Rust Bootcamp 2021

Day 6

Day 6 Activities

- ✓ Packaging & Crates
- ✓ Asynchronous
- ✓ Exercises & Homework

Packages

Remember that *cargo new* creates a package.

“A package must contain zero or one library crates, and no more. It can contain as many binary crates as you’d like, but it must contain at least one crate (either library or binary).”

src/main.rs - binary crate root

src/lib.rs - library crate root

https://doc.rust-lang.org/book/ch07-01-packages-and-crates.html

Modules

Modules allow us to control scope and privacy.

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
        fn seat_at_table() {}
    }
    mod serving {
        fn take_order() {}
        fn serve_order() {}
        fn take_payment() {}
    }
}
```

```
crate
 └── front_of_house
     ├── hosting
     │   ├── add_to_waitlist
     │   └── seat_at_table
     └── serving
         ├── take_order
         ├── serve_order
         └── take_payment
```

https://doc.rust-lang.org/book/ch07-02-defining-modules-to-control-scope-and-privacy.html

Modules

To use something from a module, you need to specify the path

```
// Absolute path
crate::front_of_house::hosting::add_to_waitlist();

// Relative path
front_of_house::hosting::add_to_waitlist();
```

Absolute paths are preferred, as they do not change if you move the caller to a different module.

https://doc.rust-lang.org/book/ch07-03-paths-for-referring-to-an-item-in-the-module-tree.html

Modules - Super

The *super* keyword is a special relative path, that indicates the next module up in the hierarchy.

```
fn serve_order() {}

mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        super::serve_order();
    }

    fn cook_order() {}
}
```

https://doc.rust-lang.org/book/ch07-03-paths-for-referring-to-an-item-in-the-module-tree.html

Modules - Privacy

Things inside modules are private by default. The *pub* keyword makes them public.

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}
```

Here we make the second level module and third level function public, so we can call them from inside the crate. This also applies to structs and enums, and their implementations.

https://doc.rust-lang.org/book/ch07-03-paths-for-referring-to-an-item-in-the-module-tree.html

Modules - the *use* keyword

Using an absolute or relative path for everything gets tedious though, so the *use* keyword can bring things into the current scope.

```
use crate::front_of_house::hosting;
pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

Now we have a shorter path to call *hosting* functions, and only one place to change if we relocate the code.

https://doc.rust-lang.org/book/ch07-04-bringing-paths-into-scope-with-the-use-keyword.html

Modules - the *as* keyword

Sometimes we need to rename something we bring into scope with *use*:

```
use std::io::Result as IoResult;
fn function() -> IoResult<()> {
    // --snip--
}
```

This is useful if we have duplicate names and need to disambiguate them.

https://doc.rust-lang.org/book/ch07-04-bringing-paths-into-scope-with-the-use-keyword.html

Modules - nested paths

If we use multiple items defined in the same crate or module, it can be more concise to nest the paths onto a single line.

```
use std::{cmp::Ordering, io};
use std::io::{self, Write};
```

Note the *self* keyword, which imports the root nested path.

https://doc.rust-lang.org/book/ch07-04-bringing-paths-into-scope-with-the-use-keyword.html

Multiple Source Files

Say we put the *front_of_house* module in *src/front_of_house.rs*.

Then in *src/lib.rs*, we use that with

```
mod front_of_house; // load front_of_house from file

pub use crate::front_of_house::hosting;
pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

https://doc.rust-lang.org/book/ch07-05-separating-modules-into-different-files.html

Using External Packages

crates.io is the package repository for Rust, and contains many useful crates.

As an example, using random numbers. Cargo.toml needs:

```
[dependencies]
rand = "0.5.5"
```

And then we can use *rand* in our package:

```
use rand::Rng;

let secret_number = rand::thread_rng().gen_range(1, 101);
```

https://doc.rust-lang.org/book/ch07-04-bringing-paths-into-scope-with-the-use-keyword.html#using-external-packages

Asynchronous Programming in Rust

Async in rust is designed to be zero-cost:

- futures only run when polled (they do not run in the background)
- there are no heap allocations or dynamic dispatch

But, there is no built-in runtime for async in Rust - it is provided by external crates. Only the syntax/framework for async programming is built in.

https://rust-lang.github.io/async-book/01_getting_started/02_why_async.html

Async vs Threads

OS threads are a relatively heavy-weight solution. Spawning and switching threads is expensive, and you can only have so many threads before the stack crashes.

Async has a reduced CPU and memory overhead, so you can have many more async tasks than threads. But, async programs are larger because of more complex state machines.

https://rust-lang.github.io/async-book/01_getting_started/02_why_async.html

Simple Async Example

In this example, we download two websites at the same time.

```
async fn get_two_sites_async() {
    // Create two different "futures" which, when run to
    // completion, will asynchronously download the webpages.
    let future_one = download_async("https://www.foo.com");
    let future_two = download_async("https://www.bar.com");

    // Run both futures to completion at the same time.
    join!(future_one, future_two);
}
```

https://rust-lang.github.io/async-book/01_getting_started/02_why_async.html

Setting up for Async

In order to run async code, we need to add some dependencies.

The first is for the current futures crate:

```
[dependencies]
futures = "0.3"
```

This gives us a basic executor for async functions, to call from synchronous code (like a main function).

https://rust-lang.github.io/async-book/01_getting_started/04_async_await_primer.html

Setting up for Async

Now we can run a basic async program:

```
use futures::executor::block_on;

async fn hello_world() {
    println!("hello, world!");
}

fn main() {
    let future = hello_world(); // Nothing is printed
    block_on(future); // Now the future runs
}
```

https://rust-lang.github.io/async-book/01_getting_started/04_async_await_primer.html

Setting up for Async

If we want to run an async function from another one, use the *await* keyword.

```
async fn learn_and_sing() {
    // Wait until the song has been learned before singing it.
    let song = learn_song().await;
    sing_song(song).await;
}
```

Note that *await* is on the end of the function call, instead of in front like many other languages. Rust did this to allow chained calls, like *unwrap*, after the *await*.

https://rust-lang.github.io/async-book/01_getting_started/04_async_await_primer.html

Async blocks

Rust also allows async blocks within regular functions:

```
// `foo()` returns a type that implements `Future<Output = u8>`.
// `foo().await` will result in a value of type `u8`.
async fn foo() -> u8 { 5 }

fn bar() -> impl Future<Output = u8> {
    // This `async` block results in a type that implements
    // `Future<Output = u8>`.
    async {
        let x: u8 = foo().await;
        x + 5
    }
}
```

h

Async lifetimes and types

One benefit over threads is that lifetimes work properly again. As long as you await the Future within the lifetime of the variable you pass in, Rust is happy.

One note about types: since many async executors are multi-threaded, types will need to be thread-safe. This means *Rc/RefCell* should not be used.

If you need to use a Mutex, there is a special one in *futures::lock*, that is async-aware.

https://rust-lang.github.io/async-book/03_async_await/01_chapter.html

Executing multiple futures

As shown earlier, the correct way to run multiple futures concurrently is:

```
use futures::join;

async fn get_book_and_music() -> (Book, Music) {
    let book_fut = get_book();
    let music_fut = get_music();
    join!(book_fut, music_fut)
}
```

If we had used *await* on each of them, they would run sequentially.

https://rust-lang.github.io/async-book/06_multiple_futures/02_join.html

Executing multiple futures

If a future returns a Result and could fail, there is a similar construct to fail fast. *join* will run all futures to completion, but *try_join* will immediately stop.

```
use futures::join;

async fn get_book_and_music() -> (Book, Music) {
    let book_fut = get_book();
    let music_fut = get_music();
    try_join!(book_fut, music_fut)
}
```

Note that the Result types must contain the same Error type for *try_join*.

https://rust-lang.github.io/async-book/06_multiple_futures/02_join.html

Executing multiple futures

If a future returns a Result and could fail, there is a similar construct to fail fast. *join* will run all futures to completion, but *try_join* will immediately stop.

```
use futures::join;

async fn get_book_and_music() -> (Book, Music) {
    let book_fut = get_book();
    let music_fut = get_music();
    try_join!(book_fut, music_fut)
}
```

Note that the Result types must contain the same Error type for *try_join*.

https://rust-lang.github.io/async-book/06_multiple_futures/02_join.html

Tokio

I said previously that the executor is implemented by external crates. Tokio is the most popular crate for asynchronous programming, and provides much more than just an executor.

It has additional synchronization primitives, timeouts, sleeps, and other tools.

There are also APIs for working with IO: sockets, filesystem operations, and process management

And finally, the runtime/executor, including a task scheduler, event queue, and high performance timer.

https://docs.rs/tokio/

Using Tokio

Cargo.toml needs:

```
[dependencies]
tokio = { version = "1", features = ["full"] }
```

And then we can use *tokio* in our package:

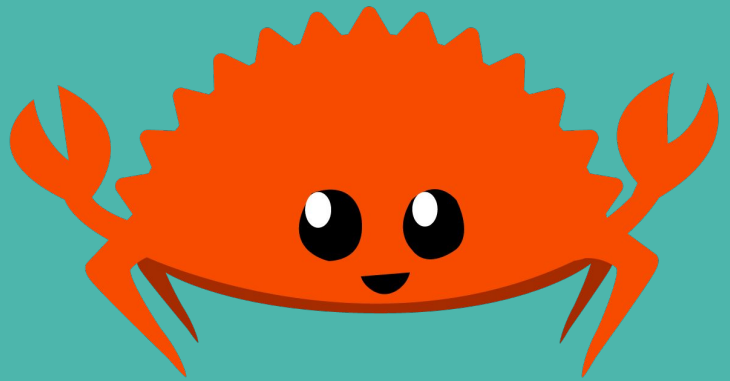
```
#[tokio::main]
async fn main() {
    do_things().await;
}
```

Yes, *tokio* has a way to run an asynchronous main function as well.

https://docs.rs/tokio/

```rust
use tokio::net::TcpListener;                                    //  A simple TCP echo server
use tokio::io::{AsyncReadExt, AsyncWriteExt};
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let listener = TcpListener::bind("127.0.0.1:8080").await?;
    loop {
        let (mut socket, _) = listener.accept().await?;
        tokio::spawn(async move {
            let mut buf = [0; 1024];
            loop { // In a loop, read data from the socket and write the data back.
                let n = match socket.read(&mut buf).await {
                    // socket closed
                    Ok(n) if n == 0 => return,
                    Ok(n) => n,
                    Err(e) => {
                        eprintln!("failed to read from socket; err = {:?}", e);
                        return;
                    }
                };
                // Write the data back
                if let Err(e) = socket.write_all(&buf[0..n]).await {
                    eprintln!("failed to write to socket; err = {:?}", e);
                    return;
                }
            }
        });
    }
}
```

End of Lecture



Exercises

Take the merge sort from the day 5 exercise, and make it async.

A single-threaded version has been provided at exercises/day6/mergesort, if you need a reminder.

Homework

As the last homework assignment, this can also be thought of as a final project.

Write a web server which serves at least one url and and error page. The goal is to see how many connections you can serve at once, with virtual points going to the one(s) with highest throughput and lowest latency.

A stub is in homework/day6, which includes a test program for statistics.
It can do about 1000 requests/s. Performance >200k requests/s is possible.