ECE459: Programming for Performance Winter 2021

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

Jeff Zarnett *2020-10-11*

Mutual Exclusion and Multiple Ownership

Mutex and Reference Counting Where we left off previously, we've identified that a mutex is not super amenable to our model of single ownership because we need multiple threads to have access to this mutex. There is a way to do it, but we have to break the single ownership rule, and that requires a little more background on smart pointers and reference counting.

We know what pointers are from C and C++, and if you have sufficient experience with C++ you will know that smart pointers exist in that language too! We'll talk about two kinds of smart pointer right now, the Box and the Reference-Counting type.

The `Box<T>` is an easy way to put some data on the heap rather than the stack. This is good for a situation where you, for example, take input from a user and you don't know in advance how big it's going to be, or when you have some data that you want to transfer ownership of rather than copy (for performance reasons, obviously). You create a Box with `Box::new(...)` as expected, and it's heap allocated with all the usual things that come with it in Rust, like ownership and that it gets dropped if the owner goes out of scope.

The reference counted smart pointer, however, is the thing that allows for shared ownership. There are some reasons why we might want this, even in a single-threaded program, such as a graph data structure. But the main idea is that you can share ownership as much as you like, and the value only goes away when the last reference to it is dropped (reference count goes to zero).

To make a reference-counted object, use type `Rc<T>`. Instantiate that type to get an object; if you want to make another reference to the same object, use `clone()`, which increases the reference count. When references are dropped, the count decreases.

It is important to note that reference types can leak memory! If you've chosen this route for managing data in your program, there is a possibility of forming a cycle in the reference types. If such a cycle is formed, the memory will never be dropped. This is undesirable, of course.

What you can't do, unlike C++'s analogous `shared_ptr`, is keep a reference to the value after the `Rc` or `Arc` goes out of scope. That's because the value is still owned by the pointer and is definitely freed when the pointer goes away.

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, because the compiler says it cannot be sent between threads safely. This is because the management of its internal counter is not done in a thread-safe way. If we want that, we need the *atomic* reference counted type, which is `Arc<T>`. It is perhaps slightly slower than the regular reference counted type, so you won't want to choose it in every scenario, but it's exactly what we need here.

Here's an example of using an atomic reference counted type for setting up a handler for the Ctrl-C (SIGINT); this is modified from a program I wrote that listens for connections and spawns threads if a client connects:

```
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): once in main and once in the handler.

We should also still remember that there exists the possibility of a deadlock in Rust, even if the mutex is automatically unlocked for us. 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, unfortunately.

Lifetimes

We've covered the idea that in Rust, the compiler can make a determination about how long a particular piece of data will live. How long it lives is sometimes referred to as the reference's lifetime. The good news is that the compiler is usually able to make a determination about how long things should live. This system is not perfect, and sometimes we have to help it a bit.

Here's a simple program in the official docs that won't compile because the type system can't figure out what's correct [KNC20]:

```
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
    }
}
```

The compiler says it can't figure out whether the return value is the borrowing of `x` or `y` and therefore it's not sure how long those strings live. 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. Alright, that's fine for now, because it was an example to show what happens when we can't know the answer at compile time anyway.

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. This is used on functions to specify what they can accept and what they can return.

If you'd like an analogy, think of it as saying something like "I will only buy eggs that have an expiration date that is at least two weeks in the future.". This rule does not change the eggs that are in the store. It does not mean that eggs that have an expiration date of next week are poison and nobody should eat them. I'll happily buy eggs that expire in a month. So we are just being clear about what we want here.

Lifetime annotations are written with an apostrophe `'` followed by a name, and names are usually short like `'a` or `'b`. Let's correct the `longest` function:

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

This does what we need! The first appearance, after the name of the function, of our lifetime annotation says that all parameters and return value must have the same lifetime. Then we say we will accept strings that live at least as long as our designated `'a` lifetime. That is, it's got to live at least as long as the smallest of `x` and `y`. The actual lifetime isn't as important, all that matters is that it follows the rule of being at least that long.

In early versions of Rust, lifetime annotations had to be specified everywhere. That was somewhat annoying, but fortunately the compiler can identify a lot of common scenarios, so the borrow checker can read them in where they're needed most of the time.

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. 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. Just because it can doesn't mean it necessarily will live forever—only that it could.

This can be used correctly to tell the compiler that a particular reference will always be valid, such as string literals that are always going to hang around. It's also used in the interface for spawning a thread, incidentally, which happens because you can pass *anything* to a thread, and the compiler wants to be sure that whatever you are providing is definitely going to live long enough for the thread which can live an arbitrarily-long life (threads are estimated to be immortal).

For the record, the kind of immortality we are talking about here is the Tolkien-Elf kind, where they won't die of old age, but can die in violence or grief. Threads can exit and be cancelled and such in Rust, and static variables can get dropped if the compiler is sure it's safe to do. But they *could* hang around indefinitely.

You can use the static lifetime to bandaid a couple of compiler errors, and the compiler might even suggest it. You shouldn't, though, you should really apply the correct lifetime annotations or fix the would-be dangling reference. The compiler can lead you down the wrong path if it says that it wants a vector being passed to a thread to be annotated as static. What you might think is that it means you should annotate the function parameter with this lifetime, but that just moves the pain to where the function is called. So you modify those and it goes up the chain until you're at creation of the vector and you're left wondering how to make it have a static lifetime. Really, what the compiler means is that a reference isn't appropriate and you need to either move the data, copy the data, or use some other construct like the `Arc` (atomic reference counter) that is appropriate to the situation.

Memory that's kept around forever that is no longer useful is fundamentally very much like a memory leak, even if it is still possible to deallocate it in a hypothetical sense.

Holodeck Safeties are Offline

There's one last thing that we need, and it is dark and terrible magic. It is *unsafe*. Unsafe exists because it has to. The compiler would rather err on the side of caution and say no to a program that is correct, than say yes to one that isn't. You might also need to interact with some other library or do some very low-level stuff. For this reason, we can override this and tell the compiler that you promise you know this is okay. You do so at your own risk, though, because you can get it wrong and if you do you get all the same problems Rust tries to avoid, like segmentation faults and memory leaks.

To do anything that qualifies as unsafe, you can go one of two ways. Either you declare a block as `unsafe`, or you specify that a given function is `unsafe` by putting that in the function signature. Inside an unsafe block or function, you can do the following things that you are not normally allowed to do [KNC20]:

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

That list is probably less extensive than you were expecting. Declaring a block as unsafe does not grant you unlimited power, sadly. The borrow checker still does its thing and there are still rules.

The design intentions for unsafe blocks are that they are supposed to be small (this reduces the chance of an error and makes it easier to find) and ideally are abstracted away a bit behind some interface...

Danger Zone. The easiest example to show is what happens when you want to call a function that is unsafe. Suppose we have a function `do_unsafe_thing()`; its function signature will be something like `unsafe fn do_unsafe_thing()` and to call it, we must wrap it in an unsafe block:

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

Unsafe things are clearly designated in both ways: when you write a function that is unsafe, you declare to the world that this function is unsafe. Then, anyone who wants to use it also has to acknowledge that they know the function in question is unsafe. (Readbacks are a safety convention used in aviation, among other places.)

If you try to use an unsafe function without it being in an unsafe block, the compiler will, naturally, forbid such a thing. Just smashing the unsafe block around it is enough to make the compiler quiet, but not a thorough code reviewer. They would ask about whether you've read carefully the documentation of the function in question and whether you are sure you're calling it with the right arguments... You did read the documentation, right? Right?

Mutable static variables. Rust tries pretty hard to discourage you from using global variables, and they are right to do so. It's a quick shortcut and we do it a lot in course assignments, exercises, labs, and even exam questions. On an exam question, the thing I want to test is something like how you use the mutex and queue constructs to solve the problem, not how well you pass the mutex and queue pointers from the main thread to the newly created threads. In production code, though, global variables are really not recommended because of how harmful it is to good software engineering principles.

But anyway, you can make global variables mutable in Rust, if you must, and do so you have to mark this as unsafe. But if you find yourself doing such a thing, please stop and think very carefully about why.

Implement an unsafe trait. Appropriately, if the trait (interface) you want to implement has unsafe in the function signature, the compiler forces you to admit that your code is unsafe. If you do, it mostly means that you have to guarantee that what you're doing does in fact meet the requirements the interface specifies (like `Send`).

Unions. In C, there exists the concept of the `union`[1]. You might not have heard of it because a lot of people don't like it (and I'm one of them). You might have to contend with it in a particular API. It's like a `struct`, except where a `struct` is all of the contents (e.g., an integer and a floating point number and a pointer), a `union` is only

[1] `https://en.wikipedia.org/wiki/Union_type`

one of those at a time (an integer or a floating point number or a pointer). Because there's no way to be totally sure that the `union` you're looking at is in fact the type you expect it to contain, you can only access the members in an unsafe block.

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.

Here's an example from the official docs [KNC20]:

```
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 and write it.

You might need this 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. . . or write a part of your application in Rust that is called from 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. . .

References

[KNC20] Steve Klabnik, Carol Nichols, and Rust Community. The Rust Programming Language, 2020. Online; accessed 2020-09-12. URL: `https://doc.rust-lang.org/book/title-page.html`.