CS128 Honors

Lecture 6

Ownership and Borrowing in Rust

Slides: Neil Kaushikkar

Goals For Today

- Review Ownership
- Introduce Borrowing
- Intro to Dereferencing in Rust
- Borrowing and Data Structures

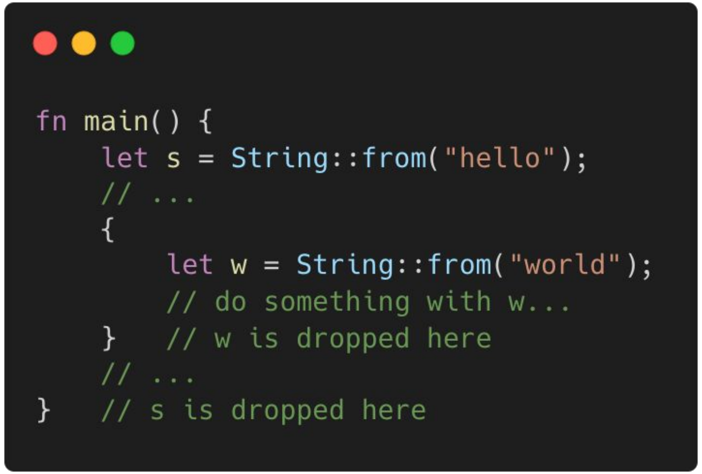
Course Announcements

- HW3 due 9/20 at 11:59 pm CT
- HW4 releasing today due 9/22 at 11:59 pm CT
- MP0 due **TOMORROW** 9/16 at 11:59 pm CT
- MP1 releasing today due 9/28 at 11:59 pm CT

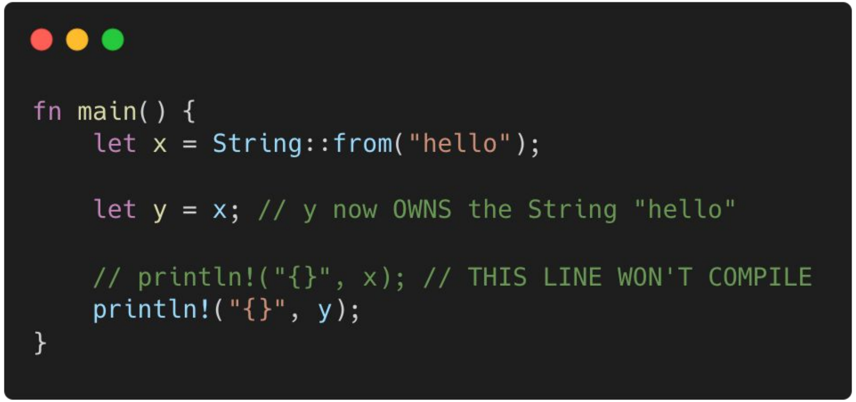
- Practice assignment goes live today - we will be adding more problems throughout the semester

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



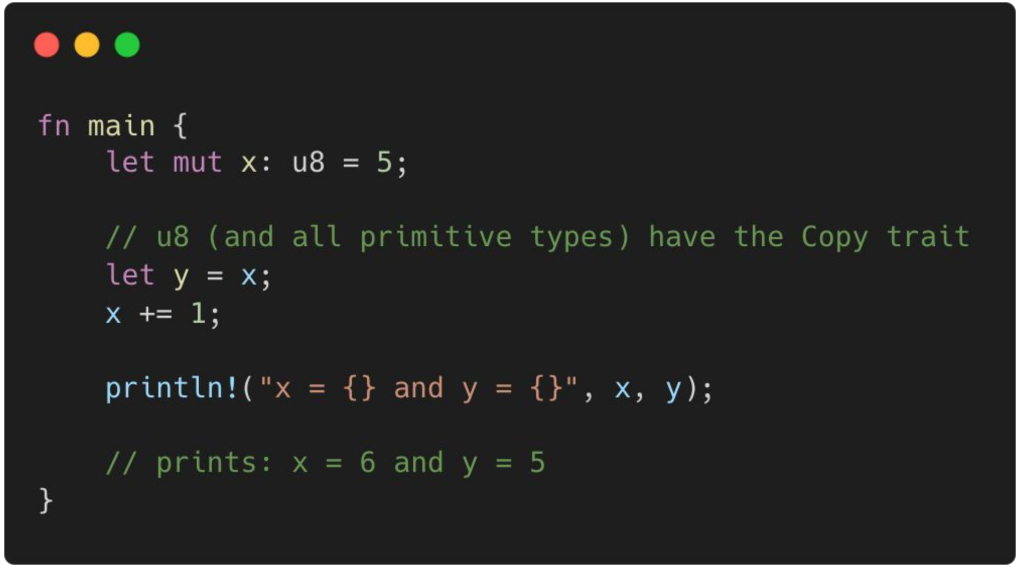
```rust
fn main() {
    let s = String::from("hello");
    // ...
    {
        let w = String::from("world");
        // do something with w...
    }   // w is dropped here
    // ...
}   // s is dropped here
```



```rust
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...)

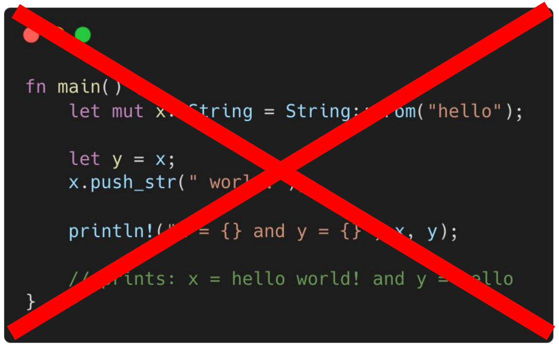


```rust
fn main {
    let mut x: u8 = 5;

    // 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
}
```

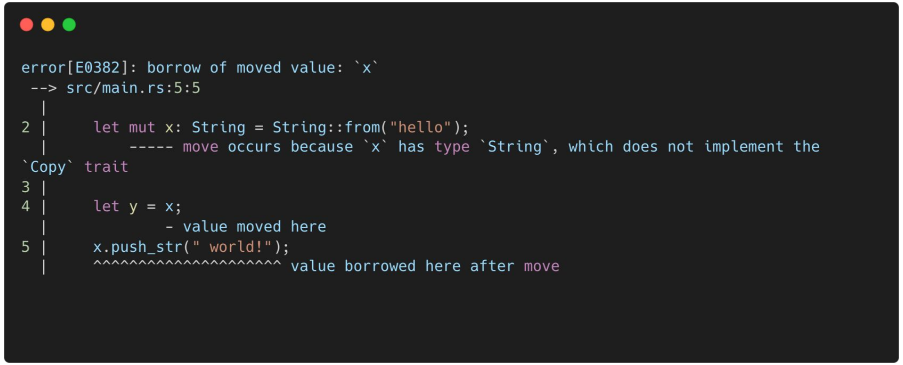
- Clone: **explicit** function call to make a deep copy of some data

Copy vs Clone

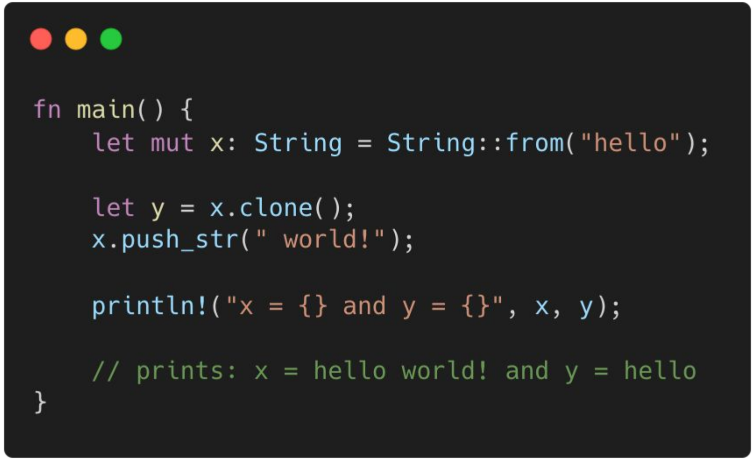
- Clone: **explicit** function call to make a deep copy of some data



```rust
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
}
```



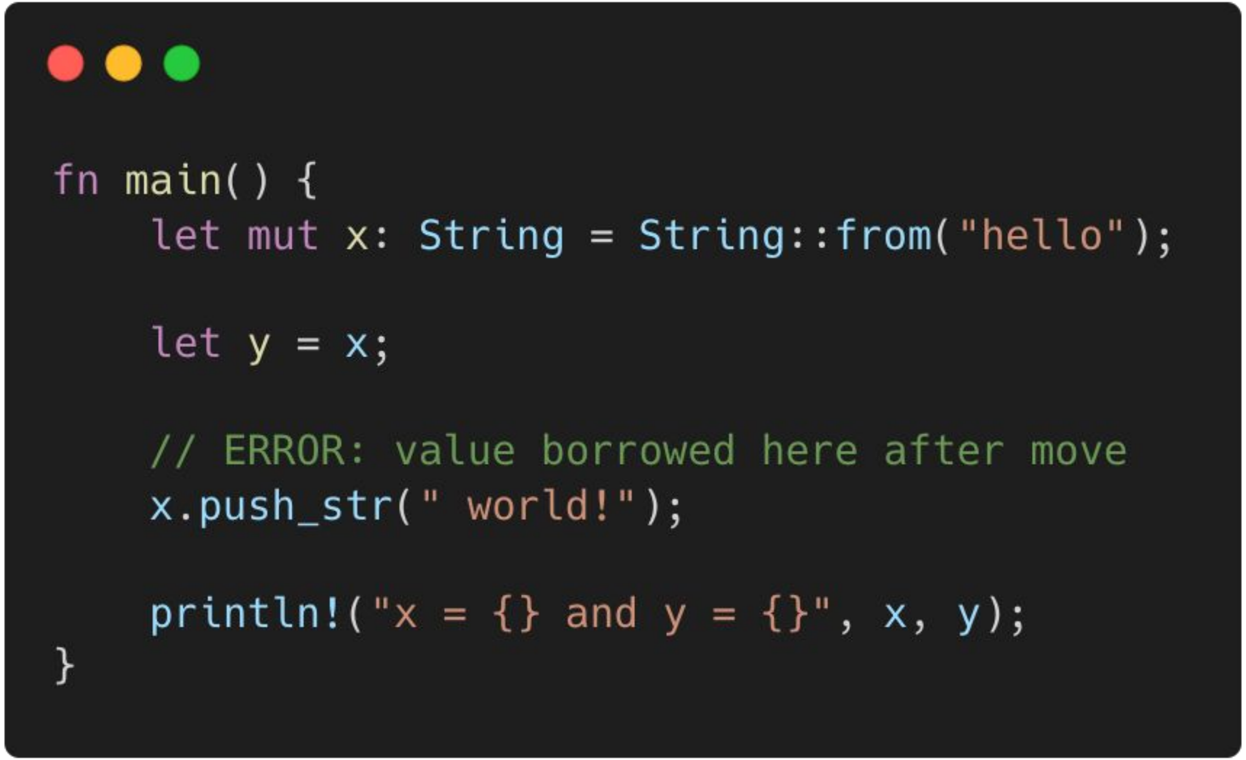
```
error[E0382]: borrow of moved value: `x`
 --> src/main.rs:5:5
  |
2 |     let mut x: String = String::from("hello");
  |         ----- move occurs because `x` has type `String`, which does not implement the
`Copy` trait
3 |
4 |     let y = x;
  |             - value moved here
5 |     x.push_str("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



```rust
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

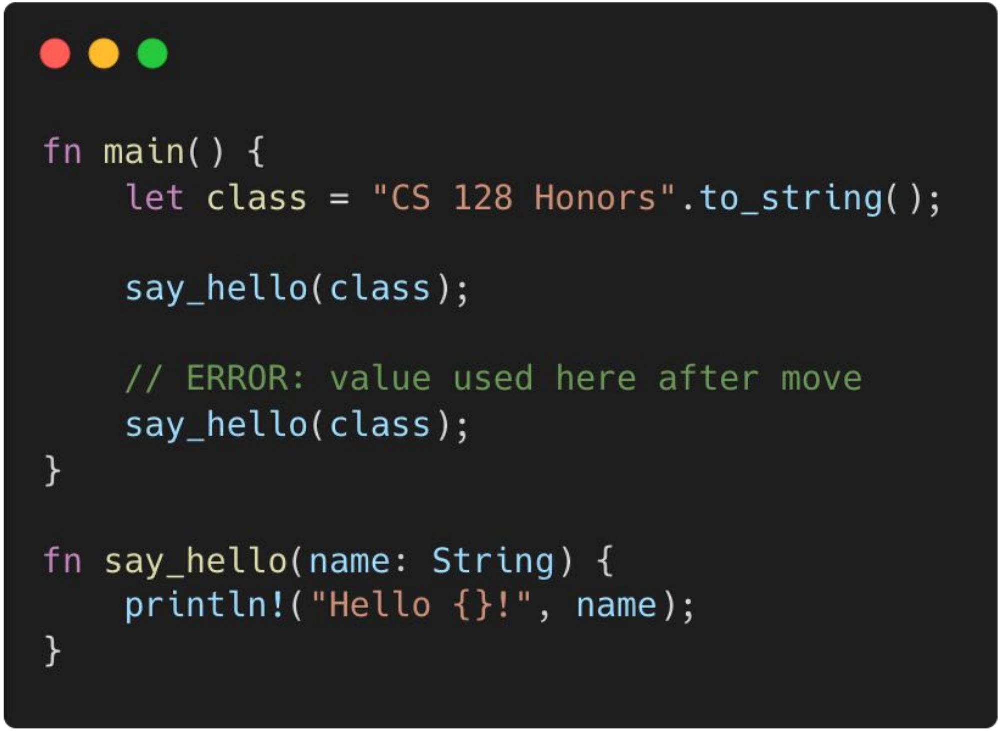
- Remember: values can only ever have 1 *owner*



```
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

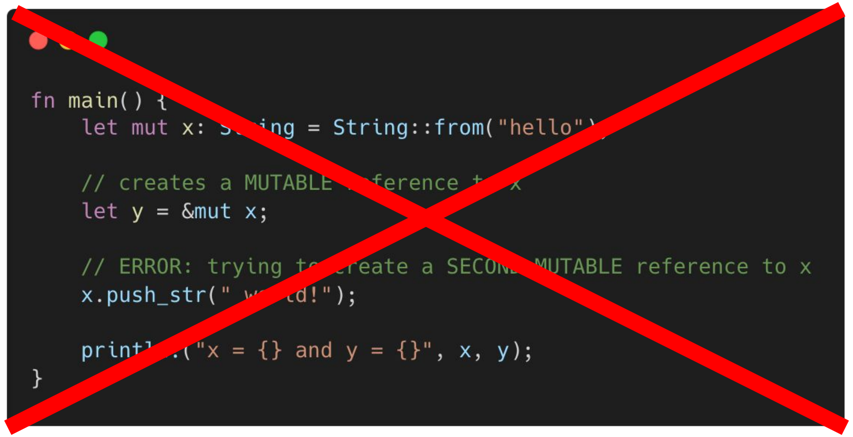
- An ampersand (&) represents a *reference*
- Allows you to refer to some value *without taking ownership* of it
- We call the action of creating a reference *borrowing*

Reference:

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

Borrowing Rules

- At any given time, you can have either:
  - **one** mutable reference using **&mut** or…
  - An **infinite** number of immutable references using **&**
- A mutable reference must be a reference to a mutable variable
  - You cannot make a mutable reference to an immutable variable
- 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

Let’s Fix Our Examples

Dereferencing Mutable References

- You can mutate the variable that a mutable reference refers to by dereferencing that reference with a * before the reference
- References are, in essence, addresses in memory
- Similar to C/C++, we can dereference an address to change the memory at that address

Reference:

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

When to Dereference

- You need to dereference mutable references to primitive types
- You need to dereference when using iterators
- You do not need to dereference when using bracket access on vectors
  - i.e. **my_vec[i]**
- Custom types like **String**s handle dereferencing for you in the methods you call on them
- More on mutable references and non-primitive types in future lectures

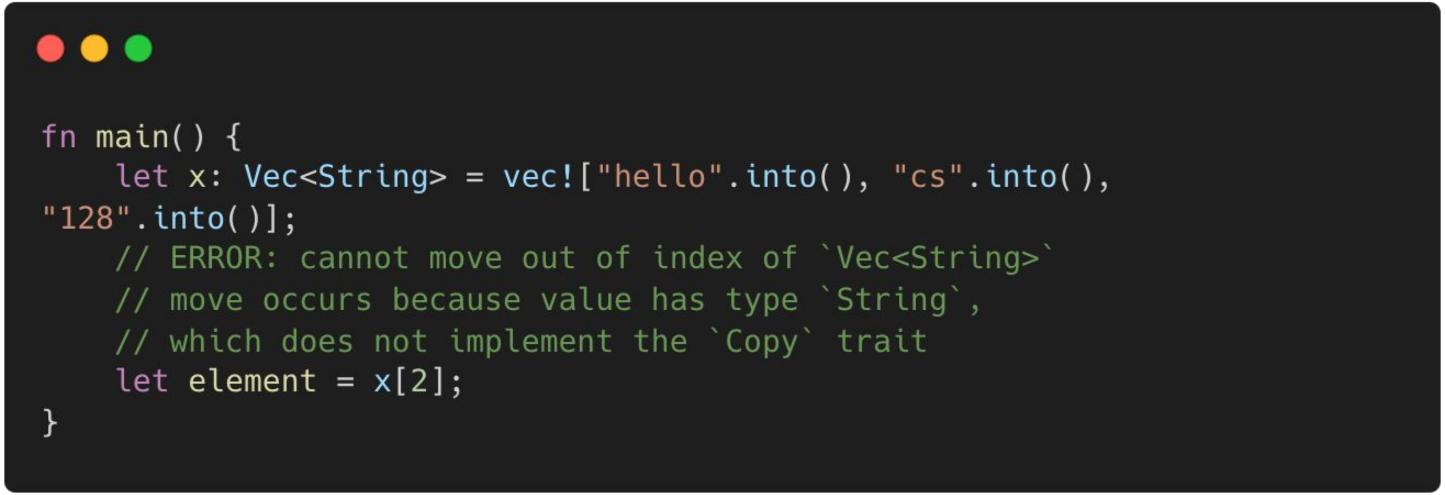
Reference:

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

Dereferencing Mutable References

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 + iterate through a vector**
  - **WARNING!!**
  - **You can no longer use the vector after calling this method on a vector**

That’s All Folks!