Rust Bootcamp 2021

Day 5

Day 5 Activities

- ✓ Closures
- ✓ Threads & Mutexes
- ✓ Message Passing
- ✓ Exercises

Closures

Closures are anonymous functions that can capture the environment, and are defined within some other function.

```
// regular function
fn function(i: i32) -> i32 { i + 1 }

// closure
let closure = |i| i+1;
```

Note that closures don't need explicit types, though you can add them if you want.

https://doc.rust-lang.org/stable/rust-by-example/fn/closures.html

Closures - Capturing

The true power of closures is to “capture” the environment:

```
let color = String::from("green");
let print = || println!("`color`: {}", color);
print();
```

Here we get a reference to *color* and can use that when we call *print*.

Note that this is an immutable reference, so color can be used immutably outside the closure as well.

https://doc.rust-lang.org/stable/rust-by-example/fn/closures/capture.html

Closures - Capturing

But how about mutable variables?

```
let mut count = 0;
let mut inc = || count += 1;
inc();
```

Here we get a mutable reference to *count* which gets incremented every time we call *inc*.

Because this is mutable, we cannot use *count* while the closure lives. So trying to use *count* before the last *inc()* call is an error.

https://doc.rust-lang.org/stable/rust-by-example/fn/closures/capture.html

Closures - Capturing and Moving

Closures can also move captured variables.

```
let haystack = vec![1, 2, 3];
let contains = move |needle| haystack.contains(needle);
println!("{}", contains(&1));
```

Here we move the haystack into the closure, by explicitly using *move*.

After we create the closure, the *haystack* is not available outside the closure.

https://doc.rust-lang.org/stable/rust-by-example/fn/closures/capture.html

Closures - Capturing

These three types of capturing are encoded as *Fn* traits as follows:

- *FnOnce* - consumes the captured variables, moving them into the closure
- *FnMut* - mutable references borrowed from the environment
- *Fn* - immutable references

When creating a closure, the type trait is inferred by the compiler. This can be used to match on.

https://doc.rust-lang.org/book/ch13-01-closures.html#capturing-the-environment-with-closures

Storing Closures

Closures can also be stored and transferred.

```
struct Cacher<T>
where
    T: Fn(u32) -> u32,
{
    calculation: T,
    value: Option<u32>,
}
```

This struct can store a closure for a calculation, using the function signature type.

https://doc.rust-lang.org/book/ch13-01-closures.html#storing-closures-using-generic-parameters-and-the-fn-traits

Using a Stored Closure

```
impl<T> Cacher<T>
where
    T: Fn(u32) -> u32,
{
    fn new(calculation: T) -> Cacher<T> {
        Cacher {calculation, value: None}
    }
    fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v, // return cached value
            None => {     // run closure
                let v = (self.calculation)(arg);
                self.value = Some(v);
                v
            }
        }
    }
}
```

Concurrency

Concurrency in Rust is a first-class problem. The solution: the borrow checker!

“By leveraging ownership and type checking, many concurrency errors are compile-time errors in Rust rather than runtime errors.”

Rust forces you to program in certain ways to eliminate classes of concurrency problems. Additionally, it provides tools to help even more.

https://doc.rust-lang.org/book/ch16-00-concurrency.html

Threads

Note that the Rust std library only implements native threads, a 1:1 model.

So-called “green” threads implement M:N threading, with M application threads to N OS threads. There are crates that implement this, if you want it.

This choice was made to keep the default Rust runtime small, to fit on embedded platforms.

https://doc.rust-lang.org/book/ch16-01-threads.html

Thread Creation

Thread creation typically uses closures:

```
thread::spawn(|| {
    for i in 1..10 {
        println!("hi number {} from the spawned thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
});
thread::sleep(Duration::from_millis(5));
```

Here we print numbers from a thread, until the main thread ends. This will typically print 5 numbers.

https://doc.rust-lang.org/book/ch16-01-threads.html

Thread Joining

To wait for a thread to finish, we need to call *join*.

```
let handle = thread::spawn(|| {
    for i in 1..10 {
        println!("hi number {} from the spawned thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
});
thread::sleep(Duration::from_millis(5));
handle.join().unwrap();
```

This will print all the numbers from the thread before exiting.

https://doc.rust-lang.org/book/ch16-01-threads.html

Capturing Variables in a Thread

Sometimes we want to pass variables into a thread closure.

```
let v = vec![1, 2, 3];
let handle = thread::spawn(|| {
    println!("Here's a vector: {:?}", v);
});
handle.join().unwrap();
```

This will fail to compile, as it only passes a reference to *v* into the thread.

The issue is that Rust can't tell how long the thread will run, so *v* needs to be valid forever, even after *main* ends (because the thread doesn't end until the program itself ends).

https://doc.rust-lang.org/book/ch16-01-threads.html

Mutexes

But what if we do want to share data between threads, instead of moving it?

Rust's Mutex interface is similar to shared pointers, but for threaded contexts.

```
let m = Mutex::new(5);

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

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

https://doc.rust-lang.org/book/ch16-03-shared-state.html

Mutexes

The *lock* on a mutex allows for a single reference in the scope it is called, and automatically releases the lock when it goes out of scope.

If another thread currently holds the lock, the current thread will wait, blocking the thread.

If the other thread panics, no one will be able to get the lock. Thus, *unwrap* is appropriate here for handling the error case.

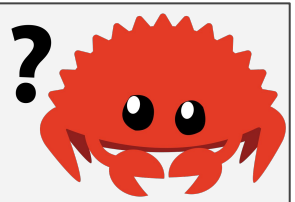
The return type from *lock* is a smart pointer, which automatically dereferences to the inner data.

https://doc.rust-lang.org/book/ch16-03-shared-state.html

Sharing Mutexes

```
let counter = Mutex::new(0);
let mut handles = vec![];

for _ in 0..10 {
    let handle = thread::spawn(move || {
        let mut num = counter.lock().unwrap();
        *num += 1;
    });
    handles.push(handle);
}
for handle in handles {
    handle.join().unwrap();
}
println!("Result: {}", *counter.lock().unwrap());
```



https://doc.rust-lang.org/book/ch16-03-shared-state.html#sharing-a-mutext-betwe

Sharing Mutexes

To have multiple owners of a Mutex, we need something similar to Rc<T>. That isn't thread-safe, but there is a new type that is: Arc<T>

The *Arc* uses atomic reference counting, which incurs a slight performance penalty in the single-threaded case. But it's exactly what we need for multiple threads.

https://doc.rust-lang.org/book/ch16-03-shared-state.html#atomic-reference-counting-with-arct

Sharing Mutexes

```
let counter = Arc::new(Mutex::new(0));
let mut handles = vec![];

for _ in 0..10 {
    let counter = Arc::clone(&counter);
    let handle = thread::spawn(move || {
        let mut num = counter.lock().unwrap();
        *num += 1;
    });
    handles.push(handle);
}
for handle in handles {
    handle.join().unwrap();
}
println!("Result: {}", *counter.lock().unwrap());
```

Shared Pointers vs Mutex/Arc

Notice the similarities between RefCell<T>/Rc<T> and Mutex<T>/Arc<T>.

Then have virtually the same interface, so can be used in similar patterns.

But this means that Mutex/Arc also has a defect: deadlocks

If two threads need to use two mutexes and grabs them in opposite order, the program will deadlock and wait forever.

https://doc.rust-lang.org/book/ch16-03-shared-state.html#similarities-between-refcelltrct-and-mutextarct

Message Passing

Another way to communicate between threads, without sharing memory, is message passing. This is especially useful in Rust, with strict ownership of memory.

The primary method in Rust is the *Channel*, a sender and receiver queue. Multiple senders are allowed, but only one receiver can exist.

https://doc.rust-lang.org/book/ch16-02-message-passing.html

Message Passing

```
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);
```

Here we make a channel and move the sender into the thread, then send a value to the main thread. There we can receive it and print it.

https://doc.rust-lang.org/book/ch16-02-message-passing.html

Message Passing

Some details on *send*/*recv*:

send returns a Result<T, E> to indicate if the channel has been closed. It also takes ownership of the value, to prevent using it again (because ownership has been transferred to the receiver).

recv will block until a value arrives, and also returns a Result<T, E> for signalling channel close.

There is another method *try_recv* which doesn't block and returns immediately if no value is available.

https://doc.rust-lang.org/book/ch16-02-message-passing.html

Message Passing

```
let (tx, rx) = mpsc::channel();
thread::spawn(move || {
    let vals = vec![String::from("hi"),String::from("main")];
    for val in vals {
        tx.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});
for received in rx {
    println!("Got: {}", received);
}
```

We can also iterate over the receiver to get multiple messages.

https://doc.rust-lang.org/book/ch16-02-message-passing.html#sending-multiple-va

Message Passing - Multiple Senders

```
let (tx, rx) = mpsc::channel();
let tx1 = tx.clone(); // here we create a copy of the sender
thread::spawn(move || {
    let val = String::from("thread 1");
    tx1.send(val).unwrap();
});
thread::spawn(move || {
    let val = String::from("thread 2");
    tx.send(val).unwrap();
});
for received in rx {
    println!("Got: {}", received);
}
```

https://doc.rust-lang.org/book/ch16-02-message-passing.html#creating-multiple-p

Concurrency Traits

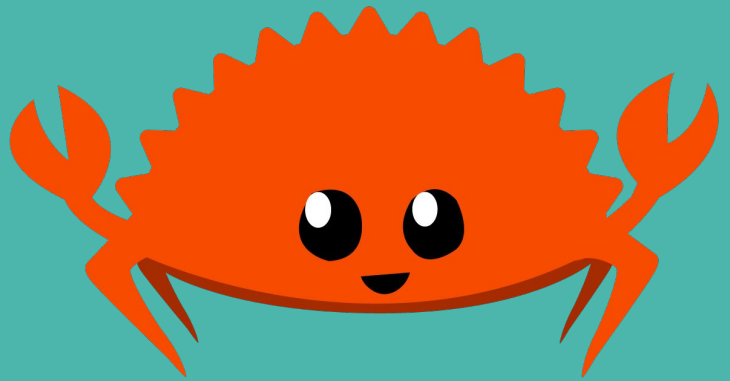
The main trait for message passing is *Send*. Most types implement it, and any new type composed of only *Send* types is automatically *Send*-enabled.

The main types that don't implement *Send* are things like *Rc<T>*, because they aren't thread-safe.

The *Sync* trait means a type is safe to use from multiple threads, or more accurately, a reference to the type is *Send*. Primitive types are *Sync*, and the Mutex is also *Sync*. A new type composed only of *Send* types is also *Send*.

https://doc.rust-lang.org/book/ch16-04-extensible-concurrency-sync-and-send.html

End of Lecture



Exercises

Write a parallel merge sort. A single-threaded version has been provided at exercises/day5/mergesort.