Lecture 6

Borrowing

Goals For Today

- Recall Ownership
- Borrowing and Dereferencing
- Common errors
- Some examples

Ownership Review

- Each value in Rust has a variable called its owner
- There can only be one owner at a time, change of owner called a move
- When variable goes out of scope, value is dropped

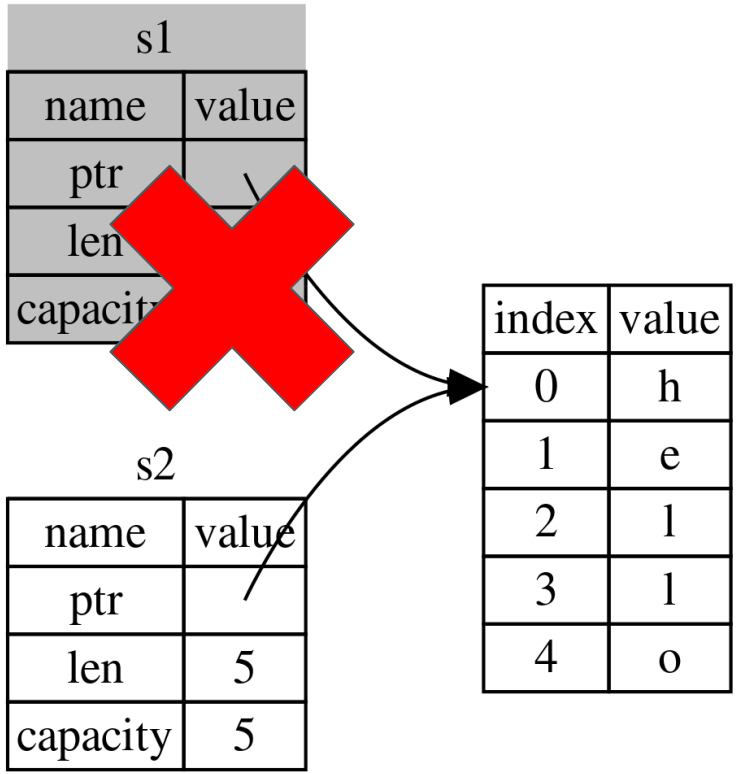
```
fn main() {

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

} // s is dropped here
```

```
fn main() {

    let x: String = String::from("hello");

    let y: String = x; // y now owns the String::from("hello")

    // println!("{}", x); // this will not compile
    println!("{}", y);
}
```

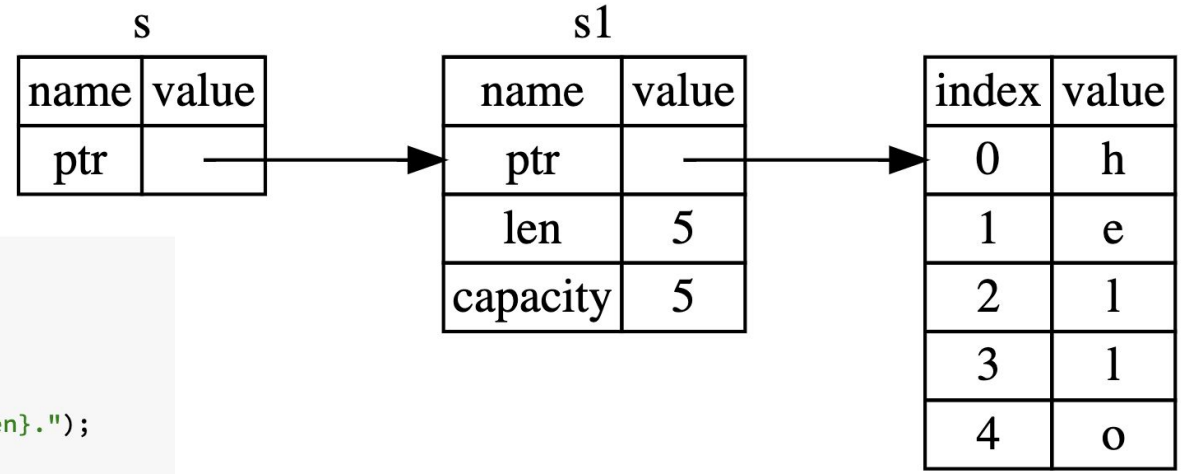
Recall: Transferring Ownership

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

- s1 moved to s2
- s1 no longer exists
- What if we don't want to move?



Solution: Borrowing




```rust
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{s1}' is {len}.");
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

- Created immutable reference to s1
- Like a pointer in C++
- s dropped at end of calculate_length, not s1

Reference

- An ampersand (&) represents a reference
- Refer to some value without taking ownership of it
- Action of referencing is called borrowing
- Similar to references/pointers in C++: refer to some value without needing a copy
- C++ problems:
  - 2 pointers to same object in different functions, one modify object without the other knowing -> bad things happen
  - An object gets dropped without reference owner knowing -> extra bad things happen
- Solution: Rust's references come with training wheels

Borrowing Rules

- At any given moment, can either have:
  - 1 mutable reference: &mut
  - Or infinite immutable references: &
- A mutable reference must refer to a mutable variable
- References must always be valid: must reference existing variables only while they are in scope (enforced by compiler)
- Want a nullable reference anyway?:
  - Option<&type>
  - With forced nullability check

```
fn main() {

    let mut x: String = String::from("hello");

    let y: &mut String = &mut x; // y is a MUTABLE reference to x

    // function `push_str` defined to take a mutable reference of x
    // ERROR: can't have a SECOND MUTABLE reference to x
    x.push_str(string: " world!");

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

Add a Borrow!


```rust
fn main() {
    let mut class: String = String::from("CS 128H");

    // a reference created here
    // and dropped while `class` still lives
    say_hello_borrow(name: &class);

    // `class` later moved into this function and dropped for good
    say_hello(name: class);
}

fn say_hello(name: String) {
    println!("Hello {}!", name);
}

fn say_hello_borrow(name: &String) {
    println!("Hello {}!", name);
} // reference dropped here
```

Dereferencing

- You can get access to the value a reference refers to by dereferencing with a *
- Similar to pointers in C++
- Why dereferencing?
  - Mutable reference: used to mutate the value -> most common use
  - Immutable reference: less useful here, as assigning that value to a variable tries to move (forbidden) or copy (if defined for type) -> avoid in most cases

```rust
fn main() {
    let mut x: u8 = 5;
    add_one(num: &mut x);
    println!("x = {}", x); // prints: "x = 6"
}

fn add_one(num: &mut u8) {
    *num += 1; // dereference and mutate the value
}
```

```rust
fn main() {
    let mut class: String = String::from("CS 128H");
}

#[allow(dead_code)]
fn take_name(name: &String) {
    // this assignment tries to move the value to `new_name`
    // ERROR: cannot move out of a reference
    let _new_name: String = *name;
}
```

When to Dereference?

- Need to dereference mutable references to primitive types
- Need to dereference when using mutable iterators
- Do not need to dereference when using bracket access on vector
  - Ex: my_vec[idx]
- Do not need to dereference when calling String methods i.e: can call them directly on a &mut to a String

Immutable Borrow after Mutable

What is the problem here?

- Only 1 mutable borrow
- No ownership transfer

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

let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM

println!("{}, {}, and {}", r1, r2, r3);
```

Immutable Borrow after Mutable

Can't make immutable borrow after mutable borrow.

Can cause issues with reading and writing data.

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
 --> src/main.rs:6:14
  |
4 |     let r1 = &s; // no problem
  |              -- immutable borrow occurs here
5 |     let r2 = &s; // no problem
6 |     let r3 = &mut s; // BIG PROBLEM
  |              ^^^^^^ mutable borrow occurs here
7 |
8 |     println!("{}, {}, and {}", r1, r2, r3);
  |                                -- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

Immutable Borrow after Mutable

Fix by changing order of borrowing. No more immutable borrow after mutable borrow.

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

let r1 = &s; // no problem
let r2 = &s; // no problem
println!("{r1} and {r2}");
// variables r1 and r2 will not be used after this point

let r3 = &mut s; // no problem
println!("{r3}");
```

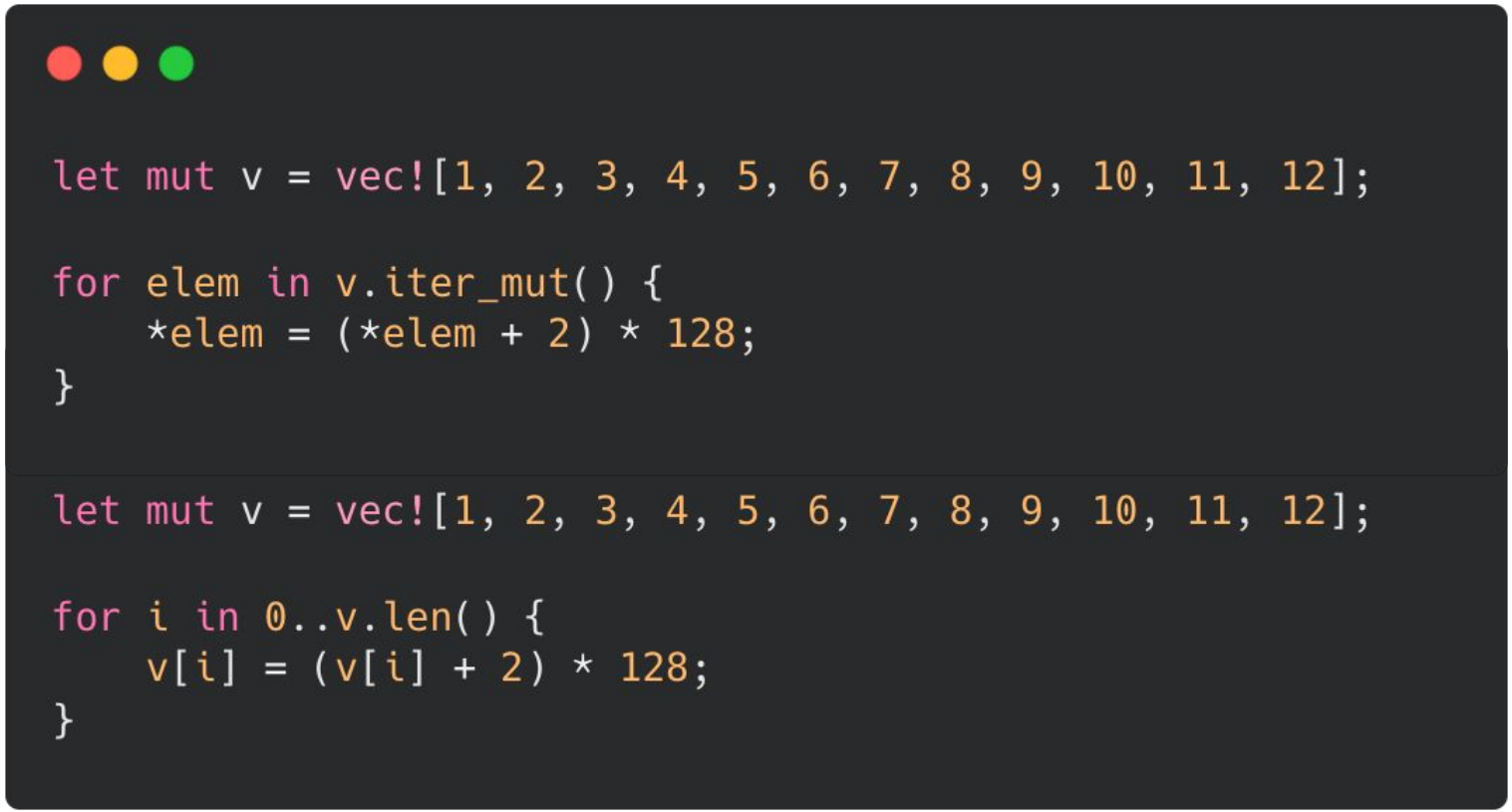
Dangling References

Can you return references?

- What if object is dropped?
- s no longer exists
  - Code will not compile
  - Must specify lifetimes
    - Cover in future lecture

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

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

    &s
}
```

Vector Iteration



```rust
let mut v = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12];

for elem in v.iter_mut() {
    *elem = (*elem + 2) * 128;
}
```

```rust
let mut v = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12];

for i in 0..v.len() {
    v[i] = (v[i] + 2) * 128;
}
```

Vector Methods

- my_vector[i]: Takes ownership (or a copy) of value at i, panics if out of bounds (OOB)
- &my_vector[i]: Immutably borrows the value at i, panics if OOB
- &mut my_vector[i]: Mutably borrows the value at i, panics if OOB
  - my_vector must be declared as mutable
- my_vector.get(i): Tries to get an immutable reference to value at i, doesn't panic
  - Returns Option<&type> (familiar?)
- my_vector.get_mut(i): Tries to get a mutable reference to value at i, doesn't panic
  - Return Option<&mut type>
  - my_vector must be declared as mutable

Announcements

HW 4 Released! Due Fri 2/21, 23:59 PM