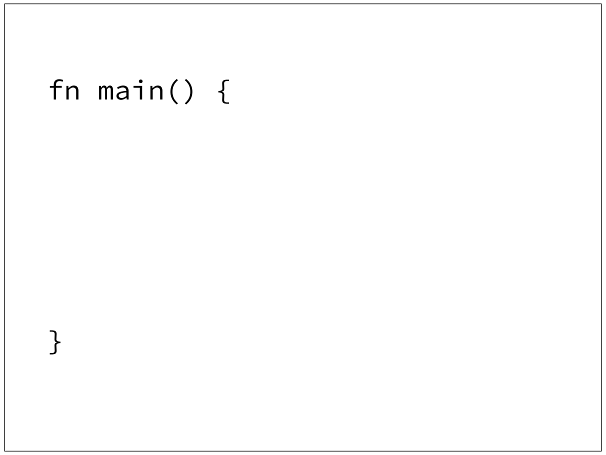
`` `String` ``:

```
let s = String::from("hello");
```

First let's talk about the difference between a string literal and a capital-s string.

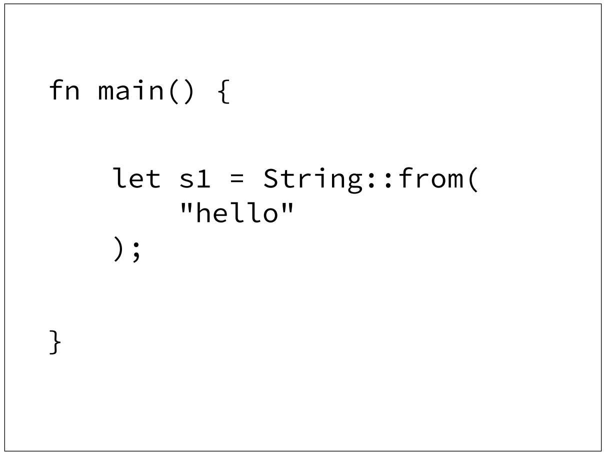
i haven't capitalized string literal here to emphasize that it's an english term to talk about what the first `s` is, it's not a Rust type. Can never change, data is read from its location in the source code itself.

Strings are a Rust type— they can be created from a string literal here, but their memory is allocated on the heap so that they can change and grow to a size that we might not know at runtime (example: user-input guesses). So we have to ask the OS for memory and give it back when we're done.



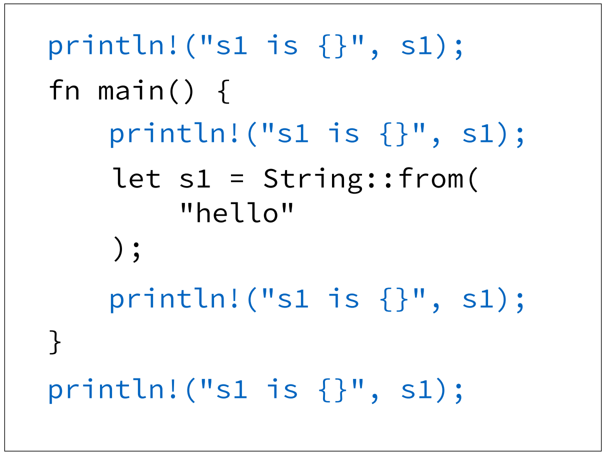
```
fn main() {





}
```

ok, so we've got a main function here, just like in the guessing game.



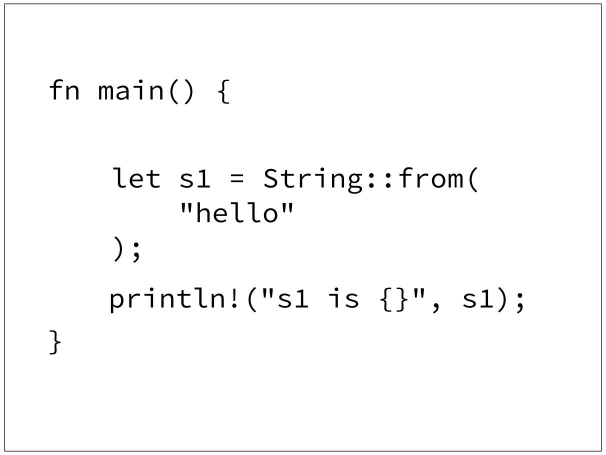
```
fn main() {

    let s1 = String::from(
        "hello"
    );

}
```

now we're going to create a new mutable String with a capital S. String::from is asking the operating system for memory.



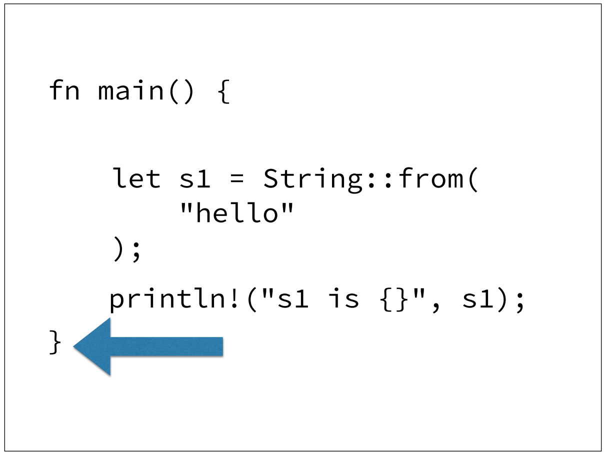
```
println!("s1 is {}", s1);
fn main() {
    println!("s1 is {}", s1);
    let s1 = String::from(
        "hello"
    );
    println!("s1 is {}", s1);
}
println!("s1 is {}", s1);
```

If we wanted to print out s, where could we do that? that is, where is s valid?



```
fn main() {

    let s1 = String::from(
        "hello"
    );
    println!("s1 is {}", s1);
}
```

This is the only place where the binding `s` is "in scope" and valid.

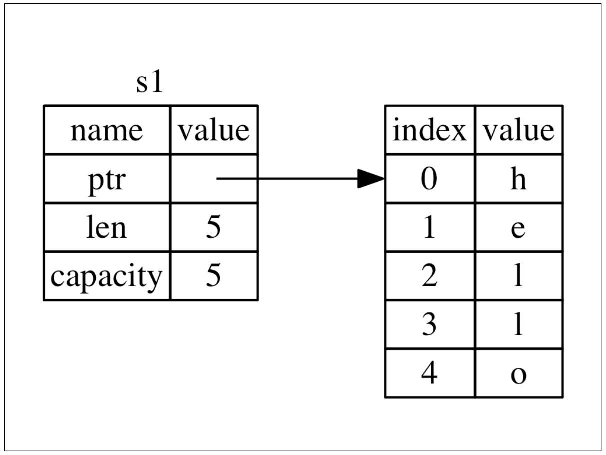


```
fn main() {

    let s1 = String::from(
        "hello"
    );
    println!("s1 is {}", s1);
}
```

At the closing curly brace of main, s "goes out of scope".

At this point, Rust automatically calls a function called "drop" on the String, which returns the memory to the operating system.

We say that the binding s1 is the owner of the memory that it points to, and it is responsible for freeing that memory when it goes out of scope.



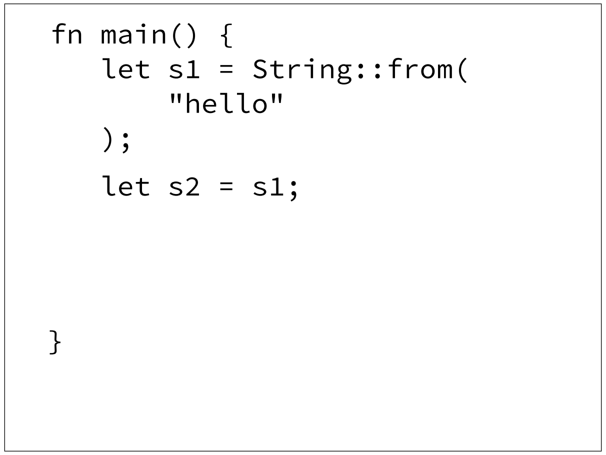
s1

name	value
ptr	
len	5
capacity	5

index	value
0	h
1	e
2	l
3	l
4	o

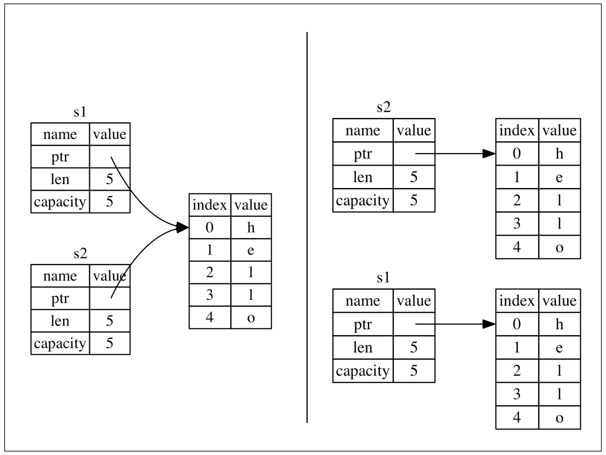
This is actually what s1 contains— a pointer to the start of some memory on the heap, and how long from that pointer you have data and that you have asked for.

This way, you could ask for more and get a different pointer to bigger memory.



```
fn main() {
   let s1 = String::from(
       "hello"
   );
   let s2 = s1;




}
```

What if we created another binding, s2, and assigned s1 to it? What are the possible things that rust could choose to do?



s1

name	value
ptr	
len	5
capacity	5

s2

name	value
ptr	
len	5
capacity	5

index	value
0	h
1	e
2	l
3	l
4	o

s2

name	value
ptr	
len	5
capacity	5

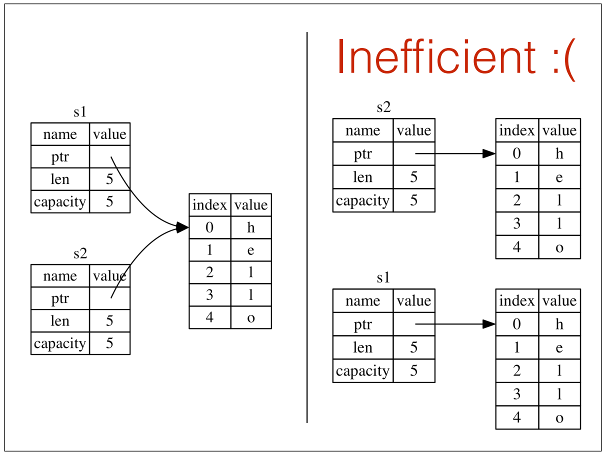
index	value
0	h
1	e
2	l
3	l
4	o

s1

name	value
ptr	
len	5
capacity	5

index	value
0	h
1	e
2	l
3	l
4	o

The two possibilities are that we do not copy the data and have the pointers of s1 and s2 point to the same data,
or we copy the data and s1 and s2 point to their own data.
Which one would take longer?



Inefficient :(

s1

name	value
ptr	
len	5
capacity	5

s2

name	value
ptr	
len	5
capacity	5

index	value
0	h
1	e
2	l
3	l
4	o

s2

name	value
ptr	
len	5
capacity	5

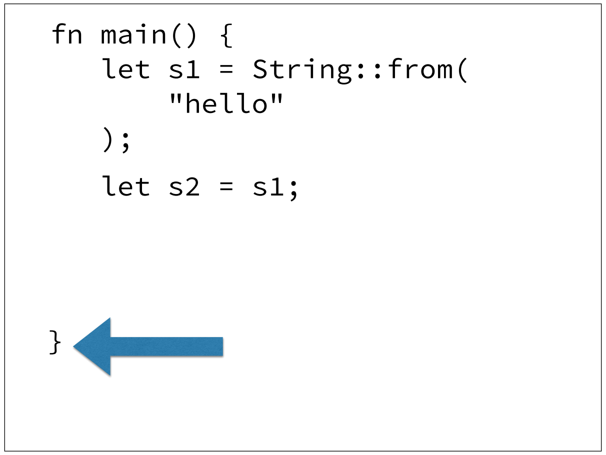
index	value
0	h
1	e
2	l
3	l
4	o

s1

name	value
ptr	
len	5
capacity	5

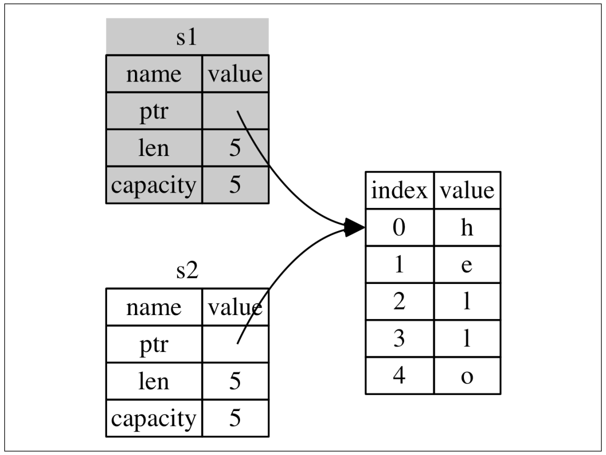
index	value
0	h
1	e
2	l
3	l
4	o

Rust is trying to be fast, so we don't copy the data.
There's a problem with the first scenario the way we have it, though:



```
fn main() {
    let s1 = String::from(
        "hello"
    );
    let s2 = s1;




}
```

What happens here? What goes out of scope? What happens when they both go out of scope? What happens if we call drop on the same memory?
I promised at the beginning we would prevent double frees???



s1

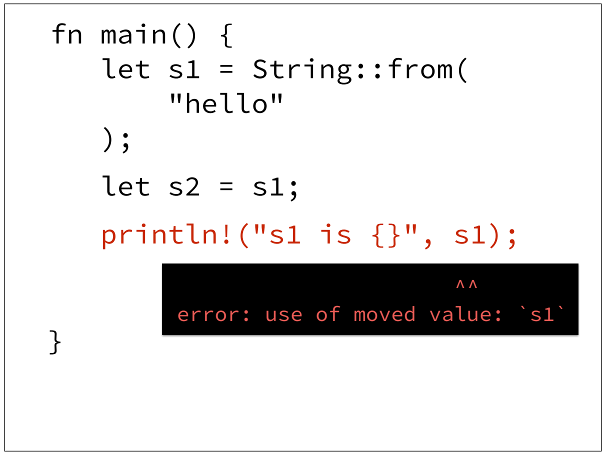
name	value
ptr	
len	5
capacity	5

s2

name	value
ptr	
len	5
capacity	5

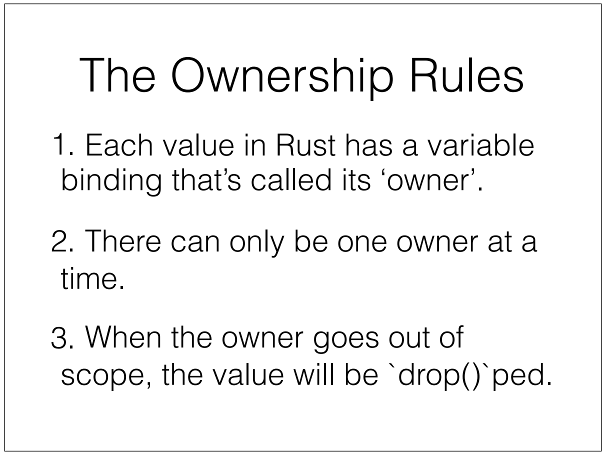
index	value
0	h
1	e
2	l
3	l
4	o

What actually happens is we invalidate s1, and we say s1 was MOVED into s2.



```
fn main() {
   let s1 = String::from(
       "hello"
   );
   let s2 = s1;
   println!("s1 is {}", s1);
                                  ^^
              error: use of moved value: `s1`
}
```

If we try to use s1 again after moving it to s2, we can't— the compiler won't let us. s2 is now the only owner of the memory and only when s2 goes out of scope will drop be called on that memory.



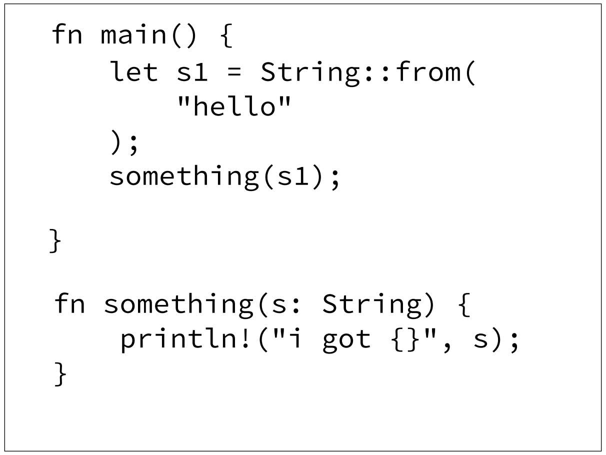
The Ownership Rules

1. Each value in Rust has a variable binding that's called its 'owner'.

2. There can only be one owner at a time.

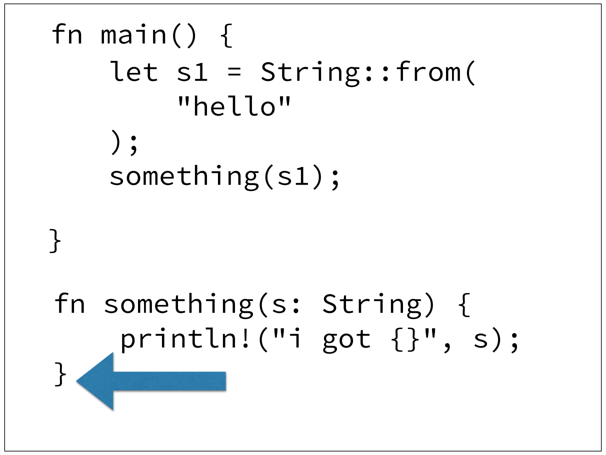
3. When the owner goes out of scope, the value will be `drop()`ped.

So, these are the ownership rules that rust enforces and checks that your program follows at compile time. These rules are what make it so that we don't have to manually ask for and return memory explicitly, but we don't have a garbage collector that has to run and we won't make memory errors.



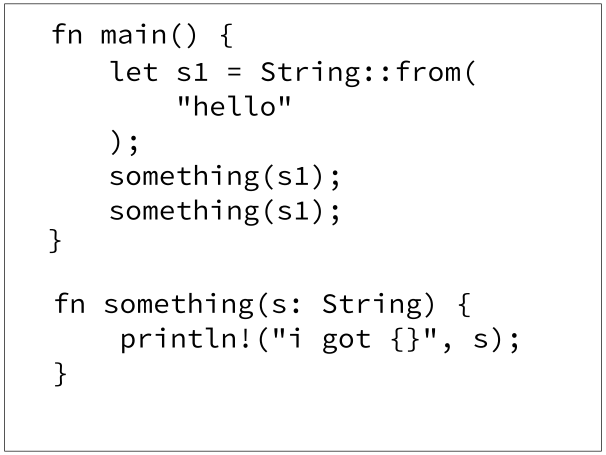
```
fn main() {
    let s1 = String::from(
        "hello"
    );
    something(s1);

}

fn something(s: String) {
    println!("i got {}", s);
}
```

Functions follow these same rules too, since functions have their own scope. By default, calling a function transfers ownership of a binding to a function.



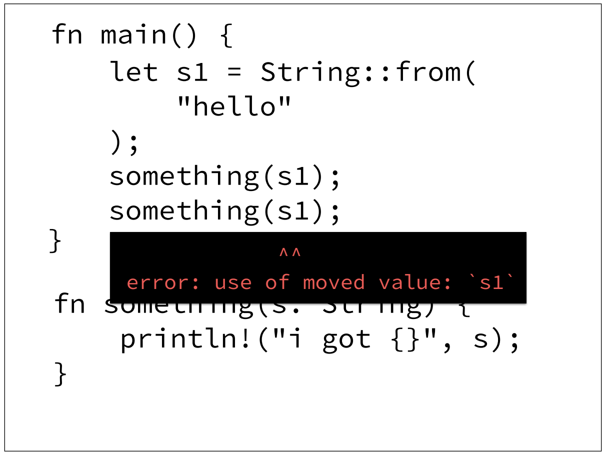
```
fn main() {
    let s1 = String::from(
        "hello"
    );
    something(s1);

}

fn something(s: String) {
    println!("i got {}", s);
}
```

That means s1 gets moved into the function something, and at the end of something is where the data that s1 points to goes out of scope and gets dropped.

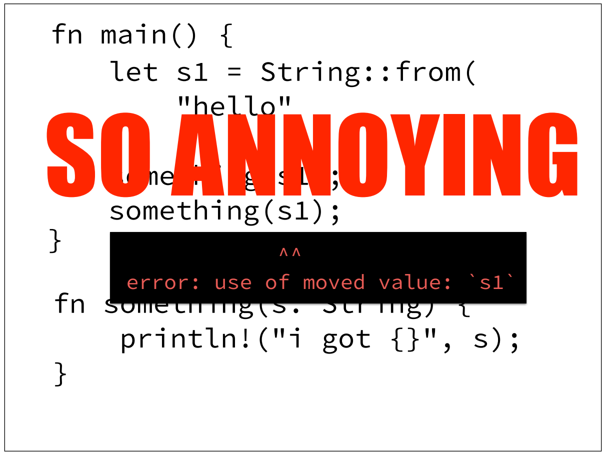


```
fn main() {
    let s1 = String::from(
        "hello"
    );
    something(s1);
    something(s1);
}

fn something(s: String) {
    println!("i got {}", s);
}
```

What this means is, like when we moved s1 into s2, we can't use s1 again.

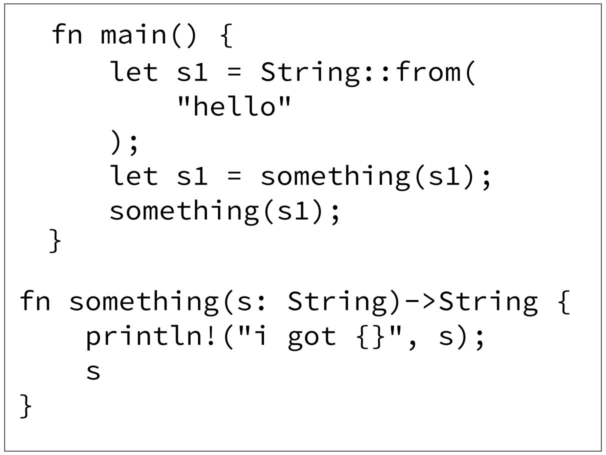


```
fn main() {
    let s1 = String::from(
        "hello"
    );
    something(s1);
    something(s1);
}
              ^^
  error: use of moved value: `s1`
fn something(s: String) {
    println!("i got {}", s);
}
```



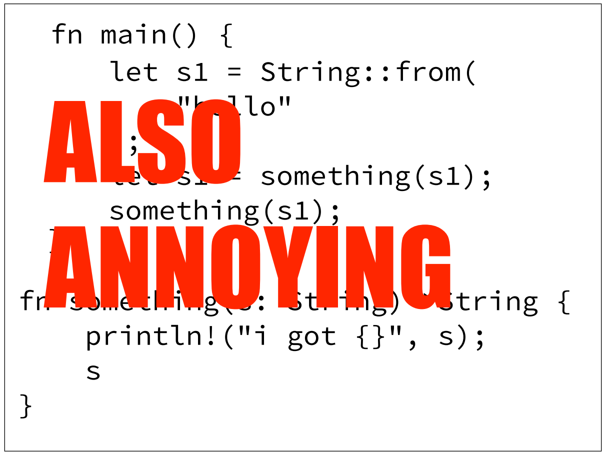
SO ANNOYING

```
fn main() {
    let s1 = String::from(
        "hello"
    );
    something(s1);
    something(s1);
}
              ^^
  error: use of moved value: `s1`
fn something(s: String) {
    println!("i got {}", s);
}
```



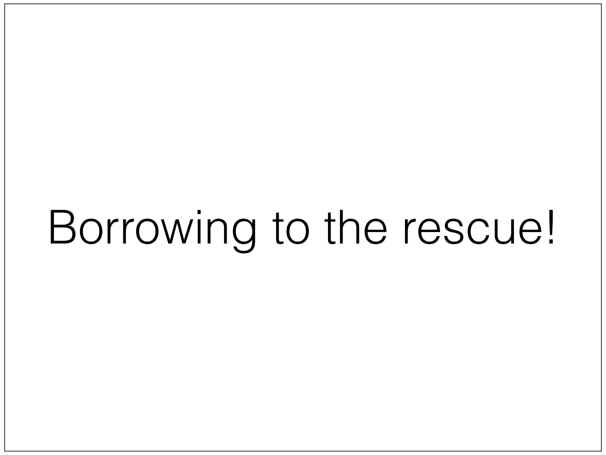
```
 fn main() {
     let s1 = String::from(
         "hello"
     );
     let s1 = something(s1);
     something(s1);
 }

fn something(s: String)->String {
    println!("i got {}", s);
    s
}
```

We could return s from something, so that main owns it again, this will compile… but this is also annoying.



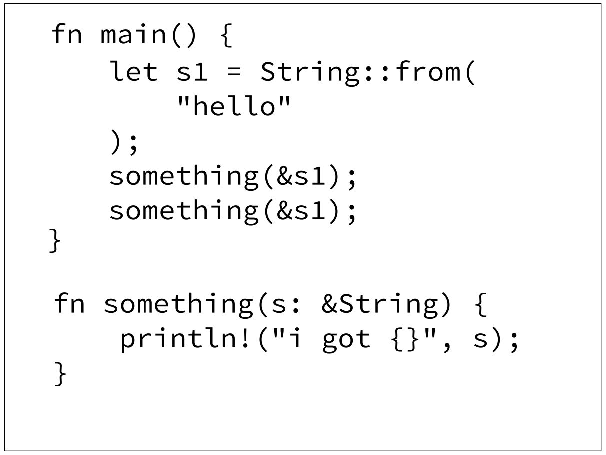
```
  fn main() {
      let s1 = String::from(
          "hello"
      );
      let s1 = something(s1);
      something(s1);
  }

fn something(s: String) -> String {
    println!("i got {}", s);
    s
}
```

ALSO ANNOYING



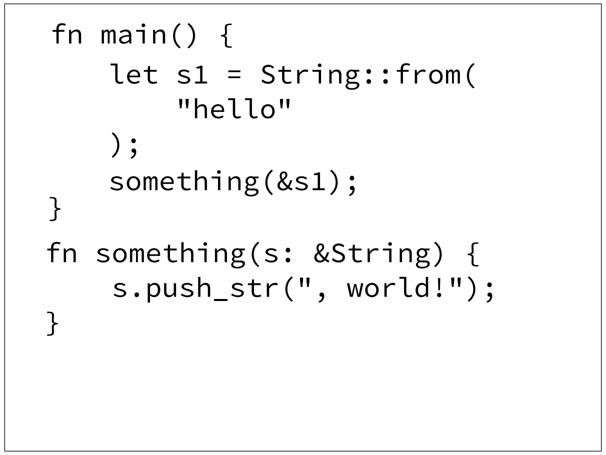
Borrowing to the rescue!

Borrowing is the feature of rust that makes this less annoying!



```
  fn main() {
      let s1 = String::from(
          "hello"
      );
      something(&s1);
      something(&s1);
  }

  fn something(s: &String) {
      println!("i got {}", s);
  }
```

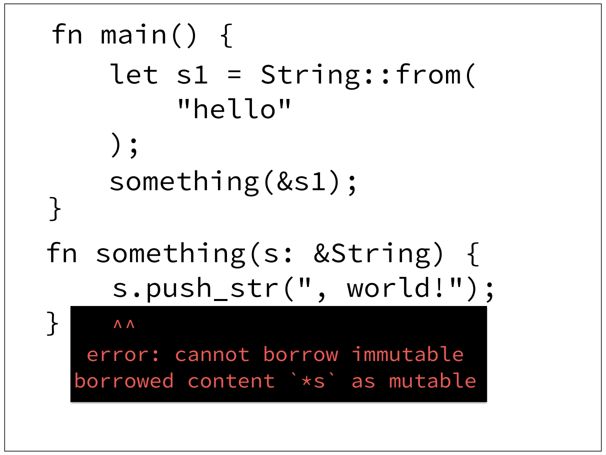
If we add an ampersand to s1, that means we're passing in a borrowed reference to s1, and main retains ownership. We also have to change the signature of something, to say that it wants a borrowed string, it doesn't want to take ownership of the argument. This compiles and works.



```
fn main() {
    let s1 = String::from(
        "hello"
    );
    something(&s1);
}
fn something(s: &String) {
    s.push_str(", world!");
}
```

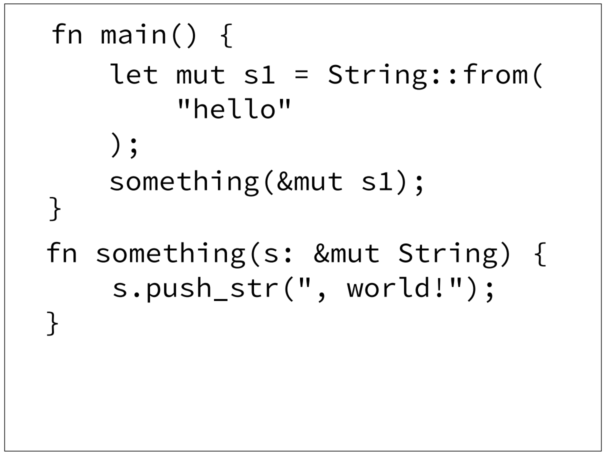
Are we allowed to modify something we're borrowing? Try it!

Nope! by default, references are immutable, just like bindings are immutable.



```
fn main() {
    let s1 = String::from(
        "hello"
    );
    something(&s1);
}
fn something(s: &String) {
    s.push_str(", world!");
}
```

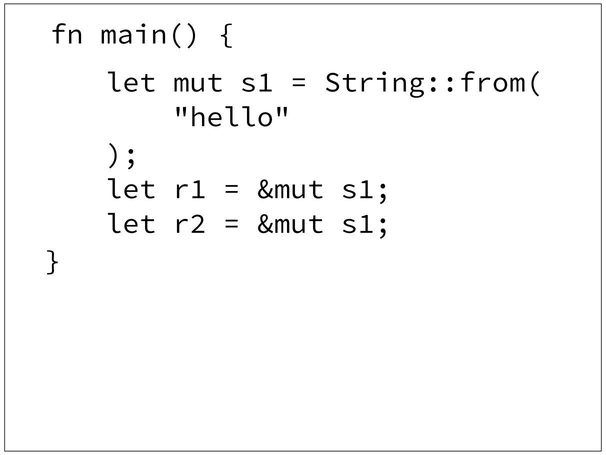
```
   ^^
 error: cannot borrow immutable
borrowed content `*s` as mutable
```



```
fn main() {
    let mut s1 = String::from(
        "hello"
    );
    something(&mut s1);
}
fn something(s: &mut String) {
    s.push_str(", world!");
}
```

Just like we can change bindings to be mutable with the `mut` keyword, we can change references to be mutable.

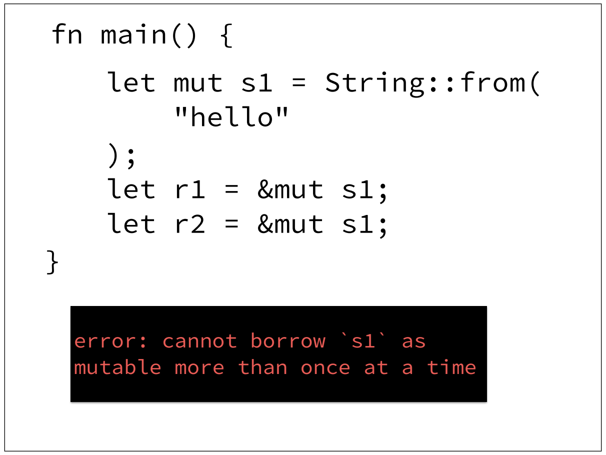
This is nice because you know by looking at the signature whether the fn is going to change its arguments or not.



```
fn main() {
    let mut s1 = String::from(
        "hello"
    );
    let r1 = &mut s1;
    let r2 = &mut s1;
}
```

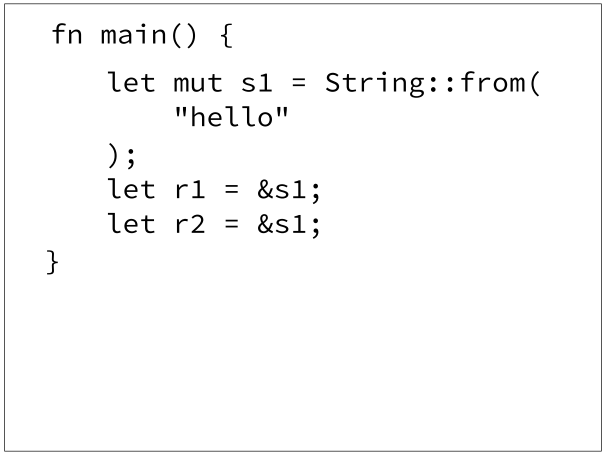
What about this code? Try running it.

This is not allowed. Only one thing can be changing a value in scope at a time— this is important to prevent data races.



```
fn main() {
    let mut s1 = String::from(
        "hello"
    );
    let r1 = &mut s1;
    let r2 = &mut s1;
}
```

```
error: cannot borrow `s1` as
mutable more than once at a time
```



```
fn main() {
    let mut s1 = String::from(
        "hello"
    );
    let r1 = &s1;
    let r2 = &s1;
}
```

Does this code compile? What does that mean?

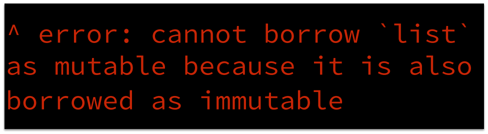
```
 fn main() {
    let mut s1 = String::from(
        "hello"
    );
    let r1 = &s1;
    let r2 = &s1;
    let r3 = &mut s1;
}
```

Does this code compile? What does that mean?

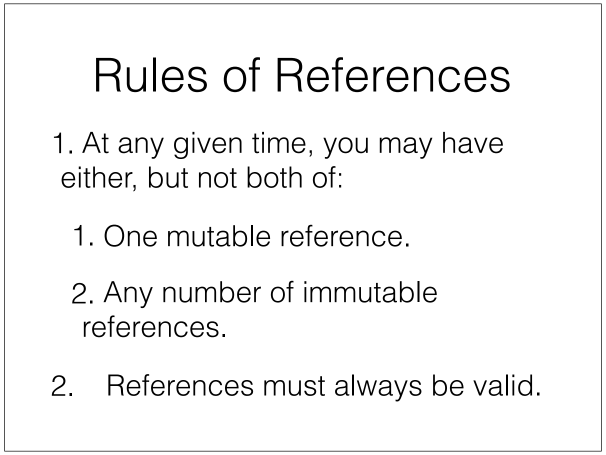
```
let mut list = vec![1, 2, 3];
for i in &list {
    println!("i is {}", i);
    list.push(i + 1);
}
```

One more example. What would happen if Rust allowed this?

```
let mut list = vec![1, 2, 3];
for i in &list {
    println!("i is {}", i);
    list.push(i + 1);
}
```



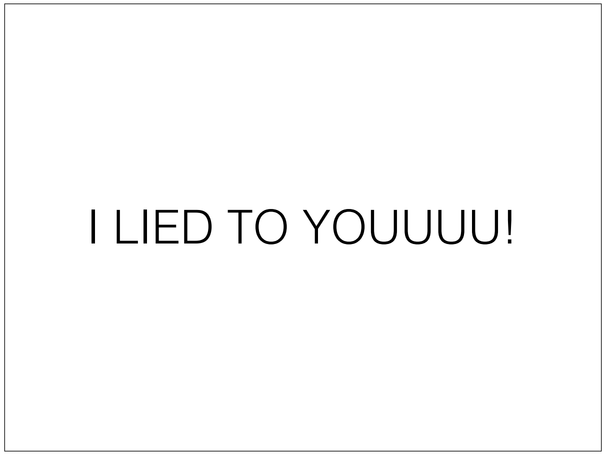
```
^ error: cannot borrow `list`
as mutable because it is also
borrowed as immutable
```



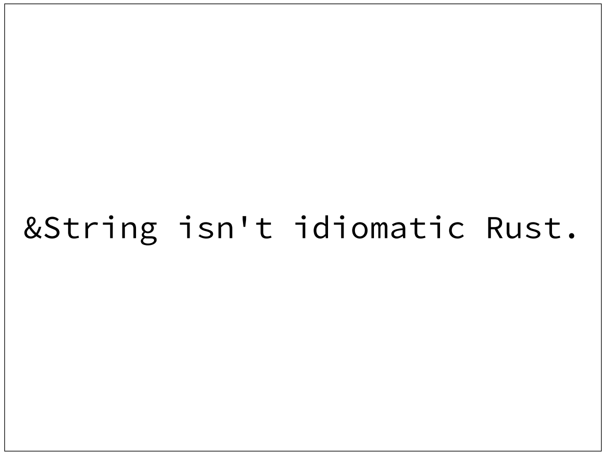
Rules of References

1. At any given time, you may have either, but not both of:
    1. One mutable reference.
    2. Any number of immutable references.
2. References must always be valid.

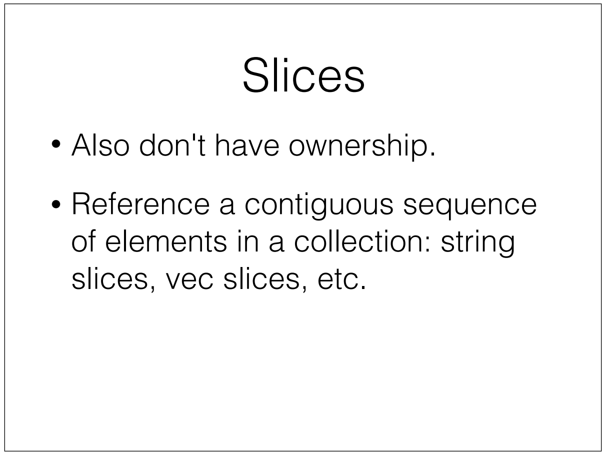
"References must always be valid" means that a reference can't exist after the data it refers to gets dropped— we say the lifetime of the data must be at least as long as the lifetime of the reference. This means there's no dangling pointers.



I LIED TO YOUUUU!

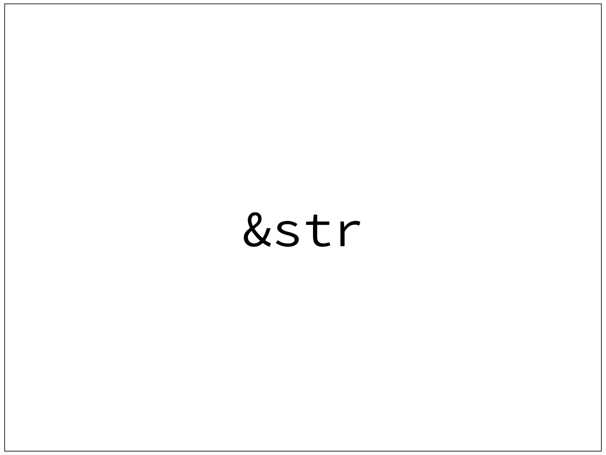


&String isn't idiomatic Rust.



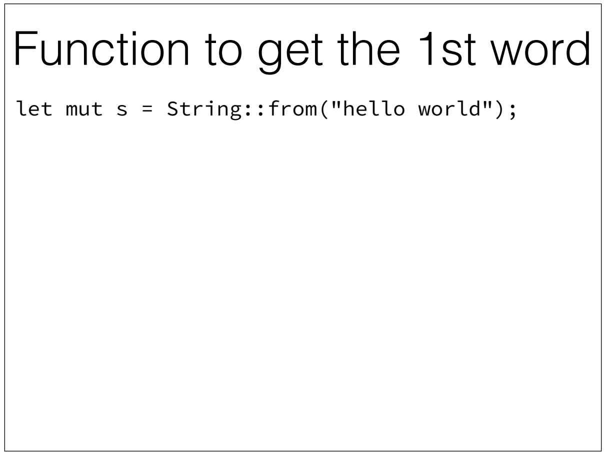
Slices

- Also don't have ownership.
- Reference a contiguous sequence of elements in a collection: string slices, vec slices, etc.



```
&str
```

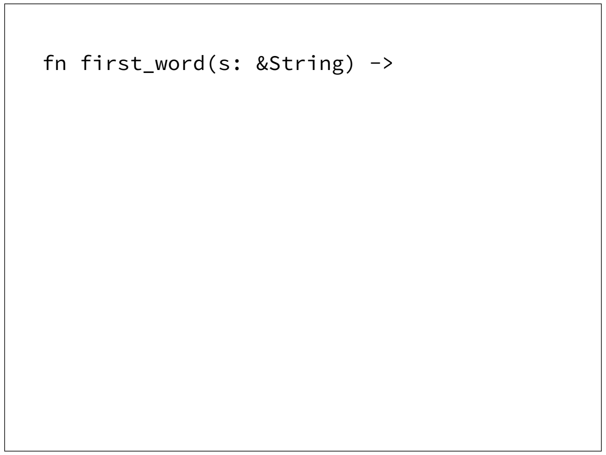
This type is pronounced "string slice" and is much more common than a borrowed capital-s String.



Function to get the 1st word

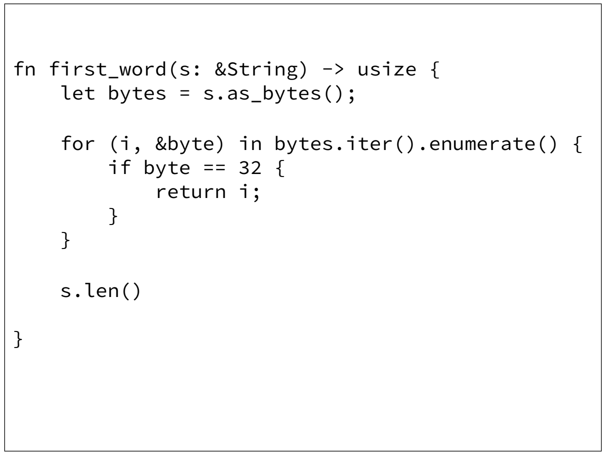
```
let mut s = String::from("hello world");
```

So if we have this string, we'd want this function to return "hello" somehow.



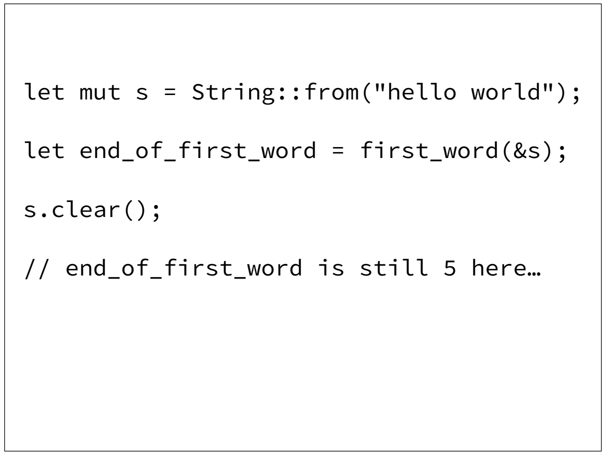
```
fn first_word(s: &String) ->
```

We don't want ownership of the string, so we take a reference to it… but what could we return?



```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &byte) in bytes.iter().enumerate() {
        if byte == 32 {
            return i;
        }
    }

    s.len()

}
```

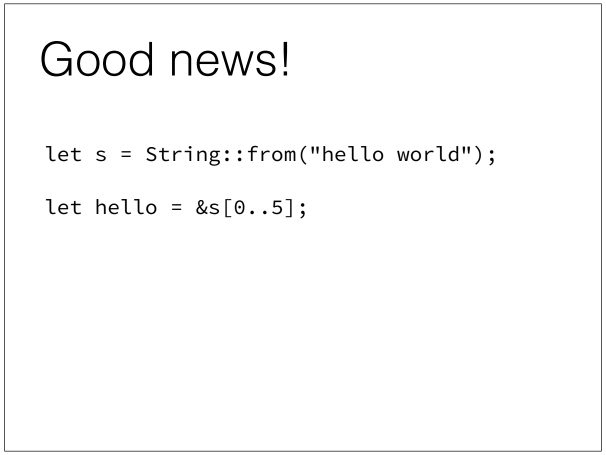
How about the index of the end of the word? Go through each byte of the string. If we find a space, return the index of the space. If we never find a space, whole string is a word.



```
let mut s = String::from("hello world");

let end_of_first_word = first_word(&s);

s.clear();

// end_of_first_word is still 5 here…
```

But this number only makes sense in the context of our string… if we don't have the data in that string anymore, what does that 5 even mean???

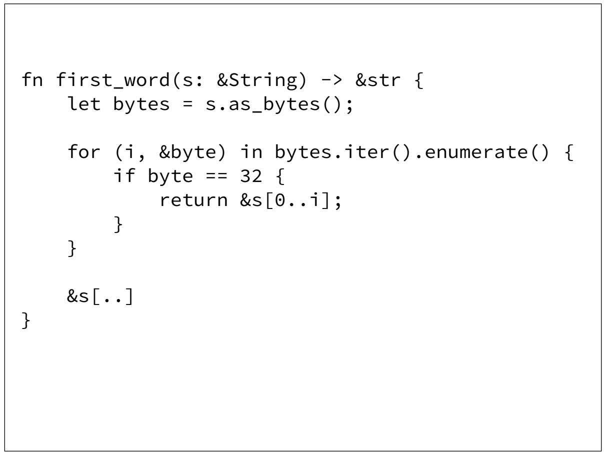
It'd be nice to have a way to tie 5 to the data in the string…



Good news!

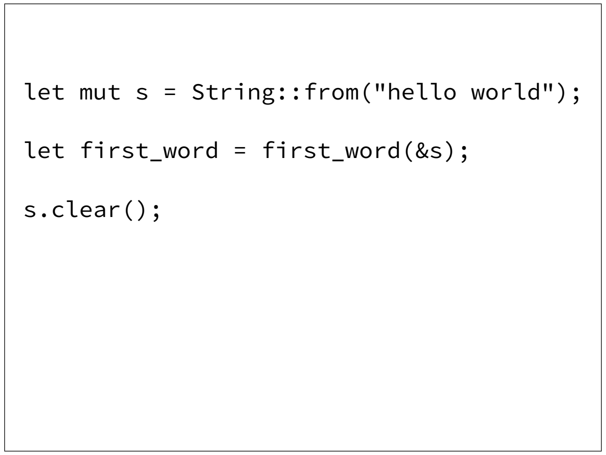
```
let s = String::from("hello world");

let hello = &s[0..5];
```

We do have that! It's the string slice! You create one by specifying a start and an end (it really stores a pointer and a length though).
This is a reference that is only valid as long as the original data is valid.



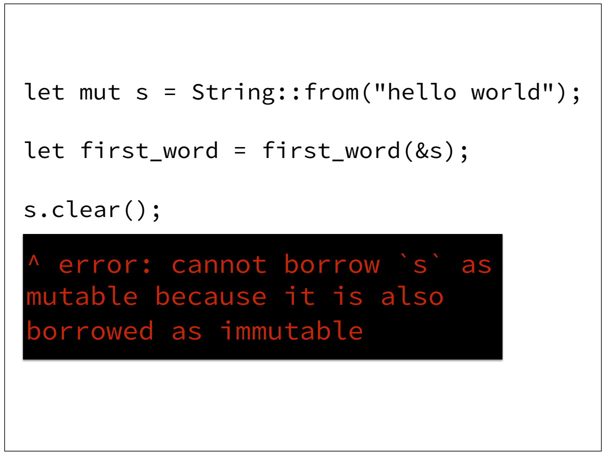
```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &byte) in bytes.iter().enumerate() {
        if byte == 32 {
            return &s[0..i];
        }
    }

    &s[..]
}
```

So now we've got a new definition for first_word that returns a slice (the indices are optional, leaving them off gets us a slice to the whole string).



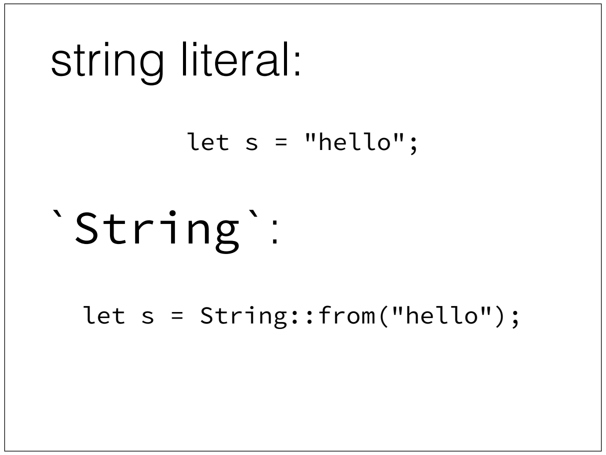
```
let mut s = String::from("hello world");

let first_word = first_word(&s);

s.clear();
```

So now in this case, where we had a number that didn't mean anything because the data it referred to is gone, the rules of references kick in— the clear function has to take a mutable reference to the string, but it's not allowed to because we have a string slice to it in scope.



```
let mut s = String::from("hello world");

let first_word = first_word(&s);

s.clear();
```

```
^ error: cannot borrow `s` as
mutable because it is also
borrowed as immutable
```



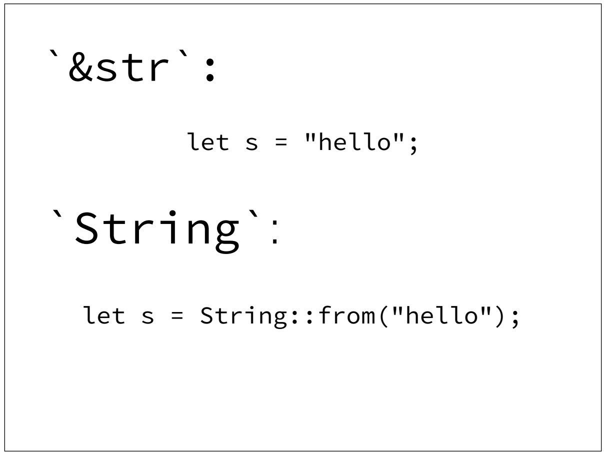
string literal:

```
let s = "hello";
```

`String`:

```
let s = String::from("hello");
```

Remember this slide from before where I said string literal wasn't a rust type?



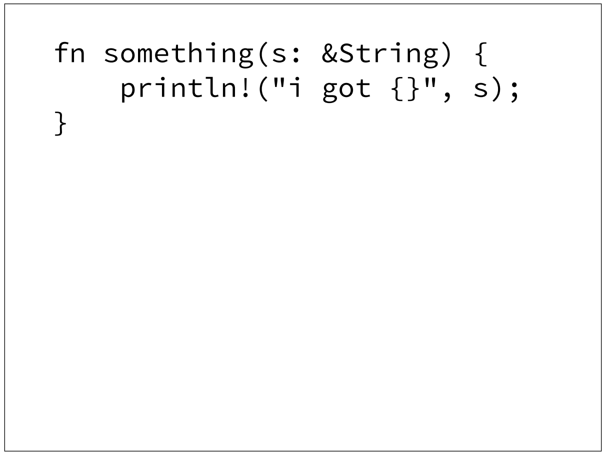
`&str`:

```
let s = "hello";
```

`String`:

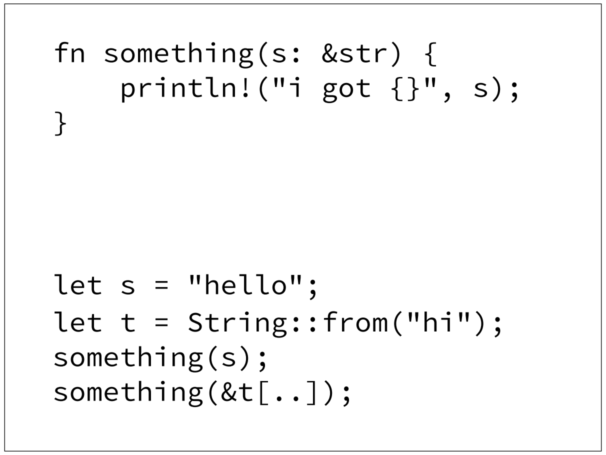
```
let s = String::from("hello");
```

String literals are actually of type &str.



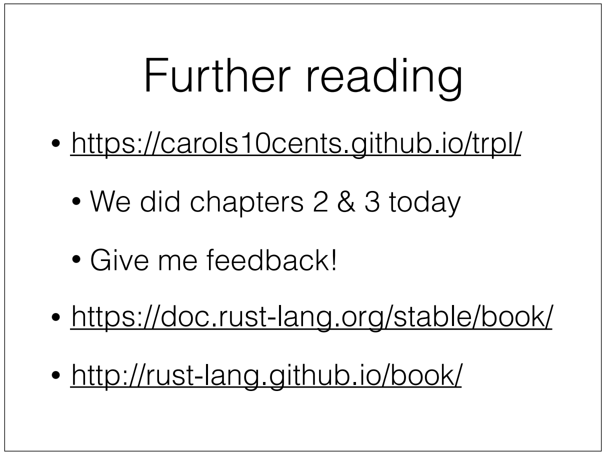
```
fn something(s: &String) {
    println!("i got {}", s);
}
```

So with this version of the something function from before,



```
fn something(s: &str) {
    println!("i got {}", s);
}




let s = "hello";
let t = String::from("hi");
something(s);
something(&t[..]);
```

… if it takes a string slice, we can call it with a string literal OR a slice from a String.



Further reading

- https://carols10cents.github.io/trpl/
  - We did chapters 2 & 3 today
  - Give me feedback!
- https://doc.rust-lang.org/stable/book/
- http://rust-lang.github.io/book/