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

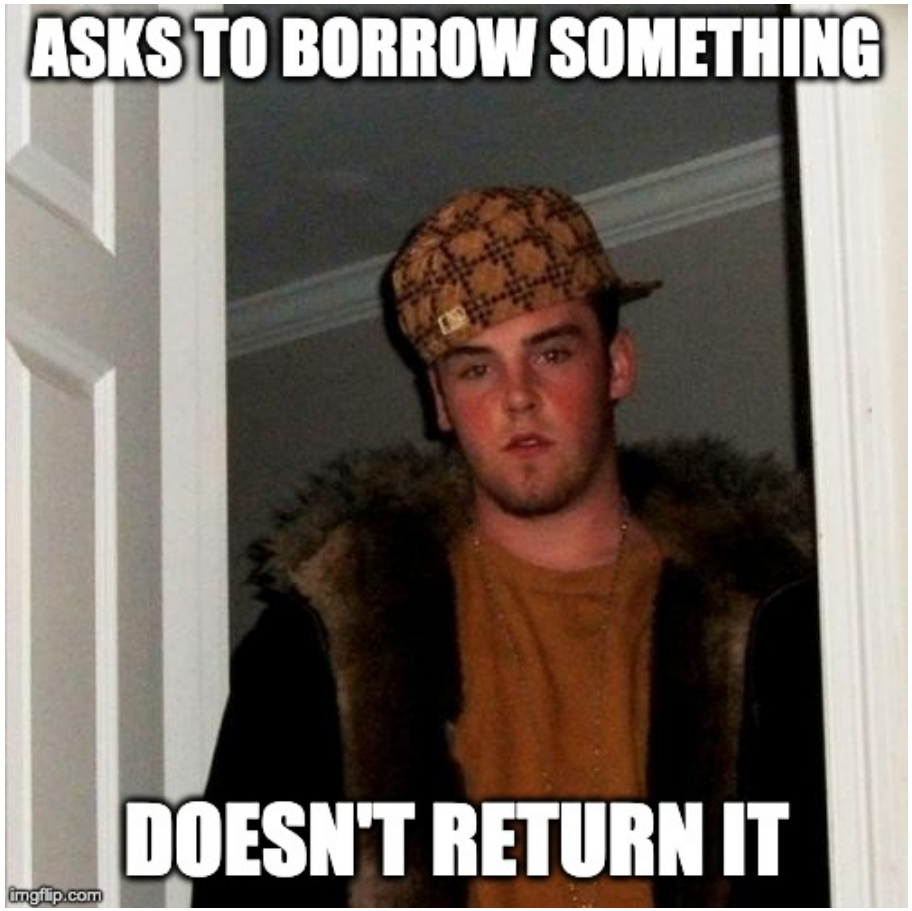
Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 4, 2022

Borrowing and References

Ownership in Rust can come with unintended consequences, like accidentally giving away something we need later.



Rust supports borrowing – you can use it but you promise you will give it back.

Borrowing has rules

Borrowing allows sharing in a controlled way.

The compiler analyzes borrowing with the borrow checker.

Its analysis always errs on the side of caution.

How to Borrow?

The feature that we need for the concept of borrowing is the reference.

The operator: &.

The reference operator appears both on the function definition and the invocation.

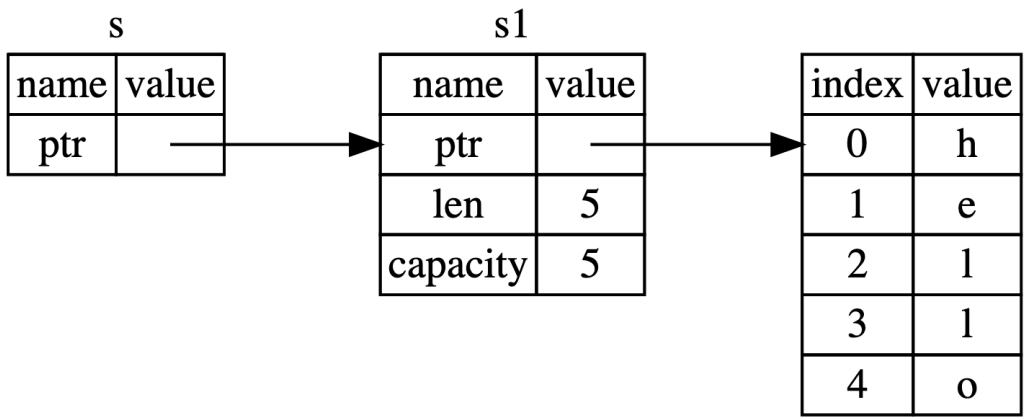
```
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of '{}' is {}.", s1, len);
}

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

Reference Rules

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.



Reference Rules

If you borrow something, you can't do with it whatever you wish.

By default, you can't change it!

Mutable references exist but must be explicitly declared:

```
fn main() {
    let 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 Reference Rules

1. While a mutable reference exists, the owner can't change the data.

2. There can be only one mutable reference at a time.
   And when there is, no immutable references!

These restrictions prevent race conditions.

Time to Die

References cannot outlive their underlying objects.

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

fn dangle() -> &String {
    let s = String::from("hello");
    &s
}
```

In C this would be a “dangling pointer”.

Non-Lexical Lifetimes

A more recent improvement to Rust's borrow checking is called non-lexical lifetimes.

```
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.

Slices

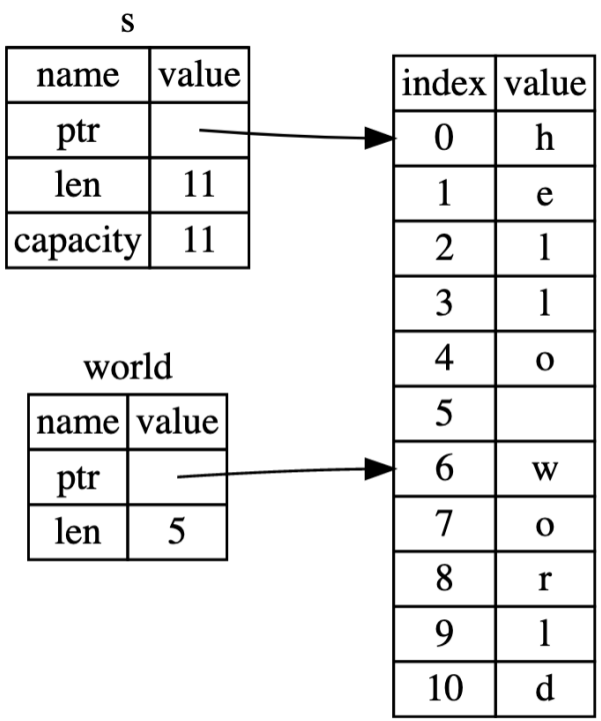
The slice concept exists in a few other programming languages, and if you have experience with them this will certainly help.

A slice is a reference to a contiguous subset of the elements of a collection.

A Less-Delicious Slice

```
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:



Why Slice?

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.

Slices prevent race conditions on collections but also avoid (as much as possible) the need to copy data (slow).

Unwrap Panic

A lot of functions we use return `Result` types.

These return either `Ok` with the type we expected, or `Err` with an error description.

We must unpack the result.

Why?

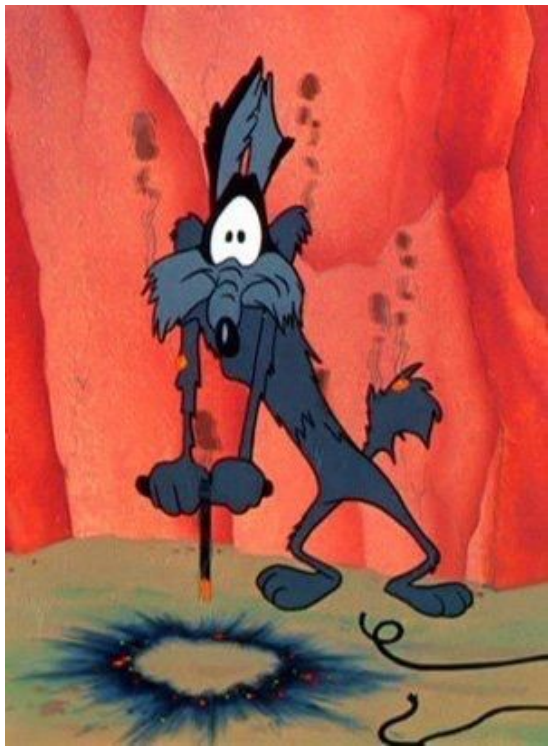
Unpack the Result

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, `unwrap()`, and `expect()`.

Yikes!

You may be tempted to just always use `unwrap()`



Don’t deny yourself the opportunity to add debug information.

It’s better to use `expect()`.

Plan for the Future

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.

Remember though, your time (debugging) is valuable!

Fearless Concurrency

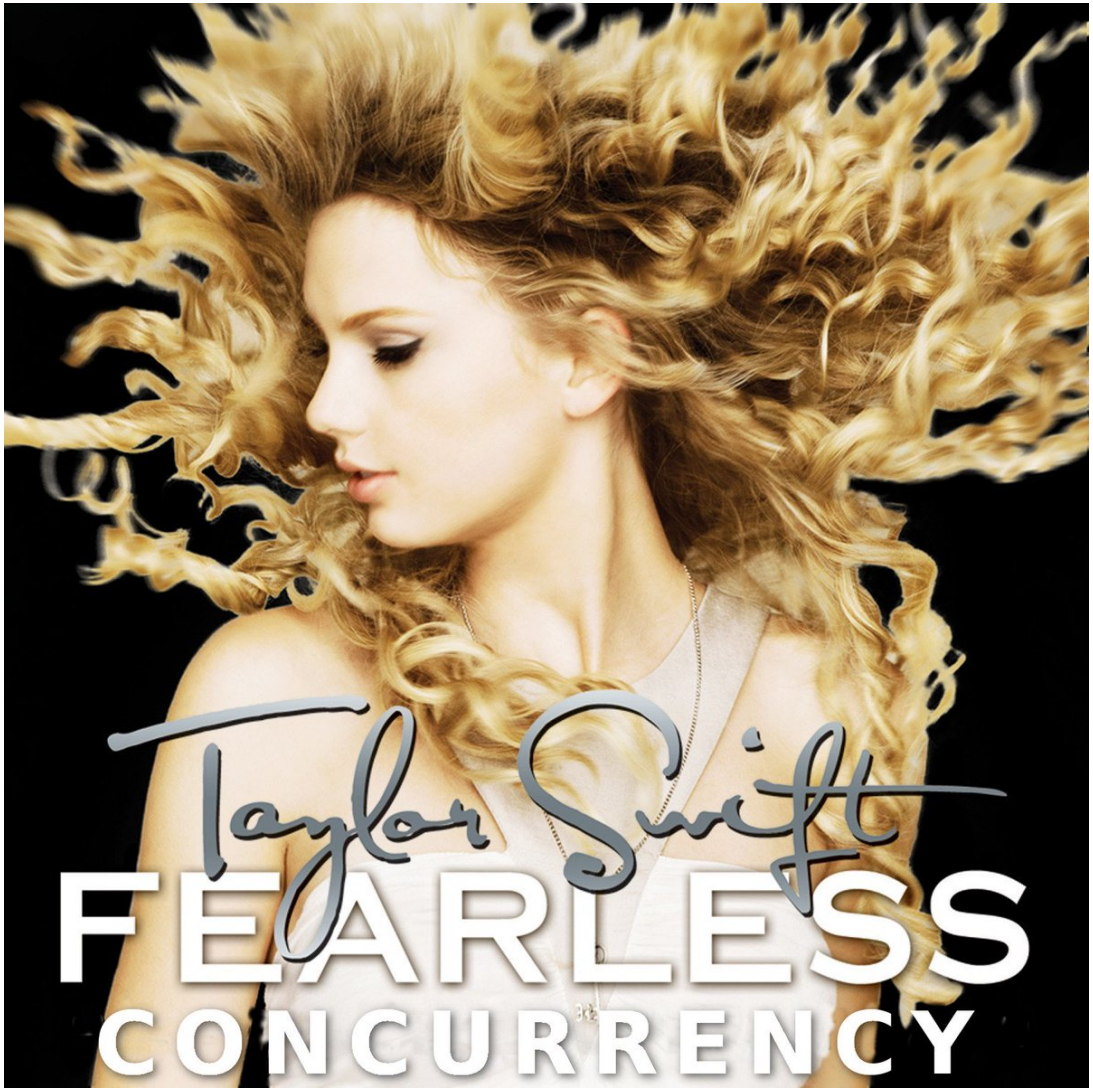
Rust is also intended to make concurrency errors compile-time problems too!

As we know, concurrency brings new problems: race conditions, deadlock, etc.

Faster is not better if it makes your program incorrect.

Fearless Concurrency

A student in a previous term thought the Dune reference was too obscure.



Wait, how does this help?

If the compiler can help with making sure your concurrent program is correct, it doesn't make your program faster directly.

If you can be (more) sure of the correctness of your code, you don't have to spend as much time testing it.

Preventing a bug from being introduced is faster than finding and fixing it.

More guarantees help you write business-critical code with confidence.

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`).

That PDF covers the POSIX thread in C, so you'll be on even footing.

So you want to create a thread...

```
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();
}
```

The thread being created takes as its argument a closure.

The spawn call creates a `JoinHandle`.

Too Easy

This fails to capture the complexity of working with threads: no data is passed.

There are three ways that we can get data from one thread to another: capturing, message passing, and shared state.

Capture

The notion of “capturing” calls back to the earlier mention that a closure captures some of its environment.

```
use std::thread;

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

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

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

The only problem is: this example does not work.

The lifetime of v in the thread might outlive the scope creating the thread.

What about Join?

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.

Turns out, I needed to understand lifetimes in Rust better. We'll return to that.

I like to move it

Anyway, the error message suggests what you actually want in this scenario: to move the variables into the thread.

Put `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 if you need.

Don’t Do It

It might look like the error message is suggesting making the lifetime of your vector `static`.

Don’t do it. We’ll talk about why when we get into lifetimes.

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.

Message passing may be familiar to you from previous courses.

Rust recommends this approach: harder to have races or invalid accesses.

Ownership in Message Passing

The ownership mechanic of message passing is like that of postal mail.

So you want to have two threads communicate.

The Rust metaphor for message-passing is called a channel.
    It has a transmit-end and receive-end.

Communication Model

The standard model is multiple-producer, single-consumer.

```
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);
}
```

`recv()` is blocking and there is a corresponding `try_recv()`

We will cover non-blocking I/O soon.

More About Sending

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.

But what is a trait? Let's talk about it.

Traits are a lot like interfaces: you specify a set of function signatures.

To implement the trait, implement the functions.

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

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

Trait Rules

You can only define traits on your own types.

You can create a default implementation.

Traits can be used as a return type or method parameter.
    Use the + to combine multiple traits in a parameter.

The Big Three

There are three traits that are really important to us right now.

`Iterator`, `Send`, and `Sync`

`Iterator` is easy: put it on a collection, you can iterate over it.

Send

Send is used to transfer ownership between threads.

Some Rust types specifically don’t implement Send as a way of saying, don’t pass this between threads.

If the compiler says no, you need a different type.

Sync

Sync means a type is thread-safe.

This does not mean all operations on a Sync type are safe and that no race conditions are possible.

If we need more than one thread to be able to modify the type, we need mutual exclusion.

Back to the Mutex..

There is also the ability to use a mutex.
I assume you know how they work from previous courses.

New in Rust: the Mutex wraps a particular type.

It is defined as `Mutex<T>` and if you want an integer counter, you create it as `Mutex::new(0);`.

The Mutex goes with the value it is protecting.

Mutex In Action

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

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

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

    println!("m = {:?}", m);
}
```

You must acquire the mutex to access the internal value.

The lock operation returns a `MutexGuard`.
When it goes out of scope, the lock is automatically unlocked.

Silly Example

This program has only 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?

Breaking the Law, Breaking the Law

We have to break a rule: we need to share ownership of some memory.