



Lecture 6

Ownership and Borrowing in Rust

Goals For Today



- Review Ownership
- Transferring Ownership
- Introduce Borrowing
- Borrowing and Data Structures

Course Announcements



- HW4 releasing today due 2/17 at 11:59 pm CT
 - MP0 due **TOMORROW** 2/11 at 11:59 pm CT
 - MP1 releasing today due 2/23 at 11:59 pm CT
-
- You can complete PrairieLearn assignments on **EITHER** of your accounts (CS128 account or normal Illinois account)
 - We will take the highest score across both accounts for all assignments

Ownership Review



- Each value in Rust has a variable that's called its *owner*
- There can only be one owner at a time
- When the owner goes out of scope, the value will be dropped

```
fn main() {  
    let s = String::from("hello");  
    // ...  
    {  
        let w = String::from("world");  
        // do something with w...  
    } // w is dropped here  
    // ...  
} // s is dropped here
```

```
fn main() {  
    let x = String::from("hello");  
  
    let y = x; // y now OWNS the String "hello"  
  
    // println!("{}", x); // THIS LINE WON'T COMPILE  
    println!("{}", y);  
}
```

Copy vs Clone



- Copy: automatically defined for primitive types (int, float, bool, char, etc...)

```
fn main {  
    let mut x: u8 = 5;  
  
    // u8 (and all primitive types) have the Copy trait  
    let y = x;  
    x += 1;  
  
    println!("x = {} and y = {}", x, y);  
  
    // prints: x = 6 and y = 5  
}
```

- Clone: explicit function call to make a deep copy of some data

Copy vs Clone

- Clone: explicit function call to make a deep copy of some data

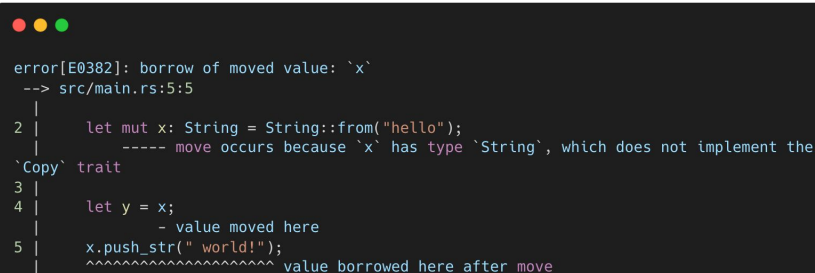


```
fn main() {
    let mut x: String = String::from("hello");

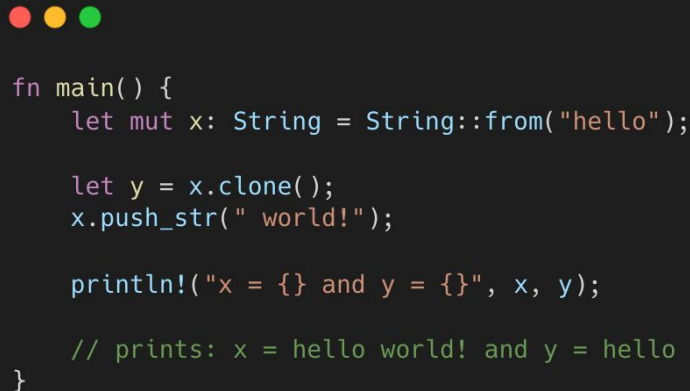
    let y = x;
    x.push_str(" world!");

    println!("x = {} and y = {}", x, y);

    // prints: x = hello world! and y = hello
}
```



```
error[E0382]: borrow of moved value: `x`
  --> src/main.rs:5:5
   |
2  |     let mut x: String = String::from("hello");
   |     ----- move occurs because `x` has type `String`, which does not implement the
   |     `Copy` trait
3  |
4  |     let y = x;
   |           - value moved here
5  |     x.push_str(" world!");
   |     ~~~~~ value borrowed here after move
```



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

    let y = x.clone();
    x.push_str(" world!");

    println!("x = {} and y = {}", x, y);

    // prints: x = hello world! and y = hello
}
```

Moving Ownership



- Remember: values can only ever have 1 *owner*



```
fn main() {  
    let mut x: String = String::from("hello");  
  
    let y = x;  
  
    // ERROR: value borrowed here after move  
    x.push_str(" world!");  
  
    println!("x = {} and y = {}", x, y);  
}
```

Moving Ownership in Function Calls



- Again, values can only have 1 *owner*



```
fn main() {  
    let class = "CS 128 Honors".to_string();  
  
    say_hello(class);  
  
    // ERROR: value used here after move  
    say_hello(class);  
}  
  
fn say_hello(name: String) {  
    println!("Hello {}", name);  
}
```


References



- An ampersand (&) represents a reference
- Allows you to refer to some value without taking ownership of it
- We call the action of creating a reference borrowing

Reference:

- <https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html>



Let's Fix Our Examples

Borrowing



- At any given time, you can have either:
 - one mutable reference using `&mut` or...
 - An infinite number of immutable references using `&`
- References must always be valid

```
fn main() {  
    let mut x: String = String::from("hello");  
  
    // creates a MUTABLE reference to x  
    let y = &mut x;  
  
    // ERROR: trying to create a SECOND MUTABLE reference to x  
    x.push_str(" world!");  
  
    println!("x = {} and y = {}", x, y);  
}
```

Reference:

- <https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html>

Ownership in Vectors (& Other Data Structures)



- Remember: values can only ever have 1 *owner*
- What happens when we add elements to a **Vec** (or any other data structure)?
 - The **Vec** now owns the value!
 - When we try to access a value from a **Vec**, we are given a reference

```
fn main() {  
    let x: Vec<String> = vec!["hello".into(), "cs".into(),  
    "128".into()];  
    // ERROR: cannot move out of index of `Vec<String>`  
    // move occurs because value has type `String`,  
    // which does not implement the `Copy` trait  
    let element = x[2];  
}
```



Vector Ownership

Vector Methods



- **my_vector[i: usize]** – Try and take ownership of (or **Copy**) the value at index **i**
- **&my_vector[i: usize]** – IMMUTABLY borrow the value at index **i**
- **&mut my_vector[i: usize]** – MUTABLY borrow the value at index **i**
 - **my_vector** MUST be declared as mutable
- **my_vector.get(i: usize)** – Try to get an IMMUTABLE reference to index **i**
 - returns **Option<&type>**
- **my_vector.get_mut(i: usize)** – Try and get a MUTABLE reference to index **i**
 - returns **Option<&mut type>**
 - **my_vector** MUST be declared as mutable

Vector Methods



- `my_vector.iter()` – Iterate over vector using IMMUTABLE references
- `my_vector.iter_mut()` – Iterate over vector using MUTABLE references
- `my_vector.into_iter(i: usize)` – Take ownership of vector
 - WARNING!!



MP1 Walkthrough