ECE459: Programming for Performance Winter 2024

Lecture 3 — Rust: Borrowing, Slices, Threads, Traits

Jeff Zarnett *2024-09-09*

Borrowing and References

We've already seen that ownership is a concept in Rust that can come with a couple of unintended consequences, e.g. from accidentally giving an argument to a function that we still need later. Rust supports "borrowing"—you need to use the data for something but you also promise you'll give it back (and the compiler forces you to live up to your promises). Borrowing allows data to be shared, but the sharing has to be done in a controlled and safe way to prevent leaks and race conditions.

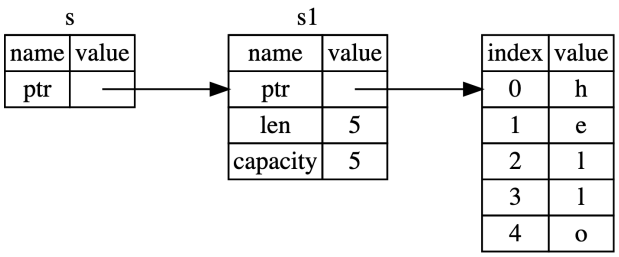
Rust's compiler analyzes all the borrowing that takes place in the program using the *borrow checker*. If the borrow checker is not certain that your code is perfectly safe, it will say no (and produce a compile time error). This can be a little bit frustrating, because the analysis is not perfect and errs on the side of caution. Eventually we will introduce some ways that you can tell the borrow checker that you guarantee the code is safe, but you have to be sure, otherwise all the usual bad things can happen!

The feature that we need for the concept of borrowing is the *reference*. To indicate that you want to use a reference, use the & operator. The reference operator appears both on the function definition and the invocation, to make sure there's no possibility of confusion as to whether a reference is being expected/provided or ownership is to be transferred. Consider this example from the official docs [KNC20]:

```
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The␣length␣of␣'{}'␣is␣{}.", s1, len);
}

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

When we invoke the `calculate_length` function, ownership of the string is not transferred, but instead a reference to it is provided. The reference goes out of scope at the end of the function where it was used, removing it from consideration. A reference is not the same as ownership and the reference cannot exist without the original owner continuing to exist. That is represented in the official docs by this diagram:



And if you borrow something, it's not yours to do with as you wish—you cannot assign ownership of it (move it), which makes sense because you can't give someone ownership of something you do not own.

By default, references are immutable: if you borrow something, you cannot change it, even if the underlying data is mutable. Attempting to do so will result in—you guessed it—a compile time error, where the compiler tells you that you are trying to change something that's immutable.

Of course, in real life, you would be much more agreeable to letting people borrow your things if there were strong guarantees that it would (1) always be returned and (2) would be returned in the same condition. That would be nice! But until such time as that magical technology is invented, no, you can't borrow my car. Sorry.

Mutable references do exist, but they have to be declared explicitly as such by tagging them as `&mut`:

```
fn main() {
    let mut s1 = String::from("hello");
    let len = calculate_length(&mut s1);
    println!("The␣length␣of␣'{}'␣is␣{}.", s1, len);
}

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

Mutable references come with some big restrictions: (1) while a mutable reference exists, the owner can't change the data, and (2) there can be only one mutable reference at a time, and while there is, there can be no immutable references. This is, once again, to prevent the possibility of a race condition. These two restrictions ensure that there aren't concurrent accesses to the data when writes are possible. There's also a potential performance increase where values can be cached (including in CPU registers; we'll come to that later) without worry that they will get out of date.

As long as there are no mutable references, there can be arbitrarily many immutable references at the same time, because reads don't interfere with reads and a race condition does not occur if there are only reads.

References cannot outlive their underlying objects. Below is an example from the official docs that will be rejected by the borrow checker, because the reference returned by `dangle` refers to memory whose owner `s` goes out of scope at the end of the function:

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

fn dangle() -> &String {
    let s = String::from("hello");
    &s // returning a thing that no longer exists upon return
}
```

In C this would be a "dangling pointer" (a pointer that's pointing to a location that is no longer valid). I see this kind of error a lot in C programs where someone has stack allocated a structure and then wants to pass it to another thread and does so with the address-of operator. It compiles and might even work sometimes at runtime, but is still wrong (technically, undefined behaviour) and can eventually be exposed as a bug that bites you.

If we actually try to compile the previous code example, the compiler says something about giving the value a lifetime. We'll come back to the idea of lifetimes soon.

Non-Lexical Lifetimes. A more recent improvement to Rust's borrow checking is called non-lexical lifetimes. Consider the small block of code below:

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

    let y = &x;
    println!("{}", y);

    let z = &mut x;
}
```

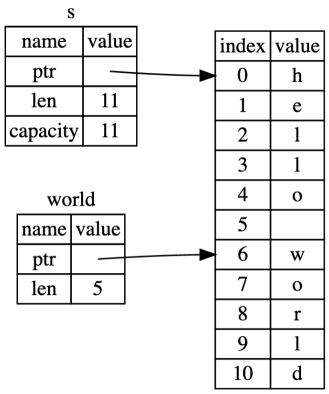
Under the old rules, the compiler would not allow creation of the mutable reference `z` because `y` has not gone out of scope. It would consider `y` to be valid until the end of the function. The improvement of NLL is that the compiler can see that `y` is no longer used after the `println!` macro and hence the `z` reference is okay to create. `y` can be dropped as soon as it's no longer needed; the `z` reference will not exist at the same time; and all is fine.

Slices

The *slice* concept exists in a few other programming languages, and if you have experience with them this will certainly help. A slice is a reference (yes, a reference in the sense of the previous section) to a contiguous subset of the elements of a collection. This is what you do if you need a part of an array (the typical example for that being a substring of an existing string). If our code looks like this:

```
fn main() {
    let s = String::from("hello world");
    let hello = &s[0..5];
    let world = &s[6..11];
}
```

The representation of the slice looks like [KNC20]:



Slices can also apply to vectors and other collections, not just strings. As with the other kinds of references we've learned about, the existence of a slice prevents modification of the underlying data. Just as with references, slices prevent race conditions on collections but also avoid (as much as possible) the need to copy data (which is slow).

Unwrap the Panic

A quick digression: a lot of functions we use return `Result` types. These return either `Ok` with the type we expected, or `Err` with an error description. To get the type you want, you need to unpack the result.

If we try to open a file but the file doesn't exist, that's an error but one that's foreseeable and we can handle it. There's three ways to handle it: a `match` expression (this is like the `switch` statement), `unwrap()`, and `expect()`.

You may be tempted to just always use `unwrap()` because it gives you the result and calls the `panic!` macro if there's an error. This, however, just shows you the lower level error that is the problem and you are denying yourself the opportunity to add information that will help you debug. For that reason, it's better to use `expect()`, which lets you add your own error message that will make it easier to find out where exactly things went wrong.

It's recommended to use `Result` types for functions you write too. Make your future self happy by giving yourself the information you need to debug what's gone wrong!

This does come at a small performance hit. CS 343 included a performance comparison of exceptions versus error codes (à la `Result`). Exceptions that don't happen (the happy path) are, indeed, faster than explicitly handling error codes / `Result` types. As an engineering trade-off, both exceptions and `Result` types force the programmer to explicitly deal with errors (at the very least, explicitly ignoring them rather than silently ignoring them). Rust doesn't have exceptions, so you always have to pay the performance hit. (Simulating exceptions with `panic!` is not a best practice.)

Fearless Concurrency

More than just trying to prevent memory problems by making them compiler errors, Rust is also intended to make concurrency errors compile-time problems too! That's actually difficult, of course, but the good news is that the key ideas of ownership and borrowing and such will help you avoid concurrency problems.

The drawback to concurrency is that it brings new problems with it: race conditions, deadlock, that sort of thing. Making your program faster is great, but not if it's at the cost of the answers being incorrect (or your program failing to produce an answer some of the time).

If the compiler can help with making sure your concurrent program is correct, it doesn't make your program faster directly, but it helps indirectly. If you can be (more) sure of the correctness of your code, you don't have to spend as much time testing it before you can deploy it and move on to the next thing. Also, if the bug is prevented from being introduced in the first place, you don't have to spend time debugging it and fixing it, which lets you spend more time on speeding up other things. And honestly, if you are looking at a piece of code that is super business critical, anything that adds to your confidence that no issue has been introduced makes it that much easier to make that change you want to make.

Threads. Rust uses threads for concurrency, with a model that resembles the create/join semantics of the POSIX pthread. If you are unfamiliar with pthreads, the course repository has a PDF refresher of the topic (`pthreads.pdf`). We will talk about the Rust way, but the background material will provide context.

OK, so you want to create a thread! The mechanism for doing so is referred to as spawning a thread. Here's a quick example from the official docs [KNC20]:

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi␣number␣{}␣from␣the␣spawned␣thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi␣number␣{}␣from␣the␣main␣thread!", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```

A few things make this significantly different from the pthread model that we are used to. First of all, the thread being created takes as its argument a *closure*—an anonymous function that can capture some bits of its environment. The spawn call creates a `JoinHandle` type and that's what we use to call `join`, which is to say, wait for that thread to be finished. As we expect from pthreads, if calling join on a thread that is not finished, the caller waits until the thread is finished.

This is a simple example that works, but fails to capture the complexity of actually working with threads, because

there's no data moved between threads. Most interesting uses of threads need some data communication. There are three ways that we can get data from one thread to another: capturing, message passing, and shared state.

Capturing. The notion of "capturing" calls back to the earlier mention that a closure captures some of its environment. That is, the body of the function can reference variables that were declared outside of that function and in the context where `thread::spawn` was called. The compiler will analyze the request and try to figure out what needs to happen to make it work, such as borrowing the value, as in this example (also from the docs):

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's␣a␣vector:␣{:?}", v); // can't do this
    });

    handle.join().unwrap();
}
```

The only problem is: this example does not work. The compiler is not sure how long the thread is going to live and therefore there's a risk that a reference to `v` held by the thread outlives the actual vector `v` in the main function. How do we fix that?

Well, I had the idea that if I put something after the `join()` call that uses `v`, then the compiler should know that `v` has to remain in existence until after the thread in question. Yet, it still reports the error E0373 that says the thread might outlive the borrowed value. This actually got me thinking about why this didn't work and I decided to ask some of the compiler devs. It has to do with the fact that a thread isn't really a first-class construct in Rust, and the "lifetime" of arguments that you pass has to be sufficiently long. We'll learn about lifetimes soon.

Anyway, the error message suggests what you actually want in this scenario: to move the variables into the thread. To do so, specify `move` before the closure: `let handle = thread::spawn(move || {...` This addition results in the transfer of ownership to the thread being created. You can also copy (i.e. clone-and-move) if you need.

One thing you don't want to do is try to make the lifetime of your vector or other construct `static`, even though the compiler might suggest this. We can revisit that when we talk about lifetimes as well.

Message Passing. Sometimes threads want to communicate in a way that isn't one-way communication at the time that the thread is being created. For that, a possibility is message-passing. This mechanism of communication may seem familiar from previous experience with various UNIX mechanisms like pipes and message queues. This strategy is very structured and generally safer than shared memory, i.e. it is harder to race or to access inappropriate locations.

The ownership mechanic of message passing is like that of postal mail. When you write a physical letter and mail it to someone, you relinquish your ownership of the letter when it goes in the mailbox, and when it is delivered to the recipient, the recipient takes ownership of that letter and can then do with it as they wish.

So you want to have two threads communicate. The metaphor that Rust (and many others) use for this is called a *channel*. It has a transmit end (where messages are submitted) and a receive end (where messages arrive). The standard model is multiple-producer, single-consumer: that is, lots of threads can send data via the sending end, but in the end it all gets delivered to one place. Think of that like postal mail as well: I can drop a letter to you in any postbox or post office, but they will all be delivered to your mailbox where you collect them in the end.

Okay, enough talk, let's make one [KNC20]:

```
use std::sync::mpsc;
use std::thread;

fn main() {
```

```
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

The channel constructor returns a tuple with the transmitting end `tx` and receiving end `rx`. We'll then send the transmitting end into the thread and have it send a message to the main thread. The main thread will wait until the message is there and then get it. This does mean that `recv()` is blocking and there is a corresponding `try_recv()` which is nonblocking. You may have already covered nonblocking I/O in a previous course; if not, we will return to that subject soon.

If you want to have multiple transmitting ends, you need only use `clone` on the transmitter and hand those out as needed.

As a small technical note, the type you want to send via a channel has to implement the `Send` trait (think of traits being like interfaces). Almost all basic types in Rust have this trait, and any programmer-defined type that is composed entirely of types that have it will also have that trait.

Traits

Okay, we have to take a detour here onto the subject of Traits. As the previous paragraph said, traits are a lot like interfaces. You specify a trait as a set of function signatures that you expect that the type in question to implement. A very simple trait and its usage are shown below:

```
pub trait FinalGrade {
    fn final_grade(&self) -> f32;
}

impl FinalGrade for Enrolled_Student {
    fn final_grade(&self) -> f32 {
        // Calculation of average according to syllabus rules goes here
    }
}
```

A couple of other notes about traits are worth mentioning. One, you can only define traits on your own types, not on external (from other packages/crates) types, so that you don't break someone else's code. Two, you can add a default implementation to the trait if you want (something Java lacked for a long time). Third, as in other languages with interfaces, a trait can be used as a return type or method parameter, so it is a kind of generic. Finally, you can use + to combine multiple traits (which is nice when you need a parameter to be two things)

With the preamble out of the way, there are three traits that are really important to us right now. They are `Iterator`, `Send`, and `Sync`.

Iterator is the easiest one to explain. You put it on a collection and it allows you to iterate over the collection. Moreover, this is often more efficient than a typical for loop construction, because it lets the compiler skip over bounds checking and other such issues. Nice.

Send was already introduced. It's necessary to transfer ownership between threads. There are some Rust built-in or standard-library types that very specifically choose not to implement this interface to give you a hint that they are not intended for this purpose. If the compiler tells you no, it's a hint that you want to use a different type. As previously mentioned, if your programmer-defined type is made entirely of types that have the Send trait, then it too has the trait. If you really must use something that is inherently not safe to send, though, you can implement

this trait on your type manually and guarantee the thread-safe transfer of ownership yourself, but it's not a good idea if you can avoid it.

Sync is the last one, and it means that a particular type is thread-safe. That means it can be referenced from multiple threads without issue. The primitive types have this trait, as do any programmer-defined types that are composed entirely of Sync types[1]. It's important to just mention here that this does not mean all operations on a Sync type are safe and that no race conditions are possible; it just means that *references* to the type can be in different threads concurrently, and we can't have multiple mutable references. No, if we want more than one thread to be able to modify the value, we need mutual exclusion...

Back to the Mutex...

If you don't want to use message passing for some reason (and performance is a reason, if it's borne out by your testing/data) then there is fortunately the ability to use a mutex for mutual exclusion. We know how these work, so let's skip the part where I make some analogy about them.

What's different about the mutex in Rust is that the Mutex wraps a particular type. So it is defined as `Mutex<T>` and if you want an integer counter initialized to 0, you create it as `Mutex::new(0);`. This way, the mutex goes with the value it is protecting, making it much more obvious what mutex goes with what data, and making it so you have to have the mutex to access the data. A sample from the docs [KNC20]:

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

    println!("m␣=␣{:?}", m);
}
```

In addition to forcing you to acquire the mutex before you can make any use of the internal value, the lock is automatically released when the `num` variable goes out of scope; the type of `num` is a `MutexGuard` which is our "possession" of the lock; when that possession ends, the mutex is automatically unlocked. This means you want, generally, to use the manual-scoping `{` and `}` braces to ensure that the lock is released when you're done with it and not just at the end of the function or loop.

The use of the mutex in the above program is obviously unnecessary, since there's only the one thread. If we want to use it in multiple threads, we need multiple threads to access it. But we can't, unfortunately, just say that references will do! The mutex type has to outlive the other threads and such and the compiler will suggest moving it... But we can't move it into more than one thread, because that violates our rule about having only one owner. What now?

It looks like we have to break a rule: we need the ability to share ownership of some memory. We don't know how to do that, but when we start with breaking rules, we might find that we like it and might break more than one...

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`.

[1] If you are yourself implementing something that implements Sync—not by composition—and you do it wrong, you can cause undefined behaviour. But if you are doing that you will be forced to tag your implementation with the `unsafe` keyword, which is beyond the scope of this course.