Lecture 4 — Rust: Breaking the Rules for Fun and Performance

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

May 20, 2025

Breaking Some Rules

Previously: mutex is not compatible with our single-ownership model.

We need multiple threads to have access to the mutex.

Background required: smart pointers, reference counting.

Smart Pointers

We know about pointers from C/C++; you may have also seen smart pointers.

We'll talk about two kinds of smart pointer right now, the Box and the Reference-Counting type.

The Box

The Box<T> is an easy way to put some data on the heap rather than the stack.

You create a Box with `Box::new(...)`.

Reference Counting

The reference counted smart pointer, however, is the thing that allows for shared ownership

The value only goes away when the last reference to it is dropped.

To make a reference-counted object, use type Rc<T>.

Reference Counting

If you want to make another reference to the same object, use `clone()`

When references are dropped, the count decreases.

It is important to note that reference types can leak memory!

Reference Counting

Unlike C++'s analogous `shared_ptr` you can't keep a reference after the Rc goes out of scope.

Right, so we have everything we need now to pass the mutex around, right?

Well, almost. Rc<T> won't work when we try to pass it between threads.

The compiler says it cannot be sent between threads safely
(because Rc<T> doesn't implement the special Send trait).

Atomic Reference Count

We need the *atomic* reference counted type, which is `Arc<T>`.

It is perhaps slightly slower than the regular reference counted type.
But exactly what we need here.

SIGINT Handler

Let's catch a SIGINT with an atomic type.

```
use std::sync::Arc;
use std::sync::atomic::{AtomicBool, Ordering};

fn main() {
    let quit = Arc::new(Mutex::new(false));
     let handler_quit = Arc::clone(&quit);
     ctrlc::set_handler(move || {
         let mut b = handler_quit.lock().unwrap();
         *b = true;
     }).expect("Error setting Ctrl-C handler");

     while !(*quit.lock().unwrap()) {
         // Do things
     }
}
```

In this example, I use a mutex to protect a boolean that's used concurrently (even if it's not in two threads).

Deadlock

There still exists the possibility of a deadlock in Rust.

Nothing prevents thread 1 from acquiring mutex A then B
and thread 2 from concurrently acquiring B then A.

This language cannot solve all concurrency problems.

Lifetimes

I want more life, father. – Roy Batty, Blade Runner (1982)

The compiler can make a determination about how long a piece of data will live.

Usually it gets it right, but we might have to help a bit.

If A predeceases B...

Here's a simple program from the official docs that won't compile because the type system can't figure out what's correct:

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}

fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

It's not sure how long the strings live.

Why Does It Not Know?

It might look like it's obvious at this point, because the two strings are known at compile time.

The compiler, however, makes decisions based on local information only
(that is, what it finds in the current function it is evaluating).

For that reason, it treats `longest` as if it could take any two string references.

Be Specific

To get this to compile, we have to specify lifetimes using annotations.

Annotations don’t change how long references live, really.

They just describe the relationships between the lifetimes of references.

Analogy: expiration dates on food.

Lifetime Annotations

Lifetime annotations are written with an apostrophe ' followed by a name, and names are usually short like 'a or 'b.

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

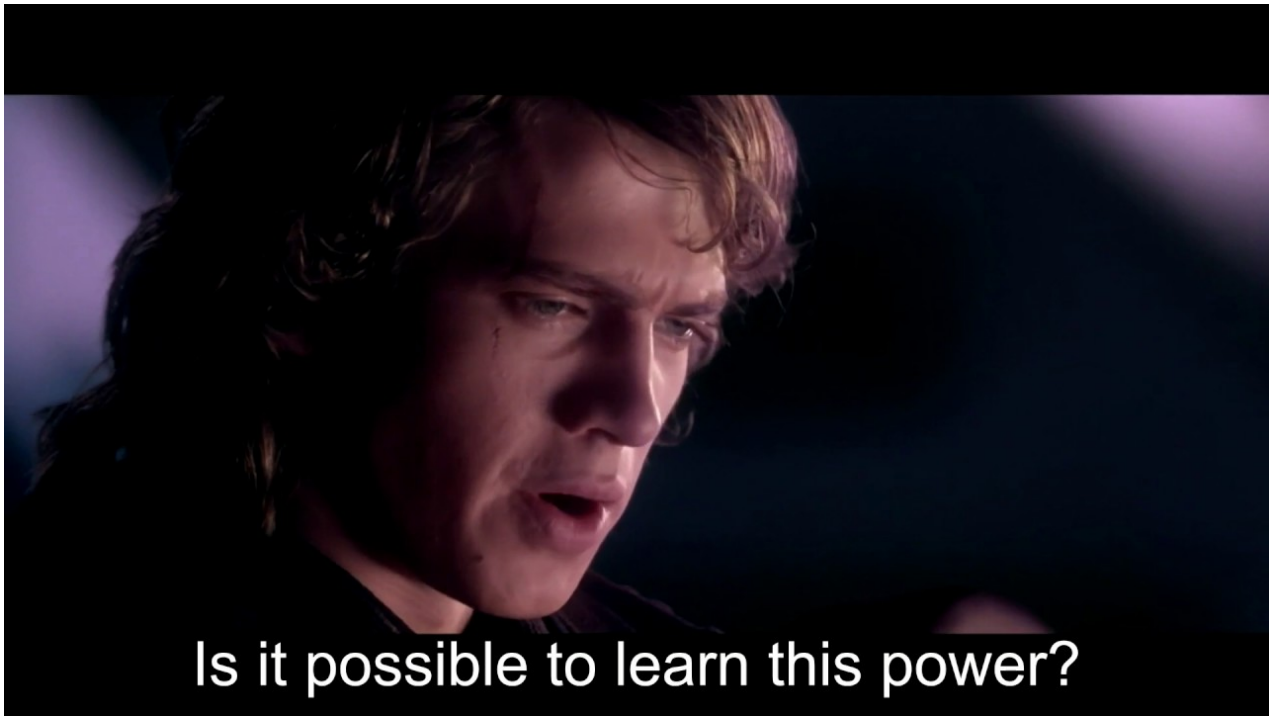
In early versions of Rust, lifetime annotations had to be specified everywhere.

Breaking the Rules?

But we're not breaking rules here, we're applying more rules. What gives?

The rule-breaking thing is the ability to grant a particular piece of memory immortality.

Immortal?



If you specify as a lifetime the special one `'static`, you grant the ability for this memory to live the entire duration of the program.

This can be used correctly to tell the compiler that a particular reference will always be valid.

Immortal?

For the record, the kind of immortality we are talking about here is the Tolkien-Elf kind.

They won’t die of old age, but can die in violence or grief.

I Said Don’t Do It

You can use the static lifetime to bandaid a couple of compiler errors, and the compiler might even suggest it.

You should really apply the correct lifetime annotations or fix the would-be dangling reference.

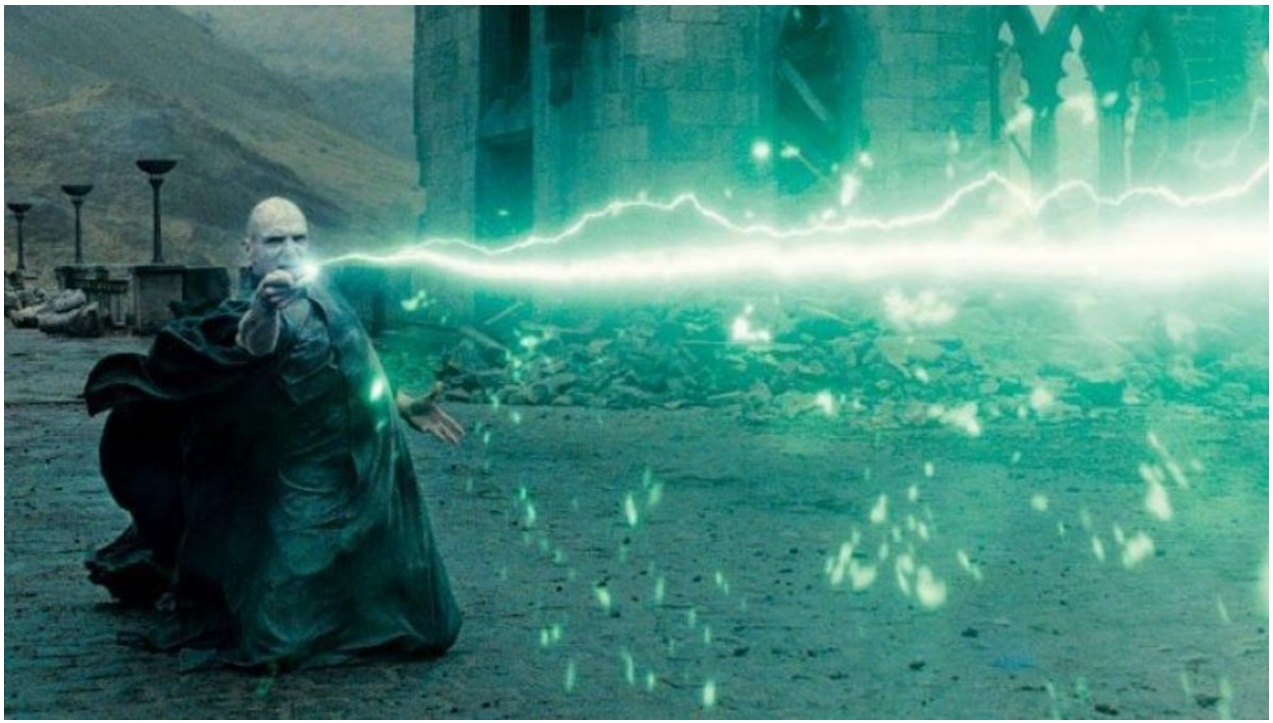
A reference isn’t appropriate and you need to either move the data, copy the data, or use some other construct like the `Arc`

Philosophy, Maybe?

Memory that's kept around forever that is no longer useful is fundamentally very much like a memory leak.

Holodeck Safeties are Offline

There's one last thing that we need, and it is dark and terrible magic.



It is unsafe.

Ancient and Terrible

This mode exists because it has to.

The compiler may err on the side of caution.

You might need to interact with a library or write low-level code.

Ancient and Terrible

You use unsafe at your own risk, though, because you can get it wrong.

If you do you get all the same problems Rust tries to avoid (aka undefined behaviour):

segmentation faults and memory leaks.

Fundamentally, you declare a block as `unsafe`.

Turn on Unsafe Mode

Expectation:

Reality:

What Does Unsafe Allow?

Inside an unsafe block, you can do the following things that you are not normally allowed to do:

1. Call an unsafe function/method
2. Access or modify a mutable static variable
3. Implement an unsafe trait
4. Access the fields of a `union`
5. Dereference a raw pointer

Unsafe blocks are supposed to be small.

Danger Zone

You can also specify that a given function is `unsafe` by putting that in the function signature.

An unsafe function is one that comes with safety conditions.
You are responsible for ensuring the safety conditions when calling the function; the compiler isn't smart enough to do it.

Or, a trait may be `unsafe`.
When implementing an unsafe trait (like `Sync`), you are responsible for ensuring that it upholds the required safety conditions.

Being Dangerous

Suppose we have an unsafe function `do_unsafe_thing()`.

Its function signature will be something like `unsafe fn do_unsafe_thing()`.

To call it, we must wrap it in an unsafe block:

```
unsafe {
  do_unsafe_thing();
}
```

Unsafe appears in the declaration and the invocation.

Unsafe blocks don't have to be in unsafe functions; if not in an unsafe function, you're saying that the function is unconditionally safe to call.

Check and Be Sure

If you try to use an unsafe function without it being in an unsafe block, the compiler will, naturally, forbid such a thing.

You can write `unsafe` around it, of course.

A good reviewer will ask if you're sure this is okay.

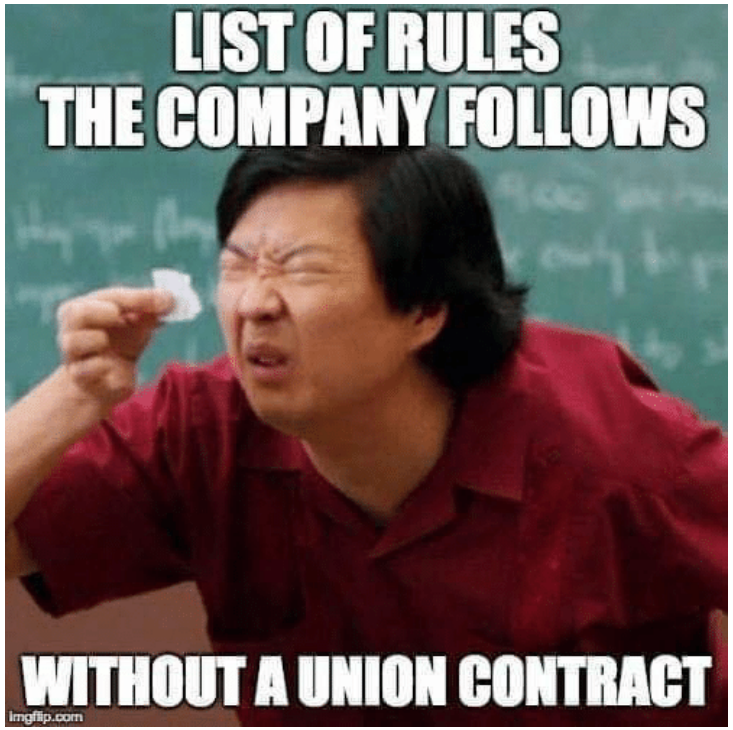
Mutable Static Variables

Rust tries pretty hard to discourage you from using global variables.

You can make global variables mutable in Rust, if you must, and do so you have to mark this as unsafe.

If you find yourself doing such a thing, please stop and think very carefully about why.

Unions



Oh, not that kind of union. The C `union`.

It's like a `struct`, except where a `struct` is all of the contents, a `union` is only one of those at a time.

Raw Pointers

You can create raw pointers anywhere you like, but to dereference them, that has to be in an unsafe block.

Creating the raw pointers can't cause a program crash;
only using them does that.

Of course, creating them incorrectly guarantees that,
when you try to use them, they blow up in your face.

I guess blame is a tricky subject.

Raw Pointers

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

You can also use raw pointers when you need to write to a particular memory address, which sometimes happens for memory-mapped I/O.

You just assign your value to an integer (i.e., `let add = 0xDEADBEEF`) and then cast it to a raw pointer (which is of type `*const`).

When you want to write some data to that address, use the unsafe block.

Crates

You might need unsafe if you are calling into a C library or function.

The Rust universe of packages (“crates”) is getting larger all the time, but sometimes you’ll have to interact with a library in C.

There is a crate for cURL, but it might be interesting to learn what one would have to do to use the C library for it...

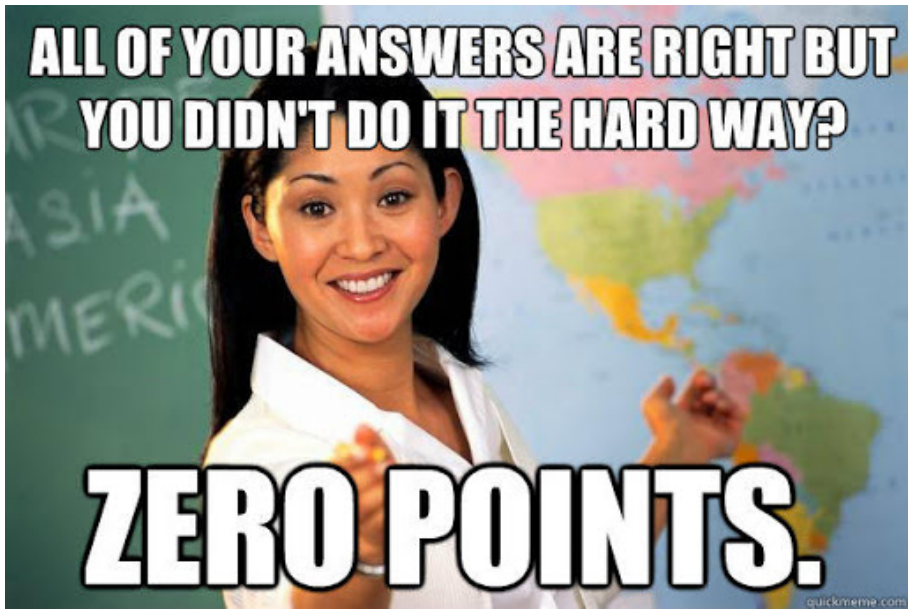
Rust Pitfalls

Some common pitfalls we've seen:

- Using `static` lifetime
- Overuse of `clone()`
- Array indexing overhead vs iterator
- Unbuffered I/O
- `resize()` on a vector
- Unguarded `unwrap()` everywhere

Doing it by hand...

In previous courses: do things the hard way: write your own!



In industry, you'll use libraries that have appropriate functionality.
... Assuming the license for them is acceptable to your project.

Appendices!

Although the topic of using libraries used to have its own lecture, we've now moved that subject matter to the appendices for the course.

It's about Crossbeam and Rayon.

Use them as appropriate in the course examples, assignments, et cetera!