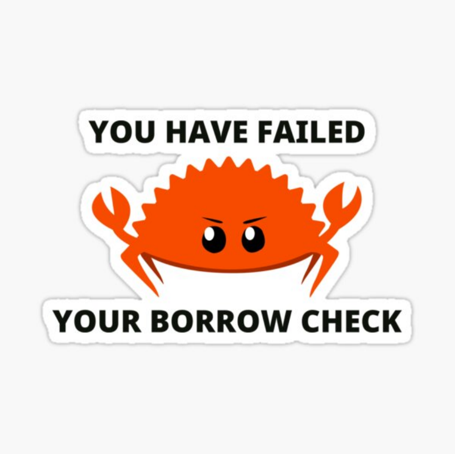
Lecture 3: Borrow checker

Barinov Denis

March 5, 2025

barinov.diu@gmail.com

Borrow Checker



What’s the problem, Rust?

```rust
let mut v = vec![1, 2, 3];
let x = &v[0];
v.push(4);
println!("{}", x);
```

What's the problem, Rust?

```
    let mut v = vec![1, 2, 3];
    let x = &v[0];
    v.push(4);
    println!("{}", x);
```

```
error[E0502]: cannot borrow `v` as mutable because it is also
borrowed as immutable
 --> src/main.rs:8:5
  |
7 |     let x = &v[0];
  |              - immutable borrow occurs here
8 |     v.push(4);
  |     ^^^^^^^^^ mutable borrow occurs here
9 |     println!("{}", x);
  |                    - immutable borrow later used here
```

What’s the problem, Rust?

```rust
fn sum(v: Vec<i32>) -> i32 {
    let mut result = 0;
    for i in v {
        result += i;
    }
    result
}

fn main() {
    let mut v = vec![1, 2, 3];
    println!("first sum: {}", sum(v));
    v.push(4);
    println!("second sum: {}", sum(v))
}
```

What’s the problem, Rust?

```
error[E0382]: borrow of moved value: `v`
  --> src/main.rs:12:5
   |
10 |     let mut v = vec![1, 2, 3];
   |         ----- move occurs because `v` has type `Vec<i32>`,
   |  which does not implement the `Copy` trait
11 |     println!("first sum: {}", sum(v));
   |                                   - value moved here
12 |     v.push(4);
   |     ^^^^^^^^^ value borrowed here after move
```

Ownership rules

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

Ownership rules

```
fn main() {
    let s = vec![1, 4, 8, 8];
    let u = s;
    println!("{:?}", u);
    println!("{:?}", s); // This won't compile!
}
```

Ownership rules

```rust
fn om_nom_nom(s: Vec<i32>) {
    println!("I have consumed {s:?}");
}

fn main() {
    let s = vec![1, 4, 8, 8];
    om_nom_nom(s);
    println!("{s:?}");
}
```

Ownership rules

```rust
fn om_nom_nom(s: Vec<i32>) {
    println!("I have consumed {s:?}");
}

fn main() {
    let s = vec![1, 4, 8, 8];
    om_nom_nom(s);
    println!("{s:?}");
}
```

- Each ”owner” has the responsibility to clean up after itself.
- When you move `s` into *om_nom_nom*, it becomes the owner of `s`, and it will free `s` when it’s no longer needed in that scope. *Technically the s parameter in om_nom_nom become the owner*.
- That means you can no longer use it in *main*!
- In C++, we would create a copy!

Ownership rules

Given what we just saw, how can the following be the valid syntax?

```rust
fn om_nom_nom(n: u32) {
    println!("{} is a very nice number", n);
}

fn main() {
    let n: u32 = 42;
    let m = n;
    om_nom_nom(n);
    om_nom_nom(m);
    println!("{}", m + n);
}
```

Ownership rules

- Say you have a group of lawyers that are reviewing and signing a contract over Google Docs (just pretend it's true :))
- What are some ground rules we'd need to set to avoid chaos?
- If someone modifies the contract before everyone else reviews/signs it, that's fine.
- But if someone modifies the contract while others are reviewing it, people might miss changes and think they're signing a contract that says something else.
- We should allow a single person to modify, or everyone to read, but not both.

Borrowing

- We can have multiple shared (immutable) references at once (with no mutable references) to a value.
- We can have only one mutable reference at once. (no shared references to it)
- This paradigm pops up a lot in systems programming, especially when you have "readers" and "writers". In fact, you've already studied it in the course of Theory and Practice of Concurrency.

Borrowing

- The lifetime of a value starts when it's created and ends the *last time it's used*
- Rust doesn't let you have a reference to a value that lasts longer than the value's lifetime
- Rust computes lifetimes at compile time using static analysis. (this is often an over-approximation!)
- Rust calls the special "drop" function on a value once its lifetime ends. (this is essentially a destructor)

Borrowing

```rust
fn main() {
    let mut x = 5;
    let y = &mut x;

    println!("y = {y}");
    x = 42; // ok
    println!("x = {x}");
}
```

Borrowing

```rust
fn main() {
    let mut x = 5;
    let y = &mut x;

    x = 42; // not ok
    println!("y = {y}");
    println!("x = {x}");
}
```

Borrowing

```
fn main() {
    let x1 = 42;
    let y1 = Box::new(84);
    { // starts a new scope
        let z = (x1, y1);
        // z goes out of scope, and is dropped;
        // it in turn drops the values from x1 and y1
    }
    // x1's value is Copy, so it was not moved into z
    let x2 = x1;

    // y1's value is not Copy, so it was moved into z
    // let y2 = y1;
}
```

Questions?