References and Borrowing

Rust, in Practice and in Theory
Lecture 4

CAS CS 392 (M1)

Outline

» Discuss Ownership and Borrowing

» **Workshop:** RustViz

Ownership

Recall: Ownership

The notion of ownership is based on two simple rules

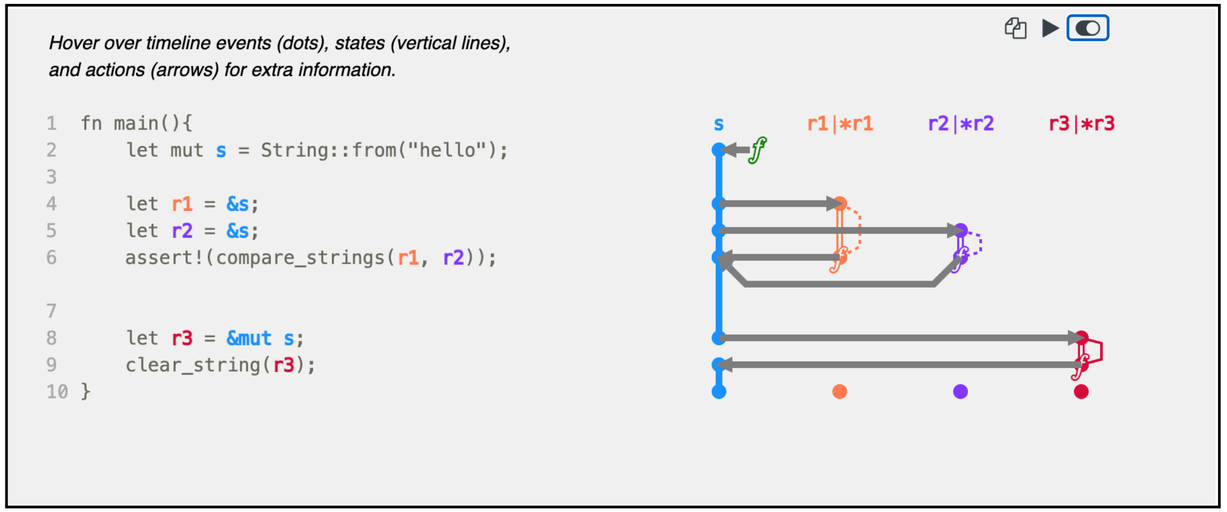
1. Every value has one owner at any given time
2. When the owner of a value goes out of scope, any memory associated with the value is freed

Recall: The Big Question

If we're not explicitly allocating/deallocating memory, when should it happen?

Rust's answer: as soon as a variable/parameter referring to it goes out of scope.

Recall: The Point



Ownership allows this stupid-simple deallocation pattern

If only one variable owns the data, then if they go out of scope, **no one owns the data**

https://github.com/rustviz/rustviz/blob/master/src/svg_generator/example.png

Drop

```
fn main() {
    let x = String::from("x");
}
```

For data on the heap, when a variable goes out of scope, Rust calls a function called **drop** *on its value* to return the memory

(It's like adding **free(x)** at the end of the block)

Drop

```
fn main() {
    let mut x = String::from("x");
    x = String::from("y");
    println!("{x}");
}
```

There is also an implicit **drop** call when a value is replaced.

Again, drop applies to *values*

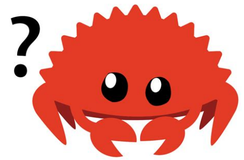
Drop (weird case)

```
fn main() {
    let mut x = String::from("x");
    x = String::from("y") + &x;
    println!("{x}");
}
```

What about this case? Should we drop the String **"x"**?

Should we drop before or after evaluating the RHS of the assignment?

Move



For data on the heap, memory needs to be returned when the owner goes out of scope

Data on the heap must be **moved** on assignment (really, the pointer must be given up)

y owns the one copy of the string that **x** originally owned

```
fn main() {
    let x = String::from("x");
    let y = x;
    println!("{x}");
    println!("{y}");
}
```

Move

Moves also happen at return values

Ownership is transferred to the parameter of **foo**, and then given to **y** as the return value of **foo**

```
fn foo(mut x : String) -> String {
    x.push_str("y");
    x
}

fn main() {
    let x = String::from("x");
    let y = foo(x);
    println!("{0}", y);
}
```

Copy

For data on the stack,
there is no memory to
return

Data on stack can be **copied**
on assignment

x and **y** both own a copy of
the value **5**

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

What's copied and what's moved?

Short answer: Stack data is copied, heap data is moved

Long answer: Everything is moved except for those types which implement the **Copy** trait

(we'll talk about traits later, they're like Type classes or interfaces)

Borrowing

We don't really need borrowing

Borrowing is, in some sense, a convenience

We can always *pass around* ownership

(Immutable borrows become more valuable in concurrent settings)

```
fn length(x : String) -> (String, i32) {
    let mut count = 0;
    for _ in x.chars() {
        count += 1;
    }
    (x, count)
}

fn main() {
    let x = String::from("xyz");
    let y = length(x);
    println!("{}", y.1);
}
```

Immutable References

A *reference* is like a pointer, guaranteed to point at a valid value

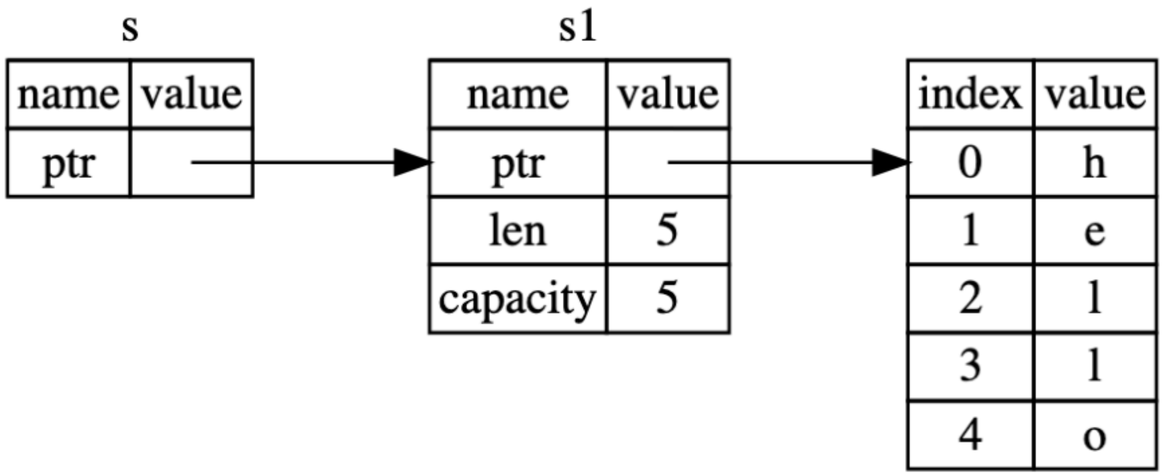
References can be used like the actual value

I prefer to think of them as an immutable "view" of a value

```
fn length(x : &String) -> i32 {
    let mut count = 0;
    for _ in x.chars() {
        count += 1;
    }
    count
}

fn main() {
    let x : String = String::from("xyz");
    let y = length(&x);
    println!("{}", y);
}
```

The Picture



In the above picture **s** has access without taking ownership

We can have as many immutable references we want

A Note on Dereferencing

It is also possible to dereference, and this looks a bit more like a pointer, but the behavior can be a bit unclear

Deref is a trait (like **Copy**) and the behavior of dereferencing can include implicit coercions

```
fn foo(x : &String) {
    let _ : &String = x;
    let _ : String = *x;
    let _ : str = **x;
}
```

Mutable References

Mutable references are the same, except that we're allowed to update the associated value

We can only have one mutable reference at a time

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

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

No Data Races


```
fn main() {
    let mut s = String::from("hello");
    let r1 = &s;
    let r2 = &s;
    let r3 = &mut s;
    println!("{}, {}, and {}", r1, r2, r3);
}
```

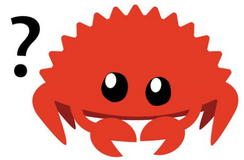
There can be no immutable references if there is a single mutable reference

No immutable reference can get different "views" of the same data

No Dangling References

We cannot use references data within the scope of the function as return values

(We'll see that *lifetimes* are actually what cause the compile-time error)




```
fn main() {
    let reference_to_nothing = dangle();
}

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

    &s
}
```

Summary

Borrowing ensures that we don't have to pass around ownership in order to work with data.

We're allowed **EITHER** one mutable reference **OR** zero or more immutable references

Workshop: RustViz

Workshop

Install rustviz, try out some of the more interesting examples.

(Very sorry assignment 2 is not ready)