



INTRO TO RUST LANG OWNERSHIP

Benjamin Owad, David Rudo, and Connor Tsui

Welcome back!

- Homework 1 due today
- You can use 7 late days over the whole semester
- If you spent over an hour on the assignment, please let us know!

Review: Scopes

Recall *scopes* in Rust.

```
                                // s is not valid here, it's not yet declared
{
    let s = "hello";           // s is valid from this point forward
    // do stuff with s
}                                // this scope is now over, and s is no longer valid
```

- There are two important points in time here:
 - When `s` comes *into* scope, it is valid
 - It remains valid until it goes *out* of scope.

Ownership

Ownership

From the official Rust Lang [book](#):

Ownership is Rust's most unique feature and has deep implications for the rest of the language. It enables Rust to make memory safety guarantees without needing a garbage collector, so it's important to understand how ownership works.

- Today we'll introduce *Ownership*, as well as several related features

What is Ownership?

Ownership is a set of rules that govern how a Rust program manages memory.

- Some languages have garbage collection to manage memory
- Other languages require you to explicitly allocate and free memory
- Rust has a third approach: memory is managed through a set of rules

Ownership rules

- Each value in Rust has an *owner*
- There can only be one owner at a time
- When the owner goes out of scope, the value will be *dropped*

Example: String Literals

We didn't explicitly talk about this last week, but every time you see a text like "Hello, World!" surrounded by double quotes, that is a *string literal*.

```
fn main() {  
    println!("Hello, world!"); // Print a string literal  
  
    let s = "Ferris is our friend"; // Another string literal  
}
```

- String literals live inside in the program binary

Problem: String Literals are Immutable

Suppose wanted to take user input and store it. This is how we do it in Python:

```
username = input("Tell me your name!")
```

- We do not know how long `username` will be
- How would we do this in Rust?
- We need a way to store a collection of characters with a dynamic size

The `String` type

- In addition to string literals, Rust has another string type, `String`
- `String` manages data allocated on the heap
- Dynamically stores an amount of text that is unknown at compile time

String example

You can create a `String` from a string literal using `String::from()`.

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

This kind of string *can* be mutated:

```
let mut s = String::from("hello");  
s.push_str(", world!"); // push_str() appends a literal to a String  
println!("{}", s); // This will print `hello, world!`
```

- Why can `String` be mutated but string literals cannot?

String vs literals

- We know the contents of string literals at compile time, so the text is hardcoded directly into the final executable
- To support a fully resizable piece of text, we need to allocate on the *heap*
 - This means we must request memory from the allocator at *runtime*
 - We need a way of returning the memory when we're done using it
 - What 2 C functions does this remind you of?

malloc and free

In C, we use `malloc` and `free` to manage heap memory for our program.

However, we need to ensure we pair exactly one `malloc` with exactly one `free`.

- If we forget to `free`, we leak memory
- If we `free` too early, we have an invalid variable
- If we `free` twice, that's a "double free" bug
- Undefined behavior!!! ☠️

Manual Memory Management

- Using `malloc` and `free` can lead to all sorts of undefined behavior
 - Unless you are the perfect developer...
 - *Who never writes an bugs...*
 - You're bound to shoot yourself in the foot
- It would be great if the compiler knew at what point the variable needs memory, and then at what point it doesn't need it anymore
- **Idea: what if we tied memory allocation to the scope of a variable?**

The Rust approach to memory

Memory is returned once the variable that owns it goes out of scope.

```
{  
    let s = String::from("hello"); // s is valid from this point forward  
    // do stuff with s  
} // this scope is now over,  
  // and s is no longer valid
```

- When `s` comes into scope, it gets memory from the allocator
- When `s` goes out of scope, it frees all of its memory
 - Rust calls a function called `drop` on `s` automatically once the program reaches the closing bracket

Multiple variables, same integer

Multiple variables can interact with the same data in different ways in Rust.

```
let s1 = String::from("hello");  
let s2 = s1.clone();  
  
println!("s1 = {}, s2 = {}", s1, s2);
```

Can we guess what this is doing?

- Bind the value `5` to `x`
- Make a copy of the value in `x` and bind it to `y`
- Simple enough, right?

Multiple variables, same `String`?

What about the `String` version?

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

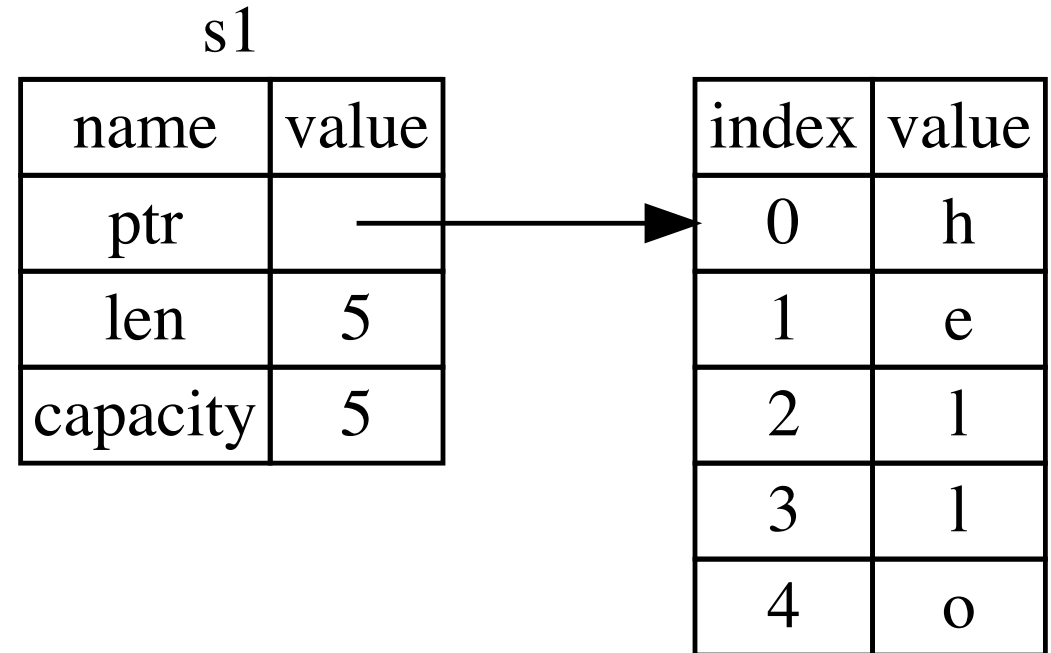
What is this code doing?

- Bind the `String` containing `"hello"` to `s1`
- Now what?
 - Do we make a copy of the `String`?

String data layout

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

- A `String` is made up of 3 fields:
 - A pointer to text somewhere in memory
 - A length
 - A capacity
- Left diagram is on the stack
- Right diagram is on the heap

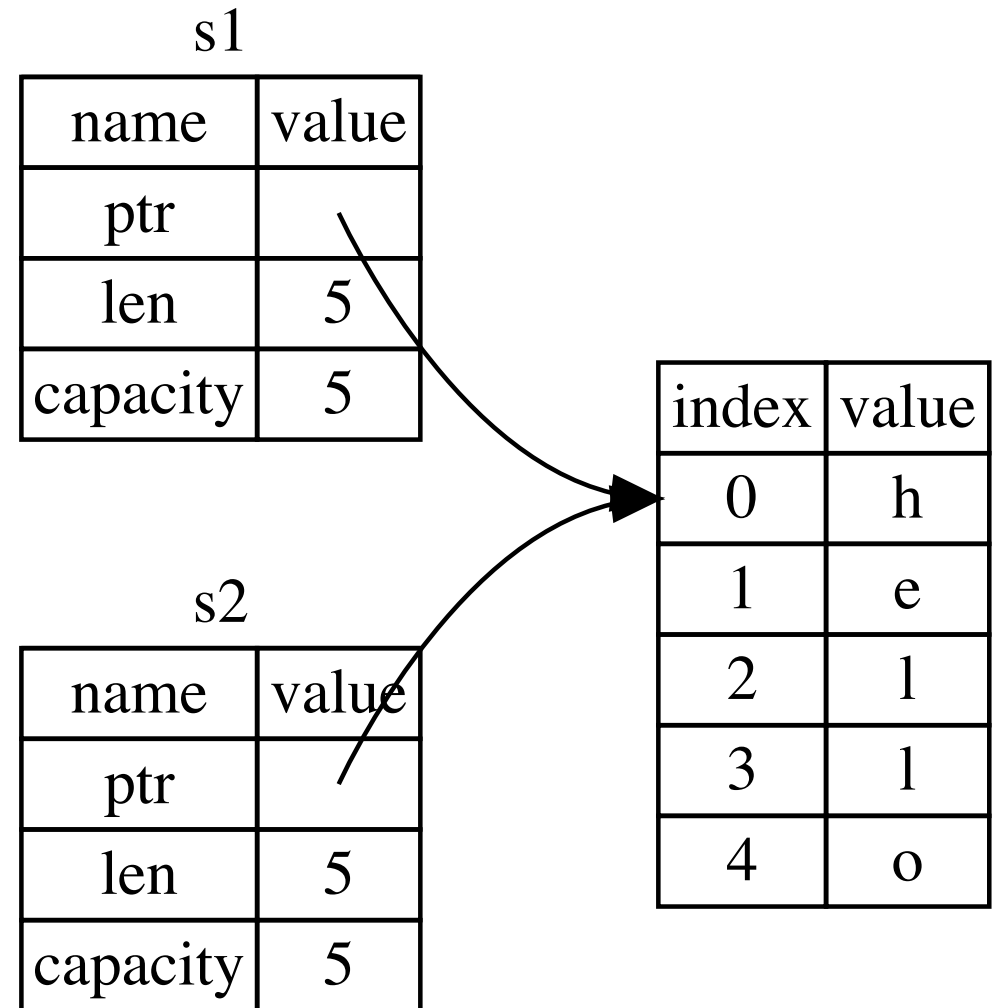


Pointer aliasing 🤔

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


Here's one option we could make:

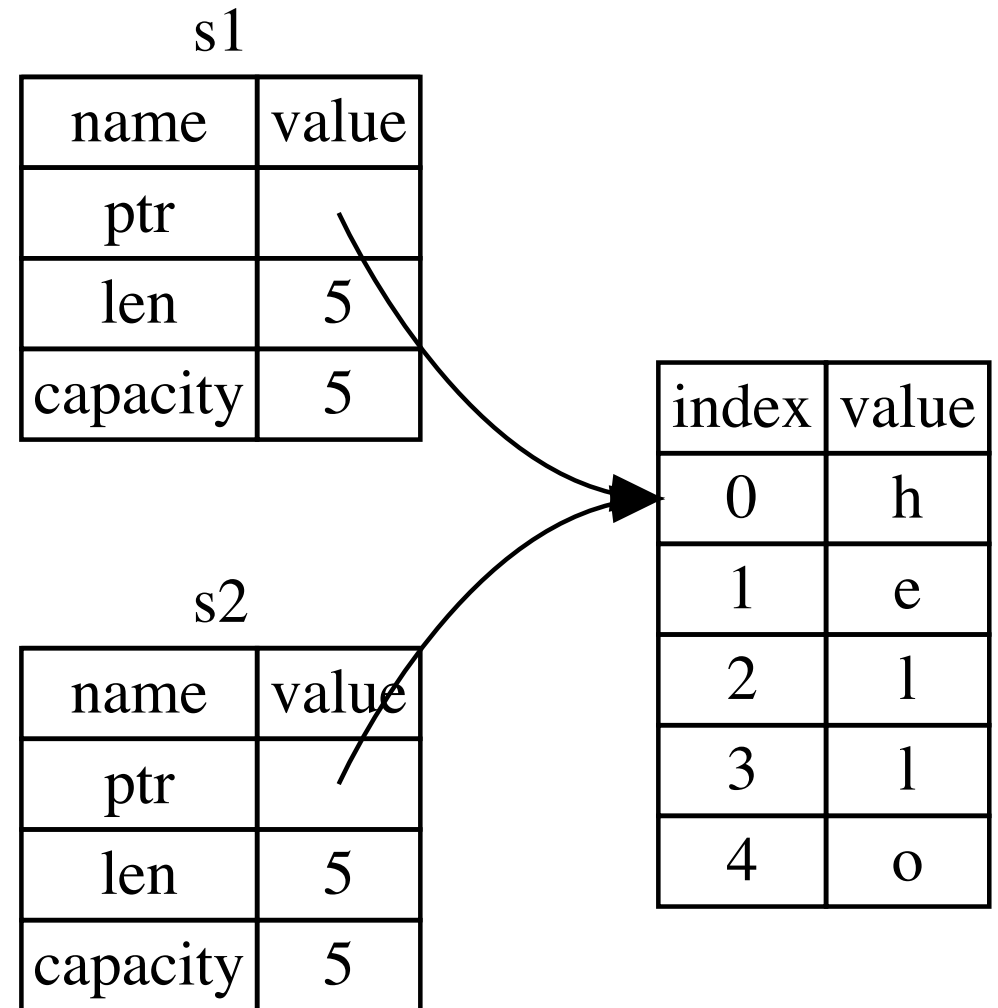
- When we assign `s1` to `s2`, only the stack data is copied
- We do *not* create a copy of the contents of the `String`
- Known as a "shallow copy" in some languages



Pointer aliasing

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

- Suppose this was how Rust actually handled this code.
- Following Rust's scope rules, what would happen if we tried to drop both `s1` and `s2`?
 - Double free! 
- How can we solve this?

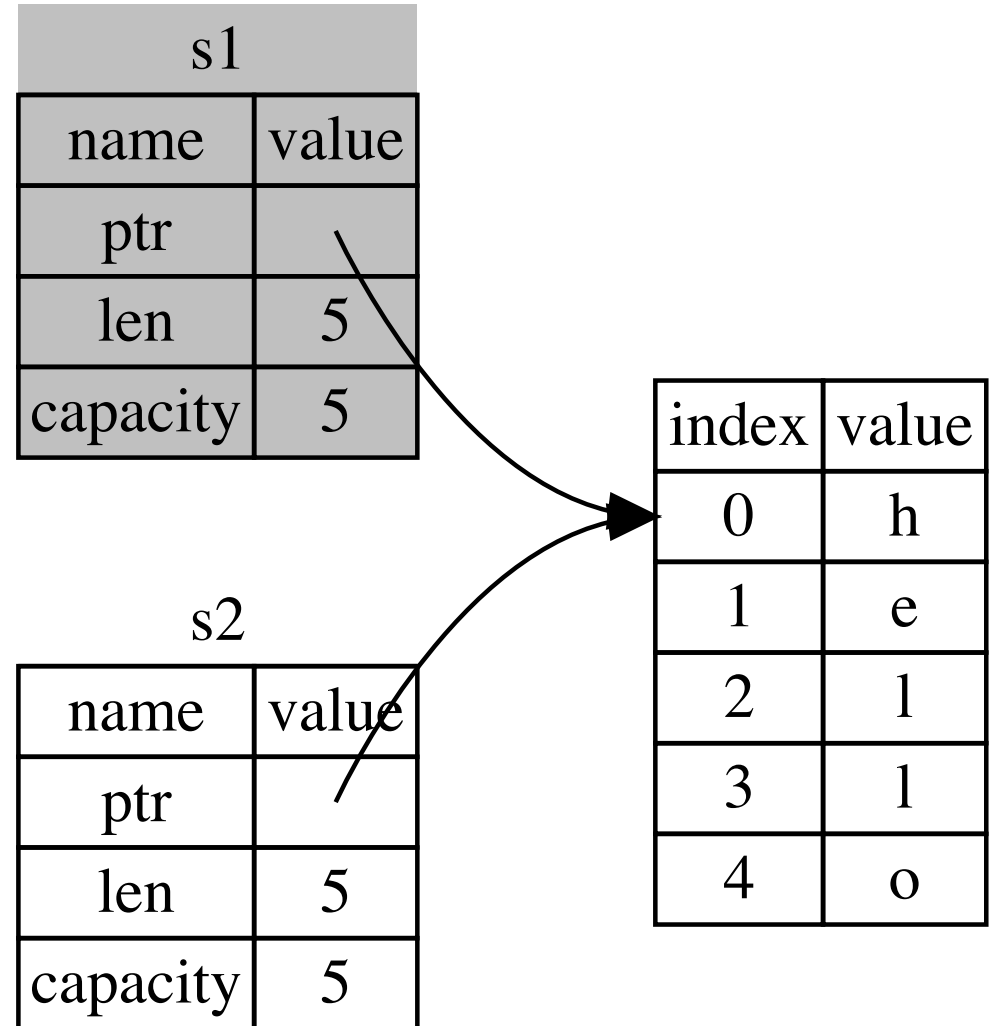


One owner at a time

To ensure memory safety, after the second line, `s1` is no longer valid.

```
let s1 = String::from("hello");  
let s2 = s1; // s1 is no longer valid
```

- *Grayed out portion is no longer accessible to the program*



What happens if we try to use `s1` after it is invalid?

```
let s1 = String::from("hello");
let s2 = s1;
println!("{}", world!", s1);
```

```
error[E0382]: borrow of moved value: `s1`
```

```
2 |     let s1 = String::from("hello");
  |           -- move occurs because `s1` has type `String`,
  |             which does not implement the `Copy` trait
3 |     let s2 = s1;
  |           -- value moved here
4 |
5 |     println!("{}", world!", s1);
  |                               ^^ value borrowed here after move
```

help: consider cloning the value if the performance cost is acceptable

```
3 |     let s2 = s1.clone();
  |           ++++++
```

Move semantics

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

- Rust calls this shallow copy plus invalidation a *move*
- We *moved* `s1` into `s2`
- This is a very intentional design choice in Rust
 - Rust will never create a "deep" copy of your data
 - No hidden costs!

Clone

If we do want to deep copy, we can use a common method called `clone`.

```
let s1 = String::from("hello");  
let s2 = s1.clone();  
  
println!("s1 = {}, s2 = {}", s1, s2);
```

```
s1 = hello, s2 = hello
```

- This copies *all* of the data contained in `s1`, both on the heap and the stack
- We'll talk more about methods next week!

What about integers?

Recall this code:

```
let x = 5;  
let y = x;  
  
println!("x = {}, y = {}", x, y);
```

```
x = 5, y = 5
```

Why does this work?

- `x` is still valid, but it looks like we moved it into `y`
- We just said that this wasn't allowed!

Copy

```
let x = 5;  
let y = x;
```

- Types such as integers have a known size at compile time
- They are also stored on the stack
- Copies of integers are quick to make
- There is no difference between a shallow copy and a deep copy here, so there is no need to call `clone`

Copy

Rust annotates certain types with a `Copy` trait, and these types allow variables that use it to be valid even after copying it to another variable.

Here are some of the types that are `Copy`:

- All number types, including integers and floating points
- Boolean type, `bool`
- Character type, `char`
- Tuples, if they only contain types that are `Copy`
 - `(i32, i32)` is `Copy`, but `(i32, String)` is not

Ownership and Functions

Passing a variable to a function will move or copy, just as assignment does.

Passing a `String` :

```
fn main() {
    let s = String::from("hello");
    takes_ownership(s);
    // println!("{}", s);
} // Because `s`'s value was moved, `s` is not dropped

        // `some_string` comes into scope
fn takes_ownership(some_string: String) {
    println!("{}", some_string);
} // Here, `some_string` goes out of scope and `drop` is called.
    // The backing memory is freed.
```

Ownership and Functions

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

```
error[E0382]: borrow of moved value: `s`
  --> src/main.rs:4:36
2 |   let s = String::from("hello");
  |       - move occurs because `s` has type `String`,
  |         which does not implement the `Copy` trait
3 |   takes_ownership(s);
  |                   - value moved here
4 |   println!("{}", s);
  |                   ^ value borrowed here after move
```

Ownership and Functions

Passing an `i32` :

```
fn main() {  
    let x = 5;  
    makes_copy(x);  
    println!("Here is {} again!", x);  
}  
  
fn makes_copy(some_integer: i32) {  
    println!("{}", just got copied", some_integer);  
}
```

Return Values and Scope

Returning values can also transfer ownership.

```
fn main() {  
    let s1 = gives_ownership();  
    println!("{}", s1);  
}  
  
fn gives_ownership() -> String {  
    let some_string = String::from("yours");  
  
    some_string // `some_string` is returned and  
                // moves out to the calling function  
}
```

Return Values and Scope

Another example of taking and giving back ownership:

```
fn main() {
    let s2 = String::from("hello");
    let s3 = takes_and_gives_back(s2);
    println!("{}", s3);
} // Here, `s3` goes out of scope and is dropped.
  // `s2` was moved, so nothing happens to `s2`.

fn takes_and_gives_back(a_string: String) -> String {
    a_string // a_string is returned and
             // moves out to the calling function
}
```


Recap: Ownership

- Ownership rules:
 - Each value in Rust has an *owner*
 - There can only be one owner at a time
 - When the owner goes out of scope, the value will be *dropped*
- With just ownership, we can either move, copy, or clone
 - Moving and copying has no overhead
 - Cloning is expensive

Moving is somewhat tedious

```
fn main() {  
    let s1 = String::from("hello");  
    let (s2, len) = calculate_length(s1);  
    println!("The length of '{}' is {}.", s2, len);  
}  
  
fn calculate_length(s: String) -> (String, usize) {  
    let length = s.len();  
    (s, length)  
}
```

- If we want to give a function some data, it seems we need to *move* the data into the function
- To get it back, it seems we need to also return the data back every time
- *What if we want to let a function use a value but not take ownership?*

References and Borrowing

References

- Moving into and returning data from a function is a lot of work
- Rust has a feature specifically for using a value without transferring ownership called *references*
- We can share memory using these *references*

Reference with &

Instead of moving a value into a function, we can provide a *reference* to the value.

We use `&` to define a reference to a value.

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

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

- Let's break this down!

Reference with &

```
fn main() {  
    let s1 = String::from("hello");  
  
    let len = calculate_length(&s1);  
  
    println!("The length of '{}' is {}.", s1, len);  
}  
  
fn calculate_length(borrowed: &String) -> usize {  
    borrowed.len()  
}
```

- The `&s1` syntax lets us create a variable that *refers* to the value of `s1`

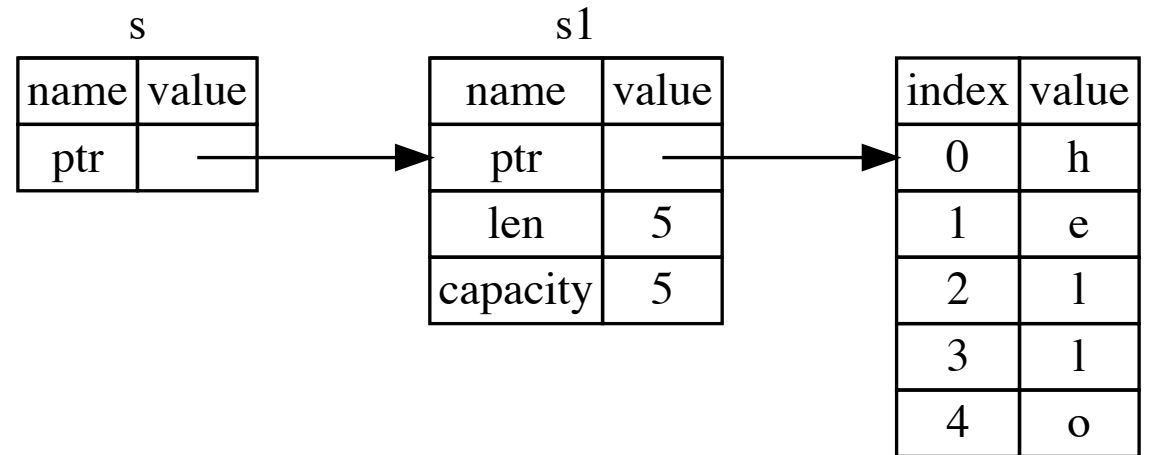
References as Function Arguments

```
                // `borrowed` is a reference to a String
fn calculate_length(borrowed: &String) -> usize {
    borrowed.len()
} // Here, `borrowed` goes out of scope
```

- `borrowed == &s1`
- We do not own `s1` if we just have a reference to it
- This means `s1` will *not* be dropped when we stop using `borrowed`
- We call this *borrowing*

Reference Data Layout

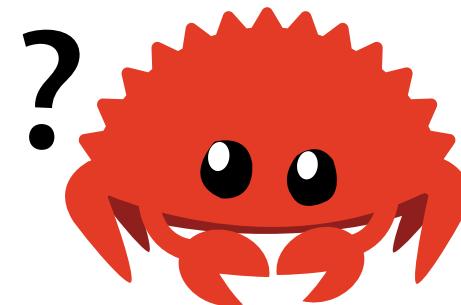
- In memory, references are just like pointers
- But they have several constraints that make them different...



Mutating a Reference

What if we want to modify the value of something we've borrowed through a reference?

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



Mutable References

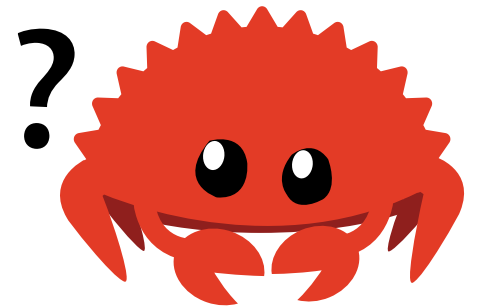
If we want to modify the value that we've borrowed, we must use a mutable reference, denoted `&mut val`.

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

Mutable References are Exclusive

If you have a mutable reference to a value, you can have no other references to that value.

```
let mut s = String::from("hello");  
  
let r1 = &mut s;  
let r2 = &mut s;  
  
println!("{}", {}, r1, r2);
```



Mutable References are Exclusive

```
let mut s = String::from("hello");
let r1 = &mut s;
let r2 = &mut s;
println!("{}", {}, r1, r2);
```

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
--> src/main.rs:5:14
```

```
4 |     let r1 = &mut s;
   |             ----- first mutable borrow occurs here
5 |     let r2 = &mut s;
   |             ^^^^^^^ second mutable borrow occurs here
6 |
7 |     println!("{}", {}, r1, r2);
   |                       -- first borrow later used here
```

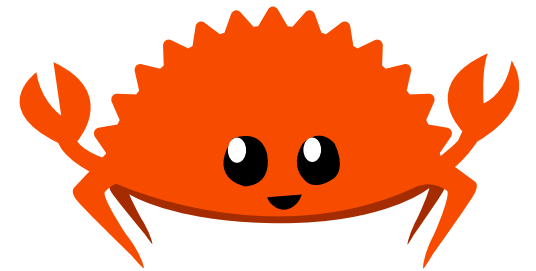
Mutable References are Exclusive

- Most languages will let you mutate anything, whenever you want
- If data can be written to from multiple places, the value can become unpredictable
- Making mutable references exclusive prevents data races at compile time!

Multiple Mutable References

We are allowed to have multiple mutable references, just not *simultaneously*.

```
let mut s = String::from("hello");  
  
{  
    let r1 = &mut s;  
} // r1 goes out of scope here, so we can make  
  // a new mutable reference with no problems  
  
let r2 = &mut s;
```



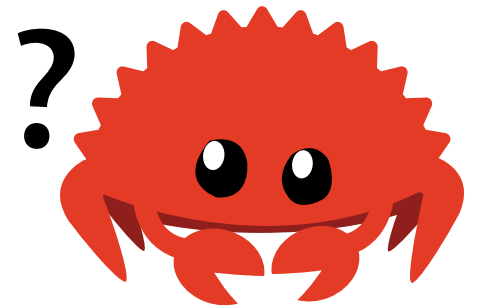
Mutable and Immutable References

We cannot have both an immutable and mutable reference to the same value.

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

let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM

println!("{}", r1, r2, r3);
```



Mutable and Immutable References

```
let mut s = String::from("hello");
let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM
println!("{}", r1, r2, r3);
```

```
error[E0502]: cannot borrow `s` as mutable because
             it is also borrowed as immutable
```

```
--> src/main.rs:6:14
```

```
4 |         let r1 = &s; // no problem
   |                   -- immutable borrow occurs here
5 |         let r2 = &s; // no problem
6 |         let r3 = &mut s; // BIG PROBLEM
   |                   ^^^^^^ mutable borrow occurs here
7 |
8 |         println!("{}", r1, r2, r3);
   |                               -- immutable borrow later used here
```

Mutable and Immutable References

Note that these rules only apply for references whose scopes overlap.

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

let r1 = &s; // no problem
let r2 = &s; // no problem
println!("{}", r1, r2);
// variables r1 and r2 will not be used after this point

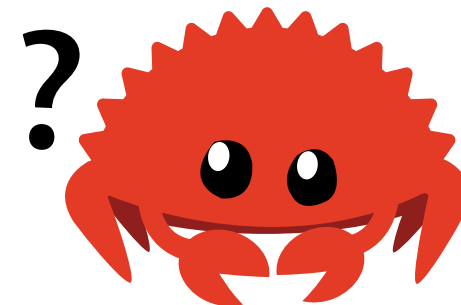
let r3 = &mut s; // no problem
println!("{}", r3);
```

- The scope of a reference starts when it is initialized
- The scope of a reference **ends at the last point it is used**
- The specific term for reference scopes are *lifetimes*
 - We'll talk about lifetimes in week 8!

Dangling References

The Rust compiler guarantees that references will never be invalid, which means it will not allow dangling references.

```
fn main() {  
    let reference_to_nothing = dangle();  
}  
  
fn dangle() -> &String {  
    let s = String::from("hello");  
  
    &s  
}
```



Dangling References

```
error[E0106]: missing lifetime specifier
  --> src/main.rs:5:16
   |
5  | fn dangle() -> &String {
   |                ^ expected named lifetime parameter
   = help: this function's return type contains a borrowed value,
         but there is no value for it to be borrowed from
   help: consider using the `static` lifetime
<-- snip -->
```

Focus on this line:

this function's return type contains a borrowed value, but there is no value for it to be borrowed from

- Again, this relies on lifetimes, so just keep this in mind!

Reference Rules

- At any given time, you can have either one mutable reference or any number of immutable references
 - A book being read by multiple people is fine
 - If more than one person writes, they may overwrite each other's work
 - References are similar to Read-Write locks
- References must always be valid

The Borrow Checker

The Borrow Checker enforces the rules of ownership and borrowing by checking:

- That all variables are initialized before they are used
- That you can't move the same value twice
- That you can't move a value while it is borrowed
- That you can't access a place while it is mutably borrowed (except through the mutable reference)
- That you can't mutate a place while it is immutably borrowed
- and more...

Slices

Slices

- *Slices* let you reference a contiguous sequence of elements in a collection rather than the whole collection
- A slice is similar to a reference, so it does not have ownership

Slices

Suppose we want to write this function:

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

- Find the first space and return all the characters before it
- What type should we return?

String Slices

A *string slice* is sometimes a reference to part of a `String`, and it looks like this:

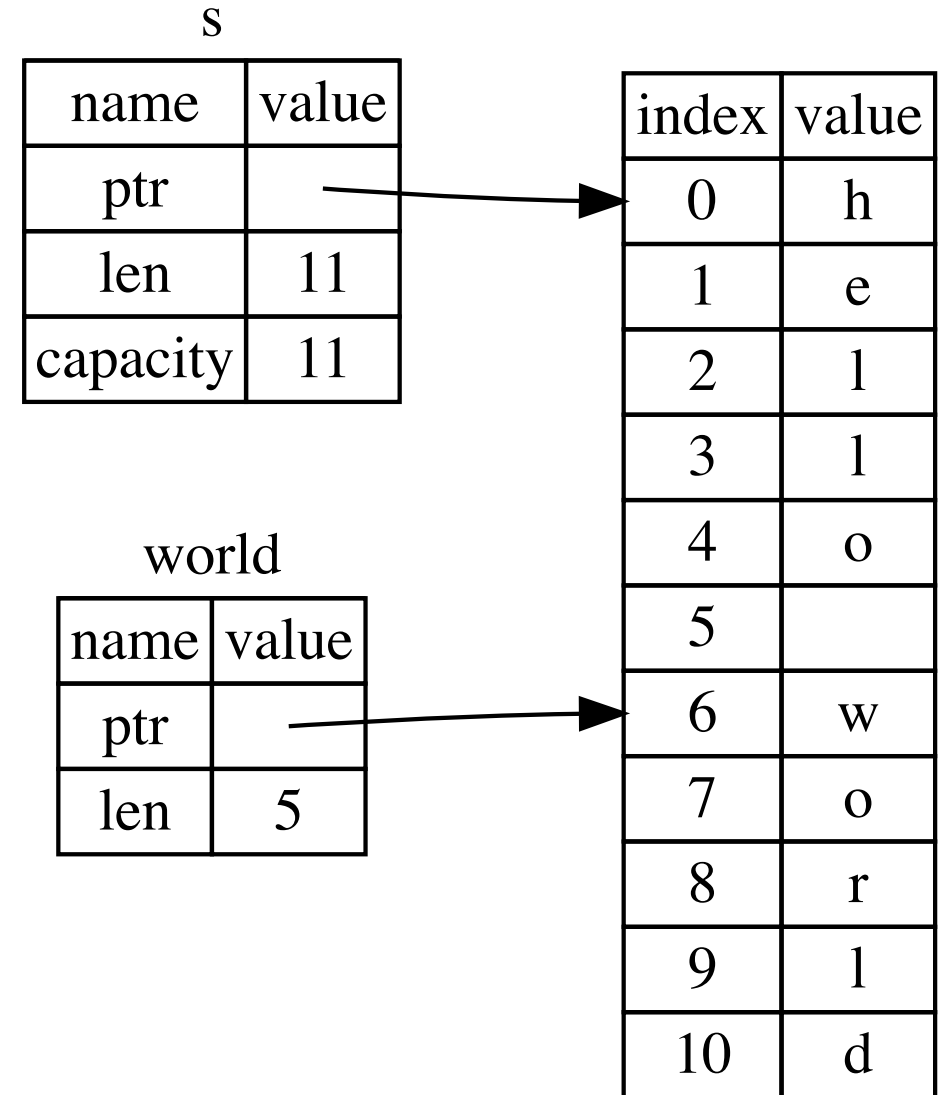
```
let s = String::from("hello world");  
  
let hello = &s[0..5];  
let world = &s[6..11];
```

- `hello` contains the first 5 characters of `s`
- `world` contains the 5 characters starting at the 6th index of `s`

String Slices

```
let s = String::from("hello world");  
let hello = &s[0..5];  
let world = &s[6..11];
```

- A string slice stores a pointer to memory and a length



String Slices

You can shorthand ranges with the `..` syntax.

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

let slice = &s[0..2];
let slice = &s[..2];

let len = s.len();
let slice = &s[3..len];
let slice = &s[3..];

let slice = &s[0..len];
let slice = &s[..];
```

String Literals are Slices

Recall that we talked about string literals being stored inside the binary. Now that we know about slices, we can explain what string literals really are:

```
let s = "Hello, world!";
```

- The type of `s` here is `&str`: it's a slice pointing to that specific point of the binary with type `str`
- Question: Why are string literals immutable?
 - `&str` is an immutable reference!

Owned Types

- String slices and string literals are immutable because they are a special type of immutable reference, so we can't mutate them
- String is an owned type, we can do whatever we want with it
- What are some other owned types?

Vectors

Vectors allow you to store a collection of values of the same type contiguously in memory. Internally, it is a dynamically sized array stored on the heap.

You can create an vector with the method `new` :

```
let v: Vec<i32> = Vec::new();
```

- The `<i32>` just means that the vector stores `i32` values. We'll talk more about this syntax in the Week 4!

Updating a `Vec`

To add elements to a `Vec`, we can use the `push` method.

```
let mut v = Vec::new();  
  
v.push(5);  
v.push(6);  
v.push(7);  
v.push(8);  
  
println!("{:?}", v);
```

```
[5, 6, 7, 8]
```


vec! Macro

Rust provides a *macro* to create vectors easily in your programs.

```
let v = vec![1, 2, 3];  
println!("{:?}", v);
```

```
[1, 2, 3]
```

- We'll talk about macros in Week 12!

Reading Elements of Vectors

You can index into a vector to retrieve a reference to an element.

```
let v = vec![1, 2, 3, 4, 5];  
  
let third: &i32 = &v[2];  
println!("The third element is {}", third);
```

- Note that Rust will panic if you try to index out of the bounds of the `Vec`

More `Vec<T>` to come...

We will talk more about `String` and `Vec<T>` in Week 4!

Homework 2

- The second homework consists entirely of 10 small puzzles
 - Refer to the `README.md` for further instructions
 - Always follow the compiler's advice!
- We *highly* recommend reading the Rust Book chapter on [ownership](#)
 - Ownership is a very tricky concept that affects almost every aspect of Rust, so understanding it is key to writing more complex Rust code
- Try your best to understand Ownership *before* attempting the homework

Next Lecture: Structs and Enums

- Thanks for coming!

