
# Ownership

# Goals For Today


- Introduce Ownership

# Don’t Forget!


- HW1 Due Tonight!
- HW2 Due Thursday
- MP0 Due Friday

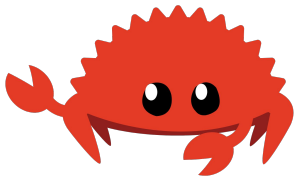
# But First...

lowkey wish prairelearn gave compilation stuff

what's the difference between the '_' keyword and 'other' keyword in the match statement? When should I use one over the other?

Do we get to see model answers for the hw and mp after the deadline? I don't think I did the hw efficiently but I got full points so I want to know how to fix it

What's the object-oriented way to become wealthy? Inheritance.



# Ownership


How do we manage memory?

# Ownership


How do we manage memory?

- Garbage Collection (Lisp, JS, Java)

# Ownership


How do we manage memory?

- Garbage Collection (Lisp, JS, Java)
- Let the programmer worry about it (C, many others)

# Ownership


How do we manage memory?

- Garbage Collection (Lisp, JS, Java)
- Let the programmer worry about it (C, many others)
- Ownership (Rust)

# Ownership?


Ownership is a new concept. It can be confusing. The goal of this lecture is to introduce it.

There are three major rules of ownership:

# Ownership?


Ownership is a new concept. It can be confusing. The goal of this lecture is to introduce it.

There are three major rules of ownership:

1. Each value in Rust has variable called it's **owner**

# Ownership?


Ownership is a new concept. It can be confusing. The goal of this lecture is to introduce it.

There are three major rules of ownership:

1. Each value in Rust has variable called it's **owner**
2. There can only be one **owner**

# Ownership?


Ownership is a new concept. It can be confusing. The goal of this lecture is to introduce it.

There are three major rules of ownership:

1. Each value in Rust has variable called it's **owner**
2. There can only be one **owner**
3. When the owner goes out of scope, the value is dropped

# Ownership - Basics

Let’s remember how scope works

```
fn main(){
    {
        // x is not in scope (not initialized)
        let x = 5;
        // x is in scope and initialized
    }
    // x is no longer in scope
}
```

# Ownership - Basics

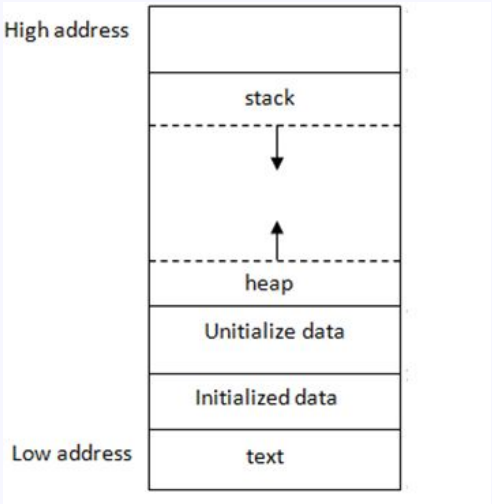
This is the same for string literals

```
fn main(){
    {
        // x is not in scope (not initialized)
        let x = "hello";
        // x is in scope and initialized
    }
    // x is no longer in scope
}
```

# Ownership - Basics

This is the same for string literals

Stack/heap refresher:

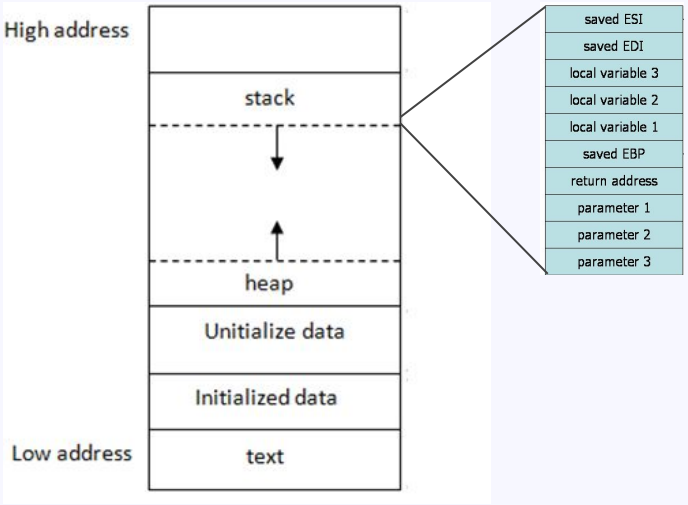



```
fn main(){
    {
        // x is not in scope (not initialized)
        let x = "hello";
        // x is in scope and initialized
    }
    // x is no longer in scope
}
```

# Ownership - Basics

This is the same for string literals

Stack/heap refresher:

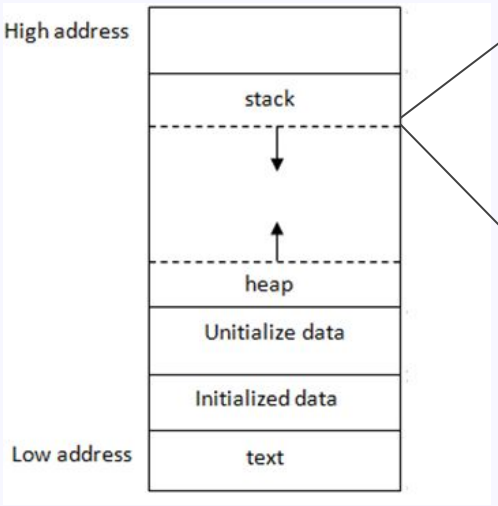


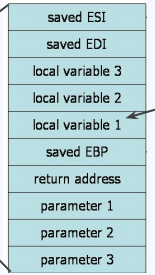

```
fn main(){
    {
        // x is not in scope (not initialized)
        let x = "hello";
        // x is in scope and initialized
    }
    // x is no longer in scope
}
```

# Ownership - Basics

This is the same for string literals

Stack/heap refresher:

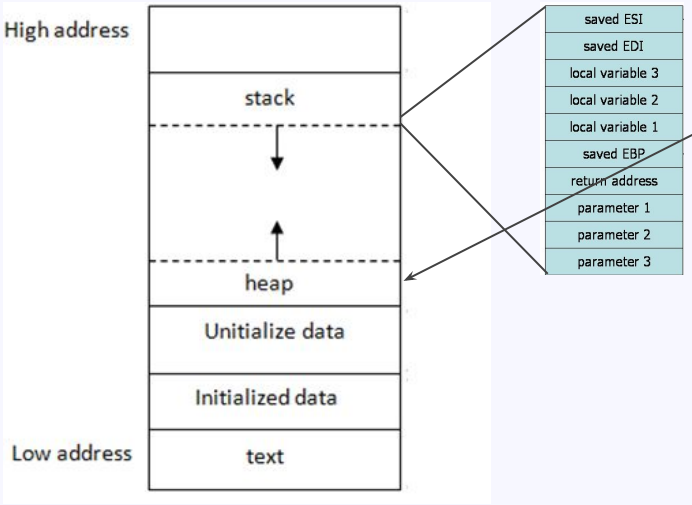



```
fn main(){
    {
        // x is not in scope (not initialized)
        let x = "hello";
        // x is in scope and initialized
    }
    // x is no longer in scope
}
```

# Ownership - Basics

This is not the same for objects we instantiate

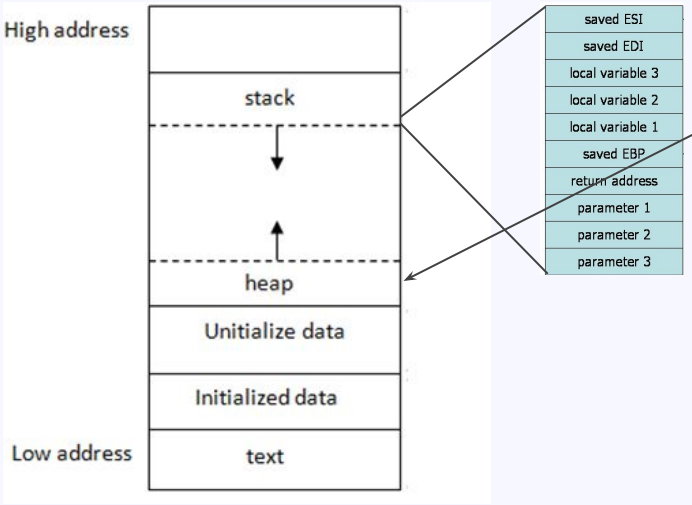



```
fn main(){
    {
        // x is not in scope (not initialized)
        let mut x = String::from("hello");
        x.push_str(", world!")
        // x is in scope and initialized
    }
    // x is no longer in scope
}
```

# Ownership - Basics

While the stack is 'torn down', the heap remains (we need to get rid of it)

Stack/heap refresher:

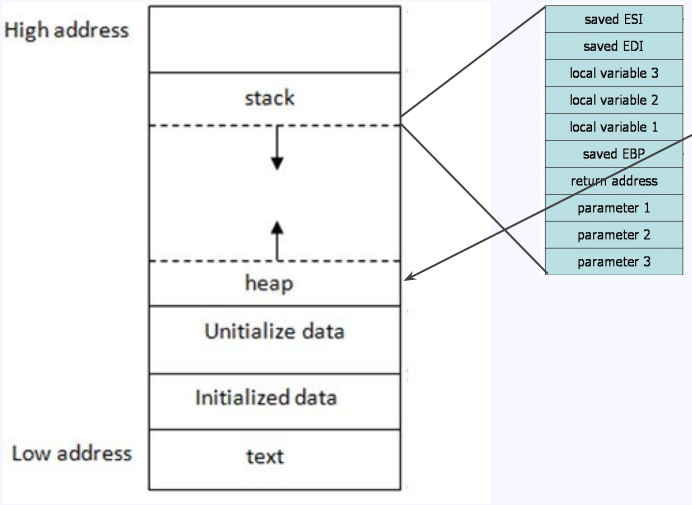



```
fn main(){
    {
        // x is not in scope (not initialized)
        let mut x = String::from("hello");
        x.push_str(", world!")
        // x is in scope and initialized
    }
    // x is no longer in scope
}
```

# Ownership - Basics

Rust will automatically ‘drop’ the data from the heap when x leaves scope.

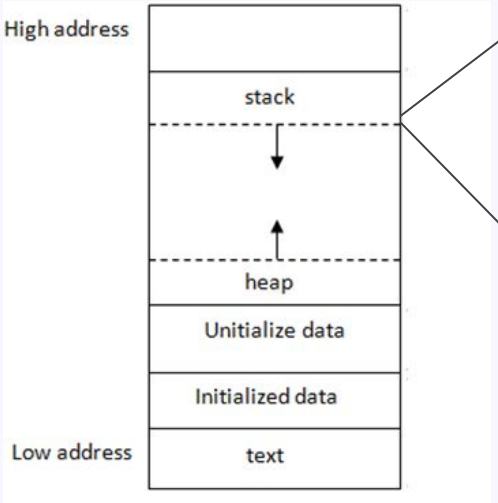
Stack/heap refresher:

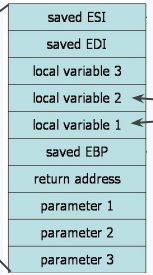



```
fn main(){
    {
        // x is not in scope (not initialized)
        let mut x = String::from("hello");
        x.push_str(", world!")
        // x is in scope and initialized
    }
    // x is no longer in scope
}
```

# Ownership - Basics

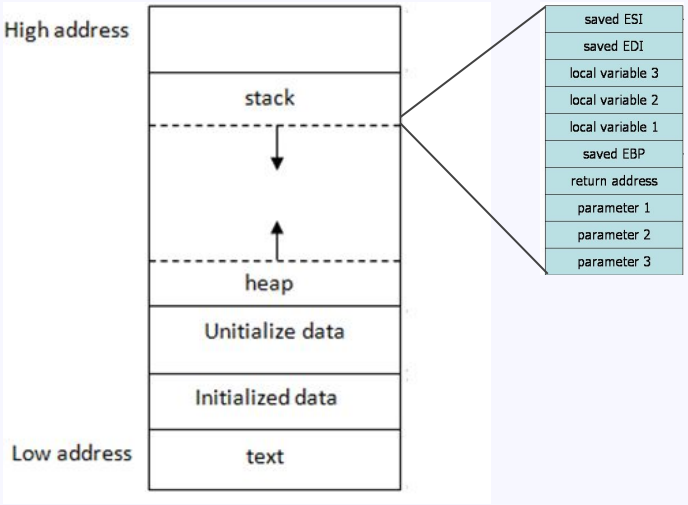
Let's take it up a notch. What's happening here?




```
fn main(){
    {
        let x = 5;
        let y = x;
        x = 6;
    }
}
```

## How about now?



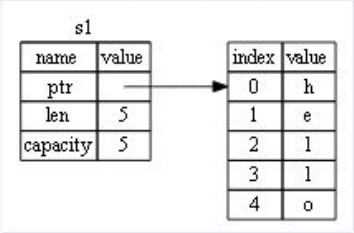

```
fn main(){
    {

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

    }
}
```

# Ownership - Basics

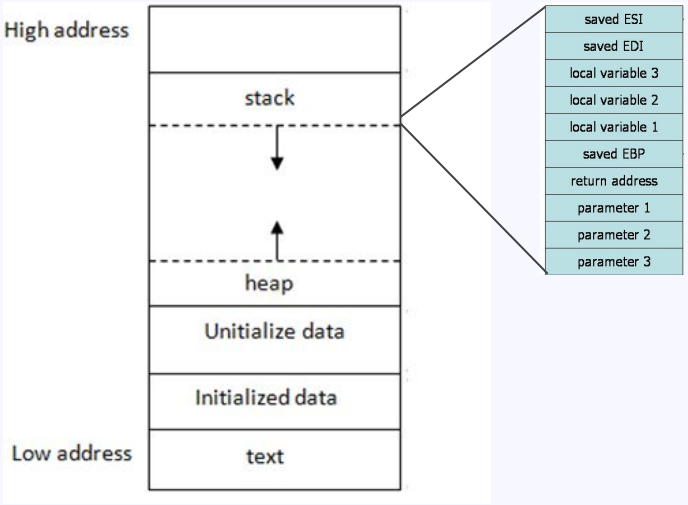
How about now?

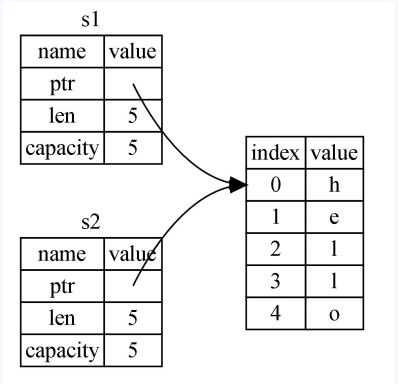


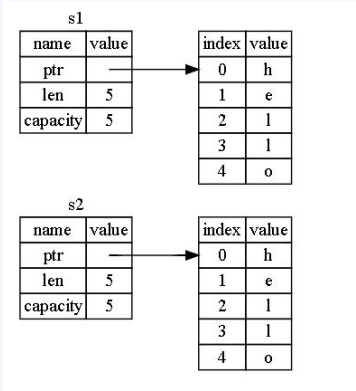

```
fn main(){
    {

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

    }
}
```

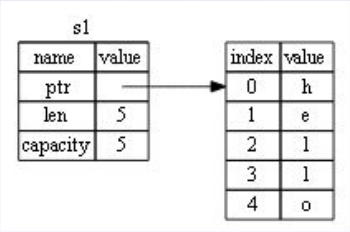


# Ownership - Basics

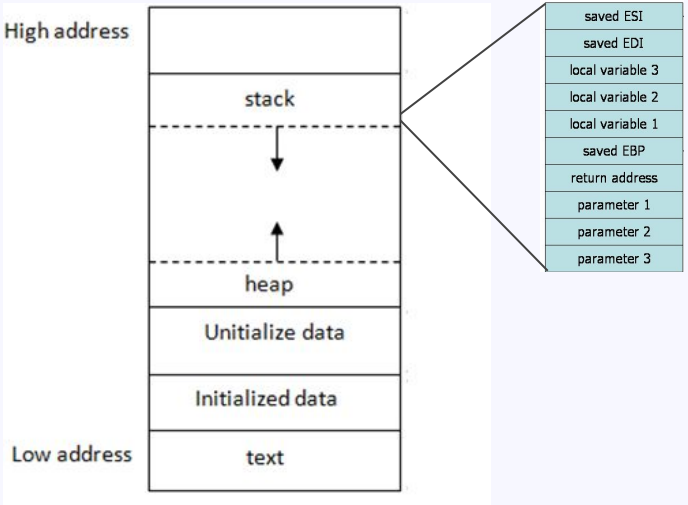
How about now?

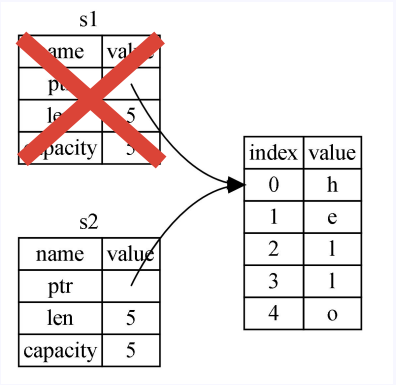



```
fn main(){
    {

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

    }
}
```



# Ownership - Basics


Let’s explore:

```
fn main(){
    {

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

# Ownership - Basics

## Let's explore:

```
fn main(){
    {

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

```
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:6:24
  |
4 |         let s1 = String::from("hello");
  |             -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
5 |         let s2 = s1;
  |                  -- value moved here
6 |         println!("{}", s1);
  |                        ^^ value borrowed here after move
```

# Ownership - Basics

Of course, we have options.

The **Clone** trait allows us to do a deep copy.

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

    println!("s1 = {}, s2 = {}", s1, s2);
}
```