Lecture 5

Introduction to Ownership

What we will cover today

Ownership!

Optional Reading:
The Rust Book Chapter 4.1 – What is Ownership?

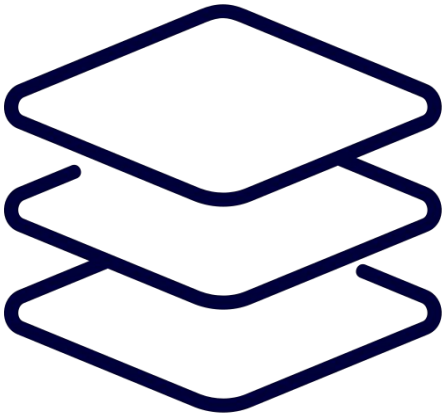
Ownership

Ownership is Rust's most unique feature!

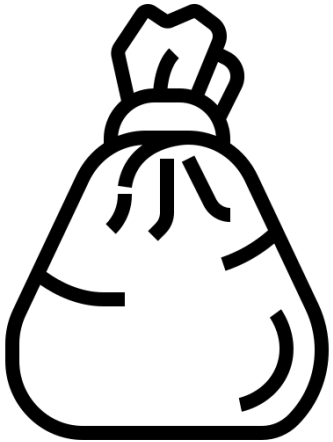
- A set of rules for how a Rust program manages memory
- Rust's ownership model is very different from many other programming languages

Brief detour: Stack vs Heap

All programs need to have some memory space to store variables when they run – Heap and stack



Stack



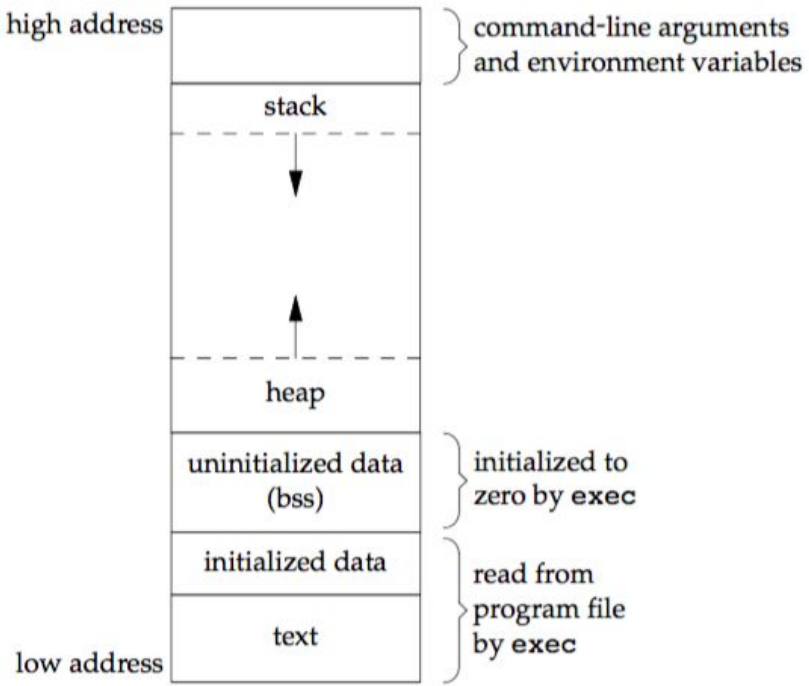
Heap

Sources: https://www.flaticon.com/free-icon/garbage-bag_5496108, https://www.flaticon.com/free-icon/layers_3171678

Brief detour: Stack vs Heap

A typical C program memory layout

Note the stack and heap

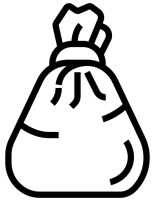


Sources: https://testtemph.blogspot.com/2017/02/memory-layout-of-c-program.html

Brief detour: Stack vs Heap

Stack

- Does not need to allocate space
- Fast to access values
- Limited size
- Only store data with known, fixed size
- E.g. Storing 1 int



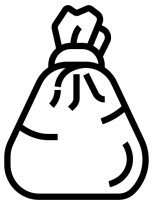
Heap

- Needs to allocate space
- Slow to access values
- Can be expanded
- Can be used for data of unknown size
- E.g. Storing the name that you get from user input

Analogy for heap

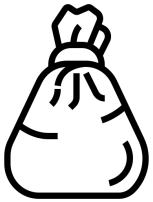
Heap is like a restaurant

- When you first come in, you tell the host how many people are in your group, and they find a table for you
- Similar to asking for memory on the heap, the amount is not known ahead of time
- Takes time to find you a table, similarly, takes time to allocate memory on the heap



Heap

Memory Management

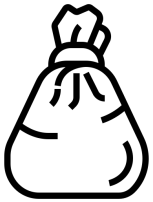


Heap

Tasks in memory management

- Ask heap for some blocks of memory
- Keep track of what memory spaces we are using
- Return memory we are not using

Memory Management

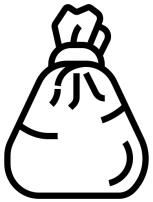


Heap

Approach 1: Manual Memory Management (E.g. C, C++)

- The programmer (that's you!) asks for some memory space
- Manually returns the space when you're done
- What is bad? Humans are error prone!
  - Free too early – Invalid variable
  - Free too late – Waste of memory space
  - Free twice – Undefined behaviour

Memory Management



Heap

Approach 2: Garbage Collection (E.g. java)

- A special process happens when the program is running
- Special process automatically returns memory you're not using anymore
- What's bad?
  - Extra process, performance is affected

Why are we talking about this?

Rust uses a different approach – Ownership!

Key idea: Ownership helps us to manage data on the heap

- Fast – Doesn't use an extra mysterious process like garbage collection does
- No manual memory management – Lesser memory errors from programmers' mistake

Ownership rules

1. Each value in Rust has an owner
2. There can only be **1 owner** at a time
3. When owner is out of scope, the value will be dropped

Brief: Variable scope

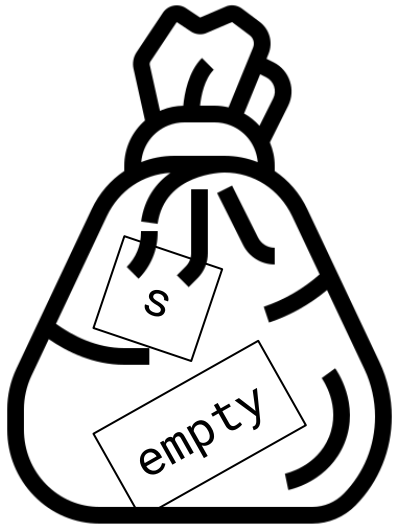
Scope – The region in which an item is valid

```
{ // s is not valid
    let s = "hello"; // s is valid
    ...
} // s is no longer valid
```

A data that lives on the heap: String

Recall: &str (string literal) vs String

```
let s = String::from("hello");
let empty = String::new();
empty.push_str("Something");
```



drop()

Recall: scope – The region in which an item is valid

When value on heap goes out of scope, drop() function is called to free the memory

```
{ // s is not valid
    // s is valid
    let s = String::from("hello");
    ...
} // s goes out of scope, Rust calls drop()
```

drop vs Garbage collection

```
{ // s is not valid
    // s is valid
    let s = String::from("hello");
    ...
} // s goes out of scope, Rust calls drop()
```

drop – Happens on compile time, compiler inserts instructions to do the drop

GC – Happens during runtime, some runtime process checks for unused memory

Why ownership?

Look at the following code example:

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

Two variables, x and y, both equal to 5

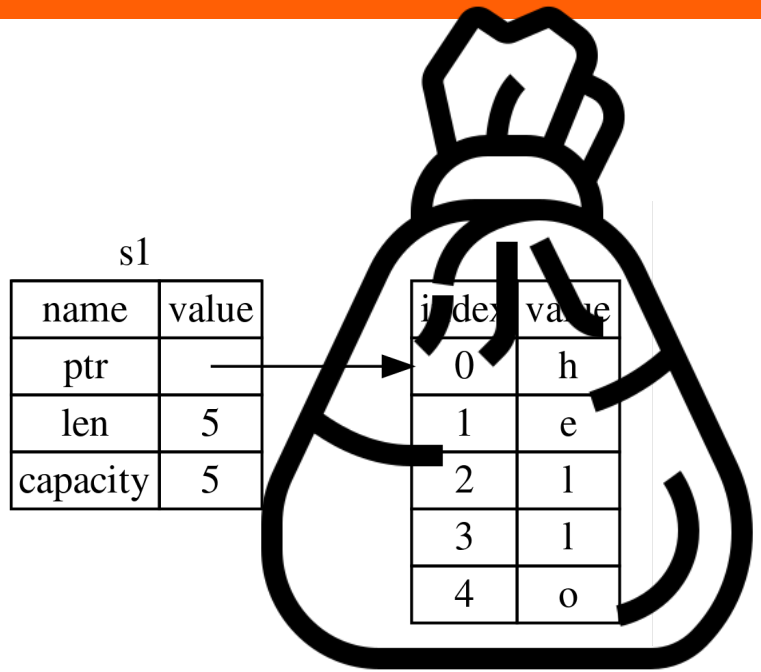
Both values are on the stack

Why ownership?

Look at the following code example:

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

What do you think is happening here?

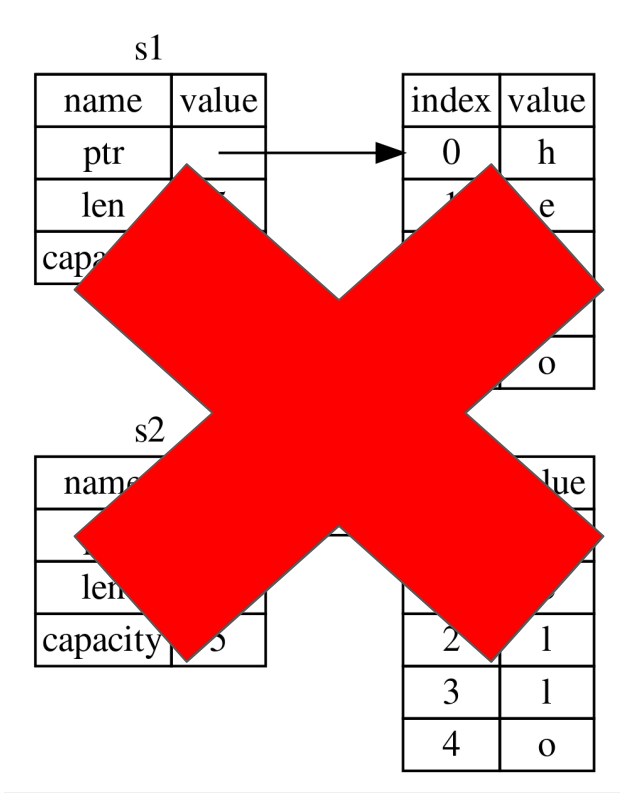


String copying??

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

Does it look like this?

Why not? Too expensive

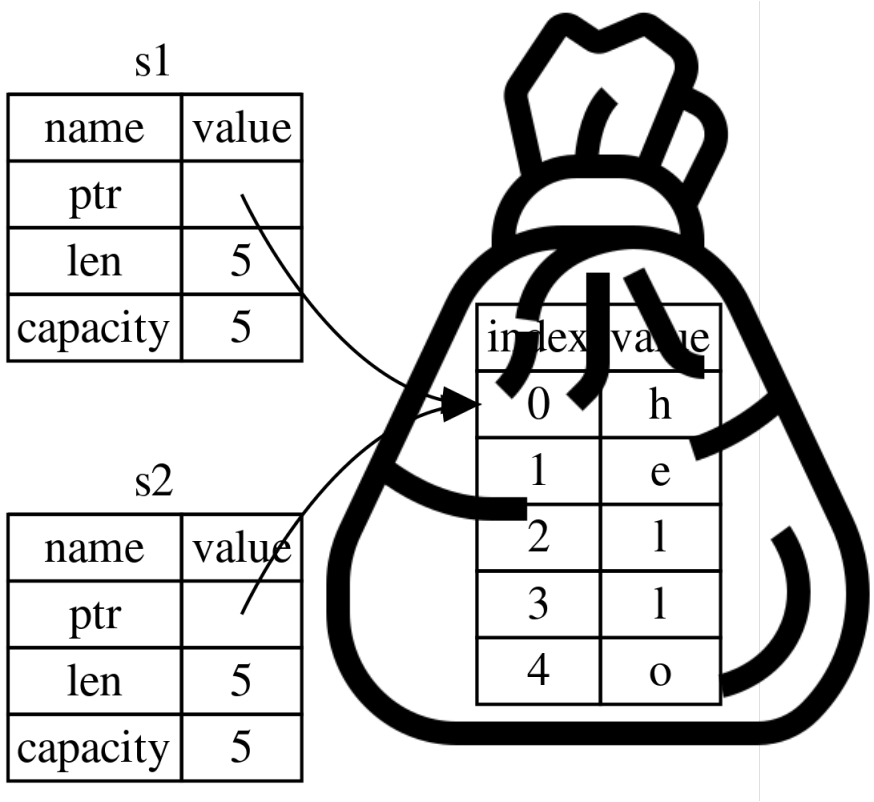


String copying??

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

This is closer to how it **might**

look like (Shallow copy)

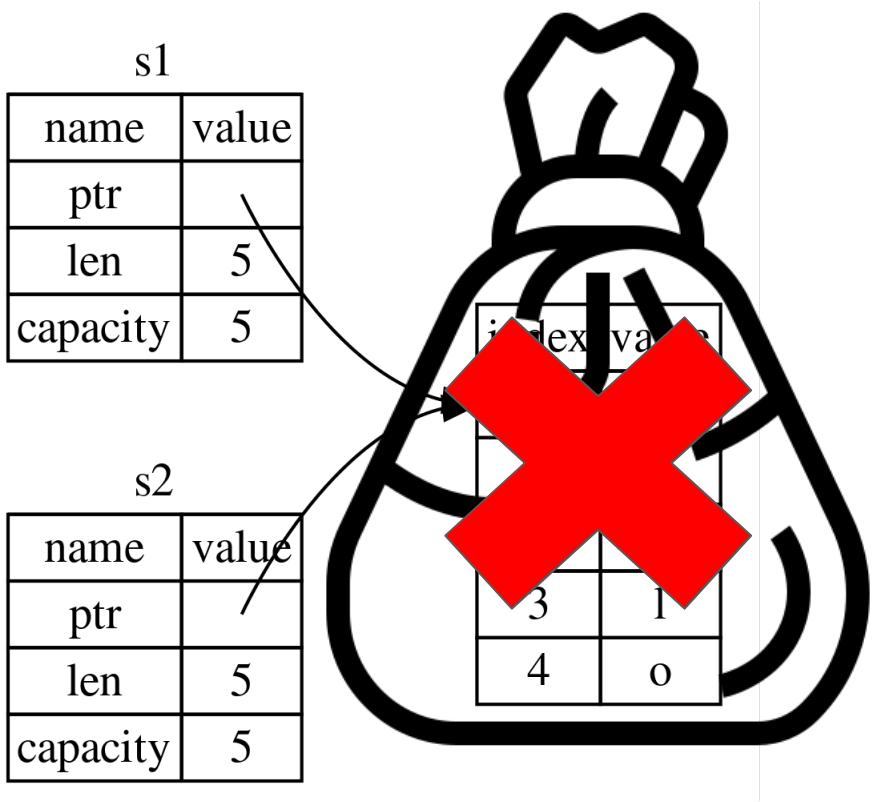


There is still a problem…

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

What happens when both s1 and s2 go out of scope?

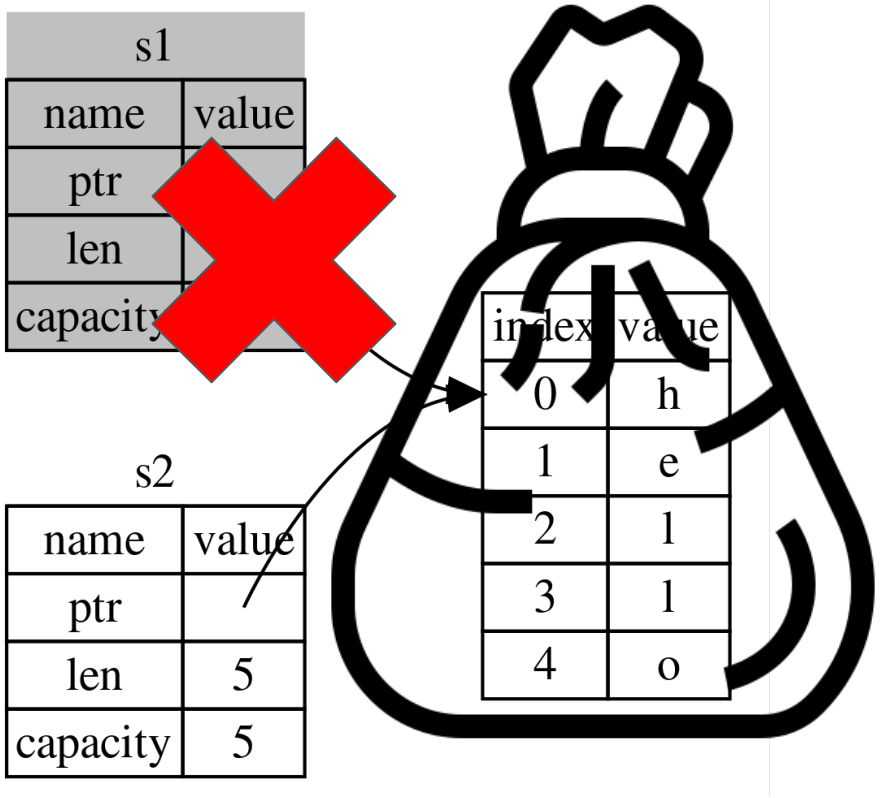
Remember that Rust calls drop() for you, so it will free the memory on the heap for you. Double free error!



What really happens…

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

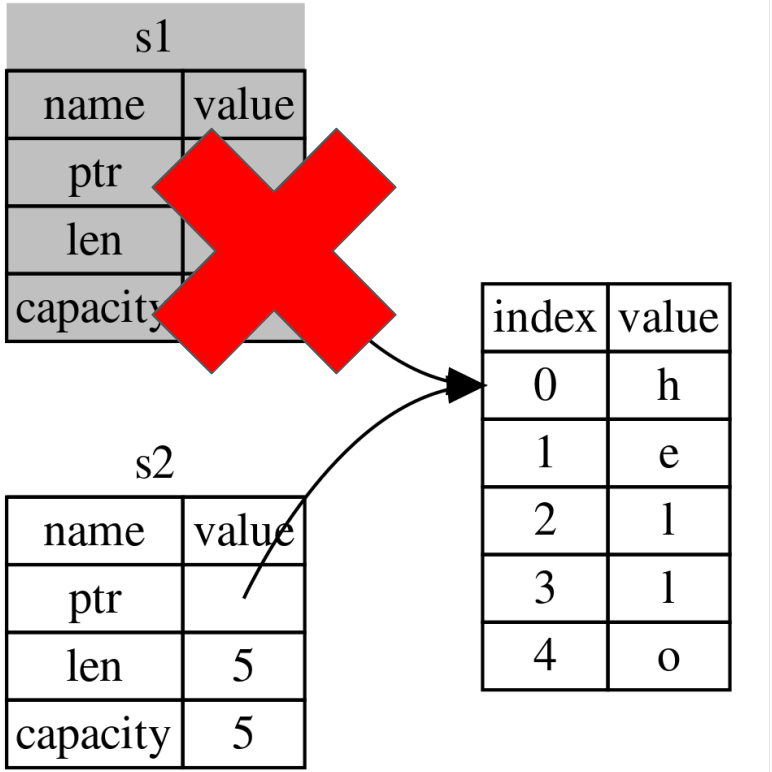
s1 gets invalidated. We call this a **move**. s1 is *moved* into s2



Recall: Ownership rules

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

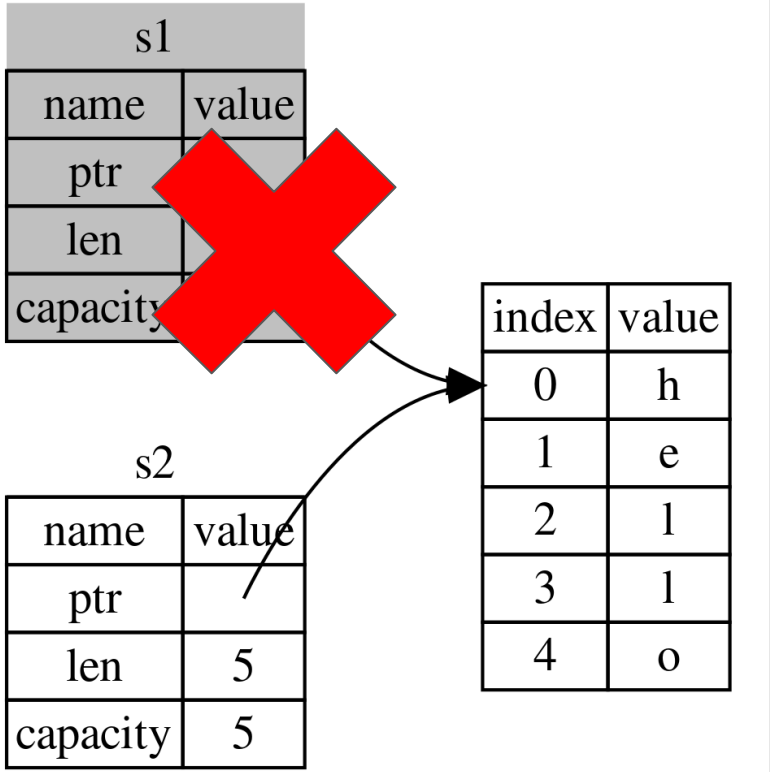
1. Each value in Rust has an owner
2. There can only **1 owner** at a time
3. When owner is out of scope, the value will be dropped



Recall: Ownership rules

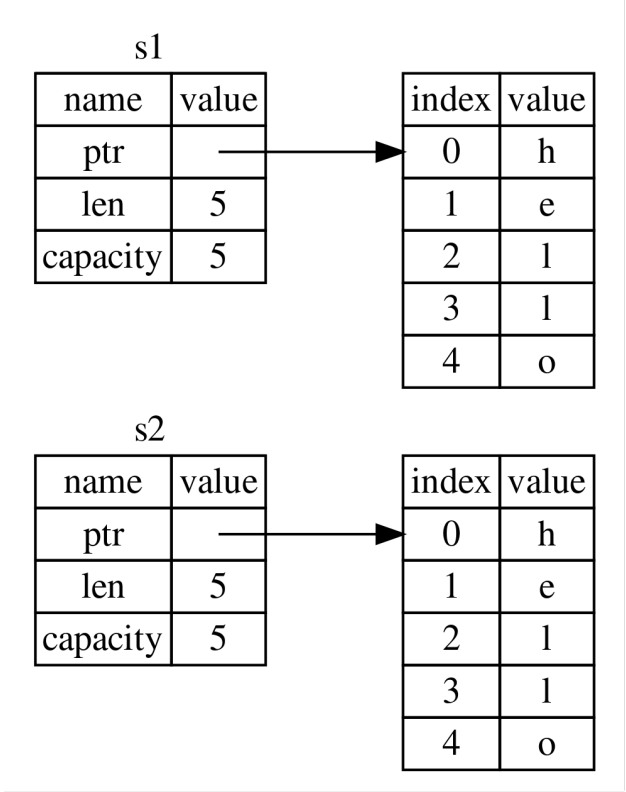
```
let s1 = String::from("hello");
let s2 = s1;
```

This means that s1 is no longer valid!



clone()

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



Demo: moving a variable

Recall: Ownership rules

1. Each value in Rust has an owner
2. There can only **1 owner** at a time
3. When owner is out of scope, the value will be dropped

Variables getting moved

```
fn main() {
    let s = String::from("hello");  // s comes into scope

    takes_ownership(s);             // s's value moves into the function...
                                    // ... and so is no longer valid here

    let x = 5;                      // x comes into scope

    makes_copy(x);                  // x would move into the function,
                                    // but i32 is Copy, so it's okay to still
                                    // use x afterward

} // Here, x goes out of scope, then s. But because s's value was moved, nothing
  // special happens.

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

fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{some_integer}");
} // Here, some_integer goes out of scope. Nothing special happens.
```

Variables getting moved

```
fn main() {
    let s1 = gives_ownership();         // gives_ownership moves its return
                                        // value into s1

    let s2 = String::from("hello");     // s2 comes into scope

    let s3 = takes_and_gives_back(s2);  // s2 is moved into
                                        // takes_and_gives_back, which also
                                        // moves its return value into s3
} // Here, s3 goes out of scope and is dropped. s2 was moved, so nothing
  // happens. s1 goes out of scope and is dropped.

fn gives_ownership() -> String {             // gives_ownership will move its
                                             // return value into the function
                                             // that calls it

    let some_string = String::from("yours"); // some_string comes into scope

    some_string                              // some_string is returned and
                                             // moves out to the calling
                                             // function
}

// This function takes a String and returns one
fn takes_and_gives_back(a_string: String) -> String { // a_string comes into
                                                      // scope

    a_string  // a_string is returned and moves out to the calling function
}
```

Announcements

HW3 released today on PrairieLearn

Due 1 week from now — Next Wednesday 02/19 23:59