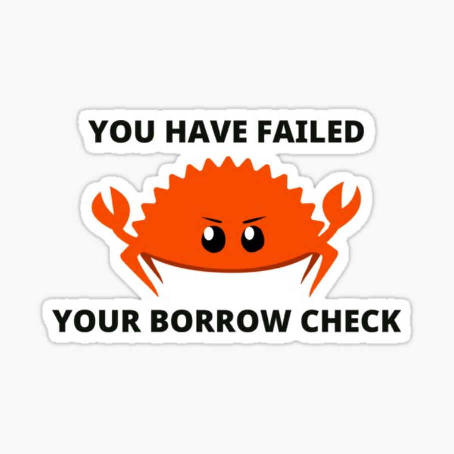
Lecture 3: Borrow checker

Barinov Denis

February 28, 2024

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?

```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

```rust
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

```
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

```rust
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;
}
```

Option[1] and Result[2]

Let's remember their definitions:

```
enum Option<T> {
    Some(T),
    None,
}

enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

[1] Option documentation

[2] Result documentation

Matching `Option`:

```rust
let result = Some("string");
match result {
    Some(s) => println!("String inside: {s}"),
    None => println!("Ooops, no value"),
}
```

`Option` API

Useful functions `.unwrap()` and `.expect()`:

```rust
fn unwrap(self) -> T;
fn expect(self, msg: &str) -> T;
```

Option API

Useful functions .unwrap() and .expect():

```
let opt = Some(22022022);
assert!(opt.is_some());
assert!(!opt.is_none());
assert_eq!(opt.unwrap(), 22022022);
let x = opt.unwrap(); // Copy!

let newest_opt: Option<i32> = None;
// newest_opt.expect("I'll panic!");

let new_opt = Some(Vec::<i32>::new());
assert_eq!(new_opt.unwrap(), Vec::<i32>::new());
// error[E0382]: use of moved value: `new_opt`
// let x = new_opt.unwrap(); // Clone!
```

`Option` API

We have a magic function:

```
fn as_ref(&self) -> Option<&T>; // &self is &Option<T>
```

Let's solve a problem:

```
let new_opt = Some(Vec::<i32>::new());
assert_eq!(new_opt.unwrap(), Vec::<i32>::new());
// error[E0382]: use of moved value: `new_opt`
// let x = new_opt.unwrap(); // Clone!

let opt_ref = Some(Vec::<i32>::new());
assert_eq!(new_opt.as_ref().unwrap(), &Vec::<i32>::new());
let x = new_opt.unwrap(); // We used reference!
// There's also .as_mut() function
```

That means if type implements `Copy`, `Option` also implements `Copy`.

Option API

We can map `Option<T>` to `Option<U>`:

```
fn map<U, F>(self, f: F) -> Option<U>;
```

Example:

```
let maybe_some_string = Some(String::from("Hello, World!"));
// `Option::map` takes self *by value*,
// consuming `maybe_some_string`
let maybe_some_len = maybe_some_string.map(|s| s.len());
assert_eq!(maybe_some_len, Some(13));
```

`Option` API

There's **A LOT** of different `Option` functions, enabling us to write beautiful functional code:

```rust
fn map_or<U, F>(self, default: U, f: F) -> U;
fn map_or_else<U, D, F>(self, default: D, f: F) -> U;
fn unwrap_or(self, default: T) -> T;
fn unwrap_or_else<F>(self, f: F) -> T;
fn and<U>(self, optb: Option<U>) -> Option<U>;
fn and_then<U, F>(self, f: F) -> Option<U>;
fn or(self, optb: Option<T>) -> Option<T>;
fn or_else<F>(self, f: F) -> Option<T>;
fn xor(self, optb: Option<T>) -> Option<T>;
fn zip<U>(self, other: Option<U>) -> Option<(T, U)>;
```

It's recommended for you to study the documentation and try to avoid `match` where possible.

`Option` and ownership

There's two cool methods to control ownership of the value inside:

```rust
fn take(&mut self) -> Option<T>;
fn replace(&mut self, value: T) -> Option<T>;
fn insert(&mut self, value: T) -> &mut T;
```

The first one takes the value out of the `Option`, leaving a `None` in its place.

The second one replaces the value inside with the given one, returning `Option` of the old value.

The third one inserts a value into the `Option`, then returns a mutable reference to it.

Option **API and ownership:** take

```rust
struct Node<T> {
    elem: T,
    next: Option<Box<Node<T>>>,
}

pub struct List<T> {
    head: Option<Box<Node<T>>>,
}

impl<T> List<T> {
    pub fn pop(&mut self) -> Option<T> {
        self.head.take().map(|node| {
            self.head = node.next;
            node.elem
        })
    }
}
```

`Option` and optimizations

Rust guarantees to optimize the following types `T` such that `Option<T>` has the same size as `T`:

- `Box<T>`
- `&T`
- `&mut T`
- `fn`, `extern "C" fn`
- `#[repr(transparent)]` struct around one of the types in this list.
- `num::NonZero*`
- `ptr::NonNull<T>`

This is called the "null pointer optimization" or NPO.

Result

Functions return Result whenever errors are expected and recoverable. In the `std` crate, `Result` is most prominently used for I/O.

Results must be used! A common problem with using return values to indicate errors is that it is easy to ignore the return value, thus failing to handle the error. `Result` is annotated with the `#[must_use]` attribute, which will cause the compiler to issue a warning when a Result value is ignored.[3]

[3] The Error Model

`Result` API

We can `match` it as a regular `enum`:

```
let version = Ok("1.1.14");
match version {
    Ok(v) => println!("working with version: {:?}", v),
    Err(e) => println!("error: version empty"),
}
```

Result API

We have pretty the same functionality as in `Option`:

```
fn is_ok(&self) -> bool;
fn is_err(&self) -> bool;
fn unwrap(self) -> T;
fn unwrap_err(self) -> E;
fn expect_err(self, msg: &str) -> E;
fn expect(self, msg: &str) -> T;
fn as_ref(&self) -> Result<&T, &E>;
fn as_mut(&mut self) -> Result<&mut T, &mut E>;
fn map<U, F>(self, op: F) -> Result<U, E>;
fn map_err<F, O>(self, op: O) -> Result<T, F>;
// And so on
```

It's recommended for you to study the documentation and try to avoid `match` where possible.

Operator ?

Consider the following structure:

```
struct Info {
    name: String,
    age: i32,
}
```

Operator ?

```
fn write_info(info: &Info) -> io::Result<()> {
    let mut file = match File::create("my_best_friends.txt") {
        Err(e) => return Err(e),
        Ok(f) => f,
    };
    if let Err(e) = file
        .write_all(format!("name: {}\n", info.name)
        .as_bytes()) {
        return Err(e)
    }
    if let Err(e) = file
        .write_all(format!("age: {}\n", info.age)
        .as_bytes()) {
        return Err(e)
    }
    Ok(())
}
```

Operator ?

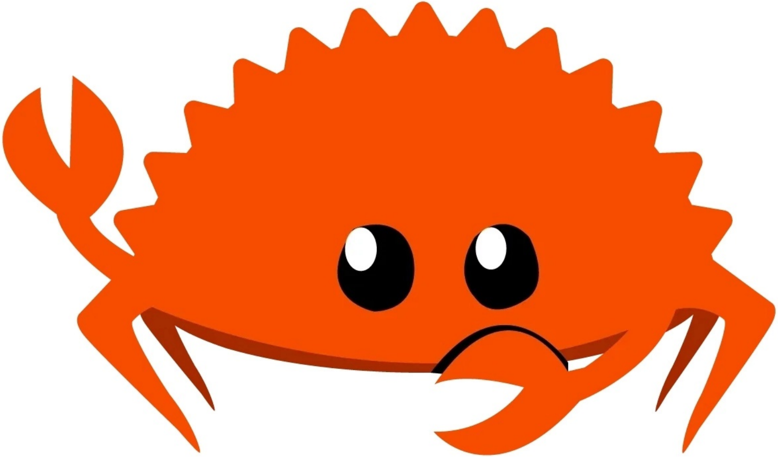
We can use the `?` operator to make the code smaller!

```
fn write_info(info: &Info) -> io::Result<()> {
    let mut file = File::create("my_best_friends.txt")?;
    file.write_all(format!("name: {}\n", info.name).as_bytes())?;
    file.write_all(format!("age: {}\n", info.age).as_bytes())?;
    Ok(())
}
```

Beautiful, isn't it?

We can use it for `Option` too!

Questions?



`Box` **and** `Rc`

We are already familiar with Box type. Let's check one advanced function:

```rust
fn leak<'a>(b: Box<T, A>) -> &'a mut T;
fn into_raw(b: Box<T, A>) -> *mut T;
```

Example:

```rust
let x = Box::new(41);
let static_ref: &'static mut usize = Box::leak(x);
*static_ref += 1;
assert_eq!(*static_ref, 42);
```

But stop! Rust is the safe language, no memory unsafety, no undefined behavior, what's wrong!?

But stop! Rust is the safe language, no memory unsafety, no undefined behavior, what's wrong!?

In reality, when you're creating global objects or interacting with other languages, you **have to** *leak objects. Moreover, it's* **safe** *to leak memory, just not good!*

`Rc` is single-threaded reference-counting pointer. "`Rc`" stands for "Reference Counted".

```
let rc = Rc::new(());
let rc2 = rc.clone(); // Clones Rc, not what inside!
let rc3 = Rc::clone(&rc); // The same
```

`Rc` is dropped when all instances of `Rc` are dropped.

Primary functions:

```
fn get_mut(this: &mut Rc<T>) -> Option<&mut T>;
fn downgrade(this: &Rc<T>) -> Weak<T>;
fn weak_count(this: &Rc<T>) -> usize;
fn strong_count(this: &Rc<T>) -> usize;
```

References to the variable inside Rc are controlled at runtime:

```
let mut rc = Rc::new(42);
println!("{}", *rc);

*Rc::get_mut(&mut rc).unwrap() -= 41;
println!("{}", *rc);

let mut rc1 = rc.clone();
println!("{}", *rc1);
// thread 'main' panicked at 'called `Option::unwrap()`
// on a `None` value'
// *Rc::get_mut(&mut rc1).unwrap() -= 1;
```

get_mut guarantees that it will return mutable reference only if there's only one pointer. If there are more, you won't have a chance to modify Rc.

`Rc` is a **strong** pointer, while `Weak` is a **weak** pointer. Both of them have *ownership over allocation*, but only `Rc` have *ownership over the value inside*:

You can upgrade `Weak` to `Rc`:

```rust
fn upgrade(&self) -> Option<Rc<T>>;
```

Weak

```rust
let rc1 = Rc::new(String::from("string"));
let rc2 = rc1.clone();
let weak1 = Rc::downgrade(&rc1);
let weak2 = Rc::downgrade(&rc1);
drop(rc1); // The string is not deallocated
assert!(weak1.upgrade().is_some());
drop(weak1); // Nothing happens
drop(rc2); // The string is deallocated
assert_eq!(weak2.strong_count(), 0);
// If no strong pointers remain, this will return zero.
assert_eq!(weak2.weak_count(), 0);
assert!(weak2.upgrade().is_none());
drop(weak2); // The Rc is deallocated
```

There's also `Arc` - a thread-safe reference-counting pointer. `Arc` stands for "Atomically Reference Counted".

We'll need it to share data safely across threads in the future.